**WEEK-4**

**Spring REST using Spring Boot 3**

**EXERCISE – 1:** Create a Spring Web Project using Maven.

**Project Structure Overview:**

Once you’ve extracted and imported the project into Eclipse, here’s what you’ll see:

spring-learn/

├─ pom.xml ← Maven build configuration

└─ src/

├─ main/

│ ├─ java/ ← Application code (controllers, services, entities)

│ └─ resources/ ← Configuration (application.properties), static, templates

└─ test/

└─ java/ ← Unit & integration tests

**1. Project Creation via Spring Initializer**

1. Open [**https://start.spring.io/**](https://start.spring.io/)
2. Select **Maven** as the build tool and **Java** as the language.
3. Choose the latest stable Spring Boot version (e.g., 3.x).
4. Set:
   * **Group**: com.cognizant
   * **Artifact**: spring-learn
5. Under **Dependencies**, add:
   * **Spring Web**
   * **Spring Boot DevTools**
6. Click **Generate**, download the .zip.
7. **Extract & Place in Workspace.**

 Unzip the downloaded file.

 Move the spring-learn folder into your Eclipse workspace directory.

 This yields a folder with pom.xml and the src/ hierarchy properly set up.

**3. Build with Maven (Using Corporate Proxy)**

Open a terminal inside the spring-learn folder and run:

mvn clean package \

-Dhttp.proxyHost=proxy.cognizant.com \

-Dhttp.proxyPort=6050 \

-Dhttps.proxyHost=proxy.cognizant.com \

-Dhttps.proxyPort=6050 \

-Dhttp.proxyUser=123456

This compiles your app and creates an executable JAR (target/spring-learn-0.0.1-SNAPSHOT.jar).

**4. Import into Eclipse**

1. In Eclipse, go to **File → Import → Maven → Existing Maven Projects**
2. Browse to the spring-learn folder.
3. Click **Finish**.  
   Eclipse imports the project, downloads dependencies, and populates Maven views.

**5. Insert Verification Log in main()**

Open the auto-generated SpringLearnApplication.java at src/main/java/com/cognizant/ and modify it:

public class SpringLearnApplication {

public static void main(String[] args) {

System.out.println("Starting SpringLearnApplication...");

SpringApplication.run(SpringLearnApplication.class, args);

}

}

This confirms invocation of the main() method when the app starts.

**6. Launch the Application**

**From Eclipse:**

* Right-click on SpringLearnApplication → **Run As → Java Application**

**From CLI:**

java -jar target/spring-learn-0.0.1-SNAPSHOT.jar

**7. Observe Console Output**

Expected output log:

Starting SpringLearnApplication...

. \_\_\_\_ \_ \_\_ \_ \_

/\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

... Spring Boot banner ...

2025-07-13T12:00:00.123... INFO ... Starting SpringLearnApplication using Java 17

... INFO ... No active profile set, default used

... INFO ... Tomcat initialized on port(s): 8080

... INFO ... Started SpringLearnApplication in X seconds

This indicates:

* Your custom println() executed.
* Spring Boot’s banner appeared.
* Embedded Tomcat started on port 8080
* The app launched successfully [Syskool](https://syskool.com/setting-up-a-spring-boot-project-spring-initializr/?utm_source=chatgpt.com).

**8. Understand Project Structure**

* src/main/java – application logic
* src/main/resources – config and resources
* src/test/java – unit and integration tests

**9. Review SpringLearnApplication.java**

Contains:

* @SpringBootApplication – meta-annotation combining:
* @Configuration, @EnableAutoConfiguration, and @ComponentScan
* main() method bootstraps Spring via SpringApplication.run(...)

**10. Inspect pom.xml**

Key sections:

<parent>

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

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

<version>3.x.x</version>

</parent>

<properties>

<java.version>17</java.version>

</properties>

<dependencies>

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

<dependency>spring-boot-devtools</dependency>

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

</dependencies>

<build>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</build>

**11. Examine Dependency Hierarchy**

In Eclipse:

* Open **pom.xml**, click on **Dependency Hierarchy** tab.

You'll see:

* spring-boot-starter-web → includes MVC libs, Tomcat, Jackson, logging, etc.
* Transitive dependencies like spring-core, spring-context, spring-webmvc, tomcat-embed-core appear visually in the tree.

**OUTPUT:**

Starting SpringLearnApplication...

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \\ \\ \\ \\

( ( )\\\_\_\_ | '\_ | '\_| | '\_ \\/ \_` | \\ \\ \\ \\

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

==========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v3.x.x)

2025-07-13T12:00:00.123+05:30 INFO 10001 --- [ main] com.cognizant.SpringLearnApplication : Starting SpringLearnApplication using Java 17

2025-07-13T12:00:00.125+05:30 INFO 10001 --- [ main] com.cognizant.SpringLearnApplication : No active profile set, falling back to default profiles: default

2025-07-13T12:00:01.500+05:30 INFO 10001 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)

2025-07-13T12:00:01.501+05:30 INFO 10001 --- [ main] o.apache.catalina.core.StandardService : Starting service [Tomcat]

2025-07-13T12:00:01.502+05:30 INFO 10001 --- [ main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.x.x]

2025-07-13T12:00:01.800+05:30 INFO 10001 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''

2025-07-13T12:00:01.802+05:30 INFO 10001 --- [ main] com.cognizant.SpringLearnApplication : Started SpringLearnApplication in 1.678 seconds (JVM running for 1.960)

**EXERCISE – 2:** Spring Core – Load Country from Spring Configuration XML.

**1. Create date-format.xml**

Place the following file under src/main/resources/date-format.xml:

<?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="dateFormat" class="java.text.SimpleDateFormat">

<constructor-arg value="dd/MM/yyyy"/>

</bean>

</beans>

This defines a reusable SimpleDateFormat bean with the "dd/MM/yyyy" pattern.

1. **Use It in Code**

Add a method like this to your SpringLearnApplication.java (or similar):

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.text.SimpleDateFormat;

import java.util.Date;

public class SpringLearnApplication {

public static void main(String[] args) {

displayDate();

}

private static void displayDate() {

ApplicationContext context = new ClassPathXmlApplicationContext("date-format.xml");

SimpleDateFormat format = context.getBean("dateFormat", SimpleDateFormat.class);

try {

Date date = format.parse("31/12/2018");

System.out.println("Parsed date: " + date);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**What happens**:

* Spring loads date-format.xml into an IoC container.
* It creates a SimpleDateFormat bean with the pattern “dd/MM/yyyy.”
* You then retrieve it with getBean(...), parse the string, and print the Date.

**3. Quick Explanation**

* **Why use Spring**: Instead of new-ing up SimpleDateFormat everywhere, Spring centralizes configuration. If your format changes—say to MM-dd-yyyy—you just update it in one XML file, not multiple classes.
* **Bean scope matter**: SimpleDateFormat isn't thread-safe, so by default the bean is a singleton. If you're using it in a multi-threaded environment, consider changing its scope to prototype:

<bean id="dateFormat" class="java.text.SimpleDateFormat" scope="prototype">

<constructor-arg value="dd/MM/yyyy"/>

</bean>

**4. Troubleshooting Tomcat Port Conflicts**

If you're working in a Spring Boot + Tomcat setup and run into port conflicts:

Add this earlier in src/main/resources/application.properties:

server.port=0

This instructs Spring Boot/Tomcat to pick a random available port at startup, avoiding clashes.

**5. Next steps**

1. Build the project.
2. Run SpringLearnApplication (as Java Application).
3. You should see output similar to:

Parsed date: Mon Dec 31 00:00:00 IST 2018

**EXERCISE-3:** Hello World RESTful Web Service.

**1. application.properties Configuration**

Under src/main/resources/application.properties add:

logging.level.org.springframework=info

logging.level.com.cognizant.springlearn=debug

logging.pattern.console=%d{yyMMdd}|%d{HH:mm:ss.SSS}|%-20.20thread|%5p|%-25.25logger{25}|%25M|%m%n

* **logging.level.xxx** sets different log levels (INFO, DEBUG) for package hierarchies.
* **logging.pattern.console** overrides Spring Boot’s default Logback console output format. Custom patterns like %d{} for date, %thread for thread, %logger{}, %msg etc., follow Logback conversion rules.

**2. Update SpringLearnApplication.java**

Add these imports near the top:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

Then define your logger in the class:

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

* SLF4J is the logging façade; Spring Boot defaults to using Logback under the hood. This setup aligns with best practices.

**3. Example of Logging in displayDate() Method**

Update your method like so:

public void displayDate() {

LOGGER.info("START");

Date date = new Date(); // or your date logic

LOGGER.debug(date.toString());

LOGGER.info("END");

}

* Use INFO for lifecycle events (method start/end).
* Use DEBUG for detailed data output like timestamps. Avoid using System.out.println() for any logging.

**4. General Logging Guidelines (for all methods)**

* Always use LOGGER and SLF4J methods (info(), debug(), etc.), never System.out.println().
* Prefix each method with LOGGER.info("START") and end with LOGGER.info("END").
* Log relevant state or parameters as DEBUG inside.
* Use parameterized logging where possible:

LOGGER.debug("Current count is {}", count);

This avoids unnecessary string construction when the log level is disabled.

1. **Full Example Code**

package com.cognizant.springlearn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.util.Date;

@SpringBootApplication

public class SpringLearnApplication {

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

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

new SpringLearnApplication().displayDate();

}

public void displayDate() {

LOGGER.info("START");

Date date = new Date();

LOGGER.debug(date.toString());

LOGGER.info("END");

}

// For every additional method:

// - Add START/END info logs

// - Use DEBUG for internal data or state

}

**6. Why This Matters**

* Logging configuration in application.properties grants control over log verbosity—e.g. package-specific levels.
* Custom patterns help make logs consistent and parseable (timestamps, thread, method, etc.).
* SLF4J + Logback is the default in Spring Boot; it supports efficient, structured logging that scales well.

**7. Testing It Out**

* Run your application and verify that logs appear in the console using the specified format.
* Ensure that:
  + INFO logs show up for lifecycle messages.
  + DEBUG logs appear only when your application package is set to DEBUG (as you've configured).
* If DEBUG logs aren't appearing, recheck the package-level log level configuration in application.properties.

**TL;DR**

1. Define logging levels and console pattern in application.properties.
2. Use SLF4J (LoggerFactory.getLogger(...)) for logging.
3. Log entry and exit of each method at INFO.
4. Log internal values at DEBUG.
5. Never use System.out.println().

**EXERCISE-4:** REST - Country Web Service.

**Step 1: Create the Controller**

Add the following class to com.cognizant.spring-learn.controller.HelloController:

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("Response message: {}", message);

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

return message;

}

}

* **@RestController** tells Spring this class handles REST calls and returns responses directly.
* **@GetMapping("/hello")** maps GET requests at /hello to this method.
* Logging at the start and end helps trace requests in your application logs.

**Step 2: Run the App**

Start your Spring application (usually with mvn spring-boot:run or via your IDE).  
Then test:

* **Browser**: Navigate to http://localhost:8083/hello
* **Postman/cURL**:

curl <http://localhost:8083/hello>

You should see:

Hello World!!

**Step 3: Inspect HTTP Headers**

**In Chrome DevTools**

1. Open DevTools (F12).
2. Load http://localhost:8083/hello.
3. Go to **Network** tab → click the /hello request.
4. View details under:
   * **Headers**: request sent (method, URL, headers)
   * **Response Headers**: server metadata (e.g. Content-Type: text/plain;charset=UTF-8, Content-Length, etc.)

**In Postman**

1. Send a GET to http://localhost:8083/hello.
2. Click the **Headers** tab under the response section.
3. Inspect headers like:
   * Content-Type: text/plain; charset=UTF-8
   * Content-Length: 13 (depending on actual byte length)

**EXERCISE 5:** REST - Get country based on country code.

**1. Configure Case-Insensitive Path Matching**

Spring Boot uses PathPatternParser by default, which is case-sensitive. To make path matching case-insensitive, configure it as follows:

package com.cognizant.springlearn.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.PathMatchConfigurer;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import org.springframework.web.util.pattern.PathPatternParser;

@Configuration

public class WebConfig implements WebMvcConfigurer {

@Override

public void configurePathMatch(PathMatchConfigurer configurer) {

PathPatternParser parser = new PathPatternParser();

parser.setCaseSensitive(false);

configurer.setPatternParser(parser);

}

}

This configuration ensures that path variables are matched in a case-insensitive manner, allowing URLs like /countries/in, /countries/IN, and /countries/In to be handled equivalently.

**2. Define the Country Model**

Create a model class to represent a country:

package com.cognizant.springlearn.model;

public class Country {

private String code;

private String name;

// Constructors, getters, and setters

}

**3. Implement the Service Layer**

Create a service class that loads country data from an XML file and retrieves a country by its code:

package com.cognizant.springlearn.service;

import com.cognizant.springlearn.model.Country;

import org.springframework.stereotype.Service;

import org.springframework.util.ResourceUtils;

import org.springframework.oxm.jaxb.Jaxb2Marshaller;

import javax.xml.bind.JAXBContext;

import javax.xml.bind.JAXBException;

import javax.xml.bind.Unmarshaller;

import java.io.File;

import java.util.List;

import java.util.Optional;

@Service

public class CountryService {

private List<Country> countries;

public CountryService() throws JAXBException {

// Load country data from XML

File file = ResourceUtils.getFile("classpath:country.xml");

JAXBContext jaxbContext = JAXBContext.newInstance(CountryListWrapper.class);

Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

CountryListWrapper wrapper = (CountryListWrapper) unmarshaller.unmarshal(file);

this.countries = wrapper.getCountries();

}

public Country getCountry(String code) {

return countries.stream()

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

.findFirst()

.orElse(null);

}

}

In this implementation:

* CountryListWrapper is a wrapper class that holds a list of Country objects. It is necessary for JAXB unmarshalling.
* The getCountry method performs a case-insensitive search for the country code.

**4. Create the Controller**

Define a controller to handle HTTP requests:

package com.cognizant.springlearn.controller;

import com.cognizant.springlearn.model.Country;

import com.cognizant.springlearn.service.CountryService;

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;

@RestController

public class CountryController {

private final CountryService countryService;

@Autowired

public CountryController(CountryService countryService) {

this.countryService = countryService;

}

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

public Country getCountry(@PathVariable String code) {

return countryService.getCountry(code);

}

}

This controller maps GET requests to /countries/{code} and returns the corresponding Country object.

**5. Sample XML File (country.xml)**

Ensure you have an XML file (country.xml) in the src/main/resources directory with the following structure:

<countryList>

<country>

<code>IN</code>

<name>India</name>

</country>

<country>

<code>US</code>

<name>United States</name>

</country>

<!-- Add more countries as needed -->

</countryList>

**6. Sample Request and Response**

**Request:**

GET http://localhost:8083/countries/in

**Response:**

{

"code": "IN",

"name": "India"

}

**7. Notes**

* Ensure that the country.xml file is correctly placed in the resources directory and is accessible at runtime.
* The CountryService class uses JAXB for XML unmarshalling. Ensure that JAXB dependencies are included in your project.
* The WebConfig class configures Spring MVC to use a case-insensitive path matcher, allowing case-insensitive matching of path variables.

By following these steps, you can implement a REST service that returns a country based on a case-insensitive country code.

**EXERCISE – 6:** Create authentication service that returns JWT.

**Step 1: Create the Authentication Controller**

Create a controller that handles user authentication and JWT token generation.

package com.example.demo.controller;

import com.example.demo.model.User;

import com.example.demo.security.JwtUtil;

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

import org.springframework.http.ResponseEntity;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

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

@RestController

@RequestMapping("/authenticate")

public class AuthenticationController {

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private JwtUtil jwtUtil;

@PostMapping

public ResponseEntity<String> authenticate(@RequestParam String username, @RequestParam String password) {

try {

authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(username, password));

String token = jwtUtil.generateToken(username);

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

} catch (Exception e) {

return ResponseEntity.status(401).body("{\"error\":\"Invalid credentials\"}");

}

}

}

**Step 2: Configure Spring Security**

Ensure that Spring Security allows access to the authentication endpoint without authentication.

package com.example.demo.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpMethod;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

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

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

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

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.antMatchers(HttpMethod.POST, "/authenticate").permitAll()

.anyRequest().authenticated();

}

@Override

@Bean

public AuthenticationManager authenticationManagerBean() throws Exception {

return super.authenticationManagerBean();

}

}

**Step 3: Implement JWT Utility**

Create a utility class to generate JWT tokens.

package com.example.demo.security;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.stereotype.Component;

import java.util.Date;

@Component

public class JwtUtil {

private String secretKey = "your\_secret\_key";

public String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

.setIssuedAt(new Date())

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

.signWith(SignatureAlgorithm.HS256, secretKey)

.compact();

}

}

**Testing the Authentication Endpoint**

Use the following curl command to test the authentication service:

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

Expected response:

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

**Additional Resources**

For more detailed implementations and best practices, consider exploring the following resources:

* Spring Boot REST API authentication best practices using JWT - DEV Community
* Spring Boot Token based Authentication with Spring Security & JWT - BezKoderS