# Module 5 : Introduction to Spring Boot and RESTful Web Service

-Professor Anamika Dhawan

# Spring Boot

Spring Boot is an open source Java-based framework used to create a micro Service.

It is used to build stand-alone and production ready spring applications.

# What is micro service?

Micro Service is an architecture that allows the developers to develop and deploy services independently.

Each service running has its own process and this achieves the lightweight model to support business applications.

Micro services offers the following advantages to its developers −

- Easy deployment
- Simple scalability
- Compatible with Containers
- Minimum configuration
- Lesser production time

# What is spring boot?

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Advantages

Spring Boot offers the following advantages to its developers −

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

# Spring boot goals-

Spring Boot is designed with the following goals −

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

# How does it works?

Spring Boot automatically configures your application based on the dependencies you have added to the project by using @EnableAutoConfiguration annotation. For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains @SpringBootApplication annotation and the main method.

Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

# Spring Boot Starters

Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developers convenience.

For example, if you want to use Spring and JPA for database access, it is sufficient if you include spring-boot-starter-data-jpa dependency in your project.

Note that all Spring Boot starters follow the same naming pattern spring-boot-starter- *, where * indicates that it is a type of the application.

Ex:
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

# Auto Configuration

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfigu

@EnableAutoConfiguration
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

Spring Boot Auto Configuration automatically configures your Spring application based on the JAR dependencies you added in the project. For example, if MySQL database is on your class path, but you have not configured any database connection, then Spring Boot auto configures an in-memory database.

For this purpose, you need to add @EnableAutoConfiguration annotation or @SpringBootApplication annotation to your main class file. Then, your Spring Boot application will be automatically configured.

# Spring Boot Application

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootAppl

@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

The entry point of the Spring Boot Application is the class contains @SpringBootApplication annotation. This class should have the main method to run the Spring Boot application. @SpringBootApplication annotation includes Auto-Configuration, Component Scan, and Spring Boot Configuration.

If you added @SpringBootApplication annotation to the class, you do not need to add the @EnableAutoConfiguration, @ComponentScan and @SpringBootConfiguration annotation. The @SpringBootApplication annotation includes all other annotations.

# Component Scan

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the @ComponentScan annotation for your class file to scan your components added in your project.

```
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

# 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 FW annotations

**@Required:** It applies to the **bean** setter method. It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

**@Autowired:** Spring provides annotation-based auto-wiring by providing @Autowired annotation. It is used to autowire spring bean on setter methods, instance variable, and constructor. When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**@Configuration:** It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**@ComponentScan:** It is used when we want to scan a package for beans. It is used with the annotation @Configuration. We can also specify the base packages to scan for

# Spring fw stereotype annotation

**@Component:** It is a class-level annotation. It is used to mark a Java class as a bean. A Java class annotated with **@Component** is found during the classpath. The Spring Framework pick it up and configure it in the application context as a **Spring Bean**.

**@Controller:** The @Controller is a class-level annotation. It is a specialization of **@Component**. It marks a class as a web request handler. It is often used to serve web pages. By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation.

**@Service:** It is also used at class level. It tells the Spring that class contains the **business logic**.

**@Repository:** It is a class-level annotation. The repository is a **DAOs** (Data Access Object) that access the database directly. The repository does all the operations related

# Spring boot annotation

**@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**.

**@RestController:** The @RestController annotation from Spring Boot is basically a quick shortcut that saves us from always having to define @ResponseBody.

# Spring MVC and REST Annotation

**@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.

**@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)**

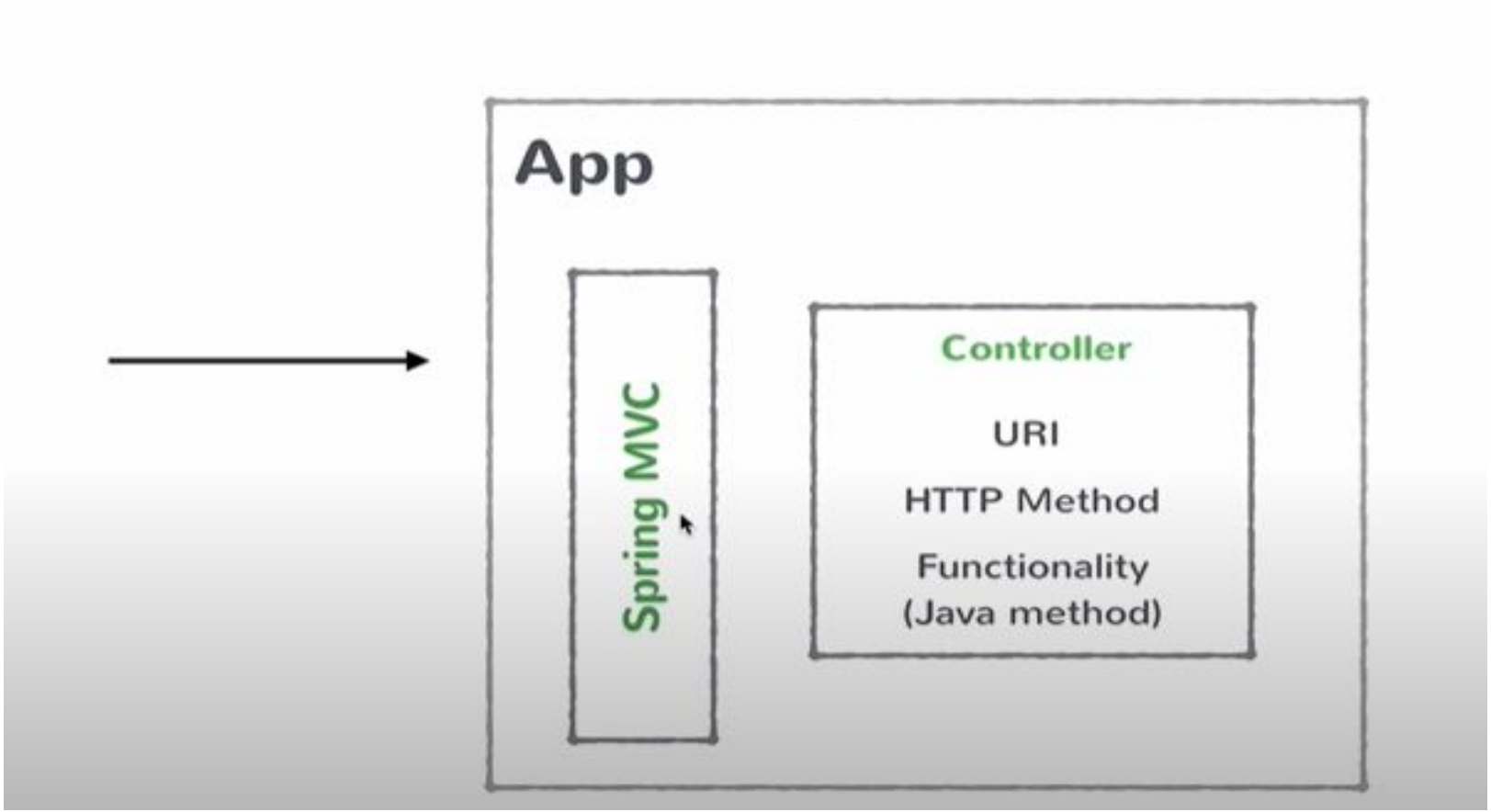
**@PostMapping:** It maps the **HTTP POST** requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**

**@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)**

# SpringApplication

`SpringApplication.run(Classname.class, args)` bootstraps a spring application as a stand-alone application from the main method.

It creates an appropriate ApplicationContext instance and load beans.


App
Spring MVC
Controller
URI
HTTP Method
Functionality
(Java method)

# Spring boot and database

**Spring boot connect to postgresql db**

**Connect to posgresql:**

Spring boot with jdbc template

Spring boot with spring data JPA & Hibernate

**Software Programs:**

JDK

Spring tool suite IDE

Postgresql

# Spring boot and database

- Configuration Steps:
  - Declare dependencies:
    - Spring Boot JDBC / Spring Data JPA
    - PostgreSQL JDBC Driver
  - Specify data source properties

# Spring Boot restful Web services

Spring Boot provides a very good support to building RESTful Web Services for enterprise applications.

Rest Controller

The @RestController annotation is used to define the RESTful web services. It serves JSON, XML and custom response. Its syntax is shown below −

```
@RestController

public class ProductServiceController {

}
```

# Spring Boot restful Web services

Request Mapping

The @RequestMapping annotation is used to define the Request URI to access the REST Endpoints. We can define Request method to consume and produce object. The default request method is GET.

```
@RequestMapping(value = "/products")

public ResponseEntity<Object> getProducts() { }
```

# Spring Boot restful Web services

Request Body

The @RequestBody annotation is used to define the request body content type.

```
public ResponseEntity<Object> createProduct(@RequestBody Product product) {
}
```

Path Variable

The @PathVariable annotation is used to define the custom or dynamic request URI. The Path variable in request URI is defined as curly braces {} as shown below −

```
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {
 }
```

# Get API

The default HTTP request method is GET.
This method does not require any Request Body.
You can send request parameters and path variables to define the custom or dynamic URL.

```
@GetMapping("/students")
public List<Student> getAllStudents() {
 return studentRepository.findAll();
}
```

In the above example getAllStudents method is annotated with @GetMapping annotation. This means that getAllStudents method will be called when an HTTP Get request is received at the /students URL

# Post API

The HTTP POST request is used to create a resource.
This method contains the Request Body.
We can send request parameters and path variables to define the custom or dynamic URL

```
@PostMapping("/students/{id}")
public void updateStudent(@PathVariable("id") String id, @RequestBody Student student) {
    // Update the student record here.
}
```

In the above example updateStudent method is annotated with @PostMapping annotation. This means that updateStudent method will be called when an HTTP Post request is received at the /students/{id} URL. Inside this updateStudent method we are also passing a parameter for Request Body to be send while making a post request. In the request body we have to pass the id for which we have to update the student details and the object for student which will contain the updated values of the student.

# Put API

The HTTP PUT request is used to update the existing resource
. This method contains a Request Body.
 We can send request parameters and path variables to define the custom or dynamic URL.

```
@PutMapping("/users/{id}")
public void updateUser(@PathVariable("id") String id, @RequestBody User user) {
	// Update the user details here
}
```

In the above example, the updateUser method is annotated with @PutMapping annotation. This means that the updateUser method will be called when the HTTP Put request is received at the /users/{id} URL. Inside this updateUser method, we are also passing a parameter for Request Body to be sent while making a PUT request. In the request body, we have to pass the id for which we have to update the user details and the object for the student which will contain the updated values of the student.

# Delete API

The HTTP Delete request is used to delete the existing resource.
This method does not contain any Request Body.
We can send request parameters and path variables to define the custom or dynamic URL.

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable("id") String id) {
        // Delete the user in this method with the id.
}
```

In the above example, the deleteUser method is annotated with @DeleteMapping annotation. This means that the deleteUser method will be called when HTTP Delete Request is received at the /users/{id} URL. Inside the deleteUser method, we are passing a parameter for Request Param to be sent while making a Delete request. In the request param, we are passing the id for the user which we have to delete while making a delete request.