**JAVA-SPRING**

**What is Spring Framework?**

Most popular frameworks for Java enterprise edition.

**Spring Core Concepts:**

**What is Dependency Injection?**

Fancy term for a rather simple concept. Dependency injection means giving an object its instance variables. Providing (injecting) dependencies for a class.

There are three ways of providing a class its dependencies in Java namely :

* Directly setting public fields – Evil Approach

public class MyMessagePublisher {

private EmailService emailService = new EmailService();

public void processMessages(String message, String receiver){

this.emailService.sendEmail(message, receiver);

}

}

public class EmailService {

public void sendEmail(String message, String receiver){

System.out.println("Email sent to " + receiver);

}

}

\*\* Potential issues with this approach:

1. EmailService initialization logic changes (it takes a constructor parameter to initialize), we would need to make changes to MyMessagePublisher class along with everywhere else in the codebase.
2. It has tight coupling. Let’s say we want to move away from sending emails and instead start sending SMSs. We will then have to write a new publisher class.
3. This code is not testable. We will be sending emails to everyone while unit testing MyMessagePublisher class.

* Constructors

public class MyMessagePublisher {

private EmailService emailService = null;

public MyMessagePublisher(EmailService emailService){

this.emailService = emailService;

}

}

We kind of solved issue 1 mentioned above. Lets assume for the time being the instantiation part for most of our classes is being handled by some central class and tomorrow if there is a change in instantiation logic that is the only place we make changes.

Class instantiating MyMessagePublisher should provide (inject) an EmailService instance (using the constructor)

* Setters

public class MyMessagePublisher {

private EmailService emailService = null;

public setEmailService(EmailService emailService){

this.emailService = emailService;

}

}

Just another way like setting with constructor.

* Accepted way of doing things – Coding to Interfaces

public interface MessageService {

void sendMessage(String message, String receiver);

}

public class EmailService implements MessageService {

@Override

public void sendMessage(String message, String receiver){

System.out.println("Email sent to " + receiver);

}

}

public class SMSService implements MessageService {

@Override

public void sendMessage(String message, String receiver){

System.out.println("SMS sent to " + receiver);

} }

public class MyMessagePublisher {

private MessageService service = null;

public MyMessagePublisher(MessageService service){

this.service = service;

}

public void processMessages(String message, String receiver){

this.service.sendMessage (message, receiver);

}

}

Benefits of DI:

* Separation of Concerns.
* Code reduction in application classes because all work to initialize dependencies is handled by the injector component
* Makes application easily extendable
* Our class is more testable as we can create mock implementations of the interface and inject them.

Another important scenario where DI plays out well: **Reduced Dependency Carrying**

Dependency carrying is when an object takes a parameter in one of its methods that it doesn't need itself but is needed by one of the objects it calls to carry out its work.

A component A boots an application and creates a configuration object, Config, that is needed by some but not all the components in the system. Then A calls a method B, B calls one in C and C calls one in D. Neither B nor C needs the Config object, but D does.

Infact not just config object is an issue here the entire instantiation logic is chained. If A needs to call some method in B it should obtain Instance of B and so goes for B, C and D.

A creates Config

A --> B --> C --> D --> Config Arrows symbolizes method calls.

One solution is to use a Static Singleton Config Object and use it wherever we want but that would be a bad design as static singleton is nothing but sort of a global which are never safe to use.

So, we use a dependency injection container you can reduce dependency carrying and the use of static singletons. The container knows about all components in the application. Therefore, it can wire the components together perfectly, without having to pass any dependencies through one component to another.

Container creates Config

container creates D and injects Config

Container creates C and injects D

Container creates B and injects C

Container creates A and injects B

**Spring achieves Dependency Injection using Configuration – XML File Based, Java Config Class and Annotation Based.**

Say we have an application class like so:

@Component

public class MyApplication {

//field-based dependency injection

@Autowired

private MessageService service;

// constructor-based dependency injection

// @Autowired

// public MyApplication(MessageService svc){

// this.service=svc;

// }

// setter-based dependency injection

// @Autowired

// public void setService(MessageService svc){

// this.service=svc;

// }

public boolean processMessage(String msg, String rec){

//some magic like validation, logging etc

return this.service.sendMessage(msg, rec);

}

} @Component annotation is added to the class, so that when Spring framework will scan for the components, this class will be treated as component. @Component annotation can be applied only to the class.

Then instead of an XML based config we create a configuration class like this:

@Configuration

@ComponentScan(value={"com.journaldev.spring.di.consumer"})

public class DIConfiguration {

@Bean

public MessageService getMessageService(){

return new EmailService();

}

}

@Configuration annotation is used to let Spring know that it’s a Configuration class. Indicates that the class can be used by the Spring IoC container as a source of bean definitions.

@ComponentScan annotation is used with @Configuration annotation to specify the packages to look for Component classes.

@Bean annotation is used to let Spring framework know that this method should be used to get the bean implementation to inject in Component classes.

And then we finally have our client using the application and config class like so:

public class ClientApplication {

public static void main(String[] args) {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(DIConfiguration.class);

MyApplication app = context.getBean(MyApplication.class);

//This is autowiring by Type… If we have multiple beans of same type then we need autowiring by name.

//For autowiring by name we need to do - @Bean(name="EmailService") in config class and then in application:

// @Autowired

//@Qualifier("EmailService")

// private MessageService service;

app.processMessage("Hi Apoorv", "apoorv@abc.com");

context.close();

}

}

Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source (for example, an xml configuration file). DI is a way to implement IoC.

**When to use Spring Data JPA? -** If you need to quickly create a JPA-based repository layer that is mainly for CRUD operations, and you do not want to create abstract DAO, implementing interfaces, Spring Data JPA is a good choice.

<https://www.journaldev.com/17034/spring-data-jpa> - Refer Full Eaxmple.

**Some popular Annotations:**

**A.**

@Configuration

public class AppConfig {

@Bean(name = "comp", initMethod = "turnOn", destroyMethod = "turnOff")

Computer computer(){ return new Computer(); }

}

turnOn and turnoff methods are defined in Computer Class and would be called during bean creation and destruction.

Instead of writing initMethod and destroyMethod we can also specify the same using annotations

public class Computer {

@PostConstruct

public void turnOn(){

System.out.println("Load operating system");

}

@PreDestroy

public void turnOff(){

System.out.println("Close all programs");

}

}

**B.**

Some JPA based annotations.

@Configuration

@EnableTransactionManagement @EnableJpaRepositories("com.journaldev.spring.repository") @PropertySource("classpath:database.properties")

public class DataConfig { //Bean definitions here }

@EnableTransactionManagement: this annotation allows users to use transaction management in application.

@EnableJpaRepositories("com.journaldev.spring.repository"): indicates where the repositories classes are present.

@PropertySource("classpath:database.properties"): says that we have property file in our classpath. The values from this file will be injected into environment variable.

@Entity

@Table(name = "people")

public class Person {

@Id

private Long id;

@Column(name = "age")

private Integer age;

@Column(name = "first\_name")

private String firstName;

}

@Entity: This annotation allows entity manager to use this class and puts it in context.

@Table(name = “people”): associates a class with a table in the database.

@Id: says that this field is the primary key.

@Column(name = "age"): denotes a column in the database with which this field will be associated.

**C.**

**Difference between @Component, @Service, @Controller, and @Repository in Spring**

During the initial release of Spring, all beans were to be declared in an XML file. For a large project, this quickly becomes a massive task, and Spring guys recognize the problem rather quickly. In later versions, they provide annotation-based dependency injection and Java-based configuration. From Spring 2.5 annotation-based dependency injection was introduced, which automatically scans and registers classes as Spring bean which is annotated using @Component annotation.

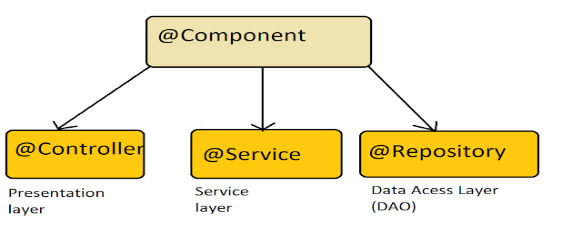
You don't need to declare that bean using the <bean> tag and inject the dependency.

This functionality was enabled and disabled using <context:component-scan> tag.

Generally called the stereotypes, @Service, @Controller, and @Repository annotations are nothing but the specialized form of @Component annotation for certain situations.

For instance Instead of using @Component on a controller class in Spring MVC, we use @Controller, which is more readable and appropriate. We do two things, first, we declare that this class is a Spring bean and should be created and maintained by Spring ApplicationContext, and also indicate that its a controller in MVC setup. This latter property is used by web-specific tools and functionalities. For example, DispatcherServlet will look for @RequestMapping on classes that are annotated using @Controller but not with @Component.

The same is true for @Service and @Repository annotation, they are a specialization of @Component in service and persistence layer. A Spring bean in the service layer should be annotated using @Service instead of @Component annotation and a spring bean in the persistence layer should be annotated with @Repository annotation.



Presentation Layer – Spring MVC

Service Layer for any Business Logic

DAO Layer for DB connectivity.

**Why and when you should use JPA**

JDBC API is still very verbose and, more importantly, lacks many features that are required when implementing a modern data access layer, like:

1. A way to fetch objects directly from the query result set. (In JDBC we need to iterate the ReusltSet and extract the column values to set the object properties.)
2. A transparent way to batch statements without having to rewrite the data access code when switching from the default non-batching mode to using batching.
3. Support for optimistic locking. Optimistic Locking is a strategy where you read a record, take note of a version number (other methods to do this involve dates, timestamps or checksums/hashes) and check that the version hasn't changed before you write the record back. When you write the record back you filter the update on the version to make sure it's atomic. (i.e. hasn't been updated between when you check the version and write the record to the disk) and update the version in one hit.

If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.

This strategy is most applicable to high-volume systems and three-tier architectures where you do not necessarily maintain a connection to the database for your session. In this situation the client cannot actually maintain database locks.

Pessimistic Locking is when you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking but requires you to be careful with your application design to avoid Deadlocks. To use pessimistic locking, you need either a direct connection to the database (as would typically be the case in a two tier client server application) or an externally available transaction ID that can be used independently of the connection.

1. A pagination API that hides the underlying database-specific Top-N and Next-N query syntax.

Later in 2001, Gavin King decided to create an ORM framework that could map database tables to POJOs (Plain Old Java Objects), and that’s how Hibernate was born.

Learning from the Hibernate project success, the Java EE platform decided to standardize the way Hibernate and Oracle TopLink, and that’s how JPA (Java Persistence API) was born.

JPA is only a specification and cannot be used on its own, providing only a set of interfaces that define the standard persistence API, which is implemented by a JPA provider, like Hibernate, EclipseLink, or OpenJPA.

JPA and Hibernate are extremely popular. The Spring Data JPA integration works like a charm. In fact, one of the biggest reasons why JPA and Hibernate are so popular is because Spring Boot uses Spring Data JPA, which, in turn, uses Hibernate behind the scenes.

**JPA Relationships:** Can be either unidirectional or bidirectional. For a bidirectional relationship, we usually define the owning side and the inverse or the referencing side. The @JoinColumn annotation helps us specify the column we'll use for joining an entity association or element collection. On the other hand, the mappedBy attribute is used to define the referencing side (non-owning side) of the relationship.

Let's say we have two entities: Employee and Email. An employee can have multiple email addresses. However, a given email address can belong exactly to a single employee. Also we'll have a foreign key employee\_id in our Email entity referring to the id attribute of an Employee. Many-To-One Relationship. The @JoinColumn annotation defines that actual physical mapping on the owning side (Many side – Email here):

@Entity

public class Email {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "employee\_id")

private Employee employee;

// ...

}

It simply means that our Email entity will have a foreign key column named employee\_id referring to the primary attribute id of our Employee entity. Once we have defined the owning side of the relationship, Hibernate already has all the information it needs to map that relationship in our database.

Now if we want to make this association bidirectional, all we'll have to do is to define the referencing side. The inverse or the referencing side simply maps to the owning side. We can easily use the mappedBy attribute of @OneToMany annotation to do so.

@Entity

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")

private List<Email> emails;

// ...

}

Here, the value of mappedBy is the name of the association-mapping attribute on the owning side.

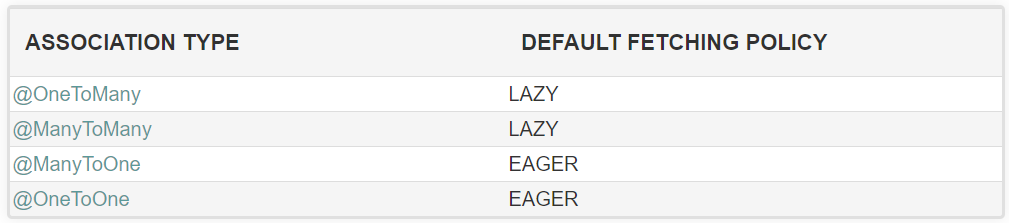
**Orphan removal Flag True vs ON DELETE CASCADE:**

orphanRemoval has nothing to do with ON DELETE CASCADE.

orphanRemoval is an entirely ORM-specific thing. It marks "child" entity to be removed when it's no longer referenced from the "parent" entity, e.g. when you remove the child entity from the corresponding collection of the parent entity.

ON DELETE CASCADE is a database-specific thing, it deletes the "child" row in the database when the "parent" row is deleted.

**Fetch Strategies**



EAGER loading of collections means that they are fetched fully at the time their parent is fetched(No delayed initialization related performance impacts but Initially loading too much unnecessary data might impact performance). So if you have Course and it has List<Student>, all the students are fetched from the database at the time the Course is fetched. LAZY on the other hand means that the contents of the List are fetched only when you try to access them(Initial load time much smaller than in the other approach, Less memory consumption than in the other approach but Delayed initialization might impact performance during unwanted moments). For example, by calling course.getStudents().iterator().

@Entity

@Table(name = "USER")

public class UserLazy implements Serializable {

@Id

@GeneratedValue

@Column(name = "USER\_ID")

private Long userId;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "user")

private Set<OrderDetail> orderDetail = new HashSet();

// standard setters and getters

// also override equals and hashcode

}

@Entity

@Table (name = "USER\_ORDER")

public class OrderDetail implements Serializable {

@Id

@GeneratedValue

@Column(name="ORDER\_ID")

private Long orderId;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name="USER\_ID")

private UserLazy user;

// standard setters and getters

// also override equals and hashcode

}

One User can have multiple OrderDetails. In eager loading strategy, if we load the User data, it will also load up all orders associated with it and will store it in a memory. But, when lazy loading is enabled, if we pull up a UserLazy, OrderDetail data won't be initialized and loaded into a memory until an explicit call is made to it.