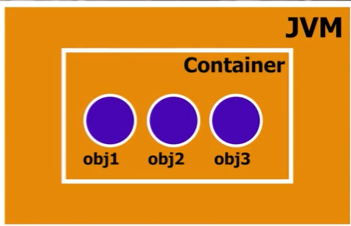
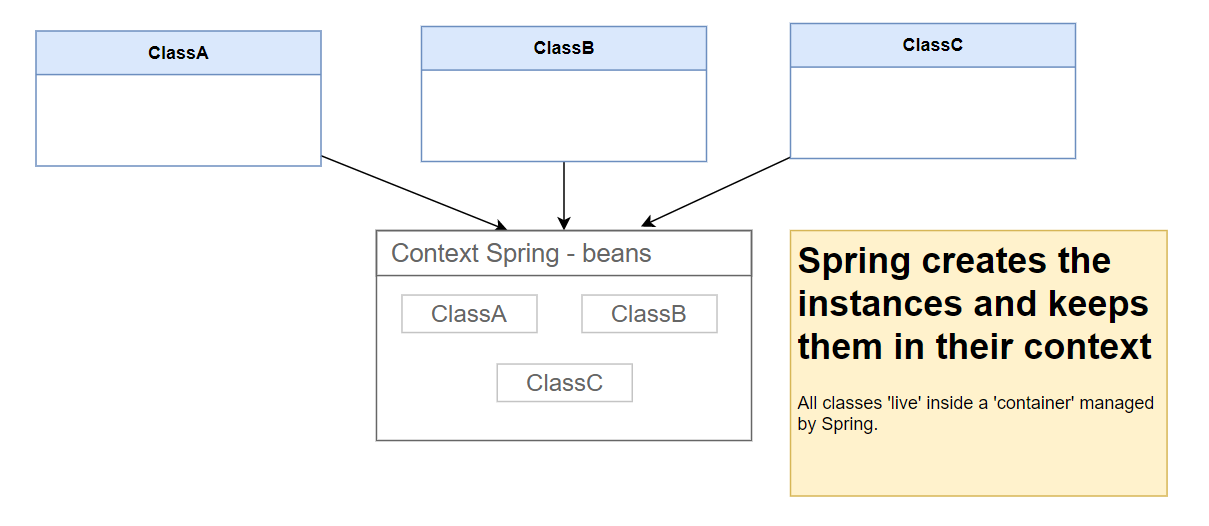
**Spring Framework**

The Spring Framework is a powerful and comprehensive framework used for enterprise Java development.

**1. Inversion of Control (IoC) and Dependency Injection (DI)**

* **IoC**: Spring promotes loose coupling through Inversion of Control, meaning that objects do not create their dependencies but are given them by an external source (the IoC container).
* 
* **DI**: Dependency Injection is a specific implementation of IoC, where dependencies are injected into objects, typically via constructor or setter injection. This makes the code more modular, testable, and maintainable.
* 

**2. Aspect-Oriented Programming (AOP)**

* **AOP**: Spring supports Aspect-Oriented Programming, which allows for the separation of cross-cutting concerns (like logging, security, transaction management) from the main business logic. This modularizes the code and enhances maintainability.

**3. Comprehensive and Flexible**

* **Modular Design**: Spring is highly modular, allowing developers to use only the parts they need (e.g., Spring Core, Spring MVC, Spring Data, Spring Security).
* **Integration**: It provides seamless integration with various technologies and frameworks, including Hibernate, JPA, JMS, and various web frameworks.

**4. Spring MVC and RESTful Web Services**

* **Spring MVC**: A robust framework for building web applications and RESTful web services, following the Model-View-Controller (MVC) design pattern.
* **REST**: Spring makes it easy to build RESTful services with its comprehensive support for REST principles and annotations, simplifying the creation and consumption of REST APIs.

**5. Spring Boot**

* **Simplification**: Spring Boot, a project within the Spring ecosystem, simplifies the setup and development of new Spring applications. It provides defaults and auto-configurations to reduce boilerplate code and configuration, allowing developers to focus on writing business logic.
* **Embedded Servers**: Spring Boot comes with embedded web servers like Tomcat or Jetty, making it easier to run standalone web applications.

**Inversion of Control**

* Inversion of Control (IoC) is a fundamental concept in the Spring Framework that addresses the problem of how objects obtain their dependencies. In traditional programming, a program controls its flow by creating objects and managing their dependencies.
* **In IoC, this control is inverted**: instead of the program creating dependencies, an external container (IoC container) manages the creation and injection of dependencies.

**IoC Container**

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets information from the XML file and works accordingly.

There are two types of IoC containers. They are:

1. **BeanFactory (**org.springframework.beans.factory.**BeanFactory** )
2. **ApplicationContext (**org.springframework.context.**ApplicationContext** )

* The **BeanFactory** and the **ApplicationContext** interfaces acts as the IoC container.
* The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory.
* So, it is better to use ApplicationContext than BeanFactory.

**ApplicationContext**

**Types of ApplicationContext**

There are several implementations of the ApplicationContext interface, each suited to different application environments:

1. **ClassPathXmlApplicationContext**: (IMPORTANT)
   * Loads the context definition from an XML file located in the classpath.

1. ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

2.

1. **FileSystemXmlApplicationContext**:
   * Loads the context definition from an XML file located in the filesystem.

1. ApplicationContext context = new FileSystemXmlApplicationContext("C:/path/to/applicationContext.xml");

2.

1. **AnnotationConfigApplicationContext**: (IMPORTANT)
   * Reads the configuration from Java annotations.

1. ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

2.

1. ***WebApplicationContext****:*
   * *A specialized ApplicationContext for web applications, which integrates with the lifecycle of a Servlet.*

**Dependency Injection**

The Dependency Injection **is a design pattern** that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing. In such case we write the code as:

1. class Employee{

2. Address address;

3.

4. Employee(Address address){

5. this.address=address;

6. }

7. public void setAddress(Address address){

8. this.address=address;

9. }

10.

11. }

12.

In such case, instance of Address class is provided by external sources such as XML file either by constructor or setter method.

Spring framework provides two ways to inject dependency

* ***By Constructor***
* ***By Setter method***

**Java Beans**

JavaBeans (often simply referred to as "beans") are fundamental building blocks for developing applications. ***A JavaBean is a reusable software component that follows certain conventions, making it easy to manage within the Spring container.***

**Key Characteristics of JavaBeans in Spring**

1. **Properties**:
   * JavaBeans have private properties accessible via public getter and setter methods. This encapsulation promotes a clean separation between the internal state and the external interface of the bean.
2. **Default Constructor**:
   * JavaBeans must have a public no-argument constructor. This allows the Spring container to create instances of the bean dynamically at runtime.

**Example of a JavaBean in Spring**

**Step 1: Define the Bean Class**

1. public class MyBean {

2. private String property;

3.

4. // Default constructor

5. public MyBean() {

6. }

7.

8. // Getter method

9. public String getProperty() {

10. return property;

11. }

12.

13. // Setter method

14. public void setProperty(String property) {

15. this.property = property;

16. }

17.

18. // Method to display property

19. public void displayProperty() {

20. System.out.println("Property value: " + property);

21. }

22. }

23.

**Step 2: Configure the Bean in XML**

1. <beans xmlns="http://www.springframework.org/schema/beans"

2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

5.

6. <bean id="myBean" class="com.example.MyBean">

7. <property name="property" value="Hello, Spring!"/>

8. </bean>

9.

10. </beans>

11.

**Or Configure the Bean Using Annotations**

1. @Configuration

2. public class AppConfig {

3. @Bean

4. public MyBean myBean() {

5. MyBean myBean = new MyBean();

6. myBean.setProperty("Hello, Spring!");

7. return myBean;

8. }

9. }

10.

**Step 3: Use the Bean in an Application**

1. public class Main {

2. public static void main(String[] args) {

3. // Load the Spring context from XML configuration

4. ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

5.

6. // Or load the Spring context from Java configuration

7. // ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

8.

9. // Get the MyBean bean from the context

10. MyBean myBean = context.getBean(MyBean.class);

11.

12. // Use the bean

13. myBean.displayProperty();

14. }

15. }

16.

17.

**Setter Injection for Objects**

**Step 1: Define the Student and Address Classes**

**Student.java**

1. public class Student {

2. private Address address;

3.

4. // Default constructor

5. public Student() {

6. }

7.

8. // Getter method

9. public Address getAddress() {

10. return address;

11. }

12.

13. // Setter method

14. public void setAddress(Address address) {

15. this.address = address;

16. }

17.

18. // Method to display address

19. public void displayAddress() {

20. System.out.println("Address: " + address.getFullAddress());

21. }

22. }

23.

**Address.java**

1. public class Address {

2. private String street;

3. private String city;

4. private String state;

5. private String zipCode;

6.

7. // Default constructor

8. public Address() {

9. }

10.

11. // Getter and setter methods

12. public String getStreet() {

13. return street;

14. }

15.

16. public void setStreet(String street) {

17. this.street = street;

18. }

19.

20. public String getCity() {

21. return city;

22. }

23.

24. public void setCity(String city) {

25. this.city = city;

26. }

27.

28. public String getState() {

29. return state;

30. }

31.

32. public void setState(String state) {

33. this.state = state;

34. }

35.

36. public String getZipCode() {

37. return zipCode;

38. }

39.

40. public void setZipCode(String zipCode) {

41. this.zipCode = zipCode;

42. }

43.

44. // Method to get the full address as a string

45. public String getFullAddress() {

46. return street + ", " + city + ", " + state + " " + zipCode;

47. }

48. }

49.

**Step 2: Configure the Beans in XML**

**applicationContext.xml**

1. <beans xmlns="http://www.springframework.org/schema/beans"

2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

5.

6. <bean id="address" class="com.example.Address">

7. <property name="street" value="123 Main St"/>

8. <property name="city" value="Springfield"/>

9. <property name="state" value="IL"/>

10. <property name="zipCode" value="62704"/>

11. </bean>

12.

13. <bean id="student" class="com.example.Student">

14. <property name="address" ref="address"/>

15. </bean>

16.

17. </beans>

18.

**Or Configure the Beans Using Annotations**

**AppConfig.java**

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

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

3.

4. @Configuration

5. public class AppConfig {

6. @Bean

7. public Address address() {

8. Address address = new Address();

9. address.setStreet("123 Main St");

10. address.setCity("Springfield");

11. address.setState("IL");

12. address.setZipCode("62704");

13. return address;

14. }

15.

16. @Bean

17. public Student student() {

18. Student student = new Student();

19. student.setAddress(address());

20. return student;

21. }

22. }

23.

**Step 3: Use the Bean in an Application**

**Main.java**

1. import org.springframework.context.ApplicationContext;

2. import org.springframework.context.support.ClassPathXmlApplicationContext;

3. import org.springframework.context.annotation.AnnotationConfigApplicationContext;

4.

5. public class Main {

6. public static void main(String[] args) {

7. // Load the Spring context from XML configuration

8. ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

9.

10. // Or load the Spring context from Java configuration

11. // ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

12.

13. // Get the Student bean from the context

14. Student student = context.getBean(Student.class);

15.

16. // Use the bean

17. student.displayAddress();

18. }

19. }

20.

**XML Configuration for Constructor Injection**

**applicationContext.xml**

1. <beans xmlns="http://www.springframework.org/schema/beans"

2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

5.

6. <bean id="address" class="com.example.Address">

7. <constructor-arg name="street" value="123 Main St"/>

8. <constructor-arg name="city" value="Springfield"/>

9. <constructor-arg name="state" value="IL"/>

10. <constructor-arg name="zipCode" value="62704"/>

11. </bean>

12.

13. <bean id="student" class="com.example.Student">

14. <constructor-arg ref="address"/>

15. </bean>

16.

17. </beans>

18.

**Java Configuration for Constructor Injection**

**AppConfig.java**

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

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

3.

4. @Configuration

5. public class AppConfig {

6. @Bean

7. public Address address() {

8. return new Address("123 Main St", "Springfield", "IL", "62704");

9. }

10.

11. @Bean

12. public Student student() {

13. return new Student(address());

14. }

15. }

16.

**Scope**

The scope of a bean defines its **lifecycle** and **how it will be shared in the Spring application context**. Here are the main scopes available for beans in Spring:

1. **Singleton (default)**: In singleton scope, the Spring IoC container creates and manages a single instance of the bean per container. This single instance is shared across the entire application context, making it the most common and default scope.

<bean id="exampleBean" class="com.example.ExampleBean" scope="singleton"/>

1. **Prototype**: In prototype scope, the Spring IoC container creates a new instance of the bean each time it is requested (i.e., injected or retrieved from the container). This means multiple instances of the bean can exist within the application context.

<bean id="exampleBean" class="com.example.ExampleBean" scope="prototype"/>

1. **Request**: Scoped to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. Each HTTP request will have its own instance of the bean.
2. **Session**: Scoped to an HTTP session. Like request scope, but scoped to an HTTP session. Each HTTP session will have its own instance of the bean.
3. **Global Session**: Scoped to a global HTTP session. Typically used in a portlet context (portlet-based web applications).
4. **Application**: Scoped to the lifecycle of a ServletContext. This means the bean is scoped to the entire web application context.

In Java Based configuration, scopes can be defined for each bean using an annotation **@Scope(“<value>”).**

1. @Configuration

2. public class AppConfig {

3.

4. @Bean

5. @Scope("prototype")

6. public ExampleBean exampleBean() {

7. return new ExampleBean();

8. }

9. }

10.

* For singleton scope, since it is a **default** scope, it **need not be mentioned explicitly** using annotations.
* **@RequestScope // or @SessionScope, @GlobalSessionScope, @ApplicationScope** annotations are used for the web-related scopes.

**Autowiring**

In Spring Framework's XML configuration, autowiring is a feature that allows Spring to automatically inject dependencies into beans without explicit configuration using <property> or <constructor-arg> tags. Autowiring simplifies configuration by letting Spring resolve dependencies based on types.

Here's how autowiring works in Spring XML configuration:

**Types of Autowiring**

1. **No Autowiring**:
   * This is the default behavior where dependencies must be explicitly configured using <property> or <constructor-arg> tags.

Example:

1. <bean id="userService" class="com.example.UserService">

2. <property name="userRepository" ref="userRepository"/>

3. </bean>

4.

5. <bean id="userRepository" class="com.example.UserRepository"/>

6.

1. **Autowiring by Type**:
   * Spring automatically injects a bean of the **same type** **as the property**.

Example:

1. <bean id="userService" class="com.example.UserService" autowire="byType"/>

2. <bean id="userRepository" class="com.example.UserRepository"/>

3.

In this case, Spring will inject UserRepository into UserService because UserService has a property of type UserRepository.

1. **Autowiring by Name**:
   * Spring injects a bean by **matching the property name to a bean name**.

Example:

1. <bean id="userService" class="com.example.UserService" autowire="byName"/>

2. <bean id="userRepository" class="com.example.UserRepository"/>

3.

Here, Spring injects UserRepository into UserService because there is a bean named userRepository, matching the property name (userRepository) in UserService.

1. **Autowiring by Constructor**:
   * Spring uses the constructor of the bean to determine which beans to inject.

Example:

1. <bean id="userService" class="com.example.UserService" autowire="constructor"/>

2. <bean id="userRepository" class="com.example.UserRepository"/>

3.

In this scenario, Spring will inject UserRepository into the constructor of UserService if UserService has a constructor that accepts UserRepository.

The code for two bean classes is given below :

**UserRepository.class**

1. 1. package com.example;

2. 2.

3. 3. public class UserRepository {

4. 4. // Repository implementation can include data access methods

5. 5. // or interactions with a database

6. 6. }

7. 7.

8.

**UserService.class**

1. package com.example;

2.

3. public class UserService {

4. private UserRepository userRepository;

5.

6. // Constructor for constructor-based autowiring

7. public UserService(UserRepository userRepository) {

8. this.userRepository = userRepository;

9. }

10.

11. // Setter method for setter-based autowiring

12. public void setUserRepository(UserRepository userRepository) {

13. this.userRepository = userRepository;

14. }

15.

16. // Example method in UserService that uses UserRepository

17. public void saveUser(User user) {

18. // Example: Save user using UserRepository

19. userRepository.save(user);

20. }

21. }

22.

**In case of Java-based configuration**, we use the annotation **@Autowired** on the constructor or setter methods of UserService which will help us in dependency injection of UserRepository beans in UserService bean.

**Autowiring “byType” is default for java-based configuration autowiring.**

1. package com.example;

2.

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

4. import org.springframework.stereotype.Service;

5.

6. @Service

7. public class UserService {

8. private UserRepository userRepository;

9.

10. // Constructor for constructor-based autowiring

11. @Autowired

12. public UserService(UserRepository userRepository) {

13. this.userRepository = userRepository;

14. }

15.

16. // Setter method for setter-based autowiring

17. @Autowired

18. public void setUserRepository(UserRepository userRepository) {

19. this.userRepository = userRepository;

20. }

21.

22. // Example method in UserService that uses UserRepository

23. public void saveUser(User user) {

24. // Example: Save user using UserRepository

25. userRepository.save(user);

26. }

27. }

28.

* **Constructor-based autowiring:** The UserService constructor is annotated with @Autowired. When Spring creates a UserService bean, it automatically injects an instance of UserRepository into this constructor.
* **Setter-based autowiring:** The setUserRepository method is annotated with @Autowired. This method can also be used for dependency injection if preferred over constructor injection.

**Autowiring by Name in Java-based configuration**

Autowiring by name can be achieved in Java-based configuration by combining **@Autowired** with **@Qualifier.**

1. package com.example;

2.

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

4. import org.springframework.beans.factory.annotation.Qualifier;

5. import org.springframework.stereotype.Service;

6.

7. @Service

8. public class UserService {

9. private UserRepository userRepository;

10.

11. @Autowired

12. public UserService(@Qualifier("userRepository") UserRepository userRepository) {

13. this.userRepository = userRepository;

14. }

15.

16. // UserService methods

17. }

18.

In this example:

* @Qualifier("userRepository") specifies the bean name (userRepository) that should be injected into the UserService constructor.
* This effectively achieves autowiring by name, ensuring that the specific named bean (userRepository) is injected.

**To summarise auto-wiring for java-based configuration:**

* **Autowiring by Type**: Achieved through @Autowired annotation without additional qualifiers. Spring looks for a bean of the same type and injects it.
* **Autowiring by Name**: Achieved through @Autowired annotation combined with @Qualifier("beanName"). Spring looks for a bean with the specified name (beanName) and injects it.

**Primary Bean**

The concept of a primary bean in Spring is **used to indicate a preferred bean when multiple beans of the same type are candidates for autowiring**. This is particularly useful when you have several implementations of an interface or multiple beans of a particular type, and you want to specify which one should be used by default for autowiring.

**Using Primary Bean in XML Configuration**

In XML configuration, you can mark a bean as primary using the **primary="true"** attribute. Here's how you would define and use a primary bean:

**UserRepository with Primary Attribute in XML**

1. <!-- Define a primary UserRepository bean -->

2. <bean id="primaryUserRepository" class="com.example.UserRepository" primary="true"/>

3.

4. <!-- Another UserRepository bean as a secondary option -->

5. <bean id="secondaryUserRepository" class="com.example.AnotherUserRepository"/>

6.

**UserService Autowiring in XML**

1. <!-- UserService bean with autowiring by type -->

2. <bean id="userService" class="com.example.UserService"/>

3.

In this XML configuration:

* primaryUserRepository is marked as primary with primary="true".
* secondaryUserRepository is another bean of the same type (UserRepository).
* When Spring autowires UserService, it will inject primaryUserRepository by default because it is marked as primary.

**Using Primary Bean in Java-based Configuration**

In Java-based configuration, you can use the @Primary annotation to achieve the same effect. Here's how you would do it:

**UserRepository with @Primary Annotation in Java**

1. package com.example;

2.

3. import org.springframework.context.annotation.Primary;

4. import org.springframework.stereotype.Repository;

5.

6. @Repository

7. @Primary

8. public class PrimaryUserRepository implements UserRepository {

9. // Implementation of UserRepository methods

10. }

11.

**AnotherUserRepository without @Primary Annotation**

1. package com.example;

2.

3. import org.springframework.stereotype.Repository;

4.

5. @Repository

6. public class AnotherUserRepository implements UserRepository {

7. // Implementation of UserRepository methods

8. }

9.

**UserService Autowiring in Java-based Configuration**

1. package com.example;

2.

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

4. import org.springframework.stereotype.Service;

5.

6. @Service

7. public class UserService {

8. private UserRepository userRepository;

9.

10. @Autowired

11. public UserService(UserRepository userRepository) {

12. this.userRepository = userRepository;

13. }

14.

15. // UserService methods

16. }

17.

In this Java-based configuration:

* PrimaryUserRepository is annotated with @Primary, indicating it as the primary bean of type UserRepository.
* AnotherUserRepository does not have @Primary, so it serves as an alternative bean of the same type.
* When Spring autowires UserService, it will inject PrimaryUserRepository by default because it is marked as primary.

**Lazy-init**

In Spring Framework, lazy initialization of beans refers to deferring the creation of a bean until it is actually requested for the first time. This can be particularly useful for beans that are expensive to create or initialize, improving application startup time and memory usage. Here’s how you can configure lazy initialization in both XML and Java-based configurations:

**Lazy Initialization in XML Configuration**

In XML configuration, you can use the lazy-init="true" attribute on the <bean> tag to specify lazy initialization for a particular bean.

1. <!-- Define a bean with lazy initialization -->

2. <bean id="userService" class="com.example.UserService" lazy-init="true">

3. <!-- Bean properties and dependencies -->

4. </bean>

5.

In this example:

* The userService bean will be lazily initialized, meaning it won't be instantiated until it is first requested by the application context.

**Lazy Initialization in Java-based Configuration**

In Java-based configuration, you can use the @Lazy annotation on the bean definition method to achieve lazy initialization.

1. package com.example.config;

2.

3. import com.example.UserService;

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

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

6. import org.springframework.context.annotation.Lazy;

7.

8. @Configuration

9. public class AppConfig {

10.

11. @Bean

12. @Lazy

13. public UserService userService() {

14. return new UserService();

15. }

16. }

17.

In this example:

* The userService() method is annotated with @Lazy, indicating that the UserService bean should be lazily initialized.
* When the Spring application context loads, UserService bean creation is deferred until it is requested for the first time.

**Using getBean() by Type**

The getBean() method allows you to fetch a bean instance by specifying its class type. This approach is useful when you need to obtain a specific implementation or interface that Spring manages as a bean.

1. import com.example.UserService;

2. import org.springframework.context.annotation.AnnotationConfigApplicationContext;

3.

4. public class MainApplication {

5.

6. public static void main(String[] args) {

7. // Initialize the Spring application context

8. AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

9.

10. // Retrieve a bean by type

11. UserService userService = context.getBean(UserService.class);

12.

13. // Use the userService instance

14. userService.saveUser(new User("John", "Doe"));

15.

16. // Close the application context

17. context.close();

18. }

19. }

20.

**Component Stereotypes Annotation**

* Component stereotypes are annotations used to **indicate the roles or purposes of classes** within your application.
* These annotations help Spring manage and configure classes more effectively by providing metadata that Spring can use to understand how to treat these classes at runtime.

Here are the main component stereotypes in Spring:

**1. @Component**

The @Component annotation is a generic stereotype for any Spring-managed component. By annotating a class with @Component, you tell Spring that this class is a Spring bean. Spring will automatically detect and instantiate beans annotated with @Component during the component scanning process.

Example:

1. import org.springframework.stereotype.Component;

2.

3. @Component

4. public class UserService {

5. // Class implementation

6. }

7.

**2. Stereotypes for Specific Layers**

Spring provides more specific stereotypes for different layers of an application, which extend @Component:

* **@Repository**: Indicates that the annotated class is a repository, typically used for database access or data storage.

Example:

1. import org.springframework.stereotype.Repository;

2.

3. @Repository

4. public class UserRepository {

5. // Repository implementation

6. }

7.

* **@Service**: Indicates that the annotated class is a service, typically used for implementing business logic.

Example:

1. import org.springframework.stereotype.Service;

2.

3. @Service

4. public class UserService {

5. // Service implementation

6. }

7.

* **@Controller**: Indicates that the annotated class is a controller, typically used in web applications to handle web requests and responses.

Example:

1. import org.springframework.stereotype.Controller;

2.

3. @Controller

4. public class UserController {

5. // Controller implementation

6. }

7.

**3. @Configuration**

The @Configuration annotation identifies a class as a configuration class for the Spring IoC container. Configuration classes define beans using @Bean methods and can include additional configuration logic, such as externalized properties or bean initialization.

Example:

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

2.

3. @Configuration

4. public class AppConfig {

5. // Bean definitions and configuration logic

6. }

7.

**@ComponentScan** is an annotation used to enable component scanning in your application.

1. **Annotation-based Configuration (@Configuration)**:

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

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

3.

4. @Configuration

5. @ComponentScan(basePackages = "com.example")

6. public class AppConfig {

7. // Other configuration beans and methods

8. }

9.

* + **@ComponentScan(basePackages = "com.example"):** This annotation tells Spring to scan the com.example package and all its sub-packages for components annotated with stereotypes like @Component, @Service, @Repository, and @Controller.

1. **XML-based Configuration**:

If you are using XML-based configuration, you can configure component scanning in your Spring configuration XML file:

1. <beans xmlns="http://www.springframework.org/schema/beans"

2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

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

5.

6. <!-- Enable component scanning -->

7. <context:component-scan base-package="com.example"/>

8.

9. <!-- Other bean definitions -->

10. ...

11.

12. </beans>

13.

* + **<context:component-scan base-package="com.example"/>:** This XML configuration achieves the same as @ComponentScan in Java-based configuration, scanning the specified base package and its sub-packages for Spring-managed components.

**Benefits of Using Component Stereotypes**

* **Automatic Bean Detection**: Spring automatically detects and manages beans annotated with stereotypes like @Component, @Service, @Repository, and @Controller.
* **Improved Code Organization**: Clearly annotating classes with their intended roles (e.g., service, repository) enhances code readability and organization.

**Customizing Component Scanning**

You can customize component scanning in Spring by specifying additional attributes in @ComponentScan or <context:component-scan>:

* **basePackages**: Specify one or more base packages to scan.
* **basePackageClasses**: Specify classes from which to derive base packages.
* **includeFilters and excludeFilters**: Define filters to include or exclude specific components based on annotations, types, or custom criteria.

**@Value(“”) Annotation in Java Based Configuration**

The @Value annotation is used to inject values from properties files, environment variables, or other Spring beans into fields of Spring-managed components (beans). This annotation simplifies the injection of externalized values into your application without needing to use cumbersome property placeholders or environment-specific configuration.

Example:

1. import org.springframework.beans.factory.annotation.Value;

2. import org.springframework.stereotype.Component;

3.

4. @Component

5. public class MyComponent {

6.

7. @Value("${app.message}")

8. private String message;

9.

10. // Constructor, methods, etc.

11. }

12.

**Spring Boot**

Spring Boot provides a variety of annotations that simplify the development of Spring-based applications by automating configuration and providing sensible defaults. These annotations help in configuring beans, defining endpoints, enabling certain behaviours, and more.

Summarizing the 3 already discussed annotations in Spring Boot :

**1. @SpringBootApplication**

The @SpringBootApplication annotation is a combination of three annotations:

* **@Configuration**: Indicates that the class declares one or more @Bean methods.
* **@EnableAutoConfiguration:** Enables Spring Boot's auto-configuration mechanism, allowing Spring to automatically configure beans based on dependencies and environment.
* **@ComponentScan:** Configures component scanning, allowing Spring to automatically discover and register beans (components, services, repositories, etc.) in the specified packages.

1. import org.springframework.boot.SpringApplication;

2. import org.springframework.boot.autoconfigure.SpringBootApplication;

3.

4. @SpringBootApplication

5. public class MyApplication {

6. public static void main(String[] args) {

7. SpringApplication.run(MyApplication.class, args);

8. }

9. }

10.

**2. @Autowired**

The @Autowired annotation injects dependencies into Spring-managed beans automatically. It can be used for constructor injection, setter injection, or field injection.

Example:

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

2. import org.springframework.stereotype.Service;

3.

4. @Service

5. public class MyService {

6.

7. private final MyRepository repository;

8.

9. @Autowired

10. public MyService(MyRepository repository) {

11. this.repository = repository;

12. }

13.

14. // Other service methods

15. }

1. **@Value**

The @Value annotation is used to inject values from properties files, environment variables, or system properties into Spring beans.

Example:

1. import org.springframework.beans.factory.annotation.Value;

2. import org.springframework.stereotype.Component;

3.

4. @Component

5. public class MyComponent {

6.

7. @Value("${app.message}")

8. private String message;

9.

10. // Constructor, methods, etc.

11. }

12.

The moment we annotate our main method in App.java with **@SpringBootApplication** and run our application using **SpringApplication.run(MyApplication.class, args),** no configuration file is needed henceforth, as Spring Boot does all configuration for you.

**Layered Architecture A Spring Boot Application**

In a typical Spring Boot application, you'll often find a three-layered architecture that helps organize and separate concerns effectively: **Controller**, **Service**, and **Repository** layers. Each layer has a distinct role and responsibility, contributing to the overall structure and functionality of the application.

**1. Controller Layer**

The **Controller** layer in Spring Boot is responsible for handling incoming HTTP requests from clients (typically web browsers or REST clients). Controllers interpret these requests, invoke the appropriate business logic (typically located in the Service layer), and then return an HTTP response to the client. They act as an entry point for handling user input and interacting with the application.

**Key Responsibilities:**

* Handling incoming HTTP requests.
* Interpreting request parameters, headers, and payloads.
* Invoking appropriate methods from the Service layer to process business logic.
* Returning an HTTP response, often in the form of JSON, XML, HTML, etc.

**Example:**

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

2. import org.springframework.web.bind.annotation.\*;

3.

4. @RestController

5. @RequestMapping("/api/users")

6. public class UserController {

7.

8. @Autowired

9. private UserService userService;

10.

11. @GetMapping("/{id}")

12. public User getUserById(@PathVariable Long id) {

13. return userService.getUserById(id);

14. }

15.

16. @PostMapping

17. public User createUser(@RequestBody User user) {

18. return userService.createUser(user);

19. }

20.

21. // Other CRUD operations and endpoint mappings

22. }

23.

**2. Service Layer**

The **Service** layer encapsulates the business logic of the application. It contains methods that implement business rules, perform calculations, validate data, and interact with the Repository layer for data persistence. Services are typically stateless and are designed to be reusable across different controllers or clients.

**Key Responsibilities:**

* Implementing business logic and rules.
* Performing data validation and transformation.
* Coordinating transactions (if required).
* Interacting with the Repository layer for data access.

**Example:**

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

2. import org.springframework.stereotype.Service;

3.

4. @Service

5. public class UserService {

6.

7. @Autowired

8. private UserRepository userRepository;

9.

10. public User getUserById(Long id) {

11. return userRepository.findById(id).orElse(null);

12. }

13.

14. public User createUser(User user) {

15. // Additional business logic for creating a user

16. return userRepository.save(user);

17. }

18.

19. // Other business methods for user management

20. }

21.

**3. Repository Layer**

The **Repository** layer is responsible for handling data access and manipulation. It typically interacts directly with the database or other data sources (e.g., file systems, external APIs) to perform CRUD (Create, Read, Update, Delete) operations on entities. Repositories encapsulate the logic for querying and storing data, abstracting away the details of data access from the Service layer.

**Key Responsibilities:**

* Defining CRUD operations for persistent entities.
* Executing queries and updates against the database.
* Managing transactions for data integrity (often handled by Spring’s transaction management).

**Example:**

1. import org.springframework.data.jpa.repository.JpaRepository;

2. import org.springframework.stereotype.Repository;

3.

4. @Repository

5. public interface UserRepository extends JpaRepository<User, Long> {

6. // Custom query methods if needed

7. }

8.

**To summarize**

* **Controller Layer**: Handles incoming HTTP requests, delegates processing to the Service layer, and returns responses to clients.
* **Service Layer**: Implements business logic, coordinates data access through the Repository layer, and performs operations required by the application.
* **Repository Layer**: Manages data access and storage, encapsulates database interactions, and provides CRUD operations for persistent entities.

This three-layered architecture in Spring Boot promotes separation of concerns, modularization of functionalities, and facilitates maintainability and scalability of applications by enforcing clear boundaries between presentation logic (Controller), business logic (Service), and data access logic (Repository).

**Spring Boot JDBC**

Spring Boot JDBC simplifies database access and configuration by providing a streamlined way to interact with relational databases using JDBC (Java Database Connectivity). It builds on top of the Spring Framework's JDBC support, offering auto-configuration, simplified configuration properties, and integration with Spring's transaction management and other features.

**Key Features and Components of Spring Boot JDBC**

1. **Auto-configuration**: Spring Boot JDBC **automatically configures a DataSource bean based on the application's configuration properties**. It detects the presence of JDBC drivers in the classpath and sets up the necessary components for database access.
2. **DataSource Configuration**: You can configure the database connection details in the **application.properties or application.yml** file using Spring Boot's configuration properties. For example:

1. spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

2. spring.datasource.username=root

3. spring.datasource.password=secret

4. spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

5.

Spring Boot auto-configures a DataSource bean using these properties.

1. **JdbcTemplate**: Spring Boot provides the JdbcTemplate class, which simplifies JDBC code by abstracting many of the complexities of working with JDBC APIs directly. It reduces boilerplate code for database operations like querying, updating, inserting, and deleting data.

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

2. import org.springframework.jdbc.core.JdbcTemplate;

3. import org.springframework.stereotype.Repository;

4. import java.util.List;

5.

6. @Repository

7. public class UserRepository {

8.

9. @Autowired

10. private JdbcTemplate jdbcTemplate;

11.

12. public List<User> findAllUsers() {

13. return jdbcTemplate.query("SELECT \* FROM users", (rs, rowNum) ->

14. new User(

15. rs.getLong("id"),

16. rs.getString("username"),

17. rs.getString("email")

18. ));

19. }

20.

21. // Other CRUD operations using JdbcTemplate

22. }

23.

**RowMapper Interface**

RowMapper is an interface used to map rows of a ResultSet to Java objects. It defines a single method, mapRow, which converts each row of the ResultSet into an object of type T.

1. public interface RowMapper<T> {

2. T mapRow(ResultSet rs, int rowNum) throws SQLException;

3. }

**Example: Using RowMapper to Retrieve All Rows from a Table**

Here's an example of how to use JdbcTemplate and RowMapper to retrieve all rows from a table users:

9. @Repository

Remember here we are providing an anonymous class implementation of RowMapper<User> functional interface

10. public class UserRepository {

11.

12. @Autowired

13. private JdbcTemplate jdbcTemplate;

14.

15. // RowMapper implementation

16. private RowMapper<User> userRowMapper = new RowMapper<User>() {

17. @Override

18. public User mapRow(ResultSet rs, int rowNum) throws SQLException {

19. return new User(

\*0. rs.getLong("id"),

21. rs.getString("username"),

22. rs.getString("email")

23. );

24. }

25. };

26.

27. public List<User> findAllUsers() {

.query() method returns a list of rowObjects from each rowMapper

28. String sql = "SELECT \* FROM users";

29. return jdbcTemplate.query(sql, userRowMapper);

30. }

31. }

32.

In this example:

* UserRepository uses JdbcTemplate injected via @Autowired.
* userRowMapper is an instance of RowMapper<User> that maps each row from the ResultSet to a User object.
* findAllUsers() executes a SQL query using jdbcTemplate.query(), passing the SQL query and userRowMapper to map each row of the result set to User objects.

**Using Lambda Expression for RowMapper**

Starting from Java 8, you can use lambda expressions to simplify the RowMapper implementation:

1. public class UserRepository {

2.

3. @Autowired

4. private JdbcTemplate jdbcTemplate;

5.

6. // RowMapper using lambda expression

7. private RowMapper<User> userRowMapper = (rs, rowNum) ->

8. new User(

9. rs.getLong("id"),

10. rs.getString("username"),

11. rs.getString("email")

12. );

13.

14. public List<User> findAllUsers() {

15. String sql = "SELECT \* FROM users";

16. return jdbcTemplate.query(sql, userRowMapper);

17. }

18. }

We could have even passed a lambda expression as second argument to .query() method.

**Using External Databases for Repository Layer**

To switch from H2 (or any default database) to an external database like MySQL or PostgreSQL, you can modify the application.properties or application.yml file accordingly. Here’s how you can configure MySQL as an example:

1. **Configure MySQL Database**:
   * Include MySQL dependency in your pom.xml:

1. <dependency>

2. <groupId>mysql</groupId>

3. <artifactId>mysql-connector-java</artifactId>

4. </dependency>

5.

1. **Update application.properties**:
   * Specify MySQL database connection details:

**properties**

1. spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

2. spring.datasource.username=root

3. spring.datasource.password=secret

4. spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

5. spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

6.

* + Replace jdbc:mysql://localhost:3306/mydatabase with your MySQL database URL, root with your MySQL username, and secret with your MySQL password.

1. **Switching Between Databases**:
   * By updating the spring.datasource.url, username, password, and driverClassName properties, Spring Boot seamlessly switches between different databases. You can also customize Hibernate dialect (spring.jpa.database-platform) for specific database features.