**HandsOn-1: Create a Spring Web Project using Maven**

// SpringLearnApplication.java

package com.cognizant.springlearn;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringLearnApplication {

public static void main(String[] args) {

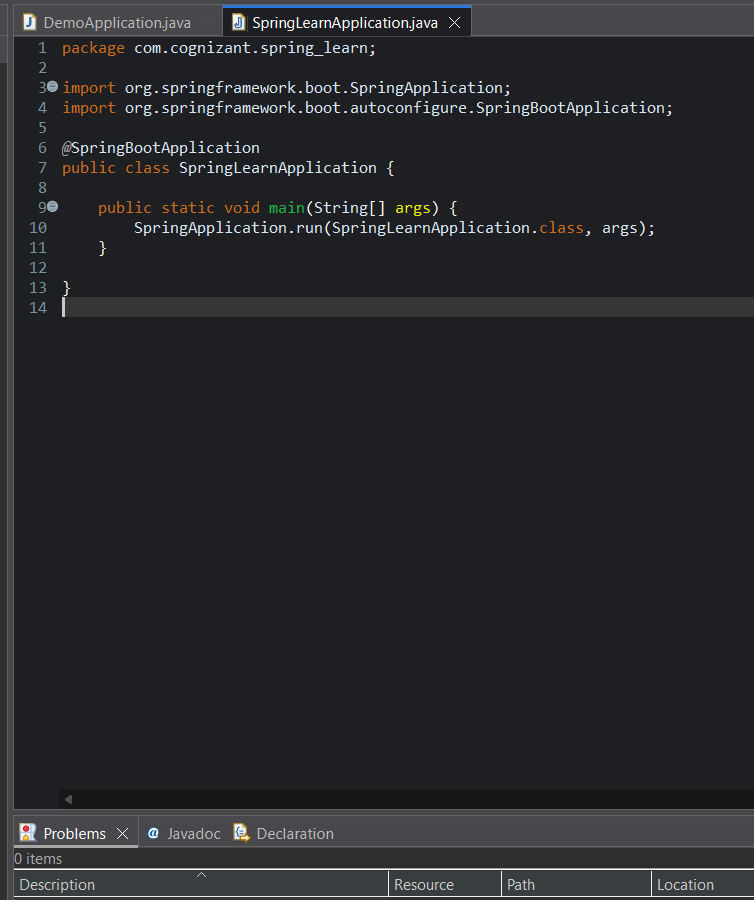
SpringApplication.run(SpringLearnApplication.class, args);

}

}

1. Steps to create project:
   * Used [start.spring.io](https://start.spring.io/) with:
     + Group: com.cognizant
     + Artifact: spring-learn
     + Dependencies: Spring Web, DevTools
   * Built with mvn clean package and imported into Eclipse.

* Key Components:
  + @SpringBootApplication: Combines:
    - @Configuration (Spring config class)
    - @EnableAutoConfiguration (auto-configures beans)
    - @ComponentScan (scans for components)
  + pom.xml: Contains Spring Boot dependencies (like spring-boot-starter-web).
* Project Structure:
  + src/main/java → Main app code (SpringLearnApplication.java)
  + src/main/resources → Config files (application.properties)



1. **Spring Core – Load SimpleDateFormat from Spring Configuration XML**
2. **XML Configuration (**date-format.xml**)**

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

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

</bean>

1. **Java Code to Load Bean**

public void displayDate() {

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

SimpleDateFormat fmt = ctx.getBean("dateFormat", SimpleDateFormat.class);

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

System.out.println(date);

}

**Explanation**:

* ClassPathXmlApplicationContext loads beans from XML.
* <constructor-arg> injects the date pattern.

**3.Spring Core - Incorporate Logging**

**1.application.properties**

properties

logging.level.org.springframework=info

logging.level.com.cognizant.springlearn=debug

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

**2.Modified**displayDate()**with Logging**

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

public void displayDate() {

LOGGER.info("START");

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

SimpleDateFormat fmt = ctx.getBean("dateFormat", SimpleDateFormat.class);

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

LOGGER.debug("Parsed Date: {}", date);

LOGGER.info("END");

}

**4.Spring Core – Load Country from Spring Configuration XML**

1. **Country.java**

public class Country {

private String code, name;

public Country() { LOGGER.debug("Inside Country Constructor."); }

// Getters & Setters with logging

public String getCode() {

LOGGER.debug("Getting code");

return code;

}

public void setCode(String code) {

LOGGER.debug("Setting code");

this.code = code;

}

// toString() implementation

}

1. **country.xml**

xml

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

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

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

</bean>

1. **displayCountry() Method**

public void displayCountry() {

LOGGER.info("START");

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

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

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

LOGGER.info("END");

}

**5.Spring Core – Demonstration of Singleton Scope and Prototype Scope**

1. **Singleton (Default)**
   * Only **one instance** per Spring container.
   * Modified displayCountry():

Country country1 = ctx.getBean("country", Country.class);

Country country2 = ctx.getBean("country", Country.class);

// Both point to the same instance

1. **Prototype Scope**
   * **New instance** per getBean() call.
   * Update country.xml:

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

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

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

</bean>

**When to Use?**

* Singleton → Stateless beans (e.g., Services).
* Prototype → Stateful beans (e.g., User sessions).

**6.Spring Core – Load list of countries from Spring Configuration XML**

1. **Updated**country.xml

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

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

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

</bean>

<!-- Define other countries (us, de, jp) similarly -->

<bean id="countryList" class="java.util.ArrayList">

<constructor-arg>

<list>

<ref bean="in"/>

<ref bean="us"/>

<ref bean="de"/>

<ref bean="jp"/>

</list>

</constructor-arg>

</bean>

1. displayCountries()**Method**

public void displayCountries() {

LOGGER.info("START");

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

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

LOGGER.debug("Countries: {}", countries);

LOGGER.info("END");

}

**1. Hello World RESTful Web Service**

@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. REST - Country Web Service

<!-- country.xml -->

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

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

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

</bean>

JAVA

@RestController

public class CountryController {

@Autowired

private ApplicationContext context;

@RequestMapping("/country")

public Country getCountryIndia() {

return (Country) context.getBean("country");

}

**}**

**REST - Get All Countries**

<bean id="countryList" class="java.util.ArrayList">

<constructor-arg>

<list>

<ref bean="country"/>

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

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

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

</bean>

<!-- Add more countries -->

</list>

</constructor-arg>

</bean>

4. REST - Get Country by Code

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

public Country getCountry(@PathVariable String code) throws CountryNotFoundException {

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

return countries.stream()

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

.findFirst()

.orElseThrow(() -> new CountryNotFoundException());

}

5.REST - Country Not Found Exception Handling

@ResponseStatus(value = HttpStatus.NOT\_FOUND, reason = "Country not found")

public class CountryNotFoundException extends Exception

1. **Controller Class**

package com.cognizant.spring\_learn.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

@RestController

public class HelloController {

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

@GetMapping("/hello")

public String sayHello() {

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

String message = "Hello World!!";

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

return message;

}

}

**Create the Country Model**

package com.cognizant.spring\_learn.model;

public class Country {

private String code;

private String name;

// Getters and Setters

public String getCode() {

return code;

}

public void setCode(String code) {

this.code = code;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**Define XML Configuration**

<?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.spring\_learn.model.Country">

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

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

</bean>

</beans>

**Step 4: Create CountryController**

package com.cognizant.spring\_learn.controller;

import com.cognizant.spring\_learn.model.Country;

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

import org.springframework.context.ApplicationContext;

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

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

@RestController

public class CountryController {

@Autowired

ApplicationContext context;

@RequestMapping("/country")

public Country getCountryIndia() {

return (Country) context.getBean("country");

}

}

<http://localhost:8083/country>

**Display Employee List and Edit Employee form using RESTful Web Service**

**1. Static Employee List from XML**

**What I did:**

* I created a Spring XML file (employee.xml) inside src/main/resources.
* I added multiple Department and Employee beans to this XML.
* I reused existing skill beans instead of creating duplicates.
* Then I wrapped the employee beans inside an ArrayList bean named employeeList.

<bean id="employeeList" class="java.util.ArrayList">

<constructor-arg>

<list>

<ref bean="employee1"/>

<ref bean="employee2"/>

<ref bean="employee3"/>

<ref bean="employee4"/>

</list>

</constructor-arg>

</bean>

**Employee Implementation**

**What I did:**

* I created a class EmployeeDao.
* Inside it, I declared a static variable EMPLOYEE\_LIST.
* In the constructor, I loaded the Spring context manually using ClassPathXmlApplicationContext and set the EMPLOYEE\_LIST by reading employeeList bean.
* I added a method getAllEmployees() which just returns this list.

public class EmployeeDao {

public static List<Employee> EMPLOYEE\_LIST;

public EmployeeDao() {

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

EMPLOYEE\_LIST = (ArrayList<Employee>) context.getBean("employeeList");

}

public List<Employee> getAllEmployees() {

return EMPLOYEE\_LIST;

}

}

**EmployeeService Setup**

**What I did:**

* Annotated the class with @Service.
* Injected the DAO inside it.
* Created a method getAllEmployees() which calls the DAO and returns the list.
* Added @Transactional annotation to indicate a read operation.

@Service

public class EmployeeService {

@Autowired

private EmployeeDao employeeDao;

@Transactional

public List<Employee> getAllEmployees() {

return employeeDao.getAllEmployees();

}

}

**EmployeeController**

**What I did:**

* Created a controller class called EmployeeController.
* Added @RestController and a method with @GetMapping("/employees").
* In the method, I called employeeService.getAllEmployees() and returned the result.

@RestController

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping("/employees")

public List<Employee> getAllEmployees() {

return employeeService.getAllEmployees();

}

}

**5. Department REST Service**

**What I did:**

* Added a list of Department beans in XML (employee.xml).
* Created DepartmentDao with a static list called DEPARTMENT\_LIST.
* Inside constructor, populated DEPARTMENT\_LIST from XML.
* Service class: DepartmentService, annotated with @Service, method getAllDepartments().
* Controller: DepartmentController with @GetMapping("/departments").

@RestController

public class DepartmentController {

@Autowired

private DepartmentService departmentService;

@GetMapping("/departments")

public List<Department> getAllDepartments() {

return departmentService.getAllDepartments();

}

}

<http://localhost:8083/departments>

**JWT Authentication with Spring Security – Assignment Summary**

**Securing RESTful Services with Spring Security**

I started by integrating Spring Security into my existing Spring Boot project using the spring-boot-starter-security dependency. Once Maven resolved the libraries, Spring’s default behavior immediately protected all endpoints with Basic Authentication. I verified this using a simple curl command, and received a 401 Unauthorized error — which confirmed that the security configuration was working as expected.

Later, by supplying credentials through curl (-u user:generated-password), I was able to successfully access /countries, validating Basic Auth at a low level. To observe how passwords are encoded and transmitted, I inspected the Authorization header and decoded the Base64 value using an online tool — a critical learning moment that revealed Basic Auth’s vulnerability in real-world scenarios.

**Role-Based Access Control (RBAC)**

Next, I implemented role-specific access by configuring in-memory users within SecurityConfig. Two roles were created: USER and ADMIN, both with hardcoded passwords (pwd). I used BCryptPasswordEncoder() to encrypt the credentials, ensuring security best practices were followed.

I defined authorization rules using antMatchers:

* /countries endpoint was restricted to users with ROLE\_USER
* /authenticate allowed access to both roles for token generation

These rules were tested with different combinations of users and passwords using curl. Incorrect credentials returned a 401 Unauthorized, and users with the wrong role received a 403 Forbidden, exactly as expected. Logging also confirmed the internal behavior was consistent.

**Token Generation – Creating JWTs**

To overcome Basic Auth's limitations, I added JWT support using the jjwt library (io.jsonwebtoken). I created an AuthenticationController with a /authenticate endpoint that accepted Basic credentials via the Authorization header.

I extracted and decoded the header value using Java’s Base64 decoding, then isolated the username. This username was passed to a new generateJwt() method, which returned a signed JWT token using HMAC SHA-256 and a fixed secret key. The token included iat and exp claims, with a 20-minute expiry for simplicity.

The endpoint returned the token as a JSON response using a map:

{ "token": "<JWT string>" }

This design clearly separates token generation logic from the controller layer, aligning well with real-world architectural patterns.

**JWT Validation – Creating a Custom Filter**

The most important step was implementing token-based request validation via a custom filter. I extended BasicAuthenticationFilter in a new class JwtAuthorizationFilter.

This filter intercepted all incoming HTTP requests, checked if the Authorization header contained a Bearer token, and validated it using the same secret key. If the token was valid, it extracted the subject (user) and marked the request as authenticated by injecting a UsernamePasswordAuthenticationToken into Spring Security’s context.

The final integration step involved modifying SecurityConfig to register this filter and update request handling rules:

.addFilter(new JwtAuthorizationFilter(authenticationManager()))

.anyRequest().authenticated()

I tested the full flow using curl:

1. Generated a token via /authenticate
2. Passed the token to /countries using Authorization: Bearer <token>
3. Received a successful response — and confirmed that modifying the token resulted in a 401 Unauthorized, proving the validation was working.