SPRING FRAMEWORK

Mentor : Rahul Dixit

TERMINOLOGY

Terminology:-Dependency Injection, IOC, Auto wiring, Auto configuration, Starter Projects.

Application :- Web app, REST API, Full Stack

Variety of other framework, tool and platform integrations: Maven, Spring Data, JPA, Hibernate, Docker and Cloud

GETTING STARTED WITH SPRING FRAMEWORK - WHY?

You can build a variety of applications using Java, Spring and Spring Boot:

- Web
- REST API
- Full Stack
- Microservices

Irrespective of the app you are building:

- Spring framework provides all the core features
  - Understanding Spring helps you learn Spring Boot easily
  - Helps in debugging problems quickly

WHY IS COUPLING IMPORTANT?

Coupling: How much work is involved in changing something?

- **Coupling is important everywhere:**
  - An engine is tightly coupled to a Car
  - A wheel is loosely coupled to a Car
  - You can take a laptop anywhere you go A computer, on the other hand, is a little bit more difficult to move
- **Coupling is even more important in building great software Only thing constant in technology is change**
  - Business requirements change
  - Frameworks change
  - Code changes
- We want Loose Coupling as much as possible
- We want to make functional changes with as less code changes as possible

Let's explore how Java Interfaces and Spring Framework help with Loose Coupling!

GETTING STARTED WITH SPRING FRAMEWORK

Design DrawingApp to draw (Circle, Rectangle, Square etc) in an iterative approach:

Iteration 1: Tightly Coupled Java Code

- Drawing class
  - Draw method
- Game classes: Circle, Rectangle, Square etc

Iteration 2: Loose Coupling - Interfaces

- Drawing class Shape interface
- Shape classes: Circle, Rectangle, Square etc

Iteration 3: Loose Coupling - Spring Level 1

- Spring Beans
- Spring framework will manage objects and wiring

Iteration 4: Loose Coupling - Spring Level 2

- Spring Annotations
- Spring framework will create, manage & auto-wire objects

GETTING STARTED WITH SPRING FRAMEWORK - GOALS

Goal: Understand core features of Spring Framework

Approach: Build a Loose Coupled Hello World Drawing App with Modern Spring Approach

- Get Hands-on with Spring and understand:
- Why Spring?
- Terminology
  - Tight Coupling and Loose Coupling
  - IOC Container → Inversion of Control
  - Application Context
  - Component Scan
  - Dependency Injection
  - Spring Beans
  - Auto Wiring

SPRING QUESTIONS YOU MIGHT BE THINKING ABOUT

Question 1: Spring Container vs Spring Context vs IOC Container vs Application Context

Question 2: Java Bean vs Spring Bean

Question 3: How can I list all beans managed by Spring Framework?

Question 4: What if multiple matching beans are available?

Question 5: Spring is managing objects and performing auto-wiring.

- BUT aren't we writing the code to create objects?
- How do we get Spring to create objects for us?

Question 6: Is Spring really making things easy?

WHAT IS SPRING CONTAINER?

Spring Container: Manages Spring beans & their lifecycle

1: **Bean Factory**: Basic Spring Container

2: **Application Context**: Advanced Spring Container with enterprise-specific features

- Easy to use in web applications
- Easy internationalization
- Easy integration with Spring AOP

Which one to use?: Most enterprise applications use Application Context

- Recommended for web applications, web services - REST API and microservices

EXPLORING SPRING DEPENDENCY INJECTION

Constructor Injection : - Dependencies are set by Creating the bean using its constructor (Default)

Setter Injection :- Dependencies are set by calling setter methods on your beans

Field :- No setter or constructor .Dependency Injected using reflection.

Q Which is better to inject Dependencies

- Spring Team Recommends Constructor based Injection. Because all Initialization Done By Only One Method i.e. Constructor

FIELD INJECTION

```
@Component
class BussinesLogic{
@Autowired
Dependency1 dependency1;
@Autowired
Dependency2 dependency2;
@Override
public String toString() {
return "BussinesLogic [dependency1=" +
dependency1 + ", dependency2=" +
dependency2 + "]";
}
```

```
@Component
class Dependency1{}
@Component
class Dependency2{}
```

SETTER INJECTION

```
@Component
class BussinesLogic{
Dependency1 dependency1;
Dependency2 dependency2;
@Autowired
public void
setDependency1(Dependency1
dependency1) {
System.out.println("Setter of
Dependency 1");
this.dependency1 = dependency1;
}
```

```
@Autowired
public void setDependency2(Dependency2
dependency2) {
System.out.println("Setter of Dependency 2");
this.dependency2 = dependency2;
}
@Override
public String toString() {
return "BussinesLogic [dependency1=" +
dependency1 + ", dependency2=" + dependency2 +
"]";
}
}
```

CONSTRUCTOR INJECTION :- EXECUTED WITH @AUTOWIRED MANAGED BY SPRING

```
@Component
class BussinesLogic{
Dependency1 dependency1;
Dependency2 dependency2;
@Autowired
public BussinesLogic(Dependency1 dependency1, Dependency2 dependency2) {
System.out.println("Constructor Injection");
this.dependency1 = dependency1;
this.dependency2 = dependency2;
}
```

```
@Override
public String toString()
{
return "BussinesLogic [dependency1=" + dependency1 + ", dependency2=" + dependency2 + "]";
}
}
```

EXPLORING AUTO-WIRING IN DEPTH

When a dependency needs to be @Autowired, IOC container looks for matches/candidates (by name and/or type)

1. **If no match is found Result:** Exception is thrown
2. **One match is found Result:** Autowiring is successful
3. **Multiple candidates Result:** Exception is thrown
4. You need to help Spring Framework choose between the candidates
   @Primary & @Qualifier Comes Into a Picture

@PRIMARY VS @QUALIFIER - WHICH ONE TO USE?

@Primary - A bean should be given preference when multiple candidates are qualified

@Qualifier - A specific bean should be auto-wired (name of the bean can be used as qualifier)

1. Just @Autowired: Give me (preferred Bean)
2. @Autowired + @Qualifier

- @Qualifier has higher priority then @Primary

INVERSION OF CONTROL

In Our Initial Program We Explicitly create an Object and Inject It Explicitly. But Here everything is managed by Spring Called IoC (Inversion Of Control)

SPRING FRAMEWORK - IMPORTANT TERMINOLOGY

@Component (..): An instance of class will be managed by Spring framework

Dependency: Drawing needs

- Shape impl! Shape Impl (Ex: Circle) is a dependency of Drawing

Component Scan: How does Spring Framework find component classes?

- It scans packages! (@ComponentScan("packageName"))

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

- Spring Beans: An object managed by Spring Framework
- IoC container: Manages the lifecycle of beans and dependencies Types: ApplicationContext (complex), BeanFactory (simpler features - rarely used)
- Autowiring: Process of wiring in dependencies for a Spring Bean

WHY DO WE HAVE A LOT OF DEPENDENCIES?

In Drawing Hello World App, we have very few classes

BUT RealWorld applications are much more complex:

- Multiple Layers (Web, Business, Data etc)
- Each layer is dependent on the layer below it!
  - Example: Business Layer class talks to a data layer class
    - Data Layer class is a dependency of Business Layer class
  - There are thousands of such dependencies in every application!

With Spring Framework:

- INSTEAD of FOCUSING on objects, their dependencies and wiring
  - You can focus on the business logic of your application!
- Spring Framework manages the lifecycle of objects:
  - Mark components using annotations: @Component (and others..)
  - Mark dependencies using @Autowired
  - Allow Spring Framework to do its magic!

Ex: BusinessCalculationService

E.G.

```
@Configuration
@ComponentScan
public class
RealWorldExampleContextLauncher {
public static void main(String[] args)
{
try(var context = new
AnnotationConfigApplicationContext(Real
WorldExampleContextLauncher.class))
{
Arrays.stream(context.getBeanDefinition
Names()).forEach(System.out::println);
System.out.println(context.getBean(Buss
inessService.class).findMax());
```

```
public interface DataService {
 public int [] retrieveData();
}
==========================
```

```java
@Component
public class BussinessService {
private DataService dataService;

@Autowired
public BussinessService(DataService dataService) {
super();
this.dataService = dataService;
}

public int findMax() {
return Arrays.stream(dataService.retrieveData()).max().orElse(0);}}
```

```
@Component
public class
MongoDbDataService
implements  DataService {

public int[] retrieveData()
{
return new int []
{1,2,3,4,5,6,7,8,9};
}


}
```

```
@Component
@Primary
public class MySqlDataService
implements DataService {

public int[] retrieveData() {
return new int []
{11,22,33,44,55,66,77,88,99};
}


}
```

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

COMPARING LAZY INITIALIZATION VS EAGER INITIALIZATION

Heading	Lazy Initialization	Eager Initialization
Initialization time	Bean initialized when it is first made use of in the application	Bean initialized at startup of the 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 in initializing?	Errors will result in runtime exceptions	Errors will prevent application 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

E.G. @LAZY

```
@Configuration
@ComponentScan
public class LazyInitializationExample {


public static void main(String[] args) {


try(var context = new
AnnotationConfigApplicationContext(LazyInitializati
onExample.class))
{
Arrays.stream(context.getBeanDefinitionNames()).for
Each(System.out::println);
System.out.println("All set for Lazy
Initialization");
context.getBean(B.class);}}}
```

```
@Component
class A{}
@Component
@Lazy
class B{
A a;
B(A a)
{
System.out.println("Dependency Injected");
this.a=a;}
}
```

SPRING BEAN SCOPES

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

- Singleton - One object instance per Spring IoC container
- Prototype - Possibly many object instances per Spring IoC 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 vs Spring Singleton

- Spring Singleton: One object instance per Spring IoC container
- Java Singleton : One object instance per JVM

E.G.

```
@Component
class NormalClass{}

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
class PrototypeClass{}

@Configuration
@ComponentScan
public class SimpleSpringContextLauncher {

public static void main(String[] args) {


try(var context = new AnnotationConfigApplicationContext(SimpleSpringContextLauncher.class)){

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

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

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

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

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

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

System.out.println(context.getBean(PrototypeClass.class));}}}
```

PROTOTYPE VS SINGLETON BEAN SCOPE

Heading	Prototype	Singleton
Instances	Possibly Many per Spring IOC Container	One per Spring IOC Container
Beans	New bean instance created every time the bean is referred to	Same bean instance reused
Default	NOT Default	Default
Code Snippet	@Scope(value= ConfigurableBeanFactory.SCOPE_PROTOTYPE)	@Scope(value= ConfigurableBeanFactory.SCOPE_SINGLETON OR Default
Usage	Rarely used	Very frequently used
Recommended Scenario	Stateful beans	Stateless beans

@POSTCONSTRUCT & @PREDESTROY

```
@Component
class SomeClass{
SomeDependency someDependency;


public SomeClass(SomeDependency
someDependency) {
super();
this.someDependency = someDependency;
System.out.println("All Dependencies are
Ready");
}
@PostConstruct
void initialize(){
someDependency.getReady();}
@PreDestroy
void cleanUp() {
System.out.println("CleanUp Activity is
Done");
}}
@Component
class SomeDependency{
public void getReady() {
System.out.println("All Initialization is
done");
}}
```

JAKARTA CONTEXTS & DEPENDENCY INJECTION (CDI)

Spring Framework V1 was released in 2004

CDI specification introduced into Java EE 6 platform in December 2009

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

- To Add Jakarta Annotation:-
  - Add Libraries By Simply Adding Dependencies in pom.xml

```
<!--
https://mvnrepository.com/artifact/jakarta.inject
/jakarta.inject-api -->
<dependency>
    <groupId>jakarta.inject</groupId>
    <artifactId>jakarta.inject-api</artifactId>
    <version>2.0.1</version>
</dependency>
```

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

What should you use?

- 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

QUICK REVIEW OF IMPORTANT SPRING ANNOTATIONS

@Configuration:- Indicates that a class declares one or more @Bean methods and may be processed by the 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. a 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.

@Named :- Jakarta Contexts & Dependency Injection (CDI) Annotation similar to Component.

@Inject :- Jakarta Contexts & Dependency Injection (CDI) Annotation similar to Autowired

QUICK REVIEW OF IMPORTANT SPRING CONCEPTS

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

FRAMEWORK, MODULES AND PROJECTS

Spring Core : IOC Container, Dependency Injection, Auto Wiring, ..

- These are the fundamental building blocks to:
  - Building web applications
  - Creating REST API
  - Implementing authentication and authorization
  - Talking to a database
  - Integrating with other systems
  - Writing great unit tests

Let's now get a Spring:

- Spring Framework
- Spring Modules
- Spring Projects

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 (Java Message Services) etc
- **Testing:** Mock Objects, Spring MVC Test etc

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 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 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

FRAMEWORK, MODULES AND PROJECTS

Hierarchy: Spring Projects > Spring Framework > Spring Modules

Why is Spring Eco system popular?

- **Loose Coupling**: Spring manages creation and wiring of beans and dependencies
  - Makes it easy to build loosely coupled applications
  - Make writing unit tests easy! (Spring Unit Testing)
- **Reduced Boilerplate Code:** Focus on Business Logic 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!

THANKS