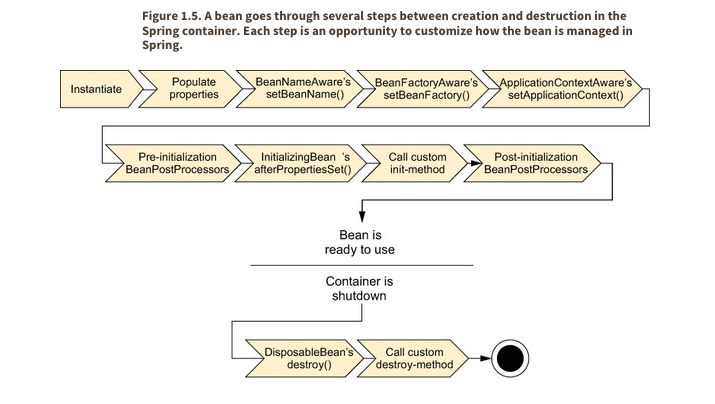
# Spring Concepts

## Spring Bean Life Cycle

* Instantiates
* Populates properties: injects values and bean reference into bean properties
* BeanNameAware 🡪 setBeanName
* BeanFactoryAware🡪setBeanFactory
* ApplicationContextAware🡪setApplicationContext
* Pre-initialization of **BeanPostProcessors**
* **InitializingBean🡪 afterPropertySet**
* **Call custom init methods**
* Post-initialization of **BeanPostProcessors**
* READY TO USE
* DisposableBean 🡪destroy
* Custom destroy methods



**Q) What is use of @import annotation?**

Used to import other configuration information into main Configuration

**Q) How do you import xml based configuration into Java based Configuration?**

**@ImportResource** annotation is used to import xml file into java configuration

**Q) How to do Configurable embedded databaes in spring?**

@Bean(destroyMethod="shutdown")

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder() .addScript("classpath:schema.sql") .addScript("classpath:test-data.sql") .build();

}

**Q) How do you activate profiles in Spring Framework?**

Spring comes with two different properies to activate profiles

1. First one is **spring.profiles.active=dev**
2. Second one is **spring.profiles.default=dev**

**Q) How many ways we can activate profiles?**

1) Intialization properies to the DispatchServlet

2) Context paramter of web application

3) Environment properties/ JNDI entries

4) JVM Argument

4) @ActiveProfiles annotation on integration test class

**Q) How do we activate profile in web.xml?**

<context-param>

<param-name>spring.profiles.default</param-name>

</param-value>dev</param-value>

</context-param>

**Q) How to achieve Conditional Configuration in spring 4?**

Using @Conditional annotation and Condition interface we can achive this

**Q) What is the Advantage of @Primary annotation?**

This annotation used to choose one among multiple annotations. This can be applied along side of @Component annotation.

**Q) What is the advantage of @Qualifier annotation?**

This is works along with @Autowired or @Inject annotation to select from multuple bean implementations

Q) What are the different scopes available in SpringFramework?

* Singleton
* Prototype
* Request
* Session
* Application
* GlobalApplication

**Q) How to you specify Scope of a Bean?**

@Scope (ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

**Q) How do you create Session Scope bean?**

**@Component**

**@Scope (value=WebApplicationContext.SCOPE\_SESSION, proxyMode=ScopedProxyMode.INTERFACES) // create proxy and implement interface**

**public ShoppingCart cart() { ... }**

**Q) When you inject SessionScope bean into Singleton bean how does it works?**

Using attribute **proxyMode** it injects proxy instead of actual Bean. In the runtime it will delegates calls to actual sessionScoped bean. This is main motton of Proxy Design Pattern.

The **proxyMode**is set to **ScopedProxyMode**.**INTERFACES**, indicating that the proxy should implement the **ShoppingCart**interface and delegate to the implementation bean.

NOTE: this works only if you are injected Interface to singleton bean

Create a JDK dynamic proxy implementing *all* interfaces exposed by the class of the target object.

**Q) How do you inject Concrete Implemenation of SessionScoped bean class into Singleton Bean?**

If the bean type is a concrete class, you must set **proxyMode**to **ScopedProxyMode.TARGET\_CLASS** to indicate that the proxy should be generated as an extension of the target class.

**NOTE: It Uses CGLib to generate extension class**

Create a class-based proxy (uses CGLIB).

**Q) How do you inject properties dynamically using SpEL?**

Using spring expression language we can inject properties dynamically.

@Value(#{“T(System).getCurrenttimemillis()”})

Using this we can call Static methods of any class

**Q) What are cross cutting concerns?**

Functions that are span into multiple points of an application are called cross cutting concerns. These are conceptually separate from the actual business logic implementations.

Ex: Logging, security, caching and transaction management

Cross cutting concerns modularizd into special classes called **aspects**.

**Q) Define AOP Terminology?**

Advice: Jo b of an Aspect is called Advice. It is fall into five main flavours. Person who measures electicity is an example of Advice.

1. Before Advice:
2. After Advice:
3. Around Advice:
4. After returning Advice:
5. After throwing Advice:

**Q) Define Joinpoint?**

Join Point is a point in the application where aspect can be executed. Ectricity measurement meters on each house an example of joinpoint. In an application we can say all the methods are joinpoints.

**Q) Define pointcut?**

Pointcuts are the points where one or more jointpoints invoked.

Ex: Person can visit few of the houses to measure electricity. For this person those houses are pointcuts. Pointcuts are specified using the Regular expressions or some other external classes.

**Q) Define Aspect?**

Combination of Advice and pointcuts are called Aspect. Which defines some functionality to achieve.

**Q) Define Introductions?**

Introductions allow introducing new methods or properties upon advice execution.

**Q) Define Weaving?**

Aspects are injected into specified joinpints to create new Proxy Objects.

Weaving can take place in the object lifetime in several places.

* **Compile time**: Aspects are woven when the target class is compiled. It requires special kind of compiler. AspectJ kind of aspects is this type.
* **Classloading time:** Aspects are weaving when the class loaded by the class loader. This kind waeving requires special kind of ClassLoader. AspectJ5’s load time weaving supports this
* **Runtime**: Aspects are woven during the runtime of the program execution. AOP Container dynamically generates proxy object that delegates the requests to target objects. Spring AOP does this.

**AOP Definition: PointCuts defines at what joinpoints advice should apply.**

In spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. As illustrated in below image, the **proxy** class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean. Between the time when the proxy intercepts the method call and the time when it invokes the target bean’s method, the proxy performs the aspect logic.

Spring doesn’t create a proxied object until that proxied bean is needed by the application. If you’re using an **ApplicationContext**, the proxied objects will be created when it loads all the beans from the BeanFactory. Because spring creates proxies at runtime, you don’t need a special compiler to weave aspects in spring’s AOP.

Q) What are the different joinpoint does spring supports?

Joinpoints can be three types

1. Method level
2. Field level
3. Constructor level

Spring only supports method level joinpoints. Other AOP frameworks such as AspectJ and Jboss are supports field and constructor joinpoints.

Ramnivas Laddad’s *AspectJ in Action*

**Q) How do u enable AOP in Java based configuration?**

Using **@EnableAspectJAutoProxy** we can enable AOP

It’s important to understand that spring’s AspectJ auto-proxying only uses @AspectJ annotations as a guide for creating proxy-based aspects. Under the covers, it’s still spring’s proxy-based aspects.

This is significant because it means that although you’re using @AspectJ annotations, you’re still limited to proxying method invocations. If you want to be able to exploit the full power of AspectJ, You’ll have to use the AspectJ runtime and not rely on spring to create proxy-based aspects.

**Q) What are introductions in spring AOP?**

Using this we can add new methods to spring dynamically

**Q) When do you prefer XML based AOP over Java Based?**

When you don’t have the source code, or if you don’t want to place AspectJ annotations in your code, spring offers another option for aspects.

**Q) What is the difference between application scope and singleton scope?**

It is a singleton per ServletContext, not per Spring 'ApplicationContext' (or which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

**Spring Web Application**

**Q) What is AbstractAnnotationConfigDispatcherServletInitializer?**

In a Servlet 3.0 environment, the container looks for any classes in the classpath that implement the **javax.servlet.ServletContainerInitializer** interface; if any are found, they’re used to configure the servlet container.

Spring supplies an implementation of that interface called **SpringServletContainerInitializer**that, in turn, seeks out any classes that implement **WebApplicationInitializer**and delegates to them for configuration. Spring 3.2 introduced a convenient base implementation of **WebApplicationInitializer**called **AbstractAnnotationConfigDispatcherServletInitializer**.

**AbstractAnnotationConfigDispatcherServletInitializer creates** both a **DispatcherServlet**and a **ContextLoaderListener**.

The@Configuration classes returned from **getServletConfigClasses()** will define beans for DispatcherServlet’s application context.

Meanwhile, the @Configuration class’s returned **getRootConfigClasses()** will be used to configure the application context created by **ContextLoaderListener**.

**Q) How to set dispatchServlet initialization parameters?**

If you plan to use Servlet 3.0 support for multipart configuration, you need to enable DispatcherServlet’s registration to enable multipart requests. You can override the customizeRegistration() method to set aMultipartConfigElement like this:

@Override

protected void customizeRegistration(Dynamic registration) { registration.setMultipartConfig(

new MultipartConfigElement("/tmp/spittr/uploads"));

}

With the ServletRegistration.Dynamic that’s given to customizeRegistration(), you can do several things, including set the load-on-startup priority by calling set-LoadOnStartup(), set an initialization parameter by calling setInitParameter(), and call setMultipartConfig() to configure Servlet 3.0 multipart support. In the preceding example, you’re setting up multipart support to temporarily store uploaded files at /tmp/spittr/uploads.

**Q) How do you register additional servlets and filter with Annotation based application?**

Override the getServletFilters() method

**Q) What exactly @SpringBootApplicatioin annotation does?**

It is combination of @Configuration, @ComponentScan, @EnableAutoConfiguration annotation to enable any application simplified.

Spring Hibernate /JPA

Spring comes with three types of session-factory beans to choose from:

* Org.springframework.orm.hibernate3.LocalSessionFactoryBean
* Org.springframeowrk.orm.hibernate3.annotation.AnnotationSessionFactoryBean
* Org.springframework.orm.hibernat4.LocalSessionFactoryBean

Q) How do you configure JPA based applications in spring?

JPA based applications use an implementation of EntityManager-Factory to get an instance of an Entitymanager.

JPA specification defines two kinds of entity manager

Application- managed: with this manager, the application is responsble for opening or closing entity managers and involving the neity manager in transactions. This thpe of entity manager is most appropriate for use in standalone applications that don’t run in a JEE containers.

Container-managed- Entity managers are created and managed by the JEE container. The application doesn’t interact with the entity manager factory at all. Instead, entity managers are obtained directly through injection or from JNDI.

Application-managed EntityManagers are created by an Entity-ManagerFactory obtained by calling the **createEntityManagerFactory()** method of the **PersistenceProvider**. Meanwhile, container-managed EntityManagerFactorys are obtained through PersistenceProvider’s create ContainerEntityManager-Factory() method.

A corresponding spring factory bean produces each flavor of entity manager factory:

* **LocalEntityManagerFactoryBean** produces an application-managed EntityManagerFactory.
* **LocalContainerEntityManagerFactoryBean** produces a container-managed EntityManagerFactory.

**NOTE:** It’s important to point out that the choice made between an application-managed **EntityManagerFactory** and a container-managed **EntityManagerFactory** is completely transparent to a Spring-based application.

The only real difference between **application-managed** and **container-managed** entity manager factories, as far as spring is concerned, is how each is configured in the spring application context.

It’s important to understand that @**PersistenceUnit** and @**PersistenceContext** aren’t spring annotations; they’re provided by the JPA specification. In order for spring to understand them and inject an EntityManagerFactory or EntityManager, spring’s **PersistenceAnnotationBeanPostProcessor** must be configured.

You also may have noticed that Repositories are annotated with @Repository and @Transactional. @Transactional indicates that the persistence methods in this repository are involved in a transactional context.

@Repository serves the same purpose here as it did when you developed the Hibernate contextual session version of the repository. Without

A template to handle exception translation, you need to annotate your repository with @Repository so **PersistenceExceptionTranslationPostProcessor** knows that this is a bean for which exceptions should be translated into one of Spring’s unified data-access exceptions.

Speaking of **PersistenceExceptionTranslationPostProcessor**, you need to remember to wire it up as a bean in spring

In java based configuration @EnableJpaRepositories annotation used to tell the container about we are using JPARepositories without any implementations are required.