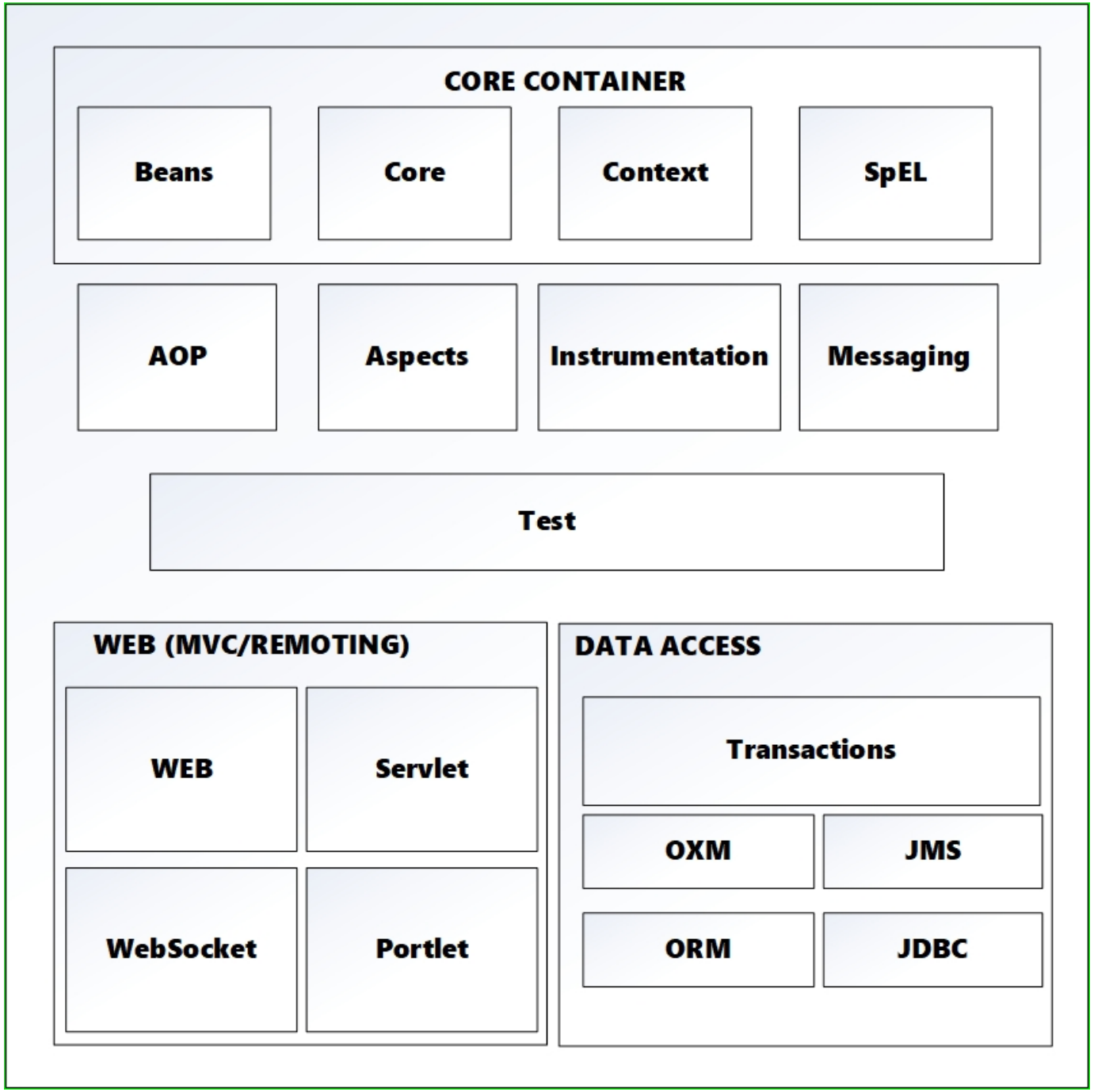

Fawry
Spring Core
By Magued Mamdouh
Download APP
www.Fawry.com

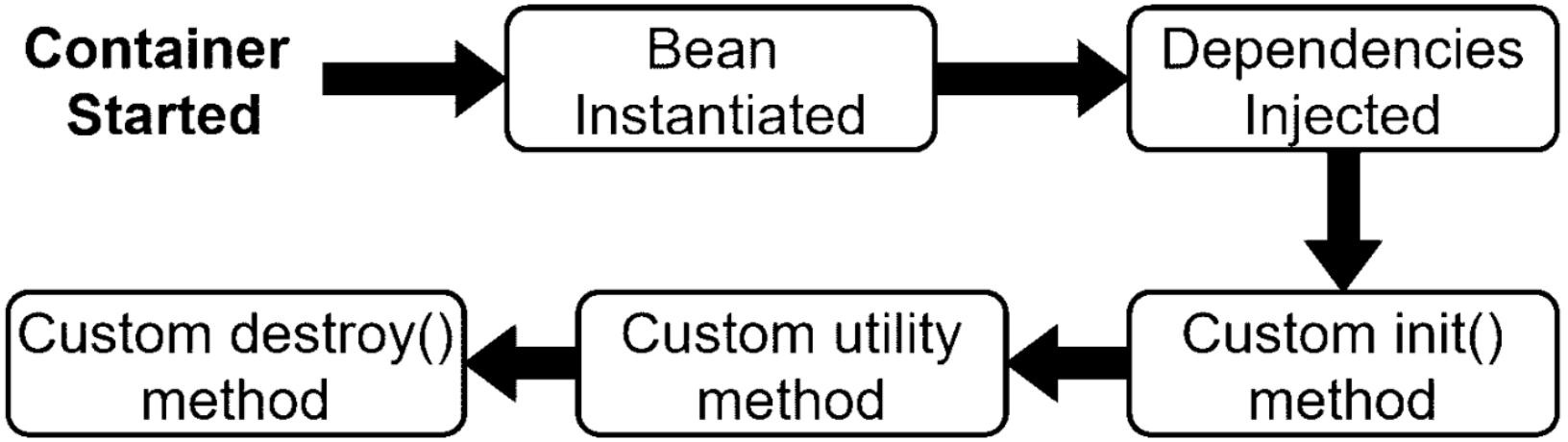
Outlines

- Spring Framework
- Spring Bean Life Cycle
- Spring Bean Scopes
- Dependency Injection
- Spring IoC Container and Spring Bean
- Spring Bean Autowiring
- Spring AOP

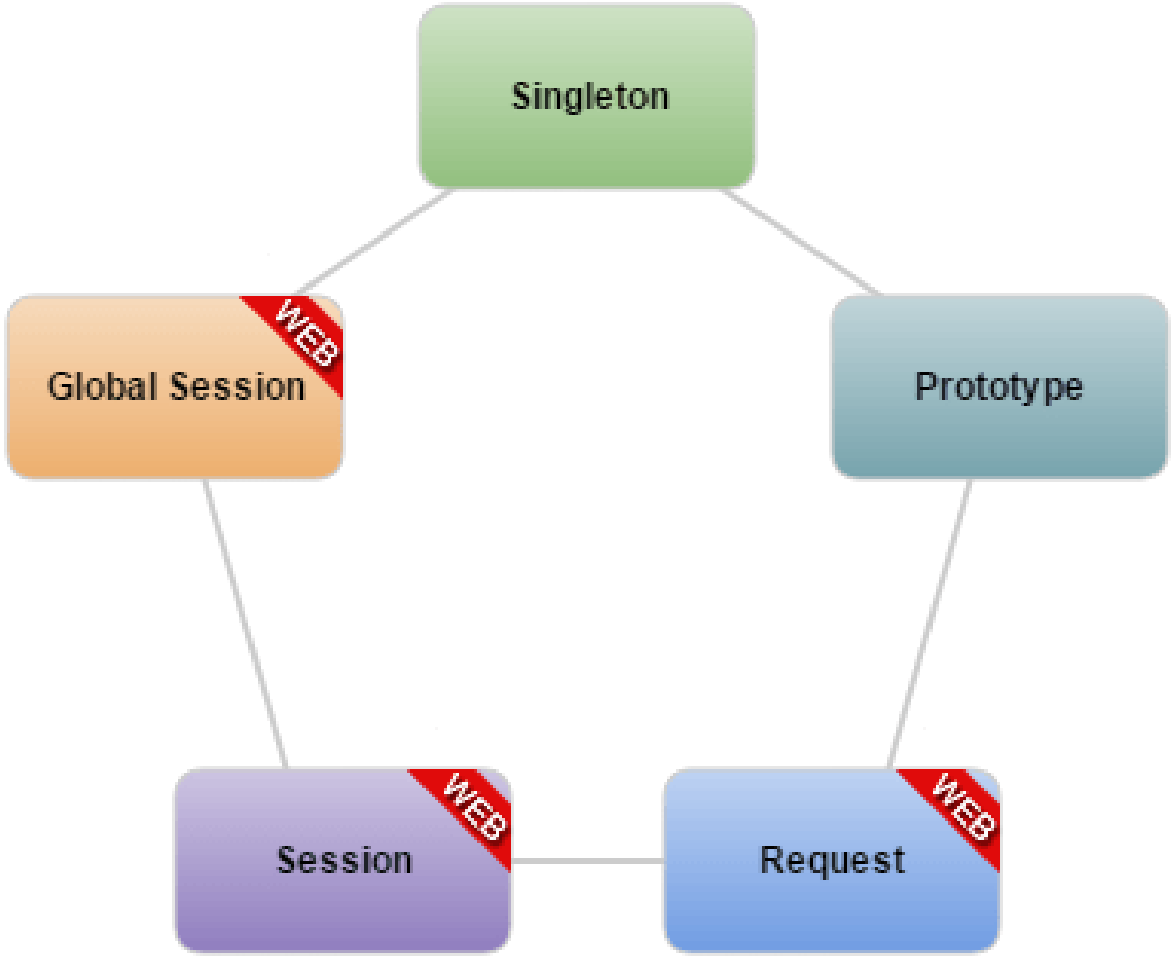


Bean life cycle in Java Spring

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



Bean Scopes


Singleton
Global Session
WEB
Prototype
Session
WEB
Request
WEB

Singleton Scope

- When we define a bean with the singleton scope, the container creates a single instance of that bean; all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

```
@Bean
@Scope("singleton")
public Person personSingleton() {
    return new Person();
}
```

Prototype Scope

- A bean with the prototype scope will return a different instance every time it is requested from the container. It is defined by setting the value prototype to the @Scope annotation in the bean definition:

```
@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

Request Scope

- The request scope creates a bean instance for a single HTTP request, while the session scope creates a bean instance for an HTTP Session.

```
@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}
```

Session Scope

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator sessionScopedBean() {
    return new HelloMessageGenerator();
}
```

Application Scope

- The application scope creates the bean instance for the lifecycle of a ServletContext.
- This is similar to the singleton scope, but there is a very important difference with regards to the scope of the bean.
- When beans are application scoped, the same instance of the bean is shared across multiple servlet-based applications running in the same ServletContext, while singleton scoped beans are scoped to a single application context only.

```
@Bean
@Scope(
  value = WebApplicationContext.SCOPE_APPLICATION, proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator applicationScopedBean() {
    return new HelloMessageGenerator();
}
```

@PostConstruct

- Spring calls the methods annotated with *@PostConstruct* only once, just after the initialization of bean properties. Keep in mind that these methods will run even if there's nothing to initialize.

```java
@Component
public class DbInit {

    @Autowired
    private UserRepository userRepository;

    @PostConstruct
    private void postConstruct() {
        User admin = new User("admin", "admin password");
        User normalUser = new User("user", "user password");
        userRepository.save(admin, normalUser);
    }
}
```

@PreDestroy

- A method annotated with @PreDestroy runs only once, just before Spring removes our bean from the application context.
- Same as with @PostConstruct, the methods annotated with @PreDestroy can have any access level, but can't be static.

```
@Component
public class UserRepository {

    private DbConnection dbConnection;
    @PreDestroy
    public void preDestroy() {
        dbConnection.close();
    }
}
```

Lets check this example

```
package com.journaldev.java.legacy;

public class EmailService {

	public void sendEmail(String message, String receiver){
		//logic to send email
		System.out.println("Email sent to "+receiver+ " with Message="+message);
	}
}
```

```
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = new EmailService();

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

```
package com.journaldev.java.legacy;

public class MyLegacyTest {

	public static void main(String[] args) {
		MyApplication app = new MyApplication();
		app.processMessages("Hi Pankaj", "pankaj@abc.com");
	}

}
```

Java Dependency Injection

1- Service Components

```
package com.journaldev.java.dependencyinjection.service;

public interface MessageService {

	void sendMessage(String msg, String rec);
}
```

```
package com.journaldev.java.dependencyinjection.service;

public class EmailServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send email
		System.out.println("Email sent to "+rec+ " with Message="+msg);
	}

}
```

```
package com.journaldev.java.dependencyinjection.service;

public class SMSServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send SMS
		System.out.println("SMS sent to "+rec+ " with Message="+msg);
	}

}
```

Service Consumer

```
package com.journaldev.java.dependencyinjection.consumer;

public interface Consumer {

	void processMessages(String msg, String rec);
}
```

My consumer class implementation is like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(MessageService svc){
		this.service=svc;
	}

	@Override
	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.service.sendMessage(msg, rec);
	}

}
```

Injectors Classes

```java
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

public interface MessageServiceInjector {

	public Consumer getConsumer();
}
```

```java
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new EmailServiceImpl());
	}

}
```

```java
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.SMSServiceImpl;

public class SMSServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new SMSServiceImpl());
	}

}
```

Client applications will use the application

```
package com.journaldev.java.dependencyinjection.test;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.injector.EmailServiceInjector;
import com.journaldev.java.dependencyinjection.injector.MessageServiceInjector;
import com.journaldev.java.dependencyinjection.injector.SMSServiceInjector;

public class MyMessageDITest {

	public static void main(String[] args) {
		String msg = "Hi Pankaj";
		String email = "pankaj@abc.com";
		String phone = "4088888888";
		MessageServiceInjector injector = null;
		Consumer app = null;

		//Send email
		injector = new EmailServiceInjector();
		app = injector.getConsumer();
		app.processMessages(msg, email);

		//Send SMS
		injector = new SMSServiceInjector();
		app = injector.getConsumer();
		app.processMessages(msg, phone);
	}

}
```

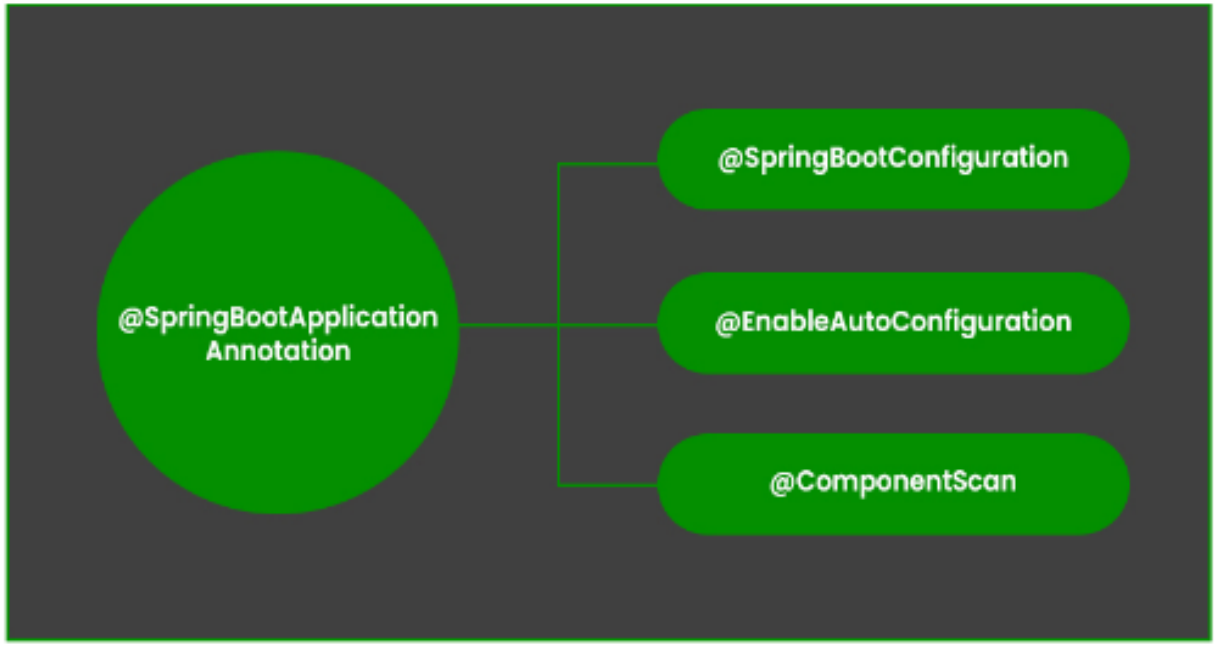
2- Setter method dependency injection

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(){}

	//setter dependency injection
	public void setService(MessageService service) {
		this.service = service;
	}

	@Override
	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.service.sendMessage(msg, rec);
	}

}
```

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		MyDIApplication app = new MyDIApplication();
		app.setService(new EmailServiceImpl());
		return app;
	}

}
```

Spring Boot Annotations

- **@EnableAutoConfiguration**: It auto-configures the bean that is present in the classpath and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.
- **@SpringBootApplication**: It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan**, and @Configuration.



Spring Boot introduces the @SpringBootApplication annotation

- This single annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan.

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

@Component *@Service*

- **We can use @Component across the application to mark the beans as Spring's managed components.** Spring will only pick up and register beans with *@Component*, and doesn't look for *@Service* and *@Repository* in general.
- They are registered in *ApplicationContext* because they are annotated with *@Component*:

```
@Component
public @interface Service {
}
```

- **We mark beans with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation.**

@Repository

- **@Repository's job is to catch persistence-specific exceptions and re-throw them as one of Spring's unified unchecked exceptions.**

- For this, Spring provides PersistenceExceptionTranslationPostProcessor, which we are required to add in our application context (already included if we're using Spring Boot):

@Service Annotation	@Repository Annotation	@Controller Annotation
@Service annotation is used with classes that provide some business functionalities.	@Repository Annotation is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects.	@Controller annotation indicates that a particular class serves the role of a controller.
@Service Annotation is a specialization of @Component Annotation.	@Repository Annotation is also a specialization of @Component Annotation.	@Controller annotation is also a specialization of @Component annotation.
It is used to mark the class as a service provider.	It is used to mark the interface as DAO (Data Access Object) provider.	It's used to mark a class as a web request handler.
It is a stereotype for the service layer.	It is also a stereotype for the DAO layer.	It is also a stereotype for the presentation layer (spring-MVC).
Switch can be possible. But it is not recommended.	Switch can be possible. But it is not recommended.	We cannot switch this annotation with any other like @Service or @Repository.
It is a Stereotype Annotation.	It is also a Stereotype Annotation.	It is also a Stereotype Annotation.

Spring Dependency Injection

Autowiring in Spring

- Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and string values. It works with reference only.

Autowiring Modes

Mode	Description
byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

First, let's define a *fooFormatter* bean:

```
@Component("fooFormatter")
public class FooFormatter {
    public String format() {
        return "foo";
    }
}
```

Then, we'll inject this bean into the *FooService* bean using *@Autowired* on the field definition:

```
@Component
public class FooService {
    @Autowired
    private FooFormatter fooFormatter;
}
```

```java
@Component("fooFormatter")
public class FooFormatter implements Formatter {
    public String format() {
        return "foo";
    }
}
```

```java
@Component("barFormatter")
public class BarFormatter implements Formatter {
    public String format() {
        return "bar";
    }
}
```

Now let's try to inject a *Formatter* bean into the *FooService* class:

```java
public class FooService {
    @Autowired
    private Formatter formatter;
}
```

In our example, there are two concrete implementations of Formatter available for the Spring container. As a result, Spring will throw a **NoUniqueBeanDefinitionException** exception when constructing the FooService:

```
public class FooService {
    @Autowired
    @Qualifier("fooFormatter")
    private Formatter formatter;
}
```

```java
public interface Manager {
    String getManagerName();
}
```

We have a *Manager* interface and two subclass beans, *DepartmentManager*:

```java
@Component
public class DepartmentManager implements Manager {
    @Override
    public String getManagerName() {
        return "Department manager";
    }
}
```

And the *GeneralManager* bean:

```java
@Component
@Primary
public class GeneralManager implements Manager {
    @Override
    public String getManagerName() {
        return "General manager";
    }
}
```

```java
@Service
public class ManagerService {

    @Autowired
    private Manager manager;

    public Manager getManager() {
        return manager;
    }
}
```

Spring AOP

Aspect Oriented Programming Core Concepts

- **Aspect**: An aspect is a class that implements enterprise application concerns that cut across multiple classes, such as transaction management. Aspects can be a normal class configured through Spring XML configuration or we can use Spring AspectJ integration to define a class as Aspect using @**Aspect** annotation.
- **Join Point**: A join point is a specific point in the application such as method execution, exception handling, changing object variable values, etc. In Spring AOP a join point is always the execution of a method.
- **Advice:** Advices are actions taken for a particular join point. In terms of programming, they are methods that get executed when a certain join point with matching pointcut is reached in the application. You can think of Advices as Struts2 interceptors or Servlet Filters.
- **Pointcut:** Pointcut is expressions that are matched with join points to determine whether advice needs to be executed or not. Pointcut uses different kinds of expressions that are matched with the join points and Spring framework uses the AspectJ pointcut expression language.
- **Target Object:** They are the object on which advices are applied. Spring AOP is implemented using runtime proxies so this object is always a proxied object. What is means is that a subclass is created at runtime where the target method is overridden and advice are included based on their configuration.

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class EmployeeAspect {

	@Before("execution(public String getName())")
	public void getNameAdvice(){
		System.out.println("Executing Advice on getName()");
	}

	@Before("execution(* com.journaldev.spring.service.*.get*())")
	public void getAllAdvice(){
		System.out.println("Service method getter called");
	}
}
```

```java
package com.journaldev.spring.aspect;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class EmployeeAspectPointcut {

	@Before("getNamePointcut()")
	public void loggingAdvice(){
		System.out.println("Executing loggingAdvice on getName()");
	}

	@Before("getNamePointcut()")
	public void secondAdvice(){
		System.out.println("Executing secondAdvice on getName()");
	}

	@Pointcut("execution(public String getName())")
	public void getNamePointcut(){}

	@Before("allMethodsPointcut()")
	public void allServiceMethodsAdvice(){
		System.out.println("Before executing service method");
	}

	//Pointcut to execute on all the methods of classes in a package
	@Pointcut("within(com.journaldev.spring.service.*)")
	public void allMethodsPointcut(){}

}
```

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class EmployeeAfterAspect {

	@After("args(name)")
	public void logStringArguments(String name){
		System.out.println("Running After Advice. String argument passed="+name);
	}

	@AfterThrowing("within(com.journaldev.spring.model.Employee)")
	public void logExceptions(JoinPoint joinPoint){
		System.out.println("Exception thrown in Employee Method="+joinPoint.toString());
	}

	@AfterReturning(pointcut="execution(* getName())", returning="returnString")
	public void getNameReturningAdvice(String returnString){
		System.out.println("getNameReturningAdvice executed. Returned String="+returnString);
	}

}
```

```
package com.journaldev.spring.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class EmployeeAroundAspect {

	@Around("execution(* com.journaldev.spring.model.Employee.getName())")
	public Object employeeAroundAdvice(ProceedingJoinPoint proceedingJoinPoint){
		System.out.println("Before invoking getName() method");
		Object value = null;
		try {
			value = proceedingJoinPoint.proceed();
		} catch (Throwable e) {
			e.printStackTrace();
		}
		System.out.println("After invoking getName() method. Return value="+value);
		return value;
	}
}
```

Spring Annotations

- Spring **@Bean** Spring @Bean Annotation 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.
- Spring **@Service** Spring @Service annotation is a specialization of @Component annotation. Spring Service annotation can be applied only to classes. It is used to mark the class as a service provider.
- Spring **@Component** Spring Component annotation is used to denote a class as Component. It means that the Spring framework will autodetect these classes for dependency injection when annotation-based configuration and classpath scanning is used.
- Spring **@RestController** Spring RestController annotation is a convenience annotation that is itself annotated with @Controller and **@ResponseBody**. This annotation is applied to a class to mark it as a request handler.
- Spring **@Controller** Spring Controller annotation is a specialization of @Component annotation. Spring Controller annotation is typically used in combination with annotated handler methods based on the RequestMapping annotation.
- Spring **@Repository** Spring @Repository annotation is used to indicate that the class provides the mechanism for storage, retrieval, search, update and delete operation on objects.
- Spring **@Configuration** Spring @Configuration annotation is part of the spring core framework. Spring Configuration annotation indicates that the class has @Bean definition methods. So Spring container can process the class and generate Spring Beans to be used in the application.
- Spring **@PostConstruct** and **@PreDestroy** When we annotate a method in Spring Bean with @PostConstruct annotation, it gets executed after the spring bean is initialized. When we annotate a Spring Bean method with PreDestroy annotation, it gets called when bean instance is getting removed from the context.

Fawry

THANK YOU

www.Fawry.com

Download APP