**IoC**

Inversion of Control is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework. It's most often used in the context of object-oriented programming.

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

Inversion of Control can be achieved through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

**DI**

Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.

The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

**Dependency Injection in Spring can be done through constructors, setters or fields.**

[Wiring](https://www.baeldung.com/spring-annotations-resource-inject-autowire) allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

There are four modes of autowiring a bean using an XML configuration:

* **no:** the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies
* **byName:** autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set
* **byType:** similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
* **constructor:** autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments

In the Spring framework, the IoC container is represented by the interface *ApplicationContext*. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their lifecycle.

The Spring framework provides several implementations of the *ApplicationContext* interface — *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext*for standalone applications, and *WebApplicationContext* for web applications.

Difference between **BeanFactory** and **ApplicationContext** are following:

1. BeanFactory uses lazy initialization **but** ApplicationContext uses eager initialization. In case of BeanFactory, bean is created when you call getBeans() method, but bean is created upfront in case of ApplicationContext when the ApplicationContext object is created.
2. BeanFactory explicitly provide a resource object using syntax **but** ApplicationContext creates and manages resource objects on its own.
3. BeanFactory doesnt support internatiolization **but** ApplicationContext supports internationalization.
4. With BeanFactory annotation based dependency injection is not supported **but** annotation based dependency injection is supported in ApplicationContext.

**Using BeanFactory:**

BeanFactory beanfactory = new XMLBeanFactory(new FileSystemResource("spring.xml"));

Triangle triangle =(Triangle)beanFactory.getBean("triangle");

**Using ApplicationContext:**

ApplicationContext context = new ClassPathXMLApplicationContext("spring.xml")

Triangle triangle = (Triangle)context.getBean("triangle");

|  | **Framework** | **Path segment** | **http query parameter** |  | |
| --- | --- | --- | --- | --- | --- |
|  | Jersey (JAX-RS) | @PathParam | @QueryParam | |
|  | Spring RESTFul | @PathVariable | @RequestParam | |
|  | example | http://xyz.ir/{segment} | http://xyz.ir/?param{param} | |

**@PathParam**reads the value from a path part of the called URI. **@QueryParam** is used to read the values ​​from QueryParameters of a URI call. These are after? listed in a URI.  
PathParams are location-dependent, while QueryParams are passed as a key value pair and therefore their order is irrelevant to more than one QueryParam.

**example**  
As an example again both calls in a URI:

<http://www.xyz.ir/> <[@PathParam](http://twitter.com/PathParam)> /? queryParamName = <[@QueryParm](http://twitter.com/QueryParm)>

@PathVariable and @RequestParam are both used to extract values from the URL, their usage is largely determined by how a site is designed.

The @PathVariable annotation is used for data passed in the URI (e.g. RESTful web services) while @RequestParam is used to extract the data found in query parameters

**Wpro**

Optional interface

@CrossOrigin(Annotations in Controller)

Annotation in Entity class and generator type ,id is compulsory in entity

How to scan other packages in spring boot

@ControllerAdvice and ExceptionHnadler

@PathParam vs @PathVariable

Native Query in Repository

Insert using @Put

Authorization and authentication in Spring

PCF config and Scaling properties

Immutable String example,

Sleep vs Wait

Ways to create thread

Volatile transient serialization

Custom exception

LinkedList vs ArrayList

HAsmAp working

**Optional**

The purpose of the Optional  class is to provide a type-level solution for representing optional values instead of null references.

Ways to create Optional:

* Optional<String> empty = Optional.empty();

To create an empty Optional object, we simply need to use its empty() static method

* String name = **"baeldung"**; Optional<String> opt = Optional.of(name);

However, the argument passed to the of() method can't be null. Otherwise, we'll get a NullPointerException:

* Optional<String> opt = Optional.ofNullable(name);

By doing this, if we pass in a null reference, it doesn't throw an exception but rather returns an empty Optional object:

**####**

When we have an Optional object returned from a method or created by us, we can check if there is a value in it or not with the isPresent() method

N@158683

Optional<String> opt = Optional.of(**"Baeldung"**);

opt.ifPresent(name -> System.out.println(name.length()));

The *ifPresent()* method enables us to run some code on the wrapped value if it's found to be non-*null*. Before *Optional*, we'd do:

**if**(name != **null**) {

System.out.println(name.length());

}

The orElse() method is used to retrieve the value wrapped inside an Optional instance. It takes one parameter, which acts as a default value. The orElse() method returns the wrapped value if it's present, and its argument otherwise:

String name = Optional.ofNullable(nullName).orElse(**"john"**);

The orElseGet() method is similar to orElse(). However, instead of taking a value to return if the Optional value is not present, it takes a supplier functional interface, which is invoked and returns the value of the invocation:

String name = Optional.ofNullable(nullName).orElseGet(() -> **"john"**);

* **orElse()** will always call the given function whether you want it or not, regardless of Optional.isPresent() value
* **orElseGet()** will only call the given function when the Optional.isPresent() == false

**Barc**

@service vs @repository

@Spring BootAppliaction

@ComponentScan is required along with SpeirngBootAnnotation for others packages

Hashmpa vs tree map

Collection hierarchy, linkedHashmap use cases

Enum vs constant class

Vector & hashtable part of current collection framework

Stream method chaining (intermediate/terminal)

[Why would you prefer Java 8 Stream API instead of direct hibernate/sql queries when working with the DB](https://stackoverflow.com/questions/43304023/why-would-you-prefer-java-8-stream-api-instead-of-direct-hibernate-sql-queries-w)

Lambda function

Relationship between lambda and functional interface, how to decide to write lambda for code block

Dispatcher servlet

Deployment descriptor

Flat file @repository is required

Spring profile

Deploy spring boot app on application server

Maven vs gradle

**@Component vs @Repository vs @Service in Spring :**

**With Spring’s auto-scanning feature, it automatically detects various beans defined in our application.** We usually annotate our beans using one of the available Spring annotations – @Component, @Repository, @Service, @Controller.

On detecting the bean, Spring simply registers it into the ApplicationContext.

In this quick tutorial, we’ll look at the difference between @Component, @Repository, and, @Service Spring annotations.

### **@Component:**

We can use @Component annotation to mark a bean as a Spring-managed component. In other words, **it’s a generic stereotype for any Spring-managed component**.

We can enable an auto-scan using <context:component-scan> tag. During auto-scan, Spring will scan and register all beans marked with a @Component annotation:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component  public class Employee {     ...    } |

### **@Repository:**

@Repository annotation is a specialization over @Component annotation:

|  |  |
| --- | --- |
| 1  2  3 | @Component  public @interface Repository {  } |

Since @Repository is a type of @Component, Spring also auto-scans and registers them.

**@Repository is a stereotype for the persistence layer. Its job is to catch all persistence related exceptions and rethrow them as a Spring DataAccessException.**

For this, we should configure PersistenceExceptionTranslationPostProcessor in our application context:

|  |  |
| --- | --- |
| 1  2 | <bean class=    "org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/> |

This bean post processor adds an advisor to all beans marked with @Repository. The advisor’s responsibility is to translate the platform-specific exceptions to the Spring’s unified unchecked exceptions.

### **@Service:**

Just like @Repository, @Service is another specialization of @Component:

|  |  |
| --- | --- |
| 1  2  3 | @Component  public @interface Service {  } |

Just like @Repository, @Service is also a type of @Component. That means Spring will also automatically detect such beans.

The **@Service annotation represents that our bean holds some business logic.**Till date, it doesn’t provide any specific behavior over @Component.

Still, **we should annotate the service-layer beans with the @Service annotation to make our intent clear.** Additionally, we never know if someday Spring chooses to add some specific functionality to it.

### **Differences in NutShell:**

Let’s quickly sum up the differences between @Component, @Repository and, @Service:

* @Component is the most generic stereotype and marks a bean as a Spring-managed component
* Both @Service and @Repository annotations are the specializations over the @Component annotation
* @Repository is a stereotype used for persistence layer. It translates any persistence related exceptions into a Spring’s DataAccessException
* @Service is used for the beans at the service layer. Currently, it doesn’t offer any additional functionality over @Component
* It’s always preferable to use @Repository and @Service annotations over @Component, wherever applicable. It communicates the bean’s intent more clearly

### **Conclusion:**

In this mini-tutorial, we have learned the difference between the popular Spring annotations – @Component, @Service, and @Repository.

To conclude, we should always prefer using the annotations based on their specific layer conventions.

**@ComponentScan is required along with SpirngBootAnnotation for others packages or not?**

The first thing that you should wonder is : why do you declare @ComponentScan while one of the goal of @SpringBootApplication is (among other things) to enable the component scan ?  
From [Spring Boot documentation](http://docs.spring.io/autorepo/docs/spring-boot/current/reference/html/using-boot-using-springbootapplication-annotation.html) :

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes

Note that when on the class of your Spring Boot Application, you declare @ComponentScan to specify a value as basePackages, it overrides the basePackages used by default by @SpringBootApplication that is the current package where the class resides. So to have as base package both the package of the Spring Boot Application class and the additional packages that were missing, you have to explicitly set them.

Besides basePackages is recursive. So to enable the scan both for classes locating in the "org.example" and "org.example.model" packages, specifying "org.example" is enough as "org.example.model" is a sub-package of it.

Try that :

@SpringBootApplication(scanBasePackages={"org.example"})

Or alternatively :

@SpringBootApplication

@ComponentScan("org.example")

[**Why would you prefer Java 8 Stream API instead of direct hibernate/sql queries when working with the DB**](https://stackoverflow.com/questions/43304023/why-would-you-prefer-java-8-stream-api-instead-of-direct-hibernate-sql-queries-w)

**1--** If the data originally comes from a DB it is better to do the filtering in the DB rather than fetching everything and filtering locally.

First, Database management systems are good at filtering, it is part of their main job and they are therefore optimized for it. The filtering can also be sped up by using indexes.

Second, fetching and transmitting many records and to unmarshal the data into objects just to throw away a lot of them when doing local filtering is a waste of bandwidth and computing resources.

**2--** Unless measured and proven for a specific scenario either could be good or equally bad. The reason you **usually** want to take these kind of queries to the database is because (among other things):

DB can handle much larger data then your java process

Queries in a database can be indexed (making them much faster)

On the other hand, if your data is small, using a Stream the way you did is effective. Writing such a Stream pipeline is **very readable.**

**3--** One case where I would prefer filtering data in Java code rather than in DB would be if you need to filter different features from the same data. For example, right now you are getting only the Author's surname. If you wanted to get all books written by the author, ages of authors, children of author, place of birth etc. Then it makes sense to get only one "read-only" copy from the DB and use parallel streams to get different information from the same data set.

**NCR**

Jvm memory type

ArrayList vs Hashset

K8s vs docker

Spring mvc vs boot

Sql injection in repository native query

Collection hierarchy

AOP

Performance testing & tool

How to improve response time performance

--

Composite key configuration in Spring JPA

<https://stackoverflow.com/questions/13032948/how-to-create-and-handle-composite-primary-key-in-jpa>

Transaction management in Spring boot

Background process while starting Spring boot application

SOAP vs REST

PUT/POST/PATCH

Method overloading with Auto boxing and widening

Parent/Child object and reference example with variable and method binding

public static void main(String[] args) {  
  
 // B b = (B) new A();//classCastException

//System.*out*.println(b.x);  
//b.dis();

A a = new B();  
 System.*out*.println(a.x); // x= 20  
 a.dis(); //class B  
 }  
}  
class A  
{  
 public int x=20;  
 public void dis()  
 {  
 System.*out*.println("Class A");  
 }  
}  
  
class B extends A  
{  
 public int x=40;  
 public void dis()  
 {  
 System.*out*.println("Class B");  
 }  
}

**CG**

Crud vs Jpa repository

ResponseEntity vs HttpEntity

Environment variables in spring boot example

Active profiles

Using JBoss instead of Tomcat in spring boot

Headers in Request object

Actuator

Optional

**S&P**

Async call and response handling in async call

Transaction in Spring boot, prorogation new/required

Thread no objects while calling method m2 from method m1 with and required

Model mapper

How to disable datasource auto configuration spring boot? <https://www.baeldung.com/spring-data-disable-auto-config>

You can use the exclude attribute of@EnableAutoConfiguration, if you find any specific auto-configuration classes that you do not want are being applied.

//By using "exclude"

@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

On the other foot, if the class is not on the classpath, you can use the excludeName attribute of the annotation and specify the fully qualified name instead.

2

//By using "excludeName"

@EnableAutoConfiguration(excludeName={Foo.class})

Spring Boot provides the facility to control the list of auto-configuration classes to exclude by using the spring.autoconfigure.exclude property. You can add into the application.properties. And you can add multiple classes with comma separated.

//By using property file

2

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

**Transaction mgmt in Spring and Propagation:**

There are six types of Transaction Propagations-

* **REQUIRED**
* **SUPPORTS**
* **NOT\_SUPPORTED**
* **REQUIRES\_NEW**
* **NEVER**
* **MANDATORY**

**REQUIRED** is the default propagation. Spring checks if there is an active transaction, then it creates a new one if nothing existed. Otherwise, the business logic appends to the currently active transaction:

@Transactional(propagation = Propagation.REQUIRED)

**public** **void** **requiredExample**(String user) { // ... }

If the method m1() is called directly then it will creates its own new transaction.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() makes use of current transaction.

If the calling service does not have a transaction the method m1() will create new transaction.

***SUPPORTS***, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional:

@Transactional(propagation = Propagation.SUPPORTS)

**public** **void** **supportsExample**(String user) {

// ...

}

If the method m1() is called directly then it doesn’t creates own new transaction.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() makes use of existing transaction.

If the calling service method does not have a transaction the method m1() will not create a new transaction.

***MANDATORY***, if there is an active transaction, then it will be used. If there isn't an active transaction, then Spring throws an exception:

@Transactional(propagation = Propagation.MANDATORY)

**public** **void** **mandatoryExample**(String user) {

// ...

}

If the method m1() is called directly it will throw an exception.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() makes use of existing transaction.

If the calling service method does not have a transaction the method m1() will throw an exception.

*NEVER* propagation, Spring throws an exception if there's an active transaction:

@Transactional(propagation = Propagation.NEVER)

**public** **void** **neverExample**(String user) {

// ...

}

If the method m1() is called directly it does not create a new transaction.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() will throws an exception.

If the calling service method does not have a transaction the method m1() will not create a new one and run without transaction.

*NOT\_SUPPORTED,* Spring at first suspends the current transaction if it exists, then the business logic is executed without a transaction.

@Transactional(propagation = Propagation.NOT\_SUPPORTED)

**public** **void** **notSupportedExample**(String user) {

// ...

}

The *JTATransactionManager* supports real transaction suspension out-of-the-box. Others simulate the suspension by holding a reference to the existing one and then clearing it from the thread context.

If the method m1() is called directly it does not create a new transaction.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() does not make use of existing transaction neither does it create its own transaction . It run without transaction.

If the calling service method does not have a transaction the method m1() will not create a new one and run without transaction.

*REQUIRES\_NEW*, Spring suspends the current transaction if it exists and then creates a new one:

@Transactional(propagation = Propagation.REQUIRES\_NEW)

**public** **void** **requiresNewExample**(String user) {

// ...

}

we need the *JTATransactionManager* for actual transaction suspension.

If the method m1() is called directly it does create a new transaction.

If the method m1() is called from another service:

If the calling service method has a transaction then the method m1() does not make use of existing transaction neither but it create its own transaction .

If the calling service method does not have a transaction the method m1() will create a new transaction.

**Transactional Isolation:**

Isolation is one of the common ACID properties: Atomicity, Consistency, Isolation, and Durability. Isolation describes how changes applied by concurrent transactions are visible to each other.

Each isolation level prevents zero or more concurrency side effects on a transaction:

* **Dirty read:** read the uncommitted change of a concurrent transaction
* **Nonrepeatable read**: get different value on re-read of a row if a concurrent transaction updates the same row and commits
* **Phantom read:** get different rows after re-execution of a range query if another transaction adds or removes some rows in the range and commits

We can set the isolation level of a transaction by *@Transactional::isolation.*It has these five enumerations in Spring: *DEFAULT*, *READ\_UNCOMMITTED*, *READ\_COMMITTED*, *REPEATABLE\_READ*, *SERIALIZABLE.*

Finicity

Overloading example based on Object and String parameters type

Static and dynamic binding

Exception hierarchy and overriding exception scenario

HashSet and Hasmap, ArrayList vs linkedList

@primary and @Qualifier

Spring annotations @Controller, @RestController @Service, @Repository

@Transaction noRollbackFor scenario

Default method in interfaces overloading example

Header req example – HandlerInterceptor

@RequestParam

ExcutorFramework scenario excuting task2 & 3 based on results of task 1

Query to select salary greater than avg salary

Query to select same salary employee name

@CondtionOnBean