Spring Boot

1. **Getting Started with Spring Boot**
    a. Introduction
    b. Spring vs Spring Boot
    c. Create your First Spring Boot Project
2. **Spring Core Concept**
    a. Inversion of Control
    b. Dependency Injection
    c. Spring Bean Lifecycle
    d. Singleton, Prototype Scope
    e. Create a Spring Bean
    f. Spring Autowiring
    g. DispatcherServlet
    h. Maven/Gradle (project build tools)
3. **Spring Boot Core Features**
    a. Architecture
    b. Annotations
    c. Auto-configuration
    d. Create a basic application
    e. Application Properties
    f. YAML Configuration
    g. Actuator
    h. Logging
    i. DevTools
4. **Spring Boot with REST API**
    a. Introduction
    b. @RestController
    c. @RequestMapping
    d. @GetMapping & @PostMapping
    e. @PutMapping & @DeleteMapping
    f. @PathVariable & @RequestParam
    g. @RequestBody
    h. REST API
    i. JSON Serialization/Deserialization
    j. Exception Handling
    k. Validation

5. **Spring Boot with Database and Data JPA**
    a. Spring Data JDBC
    b. Spring Data JPA
    c. Spring Data JPA (Operator I Pagination I Sorting)
    d. Spring Data MongoDB

e. Spring Data REST
f. Spring Data JPA Auditing | Envers
g. Password Encryption using Jasypt
h. Spring Data JPA One To Many Mapping & Join Query
i. Spring Data JPA One To Many Mapping & Join Query-2
j. Spring Data JPA Transaction Management

6. **Advanced Spring Boot Features**
    a. Scheduling Tasks
    b. Sending Emails
    c. File Handling & Uploading Files
    d. Caching
    e. Caching with other Providers
    f. Transaction Management
    g. DTO Mapping
7. **Spring Security**
    a. Spring Security : Basic Authentication & Fundamentals
    b. Spring Security: Authentication & Role Base Authorization
    c. D Spring Security Auth using Spring Data JPA + MySQL
    d. Spring Security Authentication Internal Work Flow
    e. Spring Security Json Web Token
8. **Spring AOP**
    a. Understanding Spring AOP Basics & its Components
    b. Before Advice
    c. After Advice & After returning Advice
    d. After Throwing & Around Advice
    e. Advice Realtime example
9. **Spring Boot Messaging**
    a. Apache Kafka Architecture & Components
    b. Apache Kafka Publisher Example
    c. Apache Kafka Consumer Example
10. **Spring Boot Final Project**
    a. Building an ecommerce site

Getting Started with Spring Boot

Introduction to Spring Boot

Spring is one of the most popular frameworks for building enterprise applications, but traditional Spring projects require heavy XML configuration, making them complex for beginners. Spring Boot solves this problem by providing a ready-to-use, production-grade framework on top of Spring. It eliminates boilerplate configuration, comes with an embedded server and focuses on rapid development.

Features of Spring Boot

Spring Boot is built on top of the conventional Spring framework, providing all the features of Spring while being significantly easier to use. Here are its key features:

1. Auto-Configuration: Automatically configures components (like Hibernate, JPA) based on dependencies. No need for manual XML setup.

2. Easy Maintenance and Creation of REST Endpoints: With annotations like @RestController, @GetMapping and @PostMapping, creating REST endpoints is straightforward.

Example:

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

3. Embedded Servers: Spring Boot includes an embedded Tomcat server, eliminating the need for manual setup. It also supports Jetty and Undertow, offering flexibility for different application needs.

4. Easy Deployment: Spring Boot applications can be packaged as JAR or WAR files and deployed directly to servers or cloud environments. By 2025, it offers seamless integration with Docker and Kubernetes for easier cloud-native deployment and scaling.

5. Microservice-Based Architecture: Spring Boot is well-suited for microservice architecture, where applications are split into independent, modular services. Unlike monolithic systems, this approach improves scalability, maintainability and deployment flexibility.

Evolution of Spring Boot

Spring Boot was introduced in 2013 following a JIRA request by Mike Youngstrom to simplify Spring bootstrapping.

Major Spring Boot Versions

- Spring Boot 1.0 (April 2014)
- Spring Boot 2.0 (March 2018)
- Spring Boot 3.0 (November 2022) (Introduced Jakarta EE 9+, GraalVM support)

Latest Version (2025): Spring Boot 3.x (Enhancements in observability, native images and containerization)

Applications of Spring Boot

Spring Boot is widely used in modern software development because of its simplicity, scalability and production-ready features. Here are some of its major applications:

1. Enterprise Applications: Build complex applications such as Hospital Management Systems, Banking Systems or ERP solutions with minimal configuration.

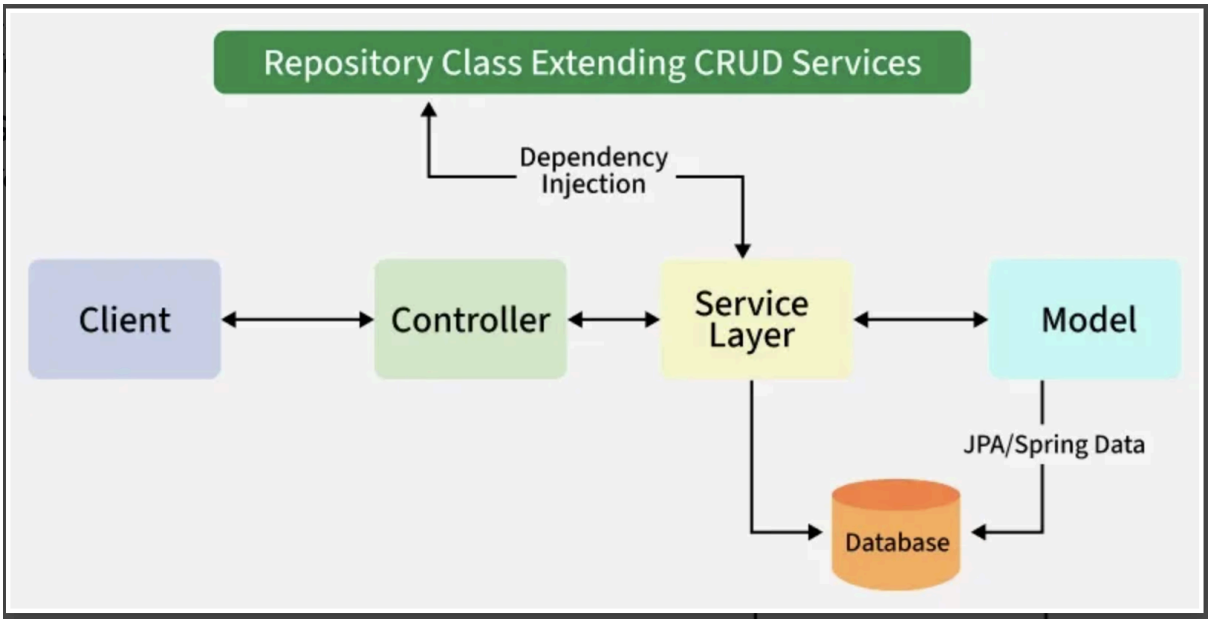
2. Cloud-Native Applications: Seamless integration with Docker, Kubernetes and cloud platforms (AWS, Azure, GCP) for deployment and scaling.

3. Real-Time Applications: Supports Reactive Programming for chat applications, streaming platforms, IoT systems and event-driven systems.

4. Batch Processing Applications: With Spring Batch, Spring Boot is used for ETL (Extract, Transform, Load), report generation and large data processing.

Spring Boot Architecture

To understand the architecture of Spring Boot, let's examine its different layers and components.



Spring Boot Flow Architecture

1. Client Layer

- This represents the external system or user that interacts with the application by sending HTTPS requests.

2. Controller Layer (Presentation Layer)

- Handles incoming HTTP requests from the client.
- Processes the request and sends a response.
- Delegates business logic processing to the Service Layer.

3. Service Layer (Business Logic Layer)

- Contains business logic and service classes.
- Communicates with the Repository Layer to fetch or update data.
- Uses Dependency Injection to get required repository services.

4. Repository Layer (Data Access Layer)

- Handles CRUD (Create, Read, Update, Delete) operations on the database.
- Extends Spring Data JPA or other persistence mechanisms.

5. Model Layer (Entity Layer)

- Represents database entities and domain models.
- Maps to tables in the database using JPA/Spring Data.

6. Database Layer

- The actual database that stores application data.
- Spring Boot interacts with it through JPA/Spring Data.

Request Flow in Spring Boot

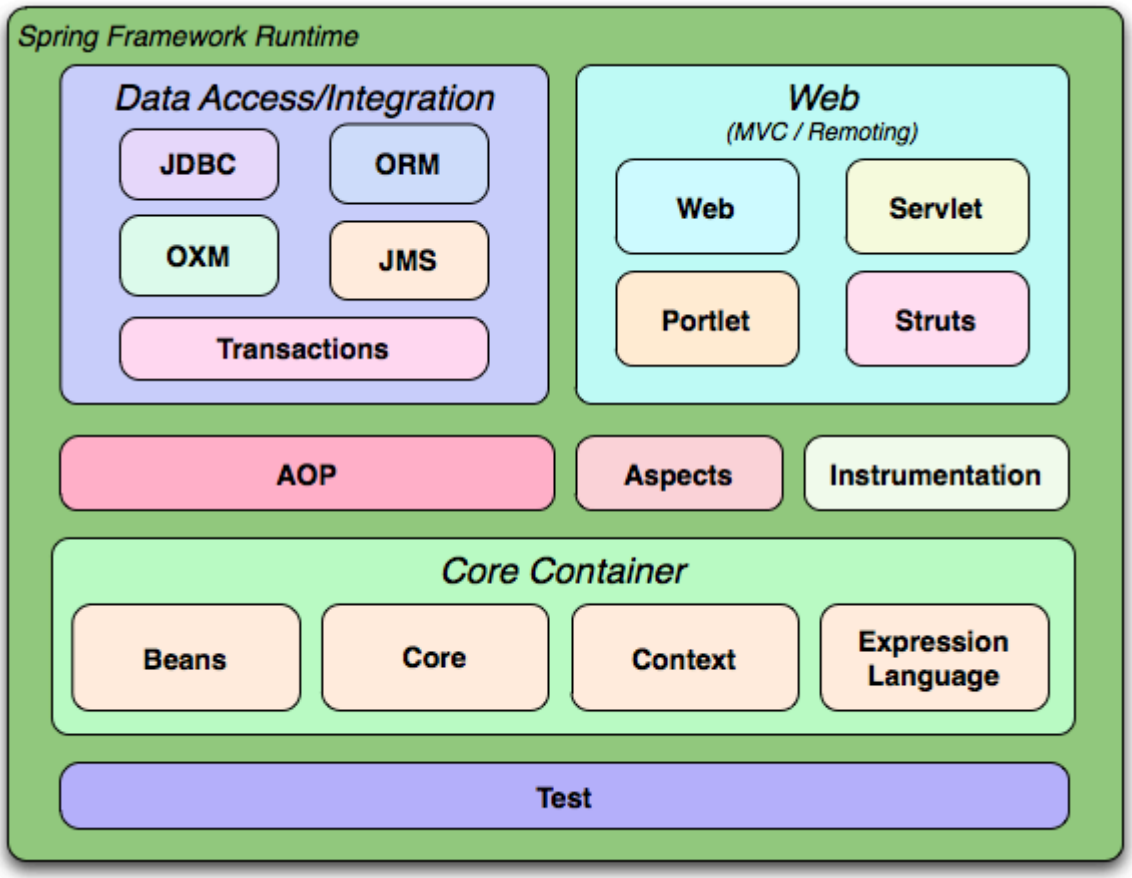
```
Client ->Controller ->Service ->Repository ->Database ->Response
```

- A Client makes an HTTPS request (GET/POST/PUT/DELETE).
- The request is handled by the Controller, which is mapped to the corresponding route.
- If business logic is required, the Controller calls the Service Layer.
- The Service Layer processes the logic and interacts with the Repository Layer to retrieve or modify data in the Database.
- The data is mapped using JPA with the corresponding Model/Entity class.
- The response is sent back to the client. If using Spring MVC with JSP, a JSP page may be returned as the response if no errors occur.

Spring vs Spring Boot

Spring

Spring is an open-source lightweight framework that allows Java developers to build simple, reliable, and scalable enterprise applications. This framework mainly focuses on providing various ways to help you manage your business objects. It made the development of Web applications much easier compared to classic Java frameworks and Application Programming Interfaces (APIs), such as Java database connectivity (JDBC), JavaServer Pages (JSP), and Java Servlet. This framework uses various new techniques such as Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), and dependency injection (DI), to develop enterprise applications. The Spring framework can be considered as a collection of sub-frameworks, also called layers, such as Spring AOP. Spring Object-Relational Mapping (Spring ORM). Spring Web Flow, and Spring Web MVC. You can use any of these modules separately while constructing a Web application. The modules may also be grouped to provide better functionalities in a Web application.



Spring Boot

Spring Boot is built on top of the conventional spring framework. So, it provides all the features of spring and is easier to use than spring. Spring Boot is a microservice-based framework and makes a production-ready application in very less time. In Spring Boot everything is auto configured. We just need to use proper configuration for utilizing a particular functionality. Spring Boot is very useful if we want to develop REST API.

Why Spring Boot over Spring?

- Dependency resolution to avoid version control
- Minimum or very less configuration.
- Embedded tomcat or jetty
- Absolutely no boilerplate code and no requirement for xml configuration.
- Provide production ready features like metrics or healthcheck.
- Autoconfiguration.

Difference between Spring and Spring Boot

Spring	Spring Boot
Spring is an open-source lightweight framework widely used to develop enterprise applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
The most important feature of the Spring Framework is dependency injection.	The most important feature of the Spring Boot is Autoconfiguration.
It helps to create a loosely coupled application.	It helps to create a stand-alone application.
To run the Spring application, we need to set the server explicitly.	Spring Boot provides embedded servers such as Tomcat and Jetty etc.
To run the Spring application, a deployment descriptor is required.	There is no requirement for a deployment descriptor.
To create a Spring application, the developers write lots of code.	It reduces the lines of code.
It doesn't provide support for the in-memory database.	It provides support for the in-memory database such as H2.
Developers need to write boilerplate code for smaller tasks.	In Spring Boot, there is reduction in boilerplate code.
Developers have to define dependencies manually in the pom.xml file.	pom.xml file internally handles the required dependencies.

Spring Core Concept

Inversion of Control

Spring IoC (Inversion of Control) Container is the core of the Spring Framework. It creates and manages objects (beans), injects dependencies and manages their life cycles. It uses Dependency Injection (DI), based on configurations from XML files, Java-based configuration, annotations or POJOs. Since the container, not the developer, controls object creating and wiring, it's called Inversion of Control (IoC).

Types Ioc Containers
There are two types of IoC containers in Spring:

- BeanFactory
- ApplicationContext

1. BeanFactory
The BeanFactory is the most basic version of the IoC container.
It provides basic support for dependency injection and bean lifecycle management. It is suitable for lightweight applications where advanced features are not required.

2. ApplicationContext
The ApplicationContext is an extension of BeanFactory with more enterprise features like event propagation, internationalization and more.
It is the preferred choice for most Spring applications due to its advanced capabilities.

Key Features of IoC Container
The key features of IoC Container are listed below

- **Dependency Injection:** Automatically injects dependencies into our classes.
- **Lifecycle Management:** Manages the lifecycle of beans, including instantiation, initialization and deletion.
- **Configuration Flexibility:** Supports both XML-based and annotation-based configurations.
- **Loose Coupling:** Promotes loose coupling by decoupling the implementation of objects from their usage.

Understanding IoC in Spring with a Practical Example

Step 1: Create the Sim Interface

So now let's understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().

```
public interface Sim {
    void calling();
    void data();
}
```

Step 2: Implement the Sim interface

Now we have created another two classes, Airtel and Banglalink which implement the Sim interface and override its methods..

```
public class Airtel implements Sim {
    @Override
    public void calling() {
        System.out.println("Airtel Calling");
    }

    @Override
    public void data() {
        System.out.println("Airtel Data");
    }
}

public class Banglalink implements Sim {
    @Override
    public void calling() {
        System.out.println("Banglalink Calling");
    }

    @Override
    public void data() {
        System.out.println("Banglalink Data");
    }
}
```

Step 3: Calling Methods Without Spring IoC

Without Spring IoC, we would manually create instances of the Sim implementation in the main method. For example:

```
public class Mobile {
    // Main driver method
    public static void main(String[] args){
        Sim sim = new Banglalink();

        // Calling methods
        sim.calling();
        sim.data();
    }
}
```

This approach tightly couples the Mobile class to the Banglalink implementation. If we want to switch to Airtel, we need to modify the source code.

Step 4: Using Spring IoC with XML Configuration
To avoid tight coupling, we can use the Spring IoC container. First, we create an XML configuration file (beans.xml) to define the beans.

Example: beans.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/
       https://www.springframework.org/schema/beans/spring-beans.xsd">

  <!-- Define the Banglalink bean -->
  <bean id="sim" class="Banglalink"></bean>

</beans>
```

Explanation: In the beans.xml file, we define beans by giving each a unique id and specifying the class name. Later, in the main method, we can use these beans — which will be shown in the next example.

Step 5: Run the Code

In the Mobile class, we use the ApplicationContext to retrieve the bean:

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Mobile {
    public static void main(String[] args) {
        // Using ApplicationContext to implement Spring IoC
        ApplicationContext applicationContext
        = new ClassPathXmlApplicationContext("beans.xml");

        // Get the bean
        Sim sim = applicationContext.getBean("sim", Sim.class);

        // Calling the methods
        sim.calling();
        sim.data();
    }
}
Output:

Banglalink Calling
Banglalink Data
```

And now if you want to use the Airtel sim you have to change only inside the beans.xml file. The main method is going to be the same.

```
<bean id="sim" class="Airtel"></bean>
```

Implementation:

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Mobile {
    public static void main(String[] args) {
        // Using ApplicationContext to implement Spring IoC
        ApplicationContext applicationContext
        = new ClassPathXmlApplicationContext("beans.xml");

        // Get the bean
        Sim sim = applicationContext.getBean("sim", Sim.class);


        // Calling the methods
```

```
        sim.calling();
        sim.data();
    }
}
Output:

Airtel Calling
Airtel Data
```

Step 6: Using Java-Based Configuration
Modern Spring applications often use Java-based configuration instead of XML. Let's see how to configure the same example using Java-based configuration.

1. Create a Configuration Class: Define a configuration class using the @Configuration annotation. Use the @Bean annotation to define beans.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public Sim sim() {
        return new Banglalink();
    }
}
```

2. Use the Configuration in the Mobile Class: Update the Mobile class to use the Java-based configuration.

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContex
t;

public class Mobile {
    public static void main(String[] args) {
        // Load the Spring IoC container using Java-based configuration
        ApplicationContext context
        = new AnnotationConfigApplicationContext(AppConfig.class);

        // Retrieve the bean
        Sim sim = context.getBean("sim", Sim.class);

        // Call methods
```

```
        sim.calling();
        sim.data();
    }
}
Output:
Banglalink Calling
Banglalink Data
```

To switch to Airtel, simply update the AppConfig class:

```
@Bean
public Sim sim() {
    return new Airtel();
}
```

Step 7: Using Annotations for Dependency Injection
Spring also supports annotation-based configuration, which is widely used in modern applications. Let's update the example to use annotations.

1. Enable Component Scanning: Add the @ComponentScan annotation to the configuration class to enable component scanning.

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

2. Annotate Classes with @Component: Annotate the Airtel and Banglalink classes with @Component

```
import org.springframework.stereotype.Component;

@Component
public class Airtel implements Sim {
    @Override
    public void calling() {
        System.out.println("Airtel Calling");
    }

    @Override
    public void data() {
        System.out.println("Airtel Data");
    }
}
```

```
@Component
public class Banglalink implements Sim {
    @Override
    public void calling() {
        System.out.println("Banglalink Calling");
    }

    @Override
    public void data() {
        System.out.println("Banglalink Data");
    }
}
```

3. Inject the Dependency Using @Autowired: Update the Mobile class to use @Autowired for dependency injection

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContex
t;
import org.springframework.stereotype.Component;

@Component
public class Mobile {
    @Autowired
    private Sim sim;

    public void useSim() {
        sim.calling();
        sim.data();
    }

    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        Mobile mobile = context.getBean(Mobile.class);
        mobile.useSim();
    }
}
Output:
Banglalink Calling
Banglalink Data
```

Dependency Injection

Dependency Injection (DI) is a design pattern in which objects receive their dependencies from an external source rather than creating them internally. It promotes loose coupling, easier testing, and better code maintainability. In Spring, DI is achieved mainly through Constructor Injection or Setter Injection.

Need for Dependency Injection

- Suppose class One requires an object of class Two to perform its operations. It means class One is dependent on class Two.
- While such dependency may seem acceptable, in real-world applications, it can lead to tight coupling, difficulty in maintenance, reduced testability or potential system failures.
- Therefore, direct dependencies should be avoided.
- Spring IoC (Inversion of Control) resolves such dependency issues using Dependency Injection (DI).
- Dependency Injection makes the code easier to test and reuse.
- Loose coupling between classes is achieved by defining interfaces for common functionality or letting the injector (Spring container) provide the appropriate implementation.
- The task of instantiating objects is done by the container according to the configurations specified by the developer.

Types of Spring Dependency Injection

There are two primary types of Spring Dependency Injection:

1. Setter Dependency Injection (SDI):

Setter DI involves injecting dependencies via setter methods. To configure SDI, the @Autowired annotation is used along with setter methods and the property is set through the <property> tag in the bean configuration file.

```
public class GFG {

    // The object of the interface Test
    private Test test;

    // Setter method for property test with @Autowired annotation
    @Autowired
    public void settest(Test test) {
        this.test = test;
    }
}
```

Bean Configuration:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
          http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="GFG" class="com.example.org.GFG">
        <property name="test"  ref ="CsvGFG" />
    </bean>

<bean id="CsvGFG" class="com.example.org.impl.CsvGFG" />
<bean id="JsonGFG" class="com.example.org.impl.JsonGFG" />

</beans>
```

This injects the CsvGFG bean into the GFG object using the setter method (settest).

2. Constructor Dependency Injection (CDI):

Constructor DI involves injecting dependencies through constructors. To configure CDI, the <constructor-arg> tag is used in the bean configuration file.

```
package com.example.org;

import com.example.org.Test;

public class GFG {

    // The object of the interface Test
    private Test test;

    // Constructor to set the CDI
    public GFG(Test test) {
        this.test = test;
    }
}
```

Bean Configuration:

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
          http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">


    <bean id="GFG" class="com.pathshala.org.GFG">
        <constructor-arg>
            <bean class="com.pathshala.org.impl.CsvGFG" />
        </constructor-arg>
    </bean>

<bean id="CsvGFG" class="com.pathshala.org.impl.CsvGFG" />
<bean id="JsonGFG" class="com.pathshala.org.impl.JsonGFG" />

</beans>
```

This injects the CsvGFG bean into the GFG object via the constructor.

Spring Bean Lifecycle

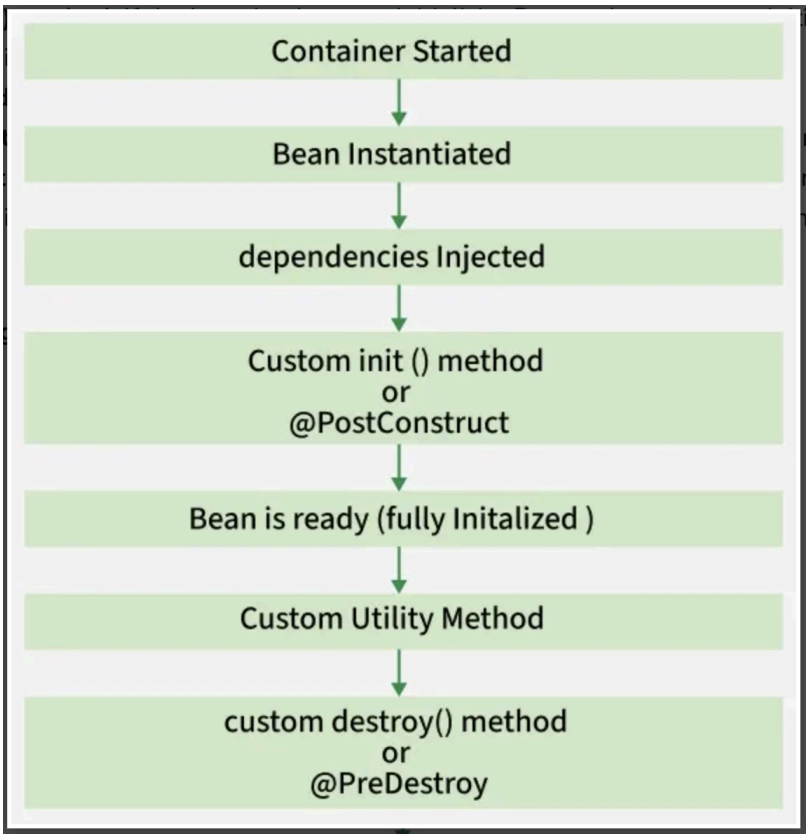
The bean lifecycle in Spring is the sequence of steps a bean goes through from creation to destruction and it's managed by the Spring container.

Bean Life Cycle Phases

The lifecycle of a Spring bean consists of the following phases, which are listed below

- **Container Started**: The Spring IoC container is initialized.
- **Bean Instantiated:** The container creates an instance of the bean.
- **Dependencies Injected:** The container injects the dependencies into the bean.
- **Custom init() method:** If the bean implements InitializingBean or has a custom initialization method specified via @PostConstruct or init-method.
- **Bean is Ready:** The bean is now fully initialized and ready to be used.
- **Custom utility method:** This could be any custom method you have defined in your bean.
- **Custom destroy() method:** If the bean implements DisposableBean or has a custom destruction method specified via @PreDestroy or destroy-method, it is called when the container is shutting down.

The below image demonstrates the Spring bean lifecycle:



Note: We can choose a custom method name instead of init() and destroy(). Here, we will use init() method to execute all its code as the spring container starts up and the bean is instantiated and destroy() method to execute all its code on closing the container.

Ways to Implement the Bean Life Cycle

Spring provides three ways to implement the life cycle of a bean. In order to understand these three ways, let's take an example. In this example, we will write and activate init() and destroy() method for our bean (HelloWorld.java) to print some messages on start and close of the Spring container. Therefore, the three ways to implement this are:

1. Using XML Configuration

In this approach, in order to avail custom init() and destroy() methods for a bean we have to register these two methods inside the Spring XML configuration file while defining a bean. Therefore, the following steps are followed:

Step1: Create the bean Class

Firstly, we need to create a bean HelloWorld.java in this case and write the init() and destroy() methods in the class.

```
package beans;

public class HelloWorld {

    // Custom init method
    public void init() throws Exception {
        System.out.println("Bean HelloWorld has been instantiated, and
I'm the init() method");
    }

    // Custom destroy method
    public void destroy() throws Exception {
        System.out.println("Container has been closed, and I'm the
destroy() method");
    }
}
```

Step 2: Configure the Spring XML File

Now, we need to configure the spring XML file spring.xml and need to register the init() and destroy() methods in it.

```
<beans xmlns="http://www.springframework.org/schema/beans//////"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans//////
    http://www.springframework.org/schema/beans///////spring-beans.xsd">

    <bean id="hw" class="beans.HelloWorld"
          init-method="init" destroy-method="destroy"/>
    </bean>
</beans>
```

Step 3: Create a Driver Class

We need to create a driver class to run this bean.

```
package test;
import beans.HelloWorld;
import org.springframework.context.ConfigurableApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {
    public static void main(String[] args) throws Exception {
        // Loading the Spring XML configuration file into the spring
container and it will create the instance of the bean as it loads into
container
        ConfigurableApplicationContext cap = new
ClassPathXmlApplicationContext("resources/spring.xml");
        cap.close();
    }
}
Output:
Bean HelloWorld has been instantiated and I'm the init() method
Container has been closed and I'm the destroy() method
```

2. Using Programmatic Approach (Interface)

To provide the facility to the created bean to invoke the custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to implement our bean with two interfaces.

In this approach, we implement the InitializingBean and DisposableBean interfaces and override their methods afterPropertiesSet() and destroy() method is invoked just after the container is closed.

Note: To invoke the destroy method we have to call a close() method of ConfigurableApplicationContext.

Step 1: Create the Bean Class

We need to create a bean class HelloWorld, by implementing InitializingBean, DisposableBean and overriding afterPropertiesSet() and destroy() method.

```
package beans;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class HelloWorld implements InitializingBean, DisposableBean {

      @Override
      public void afterPropertiesSet() throws Exception {
            System.out.println("Bean HelloWorld has been "  +
"instantiated and I'm the " + "init() method");
      }

      @Override
      public void destroy() throws Exception {
            System.out.println("Container has been closed "+ "and I'm
the destroy() method");
      }
}
```

Step 2: Configure the Spring XML File

Now, we need to configure the spring XML file spring.xml and define the bean.

```
<beans xmlns="http://www.springframework.org/schema/beans//////"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans//////

http://www.springframework.org/schema/beans///////spring-beans.xsd">

    <bean id="hw" class="beans.HelloWorld"/>
</beans>
```

Step 3: Create a Driver Class

Finally, we need to create a driver class to run this bean.

```
public class Client {
      public static void main(String[] args) throws Exception{

            ConfigurableApplicationContext cap = new
ClassPathXmlApplicationContext("resources/spring.xml");
            cap.close();
      }
```

```
}
Output:
Bean HelloWorld has been instantiated and I'm the init() method
Container has been closed and I'm the destroy() method
```

3. Using Annotations

To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to annotate init() method by @PostConstruct annotation and destroy() method by @PreDestroy annotation.

Note: To invoke the destroy() method we have to call the close() method of ConfigurableApplicationContext.

Step 1: Create the Bean Class

We need to create a bean HelloWorld.java in this case and annotate the custom init() method with @PostConstruct and destroy() method with @PreDestroy.

```
package beans;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class HelloWorld {

	@PostConstruct
	public void init() throws Exception
	{
		System.out.println(
			"Bean HelloWorld has been "   + "instantiated and I'm
the " + "init() method");
	}

	@PreDestroy
	public void destroy() throws Exception
	{
		System.out.println("Container has been closed " + "and I'm
the destroy() method");
	}
}
```

Step 2: Configure the Spring XML File

Now, we need to configure the spring XML file spring.xml and define the bean.

```
<beans xmlns="http://www.springframework.org/schema/beans//////"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans//////
http://www.springframework.org/schema/beans///////spring-beans.xsd">
    <bean
class="org.springframework.context.annotation.CommonAnnotationBeanPostPr
ocessor"/>
    <!-- Configure the bean -->
    <bean id="hw" class="beans.HelloWorld"/>
</beans>
```

Step 3: Create a Driver Class

Finally, we need to create a driver class to run this bean.

```
package test;

import org.springframework.context.ConfigurableApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import beans.HelloWorld;

public class Client {
      public static void main(String[] args) throws Exception {
            ConfigurableApplicationContext cap
                  = new ClassPathXmlApplicationContext(
                        "resources/spring.xml");
            cap.close();
      }
}
Output:
Bean HelloWorld has been instantiated and I'm the init() method
Container has been closed and I'm the destroy() method
```

Singleton, Prototype Scope

Bean Scopes refer to the lifecycle of a Bean, which means when the object of a Bean is instantiated, how long it lives, and how many objects are created for that Bean throughout its lifetime. Basically, it controls the instance creation of the bean, and it is managed by the Spring container.

Bean Scopes in Spring

The Spring framework provides five scopes for a bean. We can use three of them only in the context of a web-aware Spring ApplicationContext, and the rest of the two are available for both an IoC container and a Spring-MVC container. The following are the different scopes provided for a bean:

- **Singleton:** Only one instance will be created for a single bean definition per Spring IoC container, and the same object will be shared for each request made for that bean.
- **Prototype:** A new instance will be created for a single bean definition every time a request is made for that bean.
- **Request:** A new instance will be created for a single bean definition every time an HTTP request is made for that bean. But only valid in the context of a web-aware Spring ApplicationContext.
- **Session:** Scopes a single bean definition to the lifecycle of an HTTP Session. But only valid in the context of a web-aware Spring ApplicationContext.
- **Global-Session:** Scopes a single bean definition to the lifecycle of a global HTTP Session. It is also only valid in the context of a web-aware Spring ApplicationContext.

Difference Between Singleton and Prototype Bean

Singleton	Prototype
Only one instance is created for a single bean definition per Spring IoC container.	A new instance is created for a single bean definition every time a request is made for that bean.
Same object is shared for each request made for that bean. i.e. The same object is returned each time it is injected.	For each new request a new instance is created. i.e. A new object is created each time it is injected.
By default, the scope of a bean is singleton. So we don't need to declare a bean as a singleton explicitly.	By default, scope is not a prototype, so you have to declare the scope of a bean as prototype explicitly.
Singleton scope should be used for stateless beans.	While prototype scope is used for all beans that are stateful.

Note: In this article, all the example uses XML for clarity, in modern Spring especially in Spring Boot, we define scopes with annotations like below:

```
@Component
@Scope("prototype")  // or @Scope("singleton")
public class HelloWorld { ... }
```

Singleton Scope

If the scope is a singleton, then only one instance of that bean will be instantiated per Spring IoC container and the same instance will be shared for each request. That is when the scope of a bean is declared singleton, then whenever a new request is made for that bean, spring IOC container first checks whether an instance of that bean is already created or not. If it is already created, then the IOC container returns the same instance otherwise it creates a new instance of that bean only at the first request. By default, the scope of a bean is a singleton. Let's understand this scope with an example.

Step 1: Lets first create a bean (i.e.), the backbone of the application in the spring framework.

```
// Java program to illustrate a bean
// created in the spring framework
package bean;

public class HelloWorld {
    public String name;

    // Create a setter method to
    // set the value passed by user
    public void setName(String name)
    {
        this.name = name;
    }

    // Create a getter method so that
    // the user can get the set value
    public String getName()
    {
        return name;
    }
}
```

Step 2: Now, we write a Spring XML configuration file "spring.xml" and configure the bean defined above.

```
<!DOCTYPE beans PUBLIC
     "-//SPRING//DTD BEAN 2.0//EN"
     "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
     <!--configure the bean HelloWorld.java
         and declare its scope-->
     < bean
         id = "hw"
         class= "bean.HelloWorld"
         scope = "singleton" / >
</beans>
```

Step 3: Finally, write a driver class "Client.java" to request the above bean.

```
package driver;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support
    .ClassPathXmlApplicationContext;
import bean.HelloWorld;

public class Client {
    public static void main(String[] args){
        ApplicationContext ap
            = new ClassPathXmlApplicationContext(
                "resources/spring.xml");
        HelloWorld hello1 = (HelloWorld) ap.getBean("hw");
        hello.setName("Test");

        System.out.println("Hello object (hello1)" + " Your name is: "
            + hello.getName());

        HelloWorld hello2 = (HelloWorld)ap.getBean("hw");
        System.out.println("Hello object (hello2)" + " Your name is: "
            + hello2.getName());

        System.out.println("'hello1' and 'hello2'" + " are referring"
            + "to the same object: " + (hello1 == hello2));

        System.out.println("Address of object hello1: "+ hello1);
        System.out.println("Address of object hello2: " + hello2);
    }
}
```

```
Output:
Hello object (hello1) Your name is: Test1
Hello object (hello2) Your name is: Test1
'hello1' and 'hello2' are referring to the same object: true
Address of object hello1: bean.HelloWorld@627551fb
Address of object hello2: bean.HelloWorld@627551fb
```

Prototype Scope

If the scope is declared prototype, then spring IOC container will create a new instance of that bean every time a request is made for that specific bean. A request can be made to the bean instance either programmatically using getBean() method or by XML for Dependency Injection of secondary type. Generally, we use the prototype scope for all beans that are stateful, while the singleton scope is used for the stateless beans. Let's understand this scope with an example:

Step 1: Let us first create a bean (i.e.), the backbone of the application in the spring framework.

```
// Java program to illustrate a bean
// created in the spring framework
package bean;

public class HelloWorld {
    public String name;

    // Create a setter method to
    // set the value passed by user
    public void setName(String name)
    {
        this.name = name;
    }

    // Create a getter method so that
    // the user can get the set value
    public String getName()
    {
        return name;
    }
}
```

Step 2: Now, we write a Spring XML configuration file "spring.xml" and configure the bean defined above.

```
<!DOCTYPE beans PUBLIC
      "-//SPRING//DTD BEAN 2.0//EN"
      "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
 < beans>
      <!--configure the bean HelloWorld.java and declare its scope-->
     < bean id = "hw" class = "bean.HelloWorld" scope = "prototype" / >
</ beans>
```

Step 3: Finally, write a driver class "Client.java" to request the above bean.

```
package driver;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support
    .ClassPathXmlApplicationContext;
import bean.HelloWorld;

public class Client {
    public static void main(String[] args){
        ApplicationContext ap
            = new ClassPathXmlApplicationContext(
                "resources/spring.xml");

        HelloWorld hello1
            = (HelloWorld)ap.getBean("hw");
        hello1.setName("Test1");
        System.out.println("Hello object (hello1)"+ " Your name is: "
            + hello1.getName());

        HelloWorld hello2 = (HelloWorld)ap.getBean("hw");
        System.out.println("Hello object (hello2)"+ "Your name is: "
            + hello2.getName());

        System.out.println(
            "'hello1' and 'hello2'" + "are referring "
            + "to the same object: " + (hello1 == hello2));

        System.out.println("Address of object hello1: " + hello1);
        System.out.println("Address of object hello2: " + hello2);
    }
}
```

```
Output:
Hello object (hello1) Your name is: Test1
Hello object (hello2) Your name is: null
'hello1' and 'hello2' are referring to the same object: false
Address of object hello1: bean.HelloWorld@47ef968d
Address of object hello2: bean.HelloWorld@23e028a9
```

Create a Spring Bean

Spring is one of the most popular Java EE frameworks. It is an open-source lightweight framework that allows Java EE 7 developers to build simple, reliable, and scalable enterprise applications. This framework mainly focuses on providing various ways to help you manage your business objects. It made the development of Web applications much easier than compared to classic Java frameworks and application programming interfaces (APIs), such as Java database connectivity (JDBC), JavaServer Pages(JSP), and Java Servlet. In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Different Methods to Create a Spring Bean

Here we are going to discuss how to create a Spring Bean in 3 different ways as follows:

1. Creating Bean Inside an XML Configuration File (beans.xml)
2. Using @Component Annotation
3. Using @Bean Annotation

Method 1: Creating Bean Inside an XML Configuration File (beans.xml)

One of the most popular ways to create a spring bean is to define a bean in an XML configuration file something like this.

```
<bean id="AnyUniqueId" class="YourClassName">
</bean>
```

Let us create a simple class Student having two attributes id and studentName and later creating a simple method to print the details of the student.

Example

```
public class Student {
    private int id;
    private String studentName;

    public void displayInfo(){
        System.out.println("Student Name is " + studentName
                           + " and Roll Number is " + id);
    }
}
```

Now let’s create an XML file named beans.xml file in the project classpath. And inside this beans.xml file, we have to define our Student bean something like this. And that's it. In this way, you can create beans in spring.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/
       https://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="studentAmiya" class="Student">
    </bean>
</beans>
```

Method 2: Using @Component Annotation

Spring Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. 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. @Component is an annotation that allows Spring to automatically detect the custom beans.

Example: Suppose we have already a Java project and all the Spring JAR files are imported into that project. Now let's create a simple class named College and inside the class, we have a simple method. Below is the code for the College.java file.

A. File: College.java

```
package ComponentAnnotation;

public class College {
    public void test(){
        System.out.println("Test College Method");
    }
}
```

Now let's create a Bean for this class. So we can use @Component annotation for doing the same task. So we can modify our College.java file something like this. And that's it.

B. College.java

```
package ComponentAnnotation;
import org.springframework.stereotype.Component;

@Component("collegeBean")
public class College {
    public void test() {
        System.out.println("Test College Method");
    }
}
```

Method 3: Using @Bean Annotation

One of the most important annotations in spring is the @Bean annotation which is applied on a method to specify that it returns a bean to be managed by Spring context. Spring Bean annotation is usually declared in Configuration classes methods. Suppose we have already a Java project and all the Spring JAR files are imported into that project. Now let's create a simple class named College and inside the class, we have a simple method. Below is the code for the College.java file.

A. College.java

```
package BeanAnnotation;

import org.springframework.stereotype.Component;

public class College {

    public void test(){
        System.out.println("Test College Method");
    }
}
```

Now let's create a Configuration class named CollegeConfig. Below is the code for the CollegeConfig.java file

B. CollegeConfig.java

```
package ComponentAnnotation;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CollegeConfig {

}
```

Here, we are going to create the spring beans using the @Bean annotation. To create the College class bean using the @Bean annotation inside the configuration class we can write something like this inside our CollegeConfig.java file. Please refer to the comments for a better understanding.

```
@Bean
public College collegeBean(){
    return new College();
}
```

Implementation: Below is the complete code for the CollegeConfig.java file that is below as follows:

```
package BeanAnnotation;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CollegeConfig {
    @Bean
    public College collegeBean() {
        return new College();
    }
}
```

Spring Autowiring

Autowiring in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as Autowiring in Spring. To enable Autowiring in the Spring application we should use @Autowired annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Modes of Autowiring

Modes	Description
No	This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.
byName	It uses the name of the bean for injecting dependencies.
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect	The autodetect mode uses two other modes for autowiring - constructor and byType.

1. No

This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

```
<bean id="state" class="sample.State">
 <property name="name" value="Dhaka" />
</bean>
<bean id="city" class="sample.City"></bean>
```

2. byName

It uses the name of the bean for injecting dependencies. However, it requires that the name of the property and bean must be the same. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">
 <property name="name" value="Dhaka" />
</bean>
<bean id="city" class="sample.City" autowire="byName"></bean>
```

3. byType

It injects the dependency according to the type of the bean. It looks up in the configuration file for the class type of the property. If it finds a bean that matches, it injects the property. If not, the program throws an error. The names of the property and bean can be different in this case. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">
 <property name="name" value="Dhaka" />
</bean>
<bean id="city" class="sample.City" autowire="byType"></bean>
```

4. constructor

It injects the required dependencies by invoking the constructor. It works similar to the “byType” mode but it looks for the class type of the constructor arguments. If none or more than one bean are detected, then it throws an error, otherwise, it autowires the “byType” on all constructor arguments.

```
<bean id="state" class="sample.State">
 <property name="name" value="Dhaka" />
</bean>
<bean id="city" class="sample.City" autowire="constructor"></bean>
```

5. autodetect

The autodetect mode uses two other modes for autowiring - constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

```
<bean id="state" class="sample.State">
 <property name="name" value="Dhaka" />
</bean>
<bean id="city" class="sample.City" autowire="autodetect"></bean>
```

Example of Autowiring

Sample Program for the byType Mode

```
public class State {
    private String name;
    public String getName() { return name; }
    public void setName(String s) { this.name = s; }
}

class City {
    private int id;
    private String name;

    @Autowired
    private State s;

    public int getID() {
        return id;
    }

    public void setId(int eid) {
        this.id = eid;
    }

    public String getName() {
        return name;
    }

    public void setName(String st) {
        this.name = st;
    }

    public State getState() {
        return s;
    }

    public void showCityDetails() {
        System.out.println("City Id : " + id);
        System.out.println("City Name : " + name);
        System.out.println("State : " + s.getName());
    }
}
```

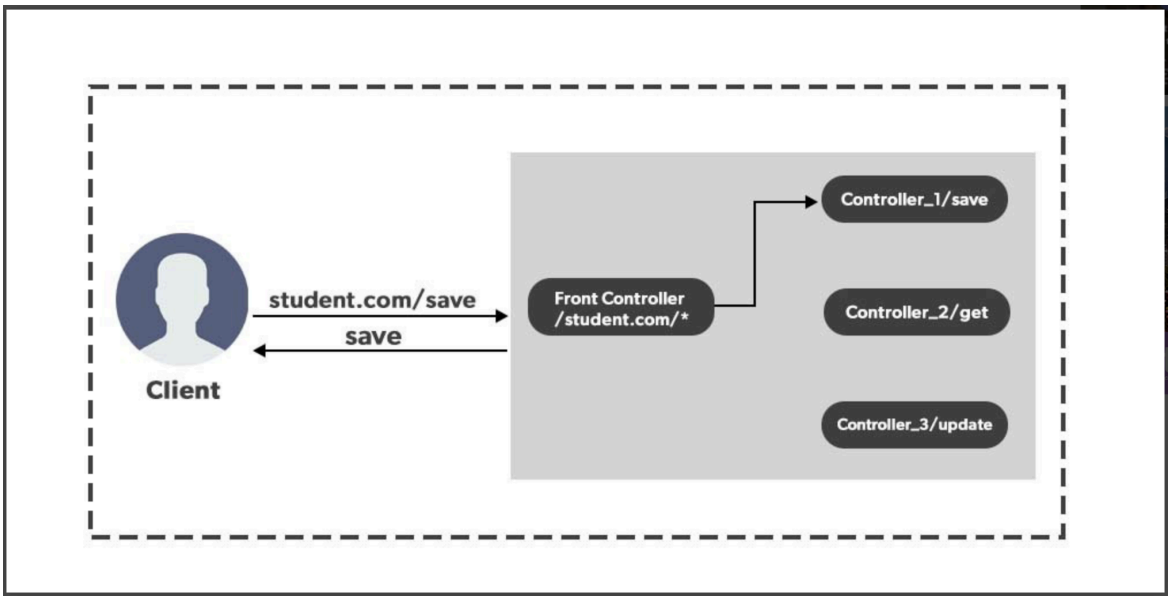
```
Spring bean configuration file:
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context/"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/
       http://www.springframework.org/schema/beans//spring-beans.xsd
       http://www.springframework.org/schema/context/

http://www.springframework.org/schema/context//spring-context.xsd">
<bean id="state" class="sample.State">
<property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="byName"></bean>
</beans>

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        City cty = context.getBean("city", City.class);
        cty.setId(01);
        cty.setName("Dhaka");
        State st = context.getBean("state", State.class);
        st.setName("Dhaka");
        cty.setState(st);

        cty.showCityDetails();
    }
}
Output:
City ID : 01
City Name : Dhaka
State : Dhaka
```

Advantage of Autowiring: It requires less code because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring: No control of the programmer and It can't be used for primitive and string values.

DispatcherServlet

DispatcherServlet is the Front Controller in a Spring web application. It acts as the entry point for all incoming HTTP requests. When a user makes a request (e.g., student.com/save), the DispatcherServlet receives it first, then decides which controller should handle it (e.g., Controller_1 for /save). After the controller processes the request, the response is sent back to the user.



DispatcherServlet handles incoming HTTP requests and delegates them to the appropriate components. It works with HandlerAdapter interfaces configured in the Spring application to process the request. It uses annotations like @Controller, @RequestMapping, etc., to identify handler methods (controller endpoints) and prepares the appropriate response objects based on the controller's output.

Setting Up Dispatcher Servlet

Step 1: Create a Dynamic Web Project in your STS IDE. You may refer to this article to create a Dynamic Web Project in STS.

Step 2: Download the spring JARs file from this link and go to the **src > main > webapp > WEB-INF > lib** folder and past these JAR files.

Step 3: Refer to this article Configuration of Apache Tomcat Server and configure the tomcat server with your application. Now we are ready to go.

Step 4: Now go to the **src > main > webapp > WEB-INF > web.xml** file and here we have to configure our front controller inside a <servlet>...</servlet> tag something like this.

```
<servlet>
    <servlet-name>frontcontroller-dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
```

Now let's tell this servlet, you have to handle all the requests coming to our website called student.com (for this example). To tell servlet this. we can write something like this.

```
<servlet-mapping>
    <servlet-name>frontcontroller-dispatcher</servlet-name>
    <url-pattern>/student.com/*</url-pattern>
</servlet-mapping>
```

When Dispatcher Servlet will be Initialized: The Dispatcher Servlet will be Initialized once we deploy the created dynamic web application inside the tomcat server. So before deploying it let's add the following line inside the web.xml file

```
<load-on-startup>1</load-on-startup>
```

So now the modified code for the servlet is

```
<servlet>
      <servlet-name>frontcontroller-dispatcher</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet
</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Why this line "<load-on-startup>1</load-on-startup>"?
This will make sure that whenever your server will get started the DispatcherServlet will get initialized. If you don't write this line of code then whenever the first request will come to your server starting from /student.com, that time only the DispatcherServlet will be initialized.

Example: File: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html"
xsi:schemaLocation="http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html
http://www.oracle.com/webfolder/technetwork/jsc/xml/ns/javaee/index.html/web-app_4_0.xsd" id="WebApp_ID" version="4.0">
  <display-name>myfirst-mvc-project</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.jsp</welcome-file>
    <welcome-file>default.htm</welcome-file>
  </welcome-file-list>

  <absolute-ordering/>

  <servlet>
      <!-- Provide a Servlet Name -->
    <servlet-name>frontcontroller-dispatcher</servlet-name>
    <!-- Provide a fully qualified path to the DispatcherServlet class -->

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
      <!-- Provide a Servlet Name that you want to map -->
    <servlet-name>frontcontroller-dispatcher</servlet-name>
    <!-- Provide a url pattern -->
    <url-pattern>/student.com/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Step 5: Now go to the **src > main > webapp > WEB-INF** and create an XML file. Actually, this is a Spring Configuration file like beans.xml file. And the name of the file must be in this format **YourServletName-servlet.xml**

For example, for this project, the name of the file must be **frontcontroller-dispatcher-servlet.xml**

Either you can create a Spring Configuration File or you can just create a simple XML file add the below lines of code inside that file. Code for the frontcontroller-dispatcher-servlet.xml is given below.

Example: frontcontroller-dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans/"
       xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context/"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context/

https://www.springframework.org/schema/context/spring-context.xsd">

</beans>
```

Maven/Gradle (project build tools)

Build Tools are programs that help to automate the steps needed to turn your source code into a working software application. These steps include:

- Compiling the code
- Running tests
- Packaging the software
- Deploying it to a production environment.

In DevOps, build tools help make sure that code is integrated and delivered smoothly, automatically, and consistently and with minimal manual effort, every time a developer makes a change.

Build tools are important for several reasons:

1. Time-Saving

Build tools automate repetitive and time-consuming tasks like compiling code, running tests, and packaging applications.

Example: Imagine you are building a Java project. Without a build tool, you have to manually compile every .java file, run each test one by one, then bundle everything into a .jar file every time you make a change. With a tool like Maven, you can do all of this with a single command:

```
mvn package
```

This saves hours of manual work over the life of a project.

2. Consistency

Build tools ensure that every developer or build server gets the same result, regardless of their environment.

Example: Suppose three developers on your team run builds on different operating systems Windows, Mac, and Linux. If you are using Gradle, it ensures the same versions of dependencies and scripts are used, producing consistent results across all machines.

3. Error Reduction
Automation removes the risk of human error during steps like compiling, testing, or deploying.

Example: A developer might forget to include a specific file when manually packaging an app. But a build tool like Ant follows a defined build script (build.xml), which consistently includes all required files every time.

4. Support for CI/CD
In Continuous Integration/Continuous Delivery (CI/CD), build tools are key to automating build, test, and deployment processes.

Example: Let's say you use GitHub Actions to run automated tests whenever someone pushes new code. GitHub Actions will call tools like Maven or npm to:

- Compile the code
- Run unit tests
- Build the application
- Deploy to a test environment

This ensures rapid feedback and smooth deployment without any manual steps.

5. Dependency Management
Build tools automatically download and manage libraries your project depends on.

Example: Suppose your Java app relies on the JUnit testing library. With Maven, all you need to do is add this to your pom.xml:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
</dependency>
```

Below are 3 most used build tools, each offering unique features, advantages and limitations.

1. Maven
Maven is a popular build tool, especially for Java-based projects. It automates the build process and simplifies the management of project dependencies.

Features:
- Dependency Management: Handles and updates libraries automatically.
- Build Automation: Uses a pom.xml file to define tasks and dependencies.
- Repository Management: Has a central place for storing and sharing libraries.
- Plugin Support: Offers plugins to automate tasks like testing and compiling.

Advantages:
- Standardized Structure: Makes Java projects have a consistent structure.
- Cross-platform: Works on different operating systems.
- Large Ecosystem: Plenty of plugins and integrations available.

Limitations:
- Complexity: Can be tricky for beginners due to its detailed XML configuration.
- Slower Builds: Larger projects may take more time to build.

2. Gradle
Gradle is a flexible and fast build tool that works well for Java and many other languages. It's known for speeding up builds by only rebuilding parts of the project that have changed.

Features:
- Groovy-based: Uses Groovy language for writing build scripts, making it easier to read.
- Incremental Builds: Rebuilds only changed parts of the project.
- Multi-language Support: Works with Java, C++, Python, and more.

Advantages:
- Fast: Gradle's incremental builds make it faster, especially for large projects.
- Flexible: Highly customizable to suit different needs.
- Scalable: Works well for both small and large projects.

Limitations:

- Learning Curve: It can take time to get used to Gradle’s flexibility.
- Less Standardized: Its flexible setup can sometimes cause inconsistency.

3. Ant

Ant is an older, but highly customizable build tool. It allows you to define tasks like compiling or deploying code using XML files, giving you full control over the build process.

Features:

- XML-based Configuration: Uses build.xml files to define tasks.
- Task-based Build System: Lets you specify and run tasks in a sequence.
- Platform Independence: Can be used across all platforms as it's written in Java.

Advantages:

- Highly Customizable: Gives you fine control over your build tasks.
- Easy to Extend: You can add custom tasks easily.

Limitations:

- No Dependency Management: Doesn’t automatically handle external libraries, unlike Maven.
- Verbose Configuration: Requires a lot of setup, making it hard to maintain for larger projects.

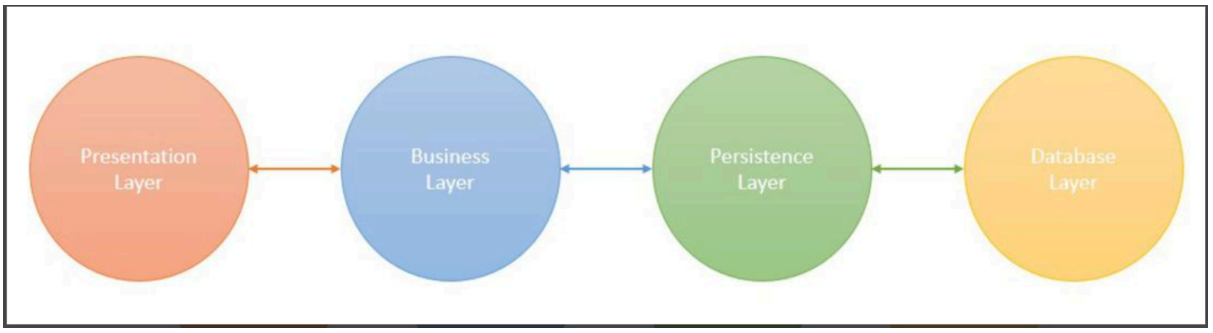
Spring Boot Core Features

Architecture

Spring Boot is built on top of the Spring Framework and follows a layered architecture. Its primary goal is to simplify application development by providing auto-configuration, embedded servers and a production-ready environment out of the box. The architecture of Spring Boot can be divided into several layers and components, each playing an important role in building and running modern applications.

Spring Boot Architecture Layers

Spring Boot consists of the following four layers:



1. Presentation Layer

- Handles HTTP requests through REST controllers (GET, POST, PUT, DELETE).
- Manages authentication, request validation and JSON serialization/deserialization.
- Forwards processed requests to the Business Layer for further logic.

2. Business Layer

The Business Layer is responsible for implementing the application's core logic. It consists of service classes that:

- Process and validate data.
- Handle authentication and authorization (integrating Spring Security if needed).
- Apply transaction management using @Transactional.
- Interact with the Persistence Layer to store or retrieve data.

3. Persistence Layer

The Persistence Layer manages database transactions and storage logic. It consists of repository classes using Spring Data JPA, Hibernate or R2DBC for data access. It is responsible for:

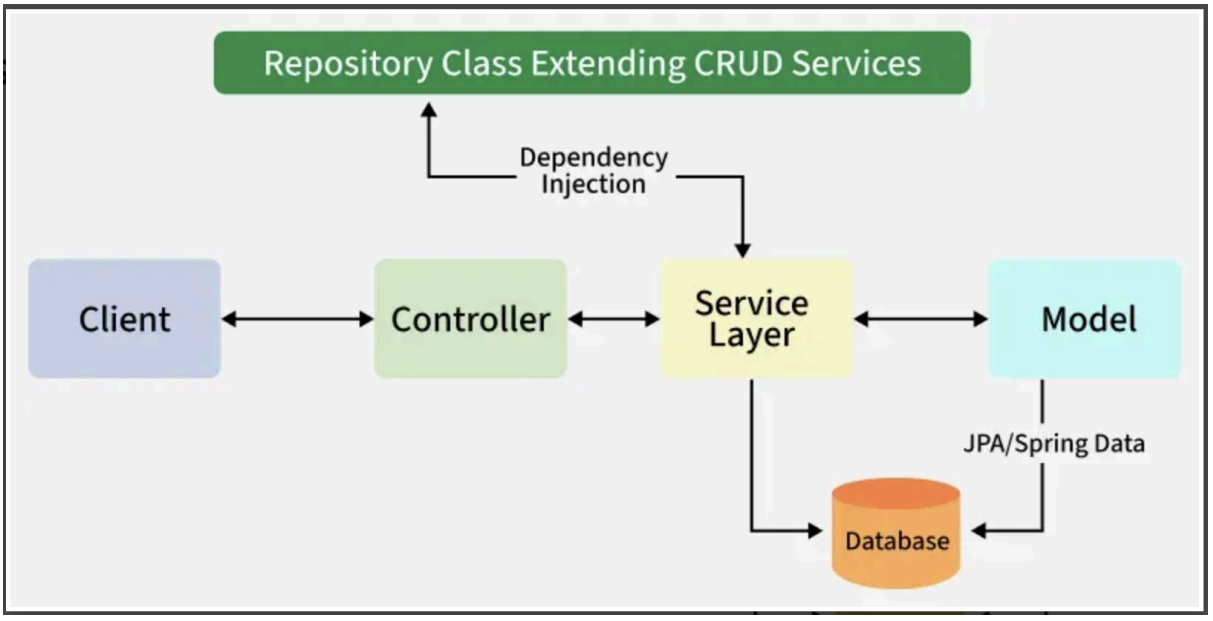
- Mapping Java objects to database records using ORM frameworks.
- Managing CRUD (Create, Read, Update, Delete) operations.
- Supporting relational and NoSQL databases.

4. Database Layer

The Database Layer contains the actual database where the application data is stored. It can support:

- Relational Databases (MySQL, PostgreSQL, Oracle, SQL Server).
- NoSQL Databases (MongoDB, Cassandra, DynamoDB, Firebase).
- Cloud-based databases for scalability.

Spring Boot Flow Architecture



Explanation:

- The client (frontend or API consumer) sends an HTTP request (GET, POST, PUT, DELETE) to the application.
- The request is handled by the Controller Layer, which maps the request to a specific handler method.
- The Service Layer processes business logic and communicates with the Persistence Layer to fetch or modify data.
- The Persistence Layer interacts with the Database Layer using Spring Data JPA or R2DBC, often through a Repository Class that extends CRUD services.
- The processed response is returned as JSON.
- Spring Boot Actuator can be used for monitoring and health checks.

Annotations

Annotations in Spring Boot are metadata that simplify configuration and development. Instead of XML, annotations are used to define beans, inject dependencies and create REST endpoints. They reduce boilerplate code and make building applications faster and easier.

Core Spring Boot Annotations

1. @SpringBootApplication Annotation

This annotation is used to mark the main class of a Spring Boot application. It encapsulates **@SpringBootConfiguration**, **@EnableAutoConfiguration** and **@ComponentScan** annotations with their default attributes.
SpringBoot-Annotation

Example:

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

2. @SpringBootConfiguration Annotation

Indicates that a class provides configuration for a Spring Boot application. Alternative to Spring's @Configuration. Included automatically with @SpringBootApplication.

Example:

```
@SpringBootConfiguration
public class Application {

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

    @Bean
    public StudentService studentService() {
        return new StudentServiceImpl();
    }
}
```

3. @EnableAutoConfiguration Annotation

This annotation auto-configures the beans that are present in the classpath. Enables Spring Boot's auto-configuration mechanism.

Example:

```
@Configuration
@EnableAutoConfiguration
public class Application {

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

4. @ComponentScan Annotation

Tell Spring where to search for components (@Controller, @Service, @Repository, etc.). Typically used with @Configuration.

Example:

```
@Configuration
@ComponentScan
// Main class
public class Application {

    // Main driver method
    public static void main(String[] args)
    {
        SpringApplication.run(Application.class, args);
    }
}
```

5. @ConditionalOnClass Annotation and @ConditionalOnMissingClass Annotation

@ConditionalOnClass Annotation used to mark auto-configuration beans if the class in the annotation's argument is present/absent.

Example:

```
@Configuration
@ConditionalOnClass(MongoDBService.class)
class MongoDBConfiguration {
    // Insert code here
}
```

6. @ConditionalOnBean Annotation and @ConditionalOnMissingBean Annotation

These annotations are used to let a bean be included based on the presence or absence of specific beans.

Example:

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager jpaTransactionManager(
    EntityManagerFactory entityManagerFactory) {
    // Insert code here
}
```

7. @ConditionalOnProperty Annotation

These annotations are used to let configuration be included based on the presence and value of a Spring Environment property.

Example:

```
@Bean
@ConditionalOnProperty(name = "usemongodb", havingValue = "local")
DataSource dataSource() {
    // Insert code here
}

@Bean
@ConditionalOnProperty(name = "usemongodb", havingValue = "prod")
DataSource dataSource(){
    // Insert code here
}
```

8. @ConditionalOnResource Annotation

These annotations are used to let configuration be included only when a specific resource is present in the classpath.

Example:

```
@ConditionalOnResource(resources = "classpath:mongodb.properties")
Properties additionalProperties() {
    // Insert code here
}
```

9. @ConditionalOnExpression Annotation
These annotations are used to let configuration be included based on the result of a SpEL (Spring Expression Language) expression.

SpEL (Spring Expression Language): It is an expression language that supports querying and manipulating an object graph at runtime.

Example:

```
@Bean
@ConditionalOnExpression("'${env}' == 'local'")
public DataSource dataSource() {
    return new HikariDataSource(); // or your DataSource impl
}
```

10. @ConditionalOnCloudPlatform Annotation
These annotations are used to let configuration be included when the specified cloud platform is active.

Example:

```
@Configuration
@ConditionalOnCloudPlatform(CloudPlatform.CLOUD_FOUNDRY)
public class CloudConfigurationExample  {
  // Insert code here
}
```

Request Handling and Controller annotations:
Some important annotations comes under this category are:

- @Controller
- @RestController
- @RequestMapping
- @RequestParam
- @PathVariable
- @RequestBody
- @ResponseBody
- @ModelAttribute

1. @Controller Annotation

- This annotation provides Spring MVC features.
- It is used to create Controller classes and simultaneously it handles the HTTP requests.
- Generally we use @Controller annotation with @RequestMapping annotation to map HTTP requests with methods inside a controller class.

Example:

```
@Controller
public class MyController{
  public String GFG(){
    //insert code here
  }
}
```

2. @RestController Annotation

- This annotation is used to handle REST APIs. and also used to create RESTful web services using Spring MVC.
- It encapsulates @Controller annotation and @ResponseBody annotation with their default attributes.

```
@RestController = @Controller + @ResponseBody
```

Example: Create a Java class and use @RestController annotation to make the class as a request handler

```
@RestController
public class HelloController{
  public String GFG(){
    //insert code here
  }
}
```

3. @RequestMapping Annotation

- Maps HTTP requests to handler methods.
- Supports GET, POST, PUT, DELETE, etc.

Example:

```
@RestController
public class TestController {
    @RequestMapping(value = "/welcome", method = RequestMethod.GET)
    public String welcome() {
        return "Welcome to Spring Boot!";
    }
}
```

For handling specific HTTP requests we can use

- @GetMapping
- @PutMapping
- @PostMapping
- @PatchMapping
- @DeleteMapping

NOTE: We can manually use GET, POST, PUT and DELETE annotations along with the path as well as we can use @RequestMapping annotation along with the method for all the above handler requests

4. @RequestParam Annotation

@RequestParam annotation is used to read the form data and binds the web request parameter to a specific controller method.

Example: Java code to demonstrate @RequestParam annotation

```
@RestController
public class MyController{
  @GetMapping("/authors")
  public String getAuthors(@RequestParam(name="authorName") String
name){
    //insert code here
  }
}
```

5. @PathVariable Annotation

This annotation is used to extract the data from the URI path. It binds the URL template path variable with method variable.

Example:

```
@RestController
public class MyController{
    @GetMapping("/author/{authorName}")
    public String getAuthorName(@PathVariable(name = "authorName")
String name){
      //insert your code here
    }
}
```

6. @RequestBody Annotation

This annotation is used to convert HTTP requests from incoming JSON format to domain objects directly from the request body. Here the method parameter binds with the body of the HTTP request.

Example: Java Code to Demonstrate @RequestBody annotation

```
@RestController
public class MyController{
  @GetMapping("/author")
  public void printAuthor(@RequestBody Author author){
    //insert code here
  }
}
```

7. @ResponseBody Annotation

This annotation is used to convert the domain object into HTTP request in the form of JSON or any other text. Here, the return type of the method binds with the HTTP response body.

Example: Java code to demonstrate @ResponseBody annotation

```
@Controller
public class MyController{
  public @ResponseBody Author getAuthor(){
    Author author = new Author();
    author.setName("GFG");
    author.setAge(20);
    return author;
  }
}
```

8. @ModelAttribute Annotation

This annotation refers to a model object in Spring MVC. It can be used on methods or method arguments as well.

Example:

```
@ModelAttribute("author")
public Author author(){
  //insert code here
}
```

Auto-configuration

Spring Boot is heavily attracting developers toward it because of three main features as follows:

1. Auto-configuration - such as checking for the dependencies, the presence of certain classes in the classpath, the existence of a bean, or the activation of some property.
2. An opinionated approach to configuration.
3. The ability to create stand-alone applications.

Auto-Configuration in Spring Boot

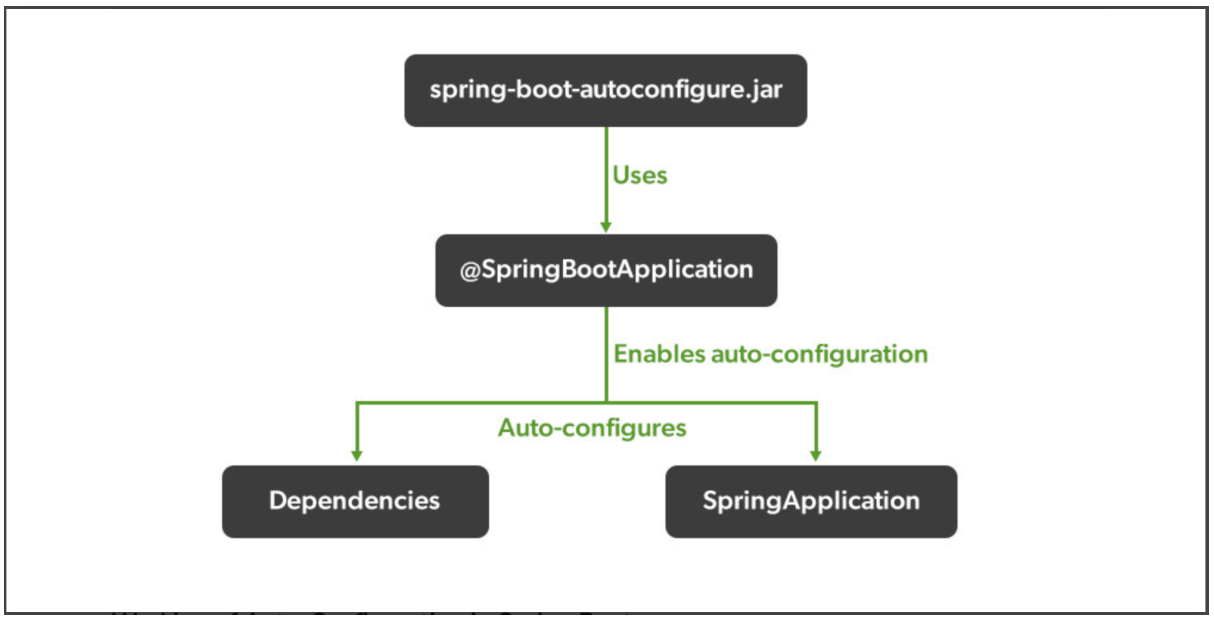
- @Conditional annotation acts as a base for the Spring Boot auto-configuration annotation extensions.
- It automatically registers the beans with @Component, @Configuration, @Bean, and meta-annotations for building custom stereotype annotations, etc.
- The annotation @EnableAutoConfiguration is used to enable the auto-configuration feature.
- The @EnableAutoConfiguration annotation enables the auto-configuration of Spring ApplicationContext by scanning the classpath components and registering the beans.
- This annotation is wrapped inside the @SpringBootApplication annotation along with @ComponentScan and @SpringBootConfiguration annotations.
- When running main() method, this annotation initiates auto-configuration.

Implementation: Bootstrapping of Application

```
package gfg;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Note: You should use the '@EnableAutoConfiguration' annotation only one time in your application.

- 'spring-boot-autoconfigure.jar' is the file that looks after all the auto-configuration.
- All auto-configuration logic for MVC, data, JMS, and other frameworks is present in a single jar



Working of Auto-Configuration in Spring Boot

A: Dependencies

- Auto-Configuration is the main focus of the Spring Boot development.
- Our Spring application needs a respective set of dependencies to work.
- Spring Boot auto-configures a pre-set of the required dependencies without a need to configure them manually.
- This greatly helps and can be seen when we want to create a stand-alone application.
- When we build our application, Spring Boot looks after our dependencies and configures both the underlying Spring Framework and required jar dependencies (third-party libraries) on the classpath according to our project build.
- It helps us to avoid errors like mismatches or incompatible versions of different libraries.
- If you want to override these defaults, you can override them after initialization.

Tool: Maven

Example 1: pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.6</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>sia</groupId>
    <artifactId>taco-cloud</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>taco-cloud</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
        <vaadin.version>14.7.5</vaadin.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jersey</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web-services</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>com.vaadin</groupId>
            <artifactId>vaadin-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
```

```
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-hateoas</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
    <dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>com.vaadin</groupId>
              <artifactId>vaadin-bom</artifactId>
              <version>${vaadin.version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
  </dependencyManagement>
  <profiles>
      <profile>
          <id>production</id>
          <build>
              <plugins>
                  <plugin>
                      <groupId>com.vaadin</groupId>
                      <artifactId>vaadin-maven-plugin</artifactId>
                      <version>${vaadin.version}</version>
                      <executions>
                          <execution>
                              <id>frontend</id>
                              <phase>compile</phase>
                              <goals>
                                  <goal>prepare-frontend</goal>
                                  <goal>build-frontend</goal>
                              </goals>
                              <configuration>
                                <productionMode>true</productionMode>
                              </configuration>
                          </execution>
                      </executions>
                  </plugin>
              </plugins>
          </build>
      </profile>
  </profiles>
</project>
```

Understanding Auto-Configuration of Dependencies

- When you build a Spring Boot project, the 'Starter Parent' dependency gets automatically added in the 'pom.xml' file.
- It notifies that the essential 'sensible' defaults for the application have been auto-configured and you therefore can take advantage of it.

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>...</version>
</parent>
```

- To add the dependency (library of tech stacks), you don't need to mention the version of it because the Spring Boot automatically configures it for you.
- Also, when you update/change the Spring Boot version, all the versions of added dependencies will also get updated/changed.

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

- It is Spring Boot's auto-configuration that makes managing dependencies supremely easy for us.
- With the help of enabling 'debug logging' in the 'application.properties' file, we can know more about auto-configuration.

Application Properties/YML

In Spring Boot applications, configuration plays a very important role in customizing the behavior of the application. Instead of hardcoding values in the code, Spring Boot provides a flexible way to configure application settings using the application.properties or application.yml file.

Why Use application.properties?

- Centralized configuration management.
- Easy to override default Spring Boot settings.
- Provides flexibility for different environments (dev, test, prod).
- Reduces hardcoding in source code.