**SPRING FRAMEWORK ADVANCED FEATURES**

In Spring Boot, lazy initialization refers to a technique where beans are not created until they are actually needed. By default, Spring Boot uses eager initialization, where all beans are created at application startup. Lazy initialization delays the creation of beans until they are first accessed.

How Lazy Initialization Works:

1. Definition: Beans marked for lazy initialization will only be instantiated when they are first requested, not at application startup.

Exploring Lazy Initialization of Spring Beans

\* Default initialization for Spring Beans: Eager

\* Eager initialization is recommended:

■ Errors in the configuration are discovered immediately at application startup

\* However, you can configure beans to be lazily initialized using Lazy annotation:

■ NOT recommended (AND) Not frequently used

\* Lazy annotation:

■ Can be used almost everywhere @Component and @Bean are used

■ Lazy-resolution proxy will be injected instead of actual dependency

■ Can be used on Configuration (@Configuration) class:

\* All @Bean methods within the @Configuration will be lazily initialized.

**package** com.in28minutes.learn\_spring\_framework.example.d1;

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

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

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

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

**import** org.springframework.stereotype.Component;

@Component

**class** ClassA

{

}

@Component

@Lazy

**class** ClassB

{

**private** ClassA classA;

**public** **void** doSomething() {

// **TODO** Auto-generated method stub

System.***out***.println("Do Something");

}

**public** ClassB(ClassA classA)

{

System.***out***.println("Some initilization logic");

**this**.classA = classA;

}

}

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.d1")

**public** **class** LazyInitializationLauncherApplication {

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(LazyInitializationLauncherApplication.**class**);

//Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

System.***out***.println("Initilization of contedxt is completed");

context.getBean(ClassB.**class**).doSomething();

}

}

Comparing Lazy Initialization vs Eager Initialization

**Heading Lazy Initialization Eager Initialization**

Initialization time Bean initialized when it is Bean initialized at startup of the

first made use of in the application application

Default NOT Default Default

Code Snippet @Lazy OR @Lazy(value=true) @Lazy(value=false) OR

(Absence of @Lazy)

What happens if there are Errors will result in runtime exception Errors will prevent application

errors in initializing? from starting up

Usage Rarely used Very frequently used

Memory Consumption Less (until bean is initialized) All beans are initialized at

startup

Recommended Scenario Beans very rarely used in your app Most of your beans

**BEANS SCOPE LAUNCHER APPLICATION**

Spring Bean Scopes

\* Spring Beans are defined to be used in a specific scope:

■ Singleton - One object instance per Spring IoC container

spring

■ Prototype

Possibly many object instances per Spring loC container

■ Scopes applicable ONLY for web-aware Spring ApplicationContext

\* Request - One object instance per single HTTP request

\* Session - One object instance per user HTTP Session

\* Application - One object instance per web application runtime

\* Websocket - One object instance per WebSocket instance

\* Java Singleton (GOF) vs Spring Singleton

■ Spring Singleton: One object instance per Spring loC container

■ Java Singleton (GOF): One object instance per JVM

--

**package** com.in28minutes.learn\_spring\_framework.example.e1;

**import** java.util.Arrays;

**import** org.springframework.beans.factory.config.ConfigurableBeanFactory;

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

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

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

**import** org.springframework.context.annotation.Scope;

**import** org.springframework.stereotype.Component;

@Component

//@Scope(value=ConfigurableBeanFactory.SCOPE\_SINGLETON)

**class** NormalClass

{

}

@Scope(value= ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***)

@Component

**class** PrototypeClass

{

}

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.e1")

**public** **class** BeanScopeLauncherApplication {

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(BeanScopeLauncherApplication.**class**);

System.***out***.println(context.getBean(NormalClass.**class**));

System.***out***.println(context.getBean(NormalClass.**class**));

//Arrays.stream(context.getBeanDefinitionNames()).forEach(System.out::println);

System.***out***.println(context.getBean(PrototypeClass.**class**));

System.***out***.println(context.getBean(PrototypeClass.**class**));

System.***out***.println(context.getBean(PrototypeClass.**class**));

}

}

**Heading Prototype Singleton**

Instances Possibly Many per Spring IOC Container One per Spring IOC Container

Beans New bean instance created every time the bean Same bean instance reused

is referred to

Default NOT Default Default

Code Snippet @Scope(value= @Scope(value=

ConfigurableBeanFactory.SCOPE\_PROTOTYPE) ConfigurableBeanFactory.SCOPE

SINGLETON or Default

Usage Rarely used Very frequently used

Recommended Stateful beans Stateless beans

In Spring Boot, @PreDestroy and @PostConstruct are annotations used to manage the lifecycle of beans. These annotations are part of the Java EE (Jakarta EE) specifications and are commonly used in Spring to execute custom initialization and cleanup logic when a bean is created and destroyed.

**@PostConstruct**

* **Definition**: The @PostConstruct annotation is used on a method that needs to be executed after the bean's dependencies have been injected and the bean has been fully initialized by the Spring container.
* **Purpose**: It is typically used for initialization logic that needs to run after the bean's properties have been set, but before the bean is put into service.
* **Usage**: Annotate a method with @PostConstruct to ensure that it runs after the bean's construction and dependency injection are complete.

**@PreDestroy**

* **Definition**: The @PreDestroy annotation is used on a method that needs to be executed just before the bean is removed from the Spring container (i.e., when the application context is closing).
* **Purpose**: It is typically used for cleanup logic, such as releasing resources, closing connections, or performing other necessary tasks before the bean is destroyed.
* **Usage**: Annotate a method with @PreDestroy to ensure that it runs before the bean is destroyed.

**Key Points:**

* **@PostConstruct** is called after the bean's initialization and dependency injection are complete.
* **@PreDestroy** is called just before the bean is destroyed, typically when the application context is being shut down.
* Both annotations are useful for managing the lifecycle of beans, especially when dealing with resource management, initialization logic, or cleanup tasks.

These annotations provide a simple and effective way to manage the initialization and destruction phases of Spring-managed beans.

**package** com.in28minutes.learn\_spring\_framework.example.f1;

**import** java.util.Arrays;

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

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

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

**import** org.springframework.stereotype.Component;

**import** jakarta.annotation.PostConstruct;

**import** jakarta.annotation.PreDestroy;

@Component

**class** SomeClass

{

**private** SomeDependency someDependency;

**public** SomeClass(SomeDependency someDependency) {

**super**();

**this**.someDependency = someDependency;

System.***out***.println("All dependencies are ready");

}

@PostConstruct

**public** **void** initialize()

{

someDependency.getReady();

}

@PreDestroy

**public** **void** cleanUp()

{

System.***out***.println("CleanUP");

}

}

@Component

**class** SomeDependency

{

// @Autowired

**public** **void** getReady() {

// **TODO** Auto-generated method stub

System.***out***.println("some logic using dependency");

}

}

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.f1")

**public** **class** PrePostAnnotationsContextLauncherApplication {

**public** **static** **void** main(String[] args) {

**var** context = **new** AnnotationConfigApplicationContext(PrePostAnnotationsContextLauncherApplication.**class**);

Arrays.*stream*(context.getBeanDefinitionNames()).forEach(System.***out***::println);

context.close();

}

}

Evolution of Jakarta EE: vs J2EE vs Java EE

128 Miantes

\* Enterprise capabilities were initially built into JDK

12EE (12,13,14)

Jakarta Et

\* With time, they were separated out:

■ J2EE - Java 2 Platform Enterprise Edition

Java EE (3108)

■ Java EE - Java Platform Enterprise Edition (Rebranding)

■ Jakarta EE (Oracle gave Java EE rights to the Foundation)

\* Important Specifications:

\* Jakarta Server Pages (JSP)

\* Jakarta Standard Tag Library (JSTL)

\* Jakarta Enterprise Beans (EJB)

\* Jakarta RESTful Web Services (JAX-RS)

\* Jakarta Bean Validation

\* Jakarta Contexts and Dependency Injection (CDI)

\* Jakarta Persistence (JPA)

\* Supported by Spring 6 and Spring Boot 3

\* That's why we use jakarta. packages (instead of javax.)

[7:54 pm, 14/08/2024] ~: Evolution of Jakarta EE: vs J2EE vs Java EE

128 Miantes

\* Enterprise capabilities were initially built into JDK

12EE (12,13,14)

Jakarta Et

\* With time, they were separated out:

■ J2EE - Java 2 Platform Enterprise Edition

Java EE (3108)

■ Java EE - Java Platform Enterprise Edition (Rebranding)

■ Jakarta EE (Oracle gave Java EE rights to the Foundation)

\* Important Specifications:

\* Jakarta Server Pages (JSP)

\* Jakarta Standard Tag Library (JSTL)

\* Jakarta Enterprise Beans (EJB)

\* Jakarta RESTful Web Services (JAX-RS)

\* Jakarta Bean Validation

\* Jakarta Contexts and Dependency Injection (CDI)

\* Jakarta Persistence (JPA)

\* Supported by Spring 6 and Spring Boot 3

\* That's why we use jakarta. packages (instead of javax.)

[8:04 pm, 14/08/2024] ~: Jakarta Contexts & Dependency Injection (CDI)

\* Spring Framework V1 was released in 2004

\* CDI specification introduced into Java EE 6 platform in December 2009

spring

\* Now called Jakarta Contexts and Dependency Injection (CDI)

\* CDI is a specification (interface)

■ Spring Framework implements CDI

\* Important Inject API Annotations:

■ Inject (~Autowired in Spring)

■ Named (~Component in Spring)

■ Qualifier

■ Scope

■ Singleton

**EXPLORING JAVA SPRING CONFIGURATIONS**

**package** com.in28minutes.learn\_spring\_framework.example.h1;

**import** java.util.Arrays;

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

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

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

@Configuration

@ComponentScan("com.in28minutes.learn\_spring\_framework.example.h1")

**public** **class** XmlConfigurationContextLauncherApplication {

**public** **static** **void** main(String[] args) {

**var** context = **new** ClassPathXmlApplicationContext("XmlConfigurationContextLauncherApplication.xml");

Arrays.*stream*(context.getBeanDefinitionNames()).forEach(System.***out***::println);

System.***out***.println(context.getBean("name"));

System.***out***.println(context.getBean("age"));

}

}

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

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

xmlns:xsi=*"http://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 definitions here -->

<bean id=*"name"* class=*"java.lang.String"*>

<constructor-arg value=*"Sivaram"*/>

</bean>

<bean id=*"age"* class=*"java.lang.Integer"*>

<constructor-arg value=*"22"*/>

</bean>

<context:component-scan base-package=*"com.in28minutes.learn\_spring\_framework.game"* />

</beans>

Spring Stereotype Annotations - @Component & more..

\* @Component - Generic annotation applicable for any class

■ Base for all Spring Stereotype Annotations

■ Specializations of @Component:

\* @Service - Indicates that an annotated class has business logic

\* @Controller - Indicates that an annotated class is a "Controller" (e.g. a web controller)

\* Used to define controllers in your web applications and REST API

\* @Repository - Indicates that an annotated class is used to retrieve and/or manipulate data in a database

Web

Busine

Data

\* What should you use?

■ (MY RECOMMENDATION) Use the most specific annotation possible

■ Why?

\* By using a specific annotation, you are giving more information to the framework about your intentions.

\* You can use AOP at a later point to add additional behavior

\* Example: For @Repository, Spring automatically wires in JDBC Exception translation features

**Annotation**

**@Configuration**

Indicates that a class declares one or more @Bean methods and may be processed by t Spring container to generate bean definitions

**@ComponentScan**

Define specific packages to scan for components. If specific packages are not defined, scanning will occur from the package of the class that declares this annotation

**@Bean**

Indicates that a method produces a bean to be managed by the Spring container

**@Component**

Indicates that an annotated class is a "component"

**@Service**

Specialization of @Component indicating that an annotated class has business logic

**@Controller**

Specialization of @Component indicating that an annotated class is a "Controller" (e.g web controller). Used to define controllers in your web applications and REST API

**@Repository**

Specialization of @Component indicating that an annotated class is used to retrieve and/or manipulate data in a database

**@Primary**

Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single- valued dependency

**@Qualifier**

Used on a field or parameter as a qualifier for candidate beans when autowiring

**@Lazy**

Indicates that a bean has to be lazily initialized. Absence of @Lazy annotation will lead to eager initialization.

**@Scope (value =**

**ConfigurableBeanFactory.SCOPE\_PROTOTYPE)**

Defines a bean to be a prototype - a new instance will be created every time you refer to the bean. Default scope is singleton - one instance per IOC container.

**@PostConstruct**

Identifies the method that will be executed after dependency injection is done to perform any initialization

**PreDestroy**

Identifies the method that will receive the callback notification to signal that the instance is in the process of being removed by the container. Typically used to release resources that it has been holding.

**Dependency Injection**

Identify beans, their dependencies and wire them together (provides IOC - Inversion of Control)

**Constr. injection**

Dependencies are set by creating the Bean using its Constructor

**Setter injection**

Dependencies are set by calling setter methods on your beans

**Field injection**

No setter or constructor. Dependency is injected using reflection.

**IOC Container**

Spring IOC Context that manages Spring beans & their lifecycle

**Bean Factory**

Basic Spring IOC Container

**Application Context**

Advanced Spring IOC Container with enterprise-specific features - Easy to use in web applications with internationalization features and good integration with Spring AOP

**Spring Beans**

Objects managed by Spring

Spring Big Picture - Framework and Modules

\* Spring Framework contains multiple Spring Modules:

■ Fundamental Features: Core (IOC Container, Dependency Injection, Auto Wiring, ..)

■ Web: Spring MVC etc (Web applications, REST API)

■ Web Reactive: Spring WebFlux etc

■ Data Access: JDBC, JPA etc

■ Integration: JMS etc

■ Testing: Mock Objects, Spring MVC Test etc

Spring

Spring

Spring

Spring Modules

A

Spring

\* No Dumb Question: Why is Spring Framework divided into Modules?

■ Each application can choose modules they want to make use of

■ They do not need to make use of everything in Spring framework!

Spring Big Picture - Spring Projects

\* Application architectures evolve continuously

■ Web > REST API > Microservices > Cloud >

\* Spring evolves through Spring Projects:

■ First Project: Spring Framework

■ Spring Security: Secure your web application or REST API or microservice

Spring Projects

■ Spring Data: Integrate the same way with different types of databases: NoSQL and Relational

■ Spring Integration: Address challenges with integration with other applications

■ Spring Boot: Popular framework to build microservices

■ Spring Cloud: Build cloud native applications

\* Hierarchy: Spring Projects > Spring Framework > Spring Modules

\* Why is Spring Eco system popular?

Spring Framew

■ Loose Coupling: Spring manages creation and wiring of beans and dependencies

Spring Bact

\* Makes it easy to build loosely coupled applications

Spring Doud

\* Make writing unit tests easy! (Spring Unit Testing)

■ Reduced Boilerplate Code: Focus on Business Logic

Spring Data

Spring Security

Spring Projects

\* Example: No need for exception handling in each method!

\* All Checked Exceptions are converted to Runtime or Unchecked Exceptions

■ Architectural Flexibility: Spring Modules and Projects

\* You can pick and choose which ones to use (You DON'T need to use all of them!)

■ Evolution with Time: Microservices and Cloud

\* Spring Boot, Spring Cloud etc!