**Unit-5**

**Spring Boot**

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



What is Spring Boot

In short, Spring Boot is the combination of Spring Framework and Embedded Servers.

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

We can use Spring STS IDE or Spring Initializer to develop Spring Boot Java applications.

**Why should we use Spring Boot Framework?**

The dependency injection approach is used in Spring Boot.

It contains powerful database transaction management capabilities.

It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.

It reduces the cost and development time of the application.

**Advantages of Spring Boot**

It creates stand-alone Spring applications that can be started using Java -jar.

It tests web applications easily with the help of different Embedded HTTP servers such as Tomcat, Jetty, etc. We don't need to deploy WAR files.

It provides opinionated 'starter' POMs to simplify our Maven configuration.

It provides production-ready features such as metrics, health checks, and externalized configuration.

There is no requirement for XML configuration.

It offers a CLI tool for developing and testing the Spring Boot application.

It offers the number of plug-ins.

It also minimizes writing multiple boilerplate codes (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.

It increases productivity and reduces development time.

**Limitations of Spring Boot**

Spring Boot can use dependencies that are not going to be used in the application. These dependencies increase the size of the application.

**Prerequisite of Spring Boot**

To create a Spring Boot application, following are the prerequisites. In this tutorial, we will use Spring Tool Suite (STS) IDE.

* Java 1.8
* Maven 3.0+
* Spring Framework 5.0.0. BUILD-SNAPSHOT
* An IDE (Spring Tool Suite) is recommended

**Spring Boot Architecture**

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts. It is developed on top of the core Spring Framework.

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

* Presentation Layer
* Business Layer
* Persistence Layer
* Database Layer



**Spring Boot Flow Architecture**



* Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except one thing: there is no need for DAO and DAOImpl classes in Spring boot.
* Creates a data access layer and performs CRUD operation.
* The client makes the HTTP requests (PUT or GET).
* The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
* In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
* A JSP page is returned to the user if no error occurred.

**Spring Boot Configuration:**

* Follow the following steps:
* Create a maven project with the help of spring initializer. <https://start.spring.io/>
* Add the dependencies
  + Spring Web
  + MYSQL
  + Hibernate JPA
* Generate the project and download the zip
* Import that project as existing maven project into spring tool suite
* Wait for some time for the installation of dependencies (will take some time)
* Remove it till then you are not using JPA otherwise it may show errors from porm.xml

<dependency>

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

<artifactId>spring-boot-starter-data- jpa</artifactId>

</dependency>

* Execute it as spring boot app
* Send request using postman or browser localost:8080
* It returns not found error because we have not defined controller
* Create a class for controller use the @RestContoller annotation with class
* Define a method with GetMapping(“/home”) annotation

**Spring Boot Annotations**

Spring Boot Annotations is a form of metadata that provides data about a program. In other words, annotations are used to provide supplemental information about a program. It is not a part of the application that we develop. It does not have a direct effect on the operation of the code they annotate. It does not change the action of the compiled program.

Core Spring Framework Annotations can also use here. Apart from those following are exclusively used with spring boot.

**@EnableAutoConfiguration:** It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.

**@SpringBootApplication:** It is a combination of three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration.

**@RequestMapping:** It is used to map the web requests. It has many optional elements like consumes, header, method, name, params, path, produces, and value. We use it with the class as well as the method.

Example

@Controller

public class BooksController

{

@RequestMapping("/computer-science/books")

public String getAllBooks(Model model)

{

//application code

return "bookList";

}

**@GetMapping:** It maps the HTTP GET requests on the specific handler method. It is used to create a web service endpoint that fetches It is used instead of using: @RequestMapping(method = RequestMethod.GET)

**@PostMapping:** It maps the HTTP POST requests on the specific handler method. It is used to create a web service endpoint that creates It is used instead of using: @RequestMapping(method = RequestMethod.POST)

**@PutMapping:** It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint that creates or updates It is used instead of using: @RequestMapping(method = RequestMethod.PUT)

**@DeleteMapping:** It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint that deletes a resource. It is used instead of using: @RequestMapping(method = RequestMethod.DELETE)

@PatchMapping: It maps the HTTP PATCH requests on the specific handler method. It is used instead of using: @RequestMapping(method = RequestMethod.PATCH)

**@RequestBody:** It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP MessageConverters to convert the body of the request. When we annotate a method parameter with @RequestBody, the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.

**@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.

**@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

**@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a method parameter. The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

**@RestController:** It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.

**@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

**Spring Boot Actuator**

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use HTTP and JMX endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the Spring Boot actuator.

**Spring Boot Actuator Features**

There are three main features of Spring Boot Actuator:

* Endpoints
* Metrics
* Audit

**Endpoint:** The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose HTTP, where the Id of the endpoint, along with the prefix of /actuator, is mapped to a URL.

For example, the /health endpoint provides the basic health information of an application. The actuator, by default, mapped it to /actuator/health.

**Metrics:** Spring Boot Actuator provides dimensional metrics by integrating with the micrometer. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for timers, gauges, counters, distribution summaries, and long task timers with a dimensional data model.

**Audit:** Spring Boot provides a flexible audit framework that publishes events to an AuditEventRepository. It automatically publishes the authentication events if spring-security is in execution.

**Spring Boot Runners**

Spring Boot provides two runner interfaces, which are ApplicationRunner and CommandLineRunner. Both of these runners are used to execute piece of code when a Spring Boot Application starts.

Both of these interfaces are Functional Interfaces, which means they have only one functional method. In order to execute specific piece of code when Spring Boot Application starts, we need to implement either of these functional interfaces and override the single method of run.

Application Runner is an interface used to execute the code after the Spring Boot application started. The example given below shows how to implement the Application Runner interface on the main class file.

package com.kiet.demo;

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication implements ApplicationRunner {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

@Override

public void run(ApplicationArguments arg0) throws Exception {

System.out.println("Hello World from Application Runner");

}

}

**Logger:**

Logging is an API that provides the ability to trace out the errors of the applications. When an application generates the logging call, the Logger records the event in the LogRecord. After that, it sends to the corresponding handlers or appenders.

**RestAPI Application:**

Follow the basic steps for creating the springboot application.

Now Create a class as MyController as follows.

MyController.java

package com.kiet.rest.kietrest.controller;

import java.util.List;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

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

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

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

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

import com.kiet.rest.kietrest.entities.Course;

import com.kiet.rest.kietrest.services.CourseService;

@RestController

public class MyController {

@Autowired

private CourseService courseService;

@GetMapping("/home")

public String home() {

return "Welcom to my first APP";

}

//get the courses

@GetMapping("/courses")

public List<Course> getCourses(){

return this.courseService.getCourses();

}

@GetMapping("/courses/{courseId}")

public Course getCourse(@PathVariable String courseId) {

return this.courseService.getCourse(Integer.parseInt(courseId));

}

@PostMapping("/courses")

public Course addCourse(@RequestBody Course course ) {

return this.courseService.addCourse(course);

}

//Update Apis

@PutMapping("/courses")

public Course updateCourse(@RequestBody Course course) {

return this.courseService.updateCourse(course);

}

//delete apis

@DeleteMapping("/courses/{courseId}")

public ResponseEntity<HttpStatus> deleteCourse(@PathVariable String courseId){

System.out.println("Delete course Invoked");

try {

this.courseService.deleteCourse(Integer.parseInt(courseId));

return new ResponseEntity<>(HttpStatus.OK);

}

catch(Exception e) {

return new ResponseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

}

**Create a bean class with getter and setters**

package com.kiet.rest.kietrest.entities;

public class Course {

private int id;

private String tiltle;

private String description;

public Course(int id, String tiltle, String description) {

super();

this.id = id;

this.tiltle = tiltle;

this.description = description;

}

public Course() {

super();

// TODO Auto-generated constructor stub

}

public long getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getTiltle() {

return tiltle;

}

public void setTiltle(String tiltle) {

this.tiltle = tiltle;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

@Override

public String toString() {

return "Course [id=" + id + ", tiltle=" + tiltle + ", description=" + description + "]";

}

}

**Now create the business logic into service**

Create an interface

package com.kiet.rest.kietrest.services;

import java.util.List;

import com.kiet.rest.kietrest.entities.Course;

public interface CourseService {

public List<Course> getCourses();

public Course getCourse(int courseId);

public Course addCourse(Course course);

public Course updateCourse(Course course);

public void deleteCourse(int id);

}

**Implement it into a service class**

package com.kiet.rest.kietrest.services;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Service;

import com.kiet.rest.kietrest.entities.Course;

@Service

public class CourseServiceImpl implements CourseService {

List<Course> list;

public CourseServiceImpl() {

// TODO Auto-generated constructor stub

list = new ArrayList<>();

list.add(new Course(123, "JAVA", "Best Course of Spring Boot"));

list.add(new Course(124, "Node", "Best Course of SNode JS"));

}

@Override

public List<Course> getCourses() {

// TODO Auto-generated method stub

return list;

}

@Override

public Course getCourse(int courseId) {

Course c = null;

for(Course course:list) {

if(course.getId()==courseId) {

c = course;

break;

}

}

return c;

}

@Override

public Course addCourse(Course course) {

list.add(course);

return course;

}

@Override

public Course updateCourse(Course course) {

for(Course e:list) {

if(e.getId()==course.getId()) {

e.setTiltle(course.getTiltle());

e.setDescription(course.getDescription());

}

}

return course;

}

@Override

public void deleteCourse(int id) {

// TODO Auto-generated method stub

Course c = getCourse(id);

list.remove(c);

}

}

Now execute the springboot app and test the api using postman tool.