Bill of Materials (BOM)

1. Spring vs Spring Boot
2. Spring Boot Features
3. Dependency Injection & Inversion of Control
4. @SpringBootAnnotation
5. @RestController, @Service, @Repository, @Configuration, @Component, @Entity, @Column, @Table, @Id, @Valid, @Data, @Bean, @RequestBody, GET, POST, PUT, DELETE
6. RESTful Web Services
7. REST TEMPLATE
8. Dispatcher Servlet
9. Aspect Oriented Programming
10. Logging
11. Spring Security (Authentication and Authorization)
12. Profiling
13. JPA
14. Maven lifecycle, Bean lifecycle
15. Spring Boot Architecture

1. Spring vs Spring Boot

Spring: Spring Framework is the most popular application development framework of Java. The main feature of the Spring Framework is dependency Injection or Inversion of Control (IoC). With the help of Spring Framework, we can develop a loosely coupled application. It is better to use if application type or characteristics are purely defined.

We have to configure application using XML file, add dependency manually with compatible version, define configuration used in class path.

Spring Boot: Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services. It is an annotation based approach, with autoconfiguration and version compatibility provided by Spring Boot. It also provide various application metrices using actuator, documentation using swagger (openAPI for upgraded version), embedded server such as Tomcat and Jetty, embedded and in-memory db such as h2. Spring Boot comes with the concept of starter in pom.xml file that internally takes care of downloading the dependencies JARs based on Spring Boot Requirement.

2. Spring Boot Features

(i) Auto configuration of class
(ii) Dependencies and version management of required JARs in Application
(iii)Annotation based development approach
(iv) Embedded server such as Tomcat and Jetty
(v) Actuator suppot
(vi) Admin console
(vii)Spring Securiy using Authentication and Authorization at Http request endpoints

3. Dependency Injection & Inversion of Control

Inversion of control is a design pattern by which it controls the creation of objects, configures and assembles their dependencies, manage their entire life cycle. It is managed by IOC by getting the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans.

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. The advantages of this architecture are:

(a) decoupling the execution of a task from its implementation
(b) making it easier to switch between different implementations
(c) greater modularity of a program
(d) greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

Dependency Injection is an implementation of IOC by which it emphasizes keeping the java classes independent of each other and the container frees them from object creation and maintainance. There are three ways by which DI can be achieved in Spring Framework, i.e. By Object Reference, By Constructor, By Setter method.

It also ensures loose coupling between java classes.

NOTE: By Object Based DI : This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection.

Application Context : In the Spring framework, the interface ApplicationContext represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles. The Spring framework provides several implementations of the ApplicationContext interface: ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications, and WebApplicationContext for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Bean : We use the @Bean annotation on a method to define a bean. If we don't specify a custom name, then the bean name will default to the method name.
For a bean with the default singleton scope, Spring first checks if a cached instance of the bean already exists, and only creates a new one if it doesn't. If we're using the prototype scope, the container returns a new bean instance for each method call.

4. @SpringBootAnnotation
@SpringBootAnnotation is the driver class level annotation in a spring boot project which is used on class which contains main() method. It is a combination of three annotation as:
    (a) : @Configuration - It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.
    (b) : @ComponentScan - It is used when we want to scan a package for beans. We specify Bean in these class.
    (c) : @EnableAutoConfiguration - It auto-configures the bean that is present in the classpath and configures it to run the methods.
@ComponentScan is used with the annotation @Configuration. We can also specify the base packages to scan for Spring Components.
Example

```
@ComponentScan(basePackages = "com.javatpoint")
@Configuration
public class ScanComponent
{
// ...
}
```

5. @RestController, @Service, @Repository, @Configuration, @Component, @Entity, @Column, @Table, @Id, @Valid, @Data, @Bean, @RequestBody, GET, POST, PUT, DELETE
@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.

@Service : It is also used at class level. It tells the Spring that class is communicating with DB via JPARepo or another API via restTemplate.

@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 to the database. By default, Hibernate ORM is used in spring for this.

@Configuration : It is used when we want to scan a package for beans. We specify Bean in these class.

@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. Is is a superset of @Configuration annotation.

@Entity : It is used to annotate Class or Model which is a table in DB.

@Table : It is a Class level annotation used to name a table in db as @Table(name ="table_name"). If we don't use this annotation over a class, by default, class name will be the name of table in DB.

@Column : It is a field level annotation used to name a column in db as @Column(name="column_name"). If we don't use this annotation over a field, by default, field name will be the name of column in DB.

@Id : It is a field level annotation used for primary key attribute.

@Valid : It is used alongside method parameter in controller starting with @Valid to validate field of a class. Some of annotation for these validation are: @Max, @Min,

@NotNull, @Past, @Future, @Size. It is available inside "spring-boot-starter-validation" atrifact or maven dependency.

@Data : It is a class level annotation used for POJO. It is available in projectlambok maven dependency. It contains @getter, @setter, @toString, @EqualsAndHashcode, @AllArgsConstructor, @NoArgsConstructor all together with only one annotation, i.e. @Data

@Bean : It is a method-level annotation. It is an alternative of XML <bean> tag. It tells the method to produce a bean to be managed by Spring Container. The class contaning this bean method must be annotated with either @Component annotation or it's subset such as @Configuration

@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

@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.

6. RESTful Web Services

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See The @Path Annotation and URI Path Templates for more information.

Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.

Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information.

7. REST TEMPLATE

Rest Template is used to create applications that consume RESTful Web Services.

To interact with REST, the client needs to create a client instance and request object, execute the request, interpret the response, map the response to domain objects, and also handle the exceptions. It is common for the Spring framework to both create an API and consume internal or external application’s APIs. This advantage also helps us in the development of microservices. To avoid such boilerplate code Spring provides a convenient way to consume REST APIs – through ‘RestTemplate’.

8. Dispatcher Servlet

DispatcherServlet handles an incoming HttpRequest, delegates the request, and processes that request according to the configured HandlerAdapter interfaces that have been implemented within the Spring application along with accompanying annotations specifying handlers, controller endpoints, and response objects.

9. Aspect Oriented Programming

The application is generally developed with multiple layers. A typical Java application has the following layers:

Web Layer: It exposes the services using the REST or web application.
Business Layer: It implements the business logic of an application.
Data Layer: It implements the persistence logic of the application.
The responsibility of each layer is different, but there are a few common aspects that apply to all layers are Logging, Security, validation, caching, etc. These common aspects are called cross-cutting concerns.

If we implement these concerns in each layer separately, the code becomes more difficult to maintain. To overcome this problem, Aspect-Oriented Programming (AOP) provides a solution to implement cross-cutting concerns.

Implement the cross-cutting concern as an aspect.
Define pointcuts to indicate where the aspect has to be applied.
It ensures that the cross-cutting concerns are defined in one cohesive code component.

AOP (Aspect-Oriented Programming) is a programming pattern that increases modularity by allowing the separation of the cross-cutting concern. These cross-cutting concerns are different from the main business logic. We can add additional behavior to existing code without modification of the code itself.

Spring's AOP framework helps us to implement these cross-cutting concerns.

Using AOP, we define common functionality in one place. We are free to define how and where this functionality is applied without modifying the class to which we are applying the new feature. The cross-cutting concern can now be modularized into special classes, called aspect.

There are two benefits of aspects:

First, the logic for each concern is now in one place instead of scattered all over the codebase.

Second, the business modules only contain code for their primary concern. The secondary concern has been moved to the aspect.

The aspects have the responsibility that is to be implemented, called advice. We can implement an aspect's functionality into a program at one or more join points.

Benefits of AOP

It is implemented in pure Java.

There is no requirement for a special compilation process.

It supports only method execution Join points.

Only run time weaving is available.

Two types of AOP proxy is available: JDK dynamic proxy and CGLIB proxy.

Cross-cutting concern

The cross-cutting concern is a concern that we want to implement in multiple places in an application. It affects the entire application.

AOP Terminology

Aspect: An aspect is a module that encapsulates advice and pointcuts and provides cross-cutting An application can have any number of aspects. We can implement an aspect using regular class annotated with @Aspect annotation.

Pointcut: A pointcut is an expression that selects one or more join points where advice is executed. We can define pointcuts using expressions or patterns. It uses different kinds of expressions that matched with the join points. In Spring Framework, AspectJ pointcut expression language is used.

Join point: A join point is a point in the application where we apply an AOP aspect. Or it is a specific execution instance of an advice. In AOP, join point can be a method execution, exception handling, changing object variable value, etc.

Advice: The advice is an action that we take either before or after the method execution. The action is a piece of code that invokes during the program execution. There are five types of advices in the Spring AOP framework: before, after, after-returning, after-throwing, and around advice. Advices are taken for a particular join point. We will discuss these advices further in this section.

Target object: An object on which advices are applied, is called the target object. Target objects are always a proxied It means a subclass is created at run time in which the target method is overridden, and advices are included based on their configuration.

Weaving: It is a process of linking aspects with other application types. We can perform weaving at run time, load time, and compile time.

Proxy: It is an object that is created after applying advice to a target object is called proxy. The Spring AOP implements the JDK dynamic proxy to create the proxy classes with target classes and advice invocations. These are called AOP proxy classes.

Types of AOP Advices

There are five types of AOP advices are as follows:

Before Advice

After Advice

Around Advice

After Throwing

After Returning

Before Advice: An advice that executes before a join point, is called before advice.It is an advice type which ensures that an advice runs before the method execution. We use @Before annotation to mark an advice as Before advice.

After Advice: An advice that executes after a join point, is called after advice. It is an advice type which ensures that an advice runs after the method execution. We use @After annotation to mark an advice as After advice.

Around Advice: An advice that executes before and after of a join point, is called around advice. It is the most powerful advice. It also provides more control for end-user to get deal with ProceedingJoinPoint.

After Throwing Advice: An advice that executes when a join point throws an exception. It ensures that an advice runs if a method throws an exception. We use @AfterThrowing annotation to implement the after throwing advice. The name (ex) that we define in the throwing attribute must correspond to the name of a parameter in the advice method. Otherwise, advice will not run.

```
@AfterThrowing(PointCut="execution(expression) ", throwing="name")

@AfterThrowing(value="execution(* com.javatpoint.service.impl.AccountServiceImpl.*(..))",throwing="ex")
public void afterThrowingAdvice(JoinPoint joinPoint, Exception ex)
{
System.out.println("After Throwing exception in method:"+joinPoint.getSignature());
System.out.println("Exception is:"+ex.getMessage());
}
```

After Returning Advice: An advice that executes when a method executes successfully. After returning is an advice in Spring AOP that invokes after the execution of join point complete (execute) normally. It does not invoke if an exception is thrown. We can implement after returning advice in an application by using @AfterReturning annotation. The annotation marks a function as an advice to be executed before the method covered by PointCut.

After returning advice runs when a matched method execution returns a value normally. The name that we define in the return attribute must correspond to the name of a parameter in the advice method. When a method returns a value, the value will be passed to the advice method as the corresponding argument value.

```
@AfterReturning(value="execution(* com.javatpoint.service.impl.AccountServiceImpl.*(..))",returning="account")
public void afterReturningAdvice(JoinPoint joinPoint, Account account)
{
System.out.println("After Returing method:"+joinPoint.getSignature());
System.out.println(account);
}
```

Before implementing the AOP in an application, we are required to add Spring AOP dependency in the pom.xml file.

Spring Boot Starter AOP is a dependency that provides Spring AOP and AspectJ. Where AOP provides basic AOP capabilities while the AspectJ provides a complete AOP framework.

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-aop</artifactId>
<version>2.2.2.RELEASE</version>
</dependency>
```

AOP in action inside a spring boot application.

(a) Add AOP dependency in pom.xml file.

(b) @EnableAspectJAutoProxy(proxyTargetClass=true) : It is a class level annotation used in class which contains main method.

It enables support for handling components marked with AspectJ’s @Aspect annotation. It is used with @Configuration annotation. We can control the type of proxy by using the proxyTargetClass attribute. Its default value is false.

(c) Add @Aspect along with @Component annotation in AspectClass where aop logic is to be written.

(d) Annotate with proper aop annotation over method, i.e.

```
@Aspect
@Component
public class EmployeeServiceAspect
{
@Before(value = "execution(* com.javatpoint.service.EmployeeService.*(..)) and args(empId, fname, sname)")      or
@After(value = "execution(* com.javatpoint.service.EmployeeService.*(..)) and args(empId, fname, sname)")    ->afterAdvice(...)
public void beforeAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
System.out.println("Before method:" + joinPoint.getSignature());
System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
}
}
```

10. Logging

Spring Boot uses Apache Commons logging for all internal logging. Spring Boot’s default configurations provides a support for the use of Java Util Logging, Log4j2, and Logback. Using these, we can configure the console logging as well as file logging.

If you are using Spring Boot Starters, Logback will provide a good support for logging. Besides, Logback also provides a use of good support for Common Logging, Util Logging, Log4J, and SLF4J.

In Spring, the log level configurations can be set in the application.properties file which is processed during runtime. Spring supports 5 default log levels, ERROR, WARN, INFO, DEBUG, and TRACE,(All in ascending order) with INFO being the default log level configuration.

```
logging.level.root=WARN
logging.level.com.mohan=TRACE
```

We can use file to keep logs by using this in application.properties file: logging.path = /var/tmp/

You can configure the ROOT level log in Logback.xml file using the code given below –

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
   <root level = "INFO">
   </root>
</configuration>
```

You can define the Log pattern in logback.xml file using the code given below in the classpath. You can also define the set of supported log patterns inside the console or file log appender using the code given below –

```
<pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}] [%C] [%t] [%L] [%-5p] %m%n</pattern>
```

NOTE: In production, using debug level is best (user error wherever required to increase application performance). Using TRACE level in prod env is not recommended at all.

11. Spring Security (Resource: https://docs.spring.io/spring-security/reference/servlet/architecture.html)

Spring Security is a framework that provides authentication, authorization, and protection against common attacks. With first class support for securing both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.

In order to add security to our Spring Boot application, we need to add the security starter dependency:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

This will also include the SecurityAutoConfiguration class containing the initial/default security configuration.

12. Profiling

The development process of an application has different stages; the typical ones are development, testing, and production. Spring Boot profiles group parts of the application configuration and make it be available only in certain environments.

A profile is a set of configuration settings. Spring Boot allows to define profile specific property files in the form of application-{profile}.properties. It automatically loads the properties in an application.properties file for all profiles, and the ones in profile-specific property files only for the specified profile. The keys in the profile-specific property override the ones in the master property file.

Note: Spring Boot properties are loaded in a particular order. If several profiles are specified, the last-wins strategy applies.

The @Profile annotation indicates that a component is eligible for registration when the specified profile or profiles are active. The default profile is called default; all the beans that do not have a profile set belong to this profile.

There are plenty of ways of defining active profiles in Spring Boot, including command line arguments, Maven/Gradle settings, JVM system parameters, environment variables, spring.profiles.active property, and SpringApplication methods.

Note: Some approaches set and replace active profiles, while other add active profiles on top of existing active profiles.

In integration tests, profiles are activated with @ActiveProfiles.

What is the difference between @profile and @ActiveProfiles?

@Profile declares which profile the bean or configuration belongs to. @ActiveProfiles comes into picture in case of an ApplicationContext and defines which profiles should be active if respective ApplicationContext is being used for test classes. When @ActiveProfiles is specified, it causes the Spring Context to check whether a bean or configuration is annotated with @Profile .

Note for application.properties file - If the property is not found while running the application, Spring Boot throws the Illegal Argument exception as Could not resolve placeholder 'spring.application.name' in value "${spring.application.name}".

To resolve the placeholder issue, we can set the default value for the property using thr syntax given below -

```
@Value("${property_key_name:default_value}")
@Value("${spring.application.name:demoservice}")
```

application.yaml : We can keep the Spring active profile properties in the single application.yml file. No need to use the separate file like application.properties.

The following is an example code to keep the Spring active profiles in application.yml file. Note that the delimiter (---) is used to separate each profile in application.yml file.

```
spring:
   application:
      name: demoservice
server:
   port: 8080

---
spring:
   profiles: dev
   application:
      name: demoservice
server:
   port: 9090

---
spring:
   profiles: prod
   application:
      name: demoservice
server:
   port: 4431
```

To run the application : java -jar demo1.0.0-SNAPSHOT.jar --spring.profiles.active=prod

13. JPA

Spring Boot JPA is a Java specification for managing relational data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows Object-Relation Mapping (ORM).There are some popular JPA implementations frameworks such as Hibernate, EclipseLink, DataNucleus, etc. It is also known as Object-Relation Mapping (ORM) tool. By default, spring uses Hibernate as as ORM tool.

In ORM, the mapping of Java objects to database tables, and vice-versa is called Object-Relational Mapping. The ORM mapping works as a bridge between a relational database (tables and records) and Java application (classes and objects). The ORM layer exists between the application and the database. It converts the Java classes and objects so that they can be stored and managed in a relational database. By default, the name that persists become the name of the table, and fields become columns. Once an application sets-up, each table row corresponds to an object.

JPA: JPA is a Java specification that is used to access, manage, and persist data between Java object and relational database. It is a standard approach for ORM.

Hibernate: It is a lightweight, open-source ORM tool that is used to store Java objects in the relational database system. It is a provider of JPA. It follows a common approach provided by JPA.

Spring Boot provides starter dependency spring-boot-starter-data-jpa to connect Spring Boot application with relational database efficiently. The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.

NOTE : We are using JPA to provide better maintainance of application. If an application uses MySQL db now, switched to MongoDB or other DB, then we need not to update our query as per DB, instead it will be done automatically by JPA by adding the respective dependency of DB in pom.xml file. This is the reason we should avoid (or minimize) using custom query in JPA, rather than use standardized method provided by JPA for your queries fetching.

14. Maven lifecycle, Bean lifecycle

Maven is a powerful project management tool that is based on POM (project object model), used for project build, dependency, and documentation. It is a tool that can be used for building and managing any Java-based project. Maven makes the day-to-day work of Java developers easier and helps with the building and running of any

Java-based project.

Maven Lifecycle: Maven lifecycle has 8 steps: Validate, Compile, Test, Package, Integration test, Verify, Install, and Deploy.

(1) Validate: This step validates if the project structure is correct. For example – It checks if all the dependencies have been downloaded and are available in the local repository.

(2) Compile: It compiles the source code, converts the .java files to .class, and stores the classes in the target/classes folder.

(3) Test: It runs unit tests for the project.

(4) Package: This step packages the compiled code in a distributable format like JAR or WAR.

(5) Integration test: It runs the integration tests for the project.

(6) Verify: This step runs checks to verify that the project is valid and meets the quality standards.

(7) Install: This step installs the packaged code to the local Maven repository.

(8) Deploy: It copies the packaged code to the remote repository for sharing it with other developers.

Maven follows a sequential order to execute the commands where if you run step n, all steps preceding it (Step 1 to n-1) are also executed. For example – if we run the Installation step (Step 7), it will validate, compile, package and verify the project along with running unit and integration tests (Step 1 to 6) before installing the built package to the local repository.

Maven Commands:

mvn clean: Cleans the project and removes all files generated by the previous build.

mvn compile: Compiles source code of the project.

mvn test-compile: Compiles the test source code.

mvn test: Runs tests for the project.

mvn package: Creates JAR or WAR file for the project to convert it into a distributable format.

mvn install: Deploys the packaged JAR/ WAR file to the local repository.

mvn site: generate the project documentation.

mvn validate: validate the project's POM and configuration.

mvn idea:idea: generate project files for IntelliJ IDEA or Eclipse.

mvn release:perform: Performs a release build.

mvn deploy: Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests and building the project.

mvn archetype:generate: This command is used to generate a new project from an archetype, which is a template for a project. This command is typically used to create new projects based on a specific pattern or structure.

mvn dependency:tree: This command is used to display the dependencies of the project in a tree format. This command is typically used to understand the dependencies of the project and troubleshoot any issues.

Bean Lifecycle:

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed. In this article, we will discuss the life cycle of the bean.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method and the destroy() method of spring.

Container Started-> Bean Instantiated-> Dependencies Injected-> Custom inti() method-> Custom utility method-> Custom destroy() method

15. Spring Boot Architecture

The Spring Boot is built on top of the core Spring framework. It is a simplified and automated version of the spring framework. The spring boot follows a layered architecture in which each layer communicates to other layers. The spring boot consists of the following four layers:

Presentation Layer – Authentication & Json Translation

Business Layer – Business Logic, Validation & Authorization

Persistence Layer – Storage Logic

Database Layer – Actual Database

1.Presentation Layer : The presentation layer is the top layer of the spring boot architecture. It consists of Views. i.e., the front-end part of the application. It handles the HTTP requests and performs authentication. It is responsible for converting the JSON field's parameter to Java Objects and vice-versa. Once it performs the authentication of the request it passes it to the next layer. i.e., the business layer.

2.Business Layer : The business layer contains all the business logic. It consists of services classes. It is responsible for validation and authorization.

3.Persistence Layer : The persistence layer contains all the database storage logic. It is responsible for converting business objects to the database row and vice-versa.

4.Database Layer : The database layer contains all the databases such as MySql, MongoDB, etc. This layer can contain multiple databases. It is responsible for performing the CRUD operations.

I   The flow of spring boot application are as below:

II The Client makes an HTTP request(GET, PUT, POST, etc.)

III The HTTP request is forwarded to the Controller. The controller maps the request. It processes the handles and calls the server logic.

IV The business logic is performed in the Service layer. The spring boot performs all the logic over the data of the database which is mapped to the spring boot model class through Java Persistence Library(JPA).

V   The JSP page is returned as Response from the controller.

****

ControllerAdvice : The @ControllerAdvice is an annotation, to handle the exceptions globally in Spring Boot Application. The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client. So we need to annotate class with @ControllerAdvice and extends the RuntimeException class, then we need to annotate @ExceptionHandler to the method where we are handling exception.

Define a class which extends RuntimeException class as below

```
public class ProductNotfoundException extends RuntimeException {
   private static final long serialVersionUID = 1L;
}
```

write the logic to handle exception if occours as below

```
@ControllerAdvice
public class ProductExceptionController {
   @ExceptionHandler(value = ProductNotfoundException.class)
   public ResponseEntity<Object> exception(ProductNotfoundException exception) {
      return new ResponseEntity<>("Product not found", HttpStatus.NOT_FOUND);
   }
}
```

And the controller class is using it to handle exception as below

```
@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
   public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
   @RequestBody Product product) {
      if(!productRepo.containsKey(id))throw new ProductNotfoundException();
      productRepo.remove(id);
      product.setId(id);
      productRepo.put(id, product);
      return new ResponseEntity<>("Product is updated successfully", HttpStatus.OK);
   }
```

****

Interceptor : The Interceptor in Spring Boot is used to perform operations under the following situations -

(A) Before sending the request to the controller

(B) Before sending the response to the client

For example, you can use an interceptor to add the request header before sending the request to the controller and add the response header before sending the response to the client.

To work with interceptor, you need to create @Component class that supports it and it should implement the HandlerInterceptor interface.

The following are the three methods we should know about while working on Interceptors -

(A) preHandle() method - This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.

(B) postHandle() method - This is used to perform operations before sending the response to the client.

(C) afterCompletion() method - This is used to perform operations after completing the request and response.

Overview to implement :

```
@Component
public class ProductServiceInterceptor implements HandlerInterceptor {
   @Override
   public boolean preHandle(
      HttpServletRequest request, HttpServletResponse response, Object handler)
      throws Exception {
      return true;
   }
   @Override
   public void postHandle(
      HttpServletRequest request, HttpServletResponse response, Object handler,
      ModelAndView modelAndView) throws Exception {}
   @Override
   public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
      Object handler, Exception exception) throws Exception {}
}
```

We need to register this Interceptor with InterceptorRegistry by using WebMvcConfigurerAdapter as shown below -

```
@Component
public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter {
   @Autowired
   ProductServiceInterceptor productServiceInterceptor;
   @Override
   public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(productServiceInterceptor);
   }
}
```

****

Servlet Filter : A filter is an object used to intercept the HTTP requests and responses of your application. By using filter, we can perform two operations at two instances -

(A) Before sending the request to the controller

(B) Before sending the response to the client

The following code shows the sample code for a Servlet Filter implementation class with @Component annotation.

```
@Component
public class SimpleFilter implements Filter {
   @Override
   public void destroy() {}
   @Override
   public void doFilter
      (ServletRequest request, ServletResponse response, FilterChain filterchain)
      throws IOException, ServletException {}

   @Override
   public void init(FilterConfig filterconfig) throws ServletException {}
}
```

The following example shows the code for reading the remote host and remote address from the ServletRequest object before sending the request to the controller.In doFilter() method, we have added the System.out.println statements to print the remote host and remote address.

```
@Component
public class SimpleFilter implements Filter {
   @Override
   public void destroy() {
   @Override
   public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterchain)
      throws IOException, ServletException {
      System.out.println("Remote Host:"+request.getRemoteHost());
      System.out.println("Remote Address:"+request.getRemoteAddr());
      filterchain.doFilter(request, response);
   }
   @Override
   public void init(FilterConfig filterconfig) throws ServletException {}
}
@SpringBootApplication
@RestController
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
   @RequestMapping(value = "/")
   public String hello() {
      return "Hello World";
   }
}
```

****

Note - If the server.port number is 0 while starting the Spring Boot application, Tomcat uses the random port number.

****

CORS : Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

For example, your web application is running on 8080 port and by using JavaScript you are trying to consuming RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.

Two requirements are needed to handle this issue –

(A) RESTful web services should support the Cross-Origin Resource Sharing.

(B) RESTful web service application should allow accessing the API(s) from the 8080 port.

>Enable CORS in Controller Method

We need to set the origins for RESTful web service by using @CrossOrigin annotation for the controller method. This @CrossOrigin annotation supports specific REST API, and not for the entire application.

```
@RequestMapping(value = "/products")
@CrossOrigin(origins = "http://localhost:8080")
public ResponseEntity<Object> getProduct() {
   return null;
}
```

>Global CORS Configuration

We need to define the shown @Bean configuration to set the CORS configuration support globally to your Spring Boot application.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
   return new WebMvcConfigurerAdapter() {
      @Override
      public void addCorsMappings(CorsRegistry registry) {
         registry.addMapping("/products").allowedOrigins("http://localhost:9000");
      }
   };
}
```

****

Schduling : Scheduling is a process of executing the tasks for the specific time period. Java Cron expressions are used to configure the instances of CronTrigger, a subclass of org.quartz.Trigger.The @EnableScheduling annotation is used to enable the scheduler for your application. This annotation should be added into the main Spring Boot application class file.

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

The @Scheduled annotation is used to trigger the scheduler for a specific time period.

```
@Scheduled(cron = "0 * 9 * * ?")
public void cronJobSch() throws Exception {
}
```

The following is a sample code that shows how to execute the task every minute starting at 9:00 AM and ending at 9:59 AM, every day

```
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@Component
public class Scheduler {
   @Scheduled(cron = "0 * 9 * * ?")
   public void cronJobSch() {
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
      Date now = new Date();
      String strDate = sdf.format(now);
      System.out.println("Java cron job expression:: " + strDate);
   }
 }
```

Fixed Rate scheduler is used to execute the tasks at the specific time. It does not wait for the completion of previous task. The values should be in milliseconds. The sample code is shown here –

```
@Scheduled(fixedRate = 1000)
public void fixedRateSch() {
}
```

A sample code for executing a task on every second from the application startup is shown here –

```
@Component
public class Scheduler {
   @Scheduled(fixedRate = 1000)
   public void fixedRateSch() {
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

      Date now = new Date();
      String strDate = sdf.format(now);
      System.out.println("Fixed Rate scheduler:: " + strDate);
   }
}
```

Fixed Delay scheduler is used to execute the tasks at a specific time. It should wait for the previous task completion. The values should be in milliseconds. A sample code is shown here –

```
@Scheduled(fixedDelay = 1000, initialDelay = 1000)
public void fixedDelaySch() {
}
```

Here, the initialDelay is the time after which the task will be executed the first time after the initial delay value.

An example to execute the task for every second after 3 seconds from the application startup has been completed is shown below –

```
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@Component
public class Scheduler {
   @Scheduled(fixedDelay = 1000, initialDelay = 3000)
   public void fixedDelaySch() {
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
      Date now = new Date();
      String strDate = sdf.format(now);
      System.out.println("Fixed Delay scheduler:: " + strDate);
   }
}
```

****