UNIT-4: SPRING BOOT FRAMEWORK-PART-1

Spring Boot Framework-Part-1: Basic concepts: Spring, Spring Boot, Testing Basics, Testing in Spring Boot. A Basic Spring Boot Application: Setting up the Development Environment, The Skeleton Web App, Spring Boot Auto configuration, Three-Tier, Three-Layer Architecture, Modeling our Domain, Business Logic, Presentation Layer.

What is 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. 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 Initializr** to develop Spring Boot Java applications.

Why should we use Spring Boot Framework?

We should use Spring Boot Framework because:

- 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.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

- **Spring Data:** It simplifies data access from the relational and **NoSQL** databases.
- **Spring Batch:** It provides powerful **batch** processing.

- **Spring Security:** It is a security framework that provides robust **security** to applications.
- **Spring Social:** It supports integration with **social networking** like LinkedIn.
- **Spring Integration:** It is an implementation of Enterprise Integration Patterns. It facilitates integration with other **enterprise applications** using lightweight messaging and declarative adapters.

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.

Goals of Spring Boot

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

- Provides Opinionated Development approach
- Avoids defining more Annotation Configuration
- Avoids writing lots of import statements
- Avoids XML Configuration.

Spring Boot Features

- **Web Development** - We can easily create a self-contained HTTP application that uses embedded servers like **Tomcat**. We can use the **spring-boot-starter-web** module to start and run the application quickly.
- **Spring Application** - The SpringApplication is a class that provides a convenient way to bootstrap a Spring application. It can be started from the main method. We can call the application just by calling a static run() method.
- **Application events and listeners** - Spring Boot uses events to handle the variety of tasks. It allows us to create factories file that is used to add listeners. We can refer it to using the **ApplicationListener key**.
- **Admin features** - Spring Boot provides the facility to enable admin-related features for the application. It is used to access and manage applications remotely. We can enable it in the Spring Boot application by using **spring.application.admin.enabled** property.
- **Externalized Configuration** - Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments. The application uses YAML files to externalize configuration.
- **Properties Files** - Spring Boot provides a rich set of **Application Properties**. So, we can use that in the properties file of our project.
- **YAML Support** - It provides a convenient way of specifying the hierarchical configuration. It is a superset of JSON. The SpringApplication class automatically supports YAML.
- **Type-safe Configuration** - The strong type-safe configuration is provided to govern and validate the configuration of the application.
- **Logging** - Spring Boot uses Common logging for all internal logging. Logging dependencies are managed by default.
- **Security** - Spring Boot applications are spring bases web applications. So, it is secure by default with basic authentication on all HTTP endpoints. A rich set of Endpoints is available to develop a secure Spring Boot application.

COMPARISON BETWEEN SPRING AND 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.
- **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.

The primary comparison between Spring and Spring Boot are discussed below:

Spring	Spring Boot
Spring Framework is a widely used Java EE framework for building applications.	**Spring Boot Framework** is widely used to develop **REST APIs**.
It aims to simplify Java EE development that makes developers more productive.	It aims to shorten the code length and provide the easiest way to develop **Web Applications**.
The primary feature of the Spring Framework is **dependency injection**.	The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement.
It helps to make things simpler by allowing us to develop **loosely coupled** applications.	It helps to create a **stand-alone** application with less configuration.
The developer writes a lot of code (**boilerplate code**) to do the minimal task.	It **reduces** boilerplate code.
To test the Spring project, we need to set up the sever explicitly.	Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc.
It does not provide support for an in-memory database.	It offers several plugins for working with an embedded and **in-memory** database such as **H2**.
Developers manually define dependencies for the Spring project in **pom.xml**.	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.

COMPARISON BETWEEN SPRING BOOT AND SPRING MVC

Spring Boot	Spring MVC
Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults.	**Spring MVC** is a model view controller-based web framework under the Spring framework.
It provides default configurations to build **Spring-powered** framework.	It provides **ready to use** features for building a web application.
There is no need to build configuration	It requires build configuration manually.

manually.	
There is **no requirement** for a deployment descriptor.	A Deployment descriptor is **required**.
It avoids boilerplate code and wraps dependencies together in a single unit.	It specifies each dependency separately.
It **reduces** development time and increases productivity.	It takes **more** time to achieve the same.

TESTING BASICS AND TESTING IN SPRING BOOT:

Testing is a crucial part of developing Spring Boot applications to ensure code quality a nd functionality. Here are some basics to get you started:

1. Unit Testing

Unit tests are used to test individual units of code, such as methods or classes, in isolatio n. Spring Boot provides support for unit testing with JUnit and Mockito.

2. Integration Testing

As the name suggests, integration tests focus on integrating different layers of the application. That also means no mocking is involved. Basically, we write integration tests for testing a feature that may involve interaction with multiple components. Integration tests verify that different components of the application work together as expected. Spring Boot provides support for integration testing with @SpringBootTest and @DataJpaTest.

- **Mocking Dependencies**
  Mocking dependencies is essential for unit testing. Spring Boot provides @MockBean to cre ate mock objects.
- **Test Configuration**-Spring Boot provides @TestConfiguration to define test-specific configurations.
- **Test Utilities** -
  Spring Boot provides utilities like MockMvc for testing web applications.

Examples:

- Integration testing of complete Employee Management Feature (EmployeeRepository, EmployeeService, EmployeeController).
- Integration testing of complete User Management Feature (UserController, UserService, and UserRepository).
- Integration testing of complete Login Feature (LoginRespository, LoginController, Login Service), etc

Development Steps

1. Create Spring Boot Application

2. Configure MySQL database
3. Create JPA Entity
4. Create Spring Data JPA Repository
5. Create Spring Boot REST Controller
6. Create Integration Tests with MySQL database
7. Run Integration Test by Adding Testcontainers to Spring Boot Project and Write Integration Tests using Testcontainers

1. Create Spring Boot Application

Using **spring initialize**, create a Spring Boot project and add the following dependencies:

- Spring Web
- Spring Data JPA
- Lombok
- MySQL Driver

Generate the Spring boot project as a zip file, extract it, and import it into IntelliJ IDEA.

2. Configure MySQL database

Use MySQL database to store and retrieve the data in this example and use Hibernate properties to create and drop tables.

Open the application.properties file and add the following configuration to it:

```
spring.datasource.url=jdbc:mysql://localhost:3306/demo?useSSL=false

spring.datasource.username=root

spring.datasource.password=Mysql@123

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

spring.jpa.hibernate.ddl-auto = create-drop
```

3. Create JPA Entity

Next, let's create a *Student* JPA entity:

```
package net.javaguides.spirngboot.entity;

import lombok.*;

import javax.persistence.*;

@Setter

@Getter

@Builder

@AllArgsConstructor
```

```
@NoArgsConstructor
@Entity
@Table(name = "students")
public class Student
{
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private long id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;
   private String email;
}
```

4. Create Spring Data JPA Repository

Let's create StudentRepository which extends the JpaRepository interface:

```
package net.javaguides.spirngboot.repository;
import net.javaguides.spirngboot.entity.Student;
import org.springframework.data.jpa.repository.JpaRepository;
public interface StudentRepository extends JpaRepository<Student, Long>
{
}
```

5. Create Spring Boot REST Controller

Let's create *StudentController* class and add these couple of REST endpoints:

```
package net.javaguides.spirngboot.controller;
import net.javaguides.spirngboot.entity.Student;
import net.javaguides.spirngboot.repository.StudentRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
```

```
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/api/students")
public class StudentController
{
   @Autowired
   private StudentRepository studentRepository;

   @PostMapping
   @ResponseStatus(HttpStatus.CREATED)
   public Student createStudent(@RequestBody Student student)
   {
      return studentRepository.save(student);
   }

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

6. Create Integration Tests with MySQL database

Now, let's create an Integration JUnit test for **GET ALL Students** REST API:

```
package net.javaguides.spirngboot;
import net.javaguides.spirngboot.entity.Student;
import net.javaguides.spirngboot.repository.StudentRepository;
import org.hamcrest.CoreMatchers;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
```

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.ResultActions;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;
import org.springframework.test.web.servlet.result.MockMvcResultMatchers;
import java.util.List;

@SpringBootTest
@AutoConfigureMockMvc
class SpringbootTestcontainersDemoApplicationTests
{
	@Autowired
	private StudentRepository studentRepository;

	@Autowired
	private MockMvc mockMvc;

	// given/when/then format - BDD style
	@Test
	public void givenStudents_whenGetAllStudents_thenListOfStudents() throws Exception
	{
		// given - setup or precondition
		List<Student> students = List.of(Student.builder().firstName("Ramesh").lastName("faadatare").email("ramesh@gmail.com").build(),
		Student.builder().firstName("tony").lastName("stark").email("tony@gmail.com").build());
		studentRepository.saveAll(students);

		// when - action
		ResultActions response = mockMvc.perform(MockMvcRequestBuilders.get("/api/students"));
```

```
        // then - verify the output
        response.andExpect(MockMvcResultMatchers.status().isOk());
        response.andExpect(MockMvcResultMatchers.jsonPath("$.size()",
CoreMatchers.is(students.size())));
    }
}
```

- *@SpringBootTest* annotation to load the application context and test various components.
- *MockMvc* provides support for Spring MVC testing. It encapsulates all web application beans and makes them available for testing.
- *@AutoConfigureMockMvc* annotation that can be applied to a test class to enable and configure auto-configuration of MockMvc. @Autowired
- private MockMvc mockMvc;
- The *MockMvc.perform()* method will call a GET request method, which returns the *ResultActions*.
- ResultActions response = mockMvc.perform(MockMvcRequestBuilders.get("/api/students"));
- Using this result, we can have assertion expectations about the response, like its content, HTTP status, or header.
- The *andExpect()* will expect the provided argument.

7. Run Integration Test by Adding Testcontainers to Spring Boot Project and Write Integration Tests using Testcontainers

Open the pom.xml file and add the following Testcontainers dependencies:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.16.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.16.2</version>
```

```
        <scope>test</scope>
</dependency>
<dependency>
        <groupId>org.testcontainers</groupId>
        <artifactId>mysql</artifactId>
        <version>1.16.2</version>
        <scope>test</scope>
</dependency>
```

BASIC SPRING BOOT APPLICATION: SETTING UP THE DEVELOPMENT ENVIRONMENT

Step 1: Install Java Development Kit (JDK)

1. **Download JDK**: Go to the Oracle JDK download page or use an open-source alternative like AdoptOpenJDK.
2. **Install JDK**: Follow the installation instructions for your operating system. Ensure that the JAVA_HOME environment variable is set correctly.

Step 2: Install an Integrated Development Environment (IDE)

1. **Choose an IDE**: Popular choices include IntelliJ IDEA, Eclipse, and Visual Studio Code.
2. **Install the IDE**: Download and install your preferred IDE.

Step 3: Install Maven

1. **Download Maven**: Go to the Maven download page and download the latest version.
2. **Install Maven**: Follow the installation instructions for your operating system. Ensure that the MAVEN_HOME environment variable is set correctly and that mvn is added to your system's PATH.

Step 4: Create a New Spring Boot Project

1. **Spring Initializr**: Go to Spring Initializr.
2. **Project Settings**:
    - **Project**: Maven Project
    - **Language**: Java
    - **Spring Boot**: 2.7.0 or later
    - **Group**: com.example
    - **Artifact**: demo
    - **Dependencies**: Spring Web

3. **Generate Project**: Click “Generate” to download the project as a ZIP file. Extract it to your desired location.

Step 5: Import the Project into Your IDE

1. **Open IDE**: Launch your IDE.
2. **Import Project**: Import the project as a Maven project. In IntelliJ IDEA, you can do this by selecting “Open” and navigating to the project’s pom.xml file.

Step 6: Verify the Setup

1. **Run the Application**: Locate the main application class (DemoApplication.java) and run it. This will start the embedded Tomcat server.
2. **Check the Console**: Ensure there are no errors in the console output. You should see a message indicating that Tomcat has started on port 8080.

This setup will get you ready to start developing Spring Boot applications.

Example:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

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

@SpringBootApplication

public class DemoApplication

{

   public static void main(String[] args)

   {

      SpringApplication.run(DemoApplication.class, args);

   }

}

@RestController

class HelloController

{

   @GetMapping("/hello")

   public String hello()

   {

      return "Hello, World!";
```

```
    }
}
```

Step 7: Test the Application

1. **Browser**: Open your web browser and navigate to http://localhost:8080/.
2. **Result**: You should see Hello, World! displayed.

THE SKELETON WEB APP, AND SPRING BOOT AUTO CONFIGURATION IN SPRINGBOOT:

Creating a skeleton web application in Spring Boot is a great way to get started with web development. Here's a step-by-step guide:

Step 1: Set Up Your Project

1. **Spring Initializr**: Go to Spring Initializr and generate a new project with the following settings:
    - **Project**: Maven Project
    - **Language**: Java
    - **Spring Boot**: 2.7.0 or later
    - **Dependencies**: Spring Web
2. **Download the Project**: Click "Generate" to download the project as a ZIP file. Extract it to your desired location.

Step 2: Import the Project into Your IDE

1. **IDE**: Open your favorite IDE (e.g., IntelliJ IDEA, Eclipse, or VS Code).
2. **Import**: Import the project as a Maven project.

Step 3: Create a Simple REST Controller

1. **Main Application Class**: Open the DemoApplication.java file (or the main class created by Spring Initializr) and ensure it looks like this:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication

{

   public static void main(String[] args)

   {
```

```
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

2. **REST Controller**: Create a new Java class named HelloController.java in the com.example.demo package:

```
package com.example.demo;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloController
{
    @GetMapping("/hello")
    public String hello()
    {
        return "Hello, World!";
    }
}
```

Step 4: Run the Application

1. **Run**: Run the DemoApplication class. Spring Boot application will start, and you should see logs indicating that Tomcat has started on port 8080.

Step 5: Test the Application

1. **Browser**: Open your web browser and navigate to http://localhost:8080/hello.
2. **Result**: You should see the message “Hello, World!”.

SPRING BOOT AUTO-CONFIGURATION

Spring Boot’s auto-configuration feature simplifies the setup of your application by automatically configuring Spring components based on the dependencies present on the classpath.

Enable Auto-Configuration: Auto-configuration is enabled by default when you use the @SpringBootApplication annotation, which is a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan.

```
@SpringBootApplication
public class DemoApplication
```

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

2. **How It Works**: Spring Boot scans the classpath for JAR files and automatically configures Spring beans based on the libraries you have included. For example, if you have spring-boot-starter-data-jpa in your dependencies, Spring Boot will configure a DataSource and an EntityManagerFactory automatically[1].
3. **Custom Auto-Configuration**: You can create your own auto-configuration classes by using the @Configuration annotation along with @Conditional annotations to control when the configuration should apply[2].

```
@Configuration
@ConditionalOnClass(DataSource.class)
public class MySQLAutoConfiguration
{
   @Bean
   @ConditionalOnMissingBean
   public DataSource dataSource()
   {
      return new HikariDataSource();
   }
}
```

4. **Excluding Auto-Configuration**: If you want to exclude specific auto-configuration classes, you can use the exclude attribute of the @SpringBootApplication annotation.

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

}

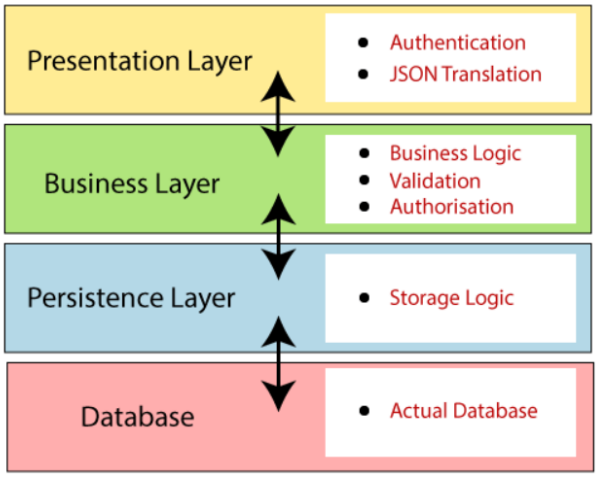
5. **Debugging Auto-Configuration**: You can start your application with the --debug switch to see which auto-configuration classes are being applied and why.

THREE-TIER, THREE-LAYER ARCHITECTURE IN SPRING BOOT

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. There are **four** layers in Spring Boot are as follows:

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



1. Presentation Layer

- **Controller classes** as the presentation layer. Keep this layer as thin as possible and limited to the mechanics of the MVC operations, e.g., receiving and validating the inputs, manipulating the model object, returning the appropriate **ModelAndView** object, and so on. All the business-related operations should be done in the service classes. Controller classes are usually put in a **controller** package.

```
package com.example.demo.controller;

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

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

@RestController

public class HelloController
```

```
{
   @GetMapping("/hello")
   public String hello()
   {
      return "Hello, World!";
   }
}
```

2. Business Logic Layer

- **Service classes** as the business logic layer. Calculations, data transformations, data processes, and cross-record validations (business rules) are usually done at this layer. They get called by the controller classes and might call repositories or other services. Service classes are usually put in a service package.

```
package com.example.demo.service;
import org.springframework.stereotype.Service;
@Service
public class HelloService
{
   public String getGreeting()
   {
      return "Hello, World!";
   }
}
```

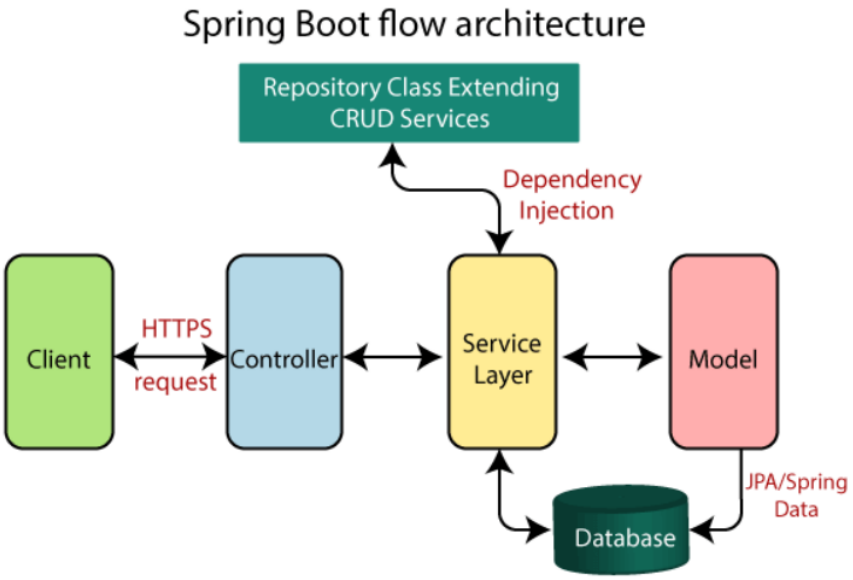
3. Data Access Layer

- **Repository classes** as data access layer. This layer's responsibility is limited to Create, Retrieve, Update, and Delete (CRUD) operations on a data source, which is usually a relational or non-relational database. Repository classes are usually put in a repository package.

```
package com.example.demo.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.model.User;
public interface UserRepository extends JpaRepository<User, Long>
{
}
```

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

Interaction between layers in a Spring Boot application:



1. **Controller**: Receives a request and delegates it to the service layer.
2. **Service**: Processes the request, applying business logic, and interacts with the repository layer if needed.
3. **Repository**: Performs CRUD operations on the database.

Example:

```
package com.example.demo.controller;

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

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

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

import com.example.demo.service.HelloService;

@RestController

public class HelloController

{

   @Autowired

   private HelloService helloService;


   @GetMapping("/hello")
```

```
    public String hello()
   {
       return helloService.getGreeting();
    }
}
```

Benefits of Three-Tier Architecture

1. **Separation of Concerns**: Each layer has a distinct responsibility, making the codebase easier to manage and understand.
2. **Scalability**: Each layer can be scaled independently.
3. **Maintainability**: Changes in one layer have minimal impact on others, making the application easier to maintain.

SPRING INITIALIZR

Spring Initializr is a web-based tool provided by the Pivotal Web Service. With the help of Spring Initializr, we can easily generate the structure of the Spring Boot Project. It offers extensible API for creating JVM-based projects.

It also provides various options for the project that are expressed in a metadata model. The metadata model allows us to configure the list of dependencies supported by JVM and platform versions, etc. It serves its metadata in a well-known that provides necessary assistance to third-party clients.

Spring Initializr Modules

Spring Initializr has the following module:

- initializr-actuator: It provides additional information and statistics on project generation. It is an optional module.
- initializr-bom: In this module, BOM stands for Bill Of Materials. In Spring Boot, BOM is a special kind of POM that is used to control the versions of a project's dependencies. It provides a central place to define and update those versions. It provides flexibility to add a dependency in our module without worrying about the versions. Outside the software world, the BOM is a list of parts, items, assemblies, and other materials required to create products. It explains what, how, and where to collect required materials.
- initializr-docs: It provides documentation.
- initializr-generator: It is a core project generation library.
- initializr-generator-spring:
- initializr-generator-test: It provides a test infrastructure for project generation.

- initializr-metadata: It provides metadata infrastructure for various aspects of the projects.
- initializr-service-example: It provides custom instances.
- initializr-version-resolver: It is an optional module to extract version numbers from an arbitrary POM.
- initializr-web: It provides web endpoints for third party clients.

Generating a Project

Before creating a project, we must be friendly with UI. Spring Initializr UI has the following labels:

- Project: It defines the kind of project. We can create either Maven Project or Gradle Project. We will create a Maven Project throughout the tutorial.
- Language: Spring Initializr provides the choice among three languages Java, Kotlin, and Groovy. Java is by default selected.
- Spring Boot: We can select the Spring Boot version. The latest version is 2.2.2.
- Project Metadata: It contains information related to the project, such as Group, Artifact, etc. Group denotes the package name; Artifact denotes the Application name. The default Group name is com.example, and the default Artifact name is demo.
- Dependencies: Dependencies are the collection of artifacts that we can add to our project.

There is another Options section that contains the following fields:

- Name: It is the same as Artifact.
- Description: In the description field, we can write a description of the project.
- Package Name: It is also similar to the Group name.
- Packaging: We can select the packing of the project. We can choose either Jar or War.
- Java: We can select the JVM version which we want to use. We will use Java 8 version throughout the tutorial.

There is a Generate button. When we click on the button, it starts packing the project and downloads the Jar or War file, which you have selected.

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

- @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 Components.
- @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.

Spring Framework Stereotype Annotations

- @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 to the database.

Spring Boot Annotations

- @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.

Spring MVC and REST Annotations

- @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)
- @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 Dependency Management

Spring Boot manages dependencies and configuration automatically. Each release of Spring Boot provides a list of dependencies that it supports. The list of dependencies is available as a part of the Bills of Materials (spring-boot-dependencies) that can be used with Maven. So, we need not to specify the version of the dependencies in our configuration. Spring Boot manages itself. Spring Boot upgrades all dependencies automatically in a consistent way when we update the Spring Boot version.

Advantages of Dependency Management

- It provides the centralization of dependency information by specifying the Spring Boot version in one place. It helps when we switch from one version to another.
- It avoids mismatch of different versions of Spring Boot libraries.
- We only need to write a library name with specifying the version. It is helpful in multi-module projects.

EXAMPLE WITH SPRING BOOT:

1. Building a Simple Web Application

This guide walks you through creating a simple web application with Spring Boot:

- **Spring Initializr**: Use Spring Initializr to generate a new project with Spring Web dependency.
- **Main Application Class**:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

   public static void main(String[] args) {

      SpringApplication.run(DemoApplication.class, args);

   }

}
```

- **REST Controller**:

Java

```
package com.example.demo;

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

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

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

- **Run the Application**: Run the DemoApplication class and navigate to http://localhost:8080/hello to see the message “Hello, World!”.

2. Spring Boot with JPA and H2 Database

This example demonstrates how to use Spring Boot with JPA and an H2 in-memory database:

- **Dependencies**: Add spring-boot-starter-data-jpa and h2 to your pom.xml.

XML

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

- **Entity Class**:

Java

```
package com.example.demo.model;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class User
{
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    // Getters and Setters
}
```

- **Repository Interface**:

Java

```
package com.example.demo.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.model.User;
public interface UserRepository extends JpaRepository<User, Long>
{
}
```

- **Service Class**:

Java

```
package com.example.demo.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.demo.repository.UserRepository;
import com.example.demo.model.User;
@Service
public class UserService
{
    @Autowired
    private UserRepository userRepository;

    public User saveUser(User user)
   {
        return userRepository.save(user);
```

```
    }
}
```

- **Controller Class**:

Java

```
package com.example.demo.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.model.User;
import com.example.demo.service.UserService;
@RestController
public class UserController
{
    @Autowired
    private UserService userService;

    @PostMapping("/users")
    public User createUser(@RequestBody User user)
   {
        return userService.saveUser(user);
    }
}
```

- **Run the Application**: Run the DemoApplication class and use a tool like Postman to send a POST request to http://localhost:8080/users with a JSON body to create a new user.

3. Spring Boot with Thymeleaf

This example shows how to create a simple web page using Thymeleaf:

- **Dependencies**: Add spring-boot-starter-thymeleaf to your pom.xml.

XML

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

```
</dependency>
```

- **Controller Class**:

Java

```
package com.example.demo.controller;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class WebController
{
   @GetMapping("/greeting")
   public String greeting(Model model)
  {
      model.addAttribute("message", "Hello, Thymeleaf!");
      return "greeting";
   }
}
```

- **Thymeleaf Template**: Create a file named greeting.html in src/main/resources/templates/.

HTML

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Greeting</title>
</head>
<body>
   <h1 th:text="${message}">Greeting</h1>
</body>
</html>
```

- **Run the Application**: Run the DemoApplication class and navigate to http://localhost:8080/greeting to see the greeting message.

1. Which of the following is a core feature of the Spring Framework?
A) Dependency Injection (DI) B) Java Persistence API (JPA)
C) Server-side scripting D) Client-side scripting

2. What does the @SpringBootApplication annotation combine?
A) @Configuration, @ComponentScan, and @EnableAutoConfiguration
B) @RestController, @Service, and @Repository
C) @Entity, @Table, and @Id
D) @RequestMapping, @GetMapping, and @PostMapping

3. Which testing framework is commonly used with Spring Boot for unit testing?
A) JUnit B) Selenium C) JMeter D) TestNG

4. The @MockBean annotation in Spring Boot testing is used for what purpose?
A) To load properties from an external file
B) To create a mock instance of a Spring bean
C) To run SQL scripts before tests
D) To configure a web client for REST API testing

5. Which component in Spring Boot provides production-ready features like monitoring and health checks?
A) Spring Data B) Spring MVC C) Spring Boot CLI **D) Spring Boot Actuator**

6. To enable mocking in Spring Boot test which annotation will be used?
A) @Mock **B) @MockBean** C) @Mockito D) @RunWith

7. What is the role of @RestController in a Spring Boot application?
A) To handle database connections **B) To create RESTful web services**
C) To configure security settings D) To manage transactions

8. Which type of testing focuses on the interaction between multiple components?
A) Unit testing **B) Integration testing**
C) Performance testing D) Load testing

9. What is the purpose of the @SpringBootTest annotation in Spring Boot?
A) To disable auto-configuration B) To configure data sources
C) To provide a comprehensive Spring application context for testing
D) To manage application properties

10. What is the use of Spring Boot's auto-configuration?
A. To automatically configure Spring applications B. To manage application security
C. To handle user authentication D. To create database schemas

11. Which tool is commonly used to create a new Spring Boot project with necessary dependencies?
A) Eclipse B) IntelliJ IDEA **C) Spring Initializr** D) NetBeans

12. What is the primary purpose of application.properties in a Spring Boot project?
A) To define database schemas **B) To configure application settings**

C) To write business logic D) To create REST endpoints

13. Which of the following is NOT a correct step in setting up the Spring Boot development environment?
A) Install JDK B) Install an IDE **C) Install a database server** D) Use Maven or Gradle

14. What is the default embedded server used by Spring Boot for running web applications?
A) Tomcat B) Jetty C) Undertow D) WebLogic

15. In a basic Spring Boot application, where do you typically define the main class?
A) src/main/resources B) src/main/webapp **C) src/main/java** D) src/test/java

16. What does the @SpringBootApplication annotation in the main class signify?
A) It enables Spring Boot auto-configuration and component scanning
B) It configures the database
C) It initializes the Spring Boot Actuator
D) It sets up a security context

17. Which of the following is necessary to run a Spring Boot application from the command line?
A) java -cp . MyApplication **B) java -jar myapp.jar**
C) javac MyApplication.java D) mvn compile

18. What is the role of the @RestController annotation in a Spring Boot web application?
A) To manage database transactions **B) To create RESTful web services**
C) To configure application properties D) To handle application security

19. What is the purpose of the Spring MVC module?
a) provides a web framework for building web applications
b) provides a data access framework for accessing databases
c) provides a caching framework for caching data
d) provides a security framework for securing web applications

20. What is the primary purpose of Spring Boot's auto-configuration feature?
A) To manually configure application components
B) To automatically configure Spring applications based on the dependencies on the classpath
C) To disable default configurations
D) To manage security settings

21. Which of the following annotations is used to exclude specific auto-configuration classes?
A) @SpringBootApplication B) @Configuration C) @EnableAutoConfiguration
D) @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})

22. What does the spring.factories file in Spring Boot do?
A) Configures the application context
B) Lists all the configuration classes to be loaded by auto-configuration
C) Manages security settings D) Defines custom beans

23. Which annotation is used to enable Spring Boot's auto-configuration feature?

A) @EnableAutoConfiguration B) @ComponentScan
C) @Configuration D) @RestController

24. How can you customize auto-configuration behavior in a Spring Boot application?
A) By modifying the application.yml or application.properties file
B) By changing the project's build.gradle file
C) By using the @Service annotation
D) By creating custom beans only

25. What will happen if you exclude a specific auto-configuration class in Spring Boot?
A) The application will throw an error
B) The specific auto-configuration will not be applied
C) All auto-configurations will be disabled D) Only security configurations will be affected

26. Which class does Spring Boot use to guess the configuration properties?
A) ApplicationProperties B) EnvironmentProperties
C) AutoConfiguration **D) SpringApplication**

27. What is the advantage of using Spring Boot's auto-configuration?
A) Increased manual configuration
B) Reduced development time and effort with sensible defaults
C) Decreased application performance
D) Complicated configuration process

28. In the Three Layer Architecture, mention the role of Controller layer?
A) handle user input and display output
B) encapsulate business logic
C) interact with the database
D) act as an intermediary between the Presentation and Business layers

29. Which of the following best describes a Three-Tier Architecture?
A) User Interface, Business Logic, Data Storage B) HTML, CSS, JavaScript
C) Presentation, Application, Network D) Model, View, Controller

30. In a Three-Layer Architecture, what is the primary responsibility of the Presentation Layer?
A) To interact with the database B) To implement business logic
C) To handle user interactions and display data D) To manage network communications

31. What is an example of a component typically found in the Business Logic layer?
A) HTML files **B) Service classes** C) Database tables D) User interface controls

32. Which annotation is commonly used to define a REST controller in the Presentation Layer in Spring Boot?
A) @Entity B) @Service C) @Repository **D) @RestController**

33. When modeling our domain in a Spring Boot application, what is typically used to represent an entity in the database?
A) @Controller B) @Service **C) @Entity** D) @Component

34. In a Three-Tier Architecture, where would you typically find the DAO (Data Access Object) classes?
A) Presentation Tier B) Business Logic Tier
C) Data Access Tier D) Network Tier

35. What is the role of the @Service annotation in Spring Boot?
A) To define a repository interface
B) To mark a class as a Spring-managed component that holds business logic
C) To configure application properties
D) To create a RESTful endpoint

36. Which layer is responsible for validating user inputs and sending appropriate responses in a Three-Tier Architecture?
A) Data Access Layer B) Business Logic Layer
C) Presentation Layer D) Middleware Layer

37. What is Spring Initializr?

a) A web-based tool to quickly generate Spring project template with predefined configurations and dependencies.

b) A command-line utility for generating a basic Spring Boot project structure.

c) A tool for initializing a Spring-based application with a specific set of dependencies.

d) A tool for creating a Spring Boot application with selected features.

38. What feature of Spring Boot simplifies the configuration process?

a) Auto-configuration (Automatic configuration of beans and dependencies).

b) Annotations to configure beans and dependencies.

c) XML-based configuration support.

d) A command-line interface for configuring beans.

39. Which annotation is used to create RESTful web services in Spring Boot?

a) *@RestController*

b) *@Controller*

c) *@Component*

d) *@Service*

40. Which Spring annotation is used to handle HTTP POST requests?

a) *@GetMapping*

b) *@PutMapping*

c) *@CreateMapping*

d) *@PostMapping*

41. Which annotation is used to mark a class as a service component in Spring?

a) *@Component*

b) *@Service*

c) *@Controller*

d) *@Repository*

42. Which class serves as the default implementation of the *JpaRepository* interface?

a) *SimpleJpaRepository*

b) *JpaRepositoryImpl*

c) *CustomJpaRepository*

d) *DefaultJpaRepository*

43. What is the main difference between Spring and Spring Boot?

a) Spring is a Java framework, while Spring Boot is a Java library.

b) Spring Boot provides pre-configured defaults, while Spring offers manual configurations.

c) Spring Boot is a lightweight version of Spring, while Spring is a full-featured framework.

d) Spring Boot is a front-end framework, while Spring is a back-end framework.

44. What is the purpose of the *@SpringBootApplication* annotation?

a) To enable Spring Boot auto-configuration.

b) To define a Spring Boot starter class.

c) To define a Spring Boot controller.

d) To define a Spring Boot service.

45. Which annotation does Spring Boot provide for integration testing?

a) *@SpringBootTest*

b) *@WebMvcTest*

c) *@DataJpaTest*

d) None of the above

46. Which annotation is used to unit test Spring MVC controllers?

a) *@SpringBootTest*

b) *@WebMvcTest*

c) *@DataJpaTest*

d) None of the above

47. What is the main purpose of the Spring Framework?

a) To provide a comprehensive programming and configuration model for Java-based enterprise applications

b) To provide a comprehensive programming and configuration model for JavaScript-based web applications

c) To provide a comprehensive programming and configuration model for PHP-based web applications

d) To provide a comprehensive programming and configuration model for Python-based web applications

48. What is the purpose of the Spring MVC module?

a) To provide a web framework for building web applications

b) To provide a data access framework for accessing databases

c) To provide a caching framework for caching data

d) To provide a security framework for securing web applications

49. What are two ways to achieve dependency Injection in Spring?

a) Using Getter and Setter methods

b) Using Setter and Constructor

c) Using Getter and Constructor

d) Using Setter and Factory methods

50. Spring MVC Framework is designed based on which Design Pattern?

a) Model-View-Controller (MVC)

b) Layered pattern

c) Client-server pattern

d) None of the above

51. What is the purpose of the Spring JDBC module?

a) To provide a data access framework for accessing databases

b) To provide a web framework for building web applications

c) To provide aspect-oriented programming functionality

d) To provide caching functionality

52. Which Spring annotation is used to create RESTful web services using Spring MVC?

a) @RestController

b) @Controller

c) @Component

d) @Rest

53. @RestController annotation is a combination of the below two annotations

a) @Component and @ResponseBody annotations

b) @Controller and @ResponseBody annotations

c) @Service and @ResponseBody annotations

d) None of the above

Part: B and C

1. Explain the basic Concepts of Spring and Spring Boot. (8)
2. How can we Set up the Development Environment needed for spring boot? (8)
3. Explain about Spring Boot Auto Configuration. (8)
4. Explain in detail about Testing in Spring Boot with needed example. (16)
5. Explain the steps to Set up a skeleton Web App in Spring Boot Application. (8)
6. How a Three-Tier Architecture build in Spring boot. Explain it with neat diagram. (16)
7. Mention the ways to model the Domain, Business Logic, and Presentation Layer. (8)