**Hands on 1**

**Create a Spring Web Project using Maven**   
  
Follow steps below to create a project: 

1. Go to <https://start.spring.io/>
2. Change Group as “com.cognizant”
3. Change Artifact Id as “spring-learn”
4. Select Spring Boot DevTools and Spring Web
5. Create and download the project as zip
6. Extract the zip in root folder to Eclipse Workspace
7. 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
8. Import the project in Eclipse "File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish"
9. Include logs to verify if main() method of SpringLearnApplication.
10. Run the SpringLearnApplication class.

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

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

Solution:

Hands on 1: Create a Spring Web Project using Maven

Follow steps below to create a project:

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

2. Change Group to “com.cognizant”

3. Change Artifact Id to “spring-learn”

4. Select Spring Boot DevTools and Spring Web

5. Create and download the project as zip

6. Extract the zip in root folder to Eclipse Workspace

7. Build the project using the following command in the command line:

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

8. Import the project in Eclipse:

File > Import > Maven > Existing Maven Projects > Click Browse and select extracted folder > Finish

9. Include logs to verify if main() method of SpringLearnApplication is executed.

10. Run the SpringLearnApplication class.

SME Walkthrough Points:

1. src/main/java

- Folder containing the application source code.

2. src/main/resources

- Folder containing configuration files like application.properties or application.yml.

3. src/test/java

- Folder containing test code written for the application.

4. SpringLearnApplication.java

- Contains the main() method which is the entry point of the Spring Boot application.

- Sample:

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

}

5. Purpose of @SpringBootApplication annotation:

- It is a convenience annotation that adds:

- @Configuration: Tags the class as a source of bean definitions.

- @EnableAutoConfiguration: Tells Spring Boot to start adding beans based on classpath settings and other beans.

- @ComponentScan: Tells Spring to look for components, configurations, and services in the specified package.

6. pom.xml

1. Walkthrough all the configuration defined in the XML file:

- Includes the project coordinates (groupId, artifactId, version)

- Contains dependencies for Spring Boot DevTools and Spring Web

- Specifies the build plugins and Maven properties

2. Open 'Dependency Hierarchy':

- In Eclipse, go to the 'pom.xml' tab > Select 'Dependency Hierarchy'

- This shows a tree view of all direct and transitive dependencies used in the project

**Hands on 4**

**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>

* 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

Solution:

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

Scenario:

An airline website supports booking from 4 countries. Each country has a name and two-character ISO code. This data is stored in a Spring XML configuration file and displayed using a Java Spring program.

Steps to implement:

1. Create `country.xml` Spring configuration file 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

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>

---------------------------------------------------

2. Create `Country.java` in `com.cognizant.springlearn` package:

---------------------------------------------------

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("Getting country code.");

return code;

}

public void setCode(String code) {

LOGGER.debug("Setting country code.");

this.code = code;

}

public String getName() {

LOGGER.debug("Getting country name.");

return name;

}

public void setName(String name) {

LOGGER.debug("Setting country name.");

this.name = name;

}

@Override

public String toString() {

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

}

}

---------------------------------------------------

3. Update `SpringLearnApplication.java` to include displayCountry() method:

---------------------------------------------------

public class SpringLearnApplication {

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

public static void main(String[] args) {

displayCountry();

}

public static void displayCountry() {

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

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

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

}

}

---------------------------------------------------

SME Notes and Explanation:

• `<bean>` tag: Defines a Spring-managed object.

• `id` attribute: Unique identifier for the bean.

• `class` attribute: Fully qualified class name of the bean.

• `<property>` tag: Used to set values in the bean.

• `name` attribute: Name of the property to set.

• `value` attribute: Value to assign to the property.

• `ApplicationContext`: Spring container that manages lifecycle and configuration of beans.

• `ClassPathXmlApplicationContext`: Reads Spring configuration from XML file in classpath.

What happens during `context.getBean("country", Country.class)`:

1. The Spring container parses the XML.

2. It creates an instance of the specified class (`Country`).

3. It calls the constructor (logging "Inside Country Constructor").

4. It sets the properties using setter methods (logging from each setter).

5. Returns the fully initialized bean.

Run the main() method and check logs to verify constructor, setters, and toString method were invoked.

**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

Solution:

Hello World RESTful Web Service - Spring Boot

Objective:

Create a RESTful endpoint that returns "Hello World!!" using Spring Web Framework.

Steps to Implement:

1. Create a new controller class: HelloController.java in package `com.cognizant.springlearn.controller`

---------------------------------------------------

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

String message = "Hello World!!";

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

return message;

}

}

---------------------------------------------------

2. Ensure that the Spring Boot application class `SpringLearnApplication.java` is in the base package or a parent package of `controller`.

3. Run the application using the main method in `SpringLearnApplication.java`.

4. Access the endpoint:

- URL: http://localhost:8083/hello

- Method: GET

- Expected Response: Hello World!!

Note: If your application is not running on port 8083, you must set it in `src/main/resources/application.properties`:

---------------------------------------------------

server.port=8083

---------------------------------------------------

SME Walkthrough:

• Network Tab in Browser (Chrome):

- Open Developer Tools (F12) > Network Tab > Visit http://localhost:8083/hello

- Click on the request for `/hello` and check the "Headers" tab.

- You will see:

- Request URL: http://localhost:8083/hello

- Request Method: GET

- Status Code: 200 OK

- Content-Type: text/plain

• Postman:

- Open Postman and send a GET request to http://localhost:8083/hello

- Go to the "Headers" tab of the response section.

- Common response headers will include:

- Content-Type: text/plain

- Date

- Content-Length

- Server

This setup ensures a simple REST service is created, responding with static content and logging method entry and exit.

**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

REST - Country Web Service

Objective:

Create a RESTful endpoint that returns India country details as a JSON response by loading it from Spring XML configuration.

Steps to Implement:

1. country.xml (place 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

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>

2. CountryController.java (in com.cognizant.springlearn.controller):

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.ClassPathXmlApplicationContext;

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);

@RequestMapping("/country")

public Country getCountryIndia() {

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

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

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

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

return country;

}

}

3. Run SpringLearnApplication.java (with main method):

public static void main(String[] args) {

SpringApplication.run(SpringLearnApplication.class, args);

}

4. Optional: Set server port in application.properties

server.port=8083

Testing:

URL: http://localhost:8083/country

Method: GET

Expected Response:

{

"code": "IN",

"name": "India"

}

SME Walkthrough:

• What happens in the controller method?

- ApplicationContext loads country.xml

- Bean with id "country" is retrieved and returned

- Spring serializes the Country object to JSON

• How the bean is converted into JSON?

- Spring Boot uses Jackson to auto-convert POJOs to JSON

- @RestController + @ResponseBody enables this conversion

• Network tab in browser:

- Press F12 > Network > Reload URL

- View headers like:

- Content-Type: application/json

- Status Code: 200 OK

• Postman Headers tab:

- View:

- Content-Type: application/json

- Content-Length

- Date

- Server

**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.

Solution:

Authentication Service to Return JWT

Objective:

Expose a secure REST API that authenticates user credentials using basic auth and returns a signed JWT token.

Example:

Request:

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

Response:

{"token":"<your\_jwt\_token>"}

---

Step 1: Create Authentication Controller

1. Create a new controller class: AuthController.java

----------------------------------------------------------

package com.cognizant.springlearn.controller;

import com.cognizant.springlearn.util.JwtUtil;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.ResponseEntity;

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;

@RestController

public class AuthController {

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

@GetMapping("/authenticate")

public ResponseEntity<?> authenticate(@RequestHeader("Authorization") String authHeader) {

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

// Step 2: Decode Basic Auth header

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

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

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 database/real user validation

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

// Step 3: Generate JWT

String token = JwtUtil.generateToken(username);

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

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

}

}

LOGGER.warn("Unauthorized attempt");

return ResponseEntity.status(401).body("Unauthorized");

}

}

----------------------------------------------------------

---

Step 2: Utility Class to Generate JWT

Create JwtUtil.java under com.cognizant.springlearn.util

----------------------------------------------------------

package com.cognizant.springlearn.util;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import java.util.Date;

public class JwtUtil {

private static final String SECRET\_KEY = "secret123"; // Store securely in production

private static final long EXPIRATION\_TIME = 1000 \* 60 \* 10; // 10 minutes

public static String generateToken(String username) {

return Jwts.builder()

.setSubject(username)

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

.setExpiration(new Date(System.currentTimeMillis() + EXPIRATION\_TIME))

.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)

.compact();

}

}

----------------------------------------------------------

---

Step 3: Configure Security

Update 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;

@SuppressWarnings("deprecation")

@Configuration

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.csrf().disable()

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

.anyRequest().authenticated()

.and().httpBasic();

}

}

----------------------------------------------------------

---

Testing:

Start the server on port 8090 (set in application.properties):

server.port=8090

Run the following in terminal:

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

Expected Response:

{"token":"<JWT\_TOKEN\_HERE>"}

---

Explanation for SME:

• In the controller:

- We read the "Authorization" header from the request.

- Decode Base64 credentials to get username and password.

- Validate user and generate token using `JwtUtil`.

• Token to JSON:

- We manually return JSON string with `{"token": "..."}` format.

- Jackson can also be used to return proper POJOs as JSON.

• Network Tab (Browser/Postman):

- Headers will show:

- Authorization: Basic base64(user:pwd)

- Content-Type: application/json

- Status Code: 200 OK or 401 Unauthorized