Spring & Spring Boot

Understand Spring boot Working

Made By

NAZ MEMON

01

Java Bean Class (Java) & Spring

Bean and its Configuration (Spring)

02

IOC Container (Spring)

03

Annotations (Spring and Spring

Boot)

04

Spring Application Starter

Guidelines (Spring Boot)

Java Bean

A JavaBean is a Java class that should follow the following conventions:

It should have a no-arg constructor.

It should be Serializable.

It should provide methods to set and get the values of the properties, known as getter and setter

methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects

into one object so that we can access this object from multiple places. Moreover, it provides easy

maintenance.

Note: There are two ways to provide values to the object. One way is by constructor and second is by

setter method.

JavaBean Class

JavaBean Class Usage Example

//Employee.java

package mypack;

public class Employee implements

java.io.Serializable{

private int id;

private String name;

public Employee(){}

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

public int getId(){return id;}

public void setName(String

name){this.name=name;}

public String getName(){return name;}

}

How to access the JavaBean class?

To access the JavaBean class, we should use

getter and setter methods.

package mypack;

public class Test{

public static void main(String args[]){

Employee e=new Employee();//object is created

e.setName("Arjun");//setting value to the object

System.out.println(e.getName());

}}

Inversion of Control

(IOC) Container

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a

program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a

framework to take control of the flow of a program and make calls to our custom code. To enable this,

frameworks use abstractions with additional behavior built in. If we want to add our own behavior, we need to

extend the classes of the framework or plugin our own classes.

The advantages of this architecture are:

decoupling the execution of a task from its implementation

making it easier to switch between different implementations

greater modularity of a program

greater ease in testing a program by isolating a component or mocking its dependencies, and allowing

components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service

Locator pattern, Factory pattern, and Dependency Injection (DI) in SPRING

IoC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container

gets informations from the XML file and works accordingly. The main tasks performed by IoC

container are:

to instantiate the application class

to configure the object

to assemble the dependencies between the objects

There are two types of IoC containers. They are:

BeanFactory

ApplicationContext

IoC Container

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.BeanFactory and the org.springframework.context.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 such as simple

integration with Spring's AOP, message resource handling , event propagation, application layer specific context (e.g. WebApplicationContext) for

web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of

XmlBeanFactory class as given below:

Resource resource=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(resource);

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the

ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

ApplicationContext context =

new ClassPathXmlApplicationContext("applicationContext.xml");

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of

ApplicationContext

IoC Container

SPRING BOOT BEAN: Introduction

Spring Bean is the key concept or backbone of the Spring Framework. Spring Bean is the object

whose life-cycle managed by the Spring IoC. It is important to understand it before we work

with the Spring Framework. In simple words Spring Bean is the core building block for any

Spring application. My goal is to give a clear answer to a basic question “What is a Spring

Bean?“. Let’s try to get an answer to this simple question.

1. Spring Bean Definition

This is the standard definition of Spring Bean from Spring documentation:

n 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. Otherwise, a bean is one of many objects in your application.

Beans, and the dependencies among them, are reflected in the configuration metadata used by a

container.

2. Spring IOC Containers

The most important concept while working on the Spring beans is the Ioc Container. Spring IoC

container is the central management system of the Spring Framework. It is responsible to create,

configure and manage the life cycle of these objects. The container gets its instructions on what

objects to instantiate, configure, and assemble by reading configuration metadata provided in the

form of either XML configurations or annotations.It allows you to express the objects that compose

your application and the rich interdependencies between such objects. Let’s look at the high level

workflow of the Spring Ioc container.

The creation of beans is pretty important to the use of Spring (probably more like very important)

allowing us to have Java classes that live within the application context that can be used within

other beans/classes without constantly creating new instances every time we need to use one of their

methods.

let’s take the following example:

Customer for our online shop.

Orders placed by Customers on our online shop.

public class Customer{

private String firstName;

private String lastName;

private List orders;

//getter and setter}

public class Order{

private String code;

private double total;

private double tax;

private Product product;

private Customer owner;

//getter and setter}

If we are working on an application with no support for the Spring Framework, we might do the following to create an instance of both Customer and

Order class:

Customer customer = new Customer(); // pass any constructor arguments

customer.setFirstName("FirstName");

// fill other data in customer mmodel

// Create order instance by getting all orders for a customer

List orders = get\_All\_Orders\_For\_Customer(Customer);

customer.setOrders(orders);

This code will work fine with no issue, however there are few points which

can make this code complex and error prone. Developer need to create

each instance and fill out dependencies (orders in our case).

For a large application with a hundred of classes, this is difficult and error

prone also if we need to change in any class, we need to make sure that all

dependencies are working correctly. We also need to take care of the

sequence

To handle all these cross dependencies in a large application, Spring Ioc

container prove its power and flexibility. As stated above, we need to pass

the class information and its dependencies to Spring container using as

metadata and Spring Ioc container will manage the bean life cycle for us.

It will also take care of any dependencies while instating the bean (if bean B

needs bean A, it will first create bean A)

3. Bean Configuration

Spring provide the following options for bean configuration:

Bean configuration using Java configurations.

XML declaration

@Component

public class Customer{

private String firstName;

private String lastName;

private List orders;

//getter and setter

}

@Component

public class Order{

private String code;

private double total;

private double tax;

private Product product;

private Customer owner;

//getter and setter

}

Here is our configuration class:

@Configuration

@ComponentScan(basePackages = { "com.javadevjournal"})

public class AppConfig{

@Bean

public Customer getCustomer() {

return new Customer();

}

}

4. Bean Dependencies

One of the main feature of Spring managed beans are the dependency management. When

Spring creates a bean which define dependency to another bean, the Spring Ioc container will

create that bean first.It will make sure that all dependencies are in place before it create the

bean.

This dependency graph

creation and making sure

that the beans are getting

created in right order is

one of the most powerful

feature of the Spring Ioc

container and it also take

away the complexity from

our code. Below, I

summarize the bean life

cycle under Spring IoC.

4. Bean LIFE CYCLE

Spring Boot

Annotations

Spring Boot Annotations is a form of metadata that

provides data about a program. In other words,

annotations are used to

provide supplemental information about a program. It is

not a part of the application that we develop. It does not

have a direct effect on the operation of the code they

annotate. It does not change the action of the compiled

program.

Core Spring Framework Annotations

@Required: It applies to the bean setter method. It

indicates that the annotated bean must be populated at

configuration time with the required property, else it

throws an exception BeanInitilizationException.

Spring Boot Annotations

Example

public class Machine

{ private Integer cost;

@Required

public void setCost(Integer

cost)

{

this.cost = cost;

}

public Integer getCost()

{

return cost;

}

}

@Autowired: Spring provides annotation-based

auto-wiring by providing @Autowired annotation. It is

used to autowire spring bean on setter methods,

instance variable, and constructor. When we use

@Autowired annotation, the spring container auto-wires

the bean by matching data-type.

Example

@Component

public class Customer

{

private Person person;

@Autowired

public Customer(Person person)

{

this.person=person;

}

}

@Configuration: It is a class-level annotation. The

class annotated with @Configuration used by Spring

Containers as a source of bean definitions.

Example

@Configuration

public class Vehicle

{

@Bean

Vehicle engine()

{

return new Vehicle();

}

}

@ComponentScan: It is used when we want to

scan a package for beans. It is used with the

annotation @Configuration. We can also specify the

base packages to scan for Spring Components.

Example

@ComponentScan(basePackages = "com.nazme

mon")

@Configuration

public class ScanComponent

{

// ...

}

@Bean: It is a method-level annotation. It is an

alternative of XML <bean> tag. It tells the

method to produce a bean to be managed by

Spring Container.

Example

@Bean

public BeanExample beanExample()

{

return new BeanExample ();

}

@Service: It is also used at class level. It tells

the Spring that class contains the business

logic.

Example

package com.nazmemon;

@Service

public class TestService

{

public void service1()

{

//business code

}

}

@Repository: It is a class-level annotation. The

repository is a DAOs (Data Access Object) that

access the database directly. The repository does all

the operations related to the database.

package com.nazmemon;

@Repository

public class TestRepository

{

public void delete()

{

//persistence code

}

}

Spring Framework Stereotype Annotations

Spring Framework Stereotype 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.

@Qualifier @Lazy

Spring MVC and REST Annotations

@RequestMapping: It is used to map the web requests. It has many optional elements like consumes,

header, method, name, params, path, produces, and value. We use it with the class as well as the

method.

Example:

@GetMapping: It maps the HTTP GET requests on the specific handler method. It is used to create a web

service endpoint that fetches It is used instead of using: @RequestMapping(method =

RequestMethod.GET)

@PostMapping: It maps the HTTP POST requests on the specific handler method. It is used to create a

web service endpoint that creates It is used instead of using: @RequestMapping(method =

RequestMethod.POST)

Spring MVC and REST Annotations

@PutMapping: It maps the HTTP PUT requests on the specific handler method. It is used to create a web service endpoint

that creates or updates It is used instead of using: @RequestMapping(method = RequestMethod.PUT)

@DeleteMapping: It maps the HTTP DELETE requests on the specific handler method. It is used to create a web service endpoint

that deletes a resource. It is used instead of using: @RequestMapping(method = RequestMethod.DELETE)

@PatchMapping: It maps the HTTP PATCH requests on the specific handler method. It is used instead of

using: @RequestMapping(method = RequestMethod.PATCH)

@RequestBody: It is used to bind HTTP request with an object in a method parameter. Internally it uses HTTP

MessageConverters to convert the body of the request. When we annotate a method parameter with @RequestBody, the Spring

framework binds the incoming HTTP request body to that parameter.

@ResponseBody: It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an

object into JSON and XML format.

@PathVariable: It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains

a path variable. We can define multiple @PathVariable in a method.

@RequestParam: It is used to extract the query parameters form the URL. It is also known as a query parameter. It is most suitable

for web applications. It can specify default values if the query parameter is not present in the URL.

@RequestHeader: It is used to get the details about the HTTP request headers. We use this annotation as a method parameter.

The optional elements of the annotation are name, required, value, defaultValue. For each detail in the header, we should specify

separate annotations. We can use it multiple time in a method

@RestController: It can be considered as a combination of @Controller and @ResponseBody annotations. The @RestController

annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with

@ResponseBody.

@RequestAttribute: It binds a method parameter to request attribute. It provides convenient access to the request attributes from a

controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

Application Starter

Interfaces

In order to use ApplicationRunner or CommandLineRunner interfaces, one needs to create a

Spring bean and implement either ApplicationRunner or CommandLineRunner interfaces, both

perform similarly. Once complete, your Spring application will detect your bean.

In addition, you can create multiple ApplicationRunner or CommandLineRunner beans, and

control the ordering by implementing either

org.springframework.core.Ordered interface

org.springframework.core.annotation.Order annotation.

use case:

One might wish to log some command line arguments

You could provide some instructions to the user on termination of this application.

consider:

@Component

public class MyBean implements CommandLineRunner {

@Override

public void run(String...args) throws Exception {

logger.info("App started with arguments: " + Arrays.toString(args));

}

}

These runners are used to run the logic on application startup, for example

spring boot has ApplicationRunner(Functional Interface) with run method.

ApplicationRunner run() will get executed, just after applicationcontext is

created and before spring boot application startup.

ApplicationRunner takes ApplicationArgument which has convenient methods

like getOptionNames(), getOptionValues() and getSourceArgs().

And CommandLineRunner is also a Functional Interface with run method

CommandLineRunner run() will get executed, just after applicationcontext is

created and before spring boot application starts up.

It accepts the argument, which are passed at time of server startup.

Both of them provides the same functionality and the only difference between

CommandLineRunner and ApplicationRunner is CommandLineRunner.run()

accepts String array[] whereas ApplicationRunner.run() accepts

ApplicationArguments as argument. You can find more information with

example at Guide To Running Logic on Startup in Spring

ApplicationRunner and CommandLineRunner

are two interfaces Spring Boot provides to run

any custom code just before application is fully

started.

Spring-batch is a batch processing framework.

This uses CommandLineRunner to register and

start batch jobs at application startup.

You can also use this interface to load some

master data into cache/perform health checks.

The use-case varies from application to

application.

1. Overview: we’ll focus on how to run logic at the startup of a Spring application.

2. Running Logic on Startup

Running logic during/after Spring application’s startup is a common scenario. But it’s also one that

causes multiple problems.

In order to benefit from Inverse of Control, we need to renounce partial control over the

application’s flow to the container. This is why instantiation, setup logic on startup, etc. need

special attention.We can’t simply include our logic in the beans’ constructors or call methods after

instantiation of any object because we aren’t in control during those processes. Let’s look at a

real-life example:

@Component

public class InvalidInitExampleBean {

@Autowired

private Environment env;

public InvalidInitExampleBean() {

env.getActiveProfiles();

}

}

Here we’re trying to access an

autowired field in the constructor.

When the constructor is called, the

Spring bean is not yet fully

initialized. This is a problem

because calling fields that are not

yet initialized will result in

NullPointerExceptions.

Let’s look at a few ways Spring gives us to manage this situation.

2.1. The @PostConstruct Annotation

We can use Javax’s @PostConstruct annotation for annotating a method that should be run once

immediately after the bean’s initialization. Keep in mind that Spring will run the annotated method

even if there is nothing to inject.

Here’s @PostConstruct in action:

@Component

public class PostConstructExampleBean {

private static final Logger LOG

= Logger.getLogger(PostConstructExampleBean.class);

@Autowired

private Environment environment;

@PostConstruct

public void init() {

LOG.info(Arrays.asList(environment.getDefaultProfiles()));

}

}

We can see that the Environment instance was safely injected and then called in the @PostConstruct

annotated method without throwing a NullPointerException.

2.2. The InitializingBean Interface

The InitializingBean approach works in a similar way. Instead of annotating a method, we need to

implement the InitializingBean interface and the afterPropertiesSet() method.

Here we implement the previous example using the InitializingBean interface:

@Component

public class InitializingBeanExampleBean implements InitializingBean {

private static final Logger LOG

= Logger.getLogger(InitializingBeanExampleBean.class);

@Autowired

private Environment environment;

@Override

public void afterPropertiesSet() throws Exception {

LOG.info(Arrays.asList(environment.getDefaultProfiles()));

}

}

2.3. An ApplicationListener

We can use this approach for running logic after the Spring context has been initialized. So, we

aren’t focusing on any particular bean. We’re instead waiting for all of them to initialize.

In order to do this, we need to create a bean that implements the

ApplicationListener<ContextRefreshedEvent> interface:

@Component

public class StartupApplicationListenerExample implements

ApplicationListener<ContextRefreshedEvent> {

private static final Logger LOG = Logger.getLogger(StartupApplicationListenerExample.class);

public static int counter;

@Override public void onApplicationEvent(ContextRefreshedEvent event) {

LOG.info("Increment counter");

counter++;

}

}

We can get the same results by using the newly introduced @EventListener

annotation:

@Component

public class EventListenerExampleBean {

private static final Logger LOGn= Logger.getLogger(EventListenerExampleBean.class);

public static int counter;

@EventListener

public void onApplicationEvent(ContextRefreshedEvent event) {

LOG.info("Increment counter");

counter++;

}

}

We want to make sure to pick an appropriate event for our needs. In this example, we chose the

ContextRefreshedEvent.

2.4. The @Bean initMethod Attribute

We can use the initMethod property to run a method after a bean’s initialization. Here’s what a bean looks like:

public class InitMethodExampleBean {

private static final Logger LOG = Logger.getLogger(InitMethodExampleBean.class);

@Autowired

private Environment environment;

public void init() {

LOG.info(Arrays.asList(environment.getDefaultProfiles())); } }

Notice we haven’t implemented any special interfaces or used any special annotations. Then we can

define the bean using the @Bean annotation:

@Bean(initMethod="init")

public InitMethodExampleBean initMethodExampleBean() {

return new InitMethodExampleBean();

}

And this is how a bean definition looks in an XML config:

<bean id="initMethodExampleBean"

class="com.baeldung.startup.InitMethodExampleBean"

init-method="init">

</bean>

2.5. Constructor Injection

If we’re injecting fields using Constructor Injection, we can simply include our logic in a

constructor:

@Component

public class LogicInConstructorExampleBean {

private static final Logger LOG

= Logger.getLogger(LogicInConstructorExampleBean.class);

private final Environment environment;

@Autowired

public LogicInConstructorExampleBean(Environment environment) {

this.environment = environment;

LOG.info(Arrays.asList(environment.getDefaultProfiles()));

}

}

2.6. Spring Boot CommandLineRunner

Spring Boot provides a CommandLineRunner interface with a callback run() method. This can be

invoked at application startup after the Spring application context is instantiated.

Let’s look at an example:

@Component

public class CommandLineAppStartupRunner implements CommandLineRunner {

private static final Logger LOG =

LoggerFactory.getLogger(CommandLineAppStartupRunner.class);

public static int counter;

@Override

public void run(String...args) throws Exception {

LOG.info("Increment counter");

counter++;

}

}

Note: As mentioned in the documentation, multiple CommandLineRunner beans can be defined

within the same application context and can be ordered using the @Ordered interface or @Order

annotation.

2.7. Spring Boot ApplicationRunner

Similar to CommandLineRunner, Spring Boot also provides an ApplicationRunner interface with a run()

method to be invoked at application startup. However, instead of raw String arguments passed to the callback

method, we have an instance of the ApplicationArguments class.

The ApplicationArguments interface has methods to get argument values that are options and plain argument

values. An argument that is prefixed with – – is an option argument.

Let’s look at an example:

@Component

public class AppStartupRunner implements ApplicationRunner {

private static final Logger LOG =

LoggerFactory.getLogger(AppStartupRunner.class);

public static int counter;

@Override

public void run(ApplicationArguments args) throws Exception {

LOG.info("Application started with option names : {}",

args.getOptionNames());

LOG.info("Increment counter");

counter++; }}

3. Combining Mechanisms

In order to have full control over our beans, we could combine the

above mechanisms together.This is the order of execution:

constructor

@PostConstruct annotated methods

InitializingBean’s afterPropertiesSet() method

initialization method specified as init-method in XML

Let’s create a Spring bean that combines all mechanisms:

CODE DESCRIBED --->

If we try to instantiate this bean, we can see logs that match the

order specified above:

[main] INFO o.b.startup.AllStrategiesExampleBean - Constructor

[main] INFO o.b.startup.AllStrategie sExampleBean -

PostConstruct

[main] INFO o.b.startup.AllStrategiesExampleBean -

InitializingBean

[main] INFO o.b.startup.AllStrategiesExampleBean - init-method

@Component

@Scope(value = "prototype")

public class AllStrategiesExampleBean implements

InitializingBean {

private static final Logger LOG =

Logger.getLogger(AllStrategiesExampleBean.class);

public AllStrategiesExampleBean() {

LOG.info("Constructor");}

@Override

public void afterPropertiesSet() throws Exception {

LOG.info("InitializingBean");}

@PostConstruct

public void postConstruct() {

LOG.info("PostConstruct"); }