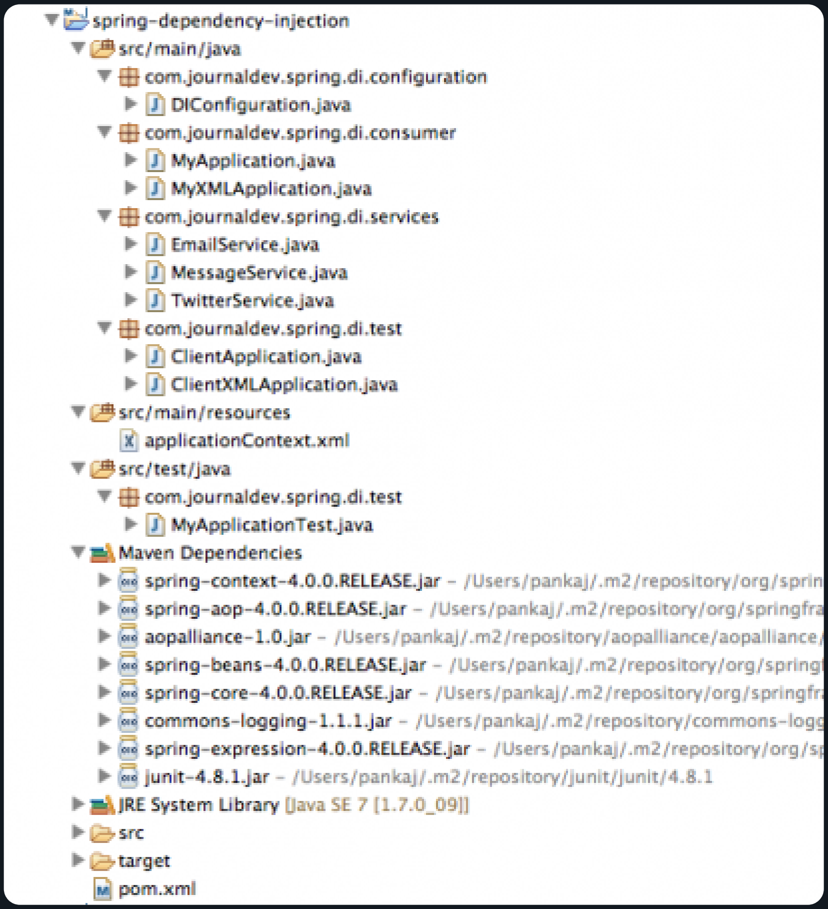
Spring Dependency Injection

Today we will look into Spring Dependency Injection. **Spring Framework** core concepts are "**Dependency Injection**" and "**Aspect Oriented Programming**". I have written earlier about **Java Dependency Injection** and how we can use **Google Guice** framework to automate this process in our applications.

Spring Dependency Injection

This tutorial is aimed to provide details about Spring Dependency Injection example with both annotation based configuration and XML file based configuration. I will also provide JUnit test case example for the application, since easy testability is one of the major benefits of dependency injection.

I have created *spring-dependency-injection* maven project whose structure looks like below image.



```
spring-dependency-injection
  src/main/java
    com.journaldev.spring.di.configuration
      DIConfiguration.java
    com.journaldev.spring.di.consumer
      MyApplication.java
      MyXMLApplication.java
    com.journaldev.spring.di.services
      EmailService.java
      MessageService.java
      TwitterService.java
    com.journaldev.spring.di.test
      ClientApplication.java
      ClientXMLApplication.java
  src/main/resources
    applicationContext.xml
  src/test/java
    com.journaldev.spring.di.test
      MyApplicationTest.java
  Maven Dependencies
    spring-context-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/sprin
    spring-aop-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/springfra
    aopalliance-1.0.jar - /Users/pankaj/.m2/repository/aopalliance/aopalliance/
    spring-beans-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/spring
    spring-core-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/springfr
    commons-logging-1.1.1.jar - /Users/pankaj/.m2/repository/commons-logg
    spring-expression-4.0.0.RELEASE.jar - /Users/pankaj/.m2/repository/org/sp
    junit-4.8.1.jar - /Users/pankaj/.m2/repository/junit/junit/4.8.1
  JRE System Library [Java SE 7 [1.7.0_09]]
  src
  target
  pom.xml
```

Let’s look at each of the components one by one.

Spring Dependency Injection – Maven Dependencies

I have added Spring and JUnit maven dependencies in pom.xml file, final pom.xml code is below.

```
<project
xmlns="https://maven.apache.org/POM/4.0.0"
xmlns:xsi="https://www.w3.org/2001/XMLSchema
-instance"
        xsi:schemaLocation="https://maven.ap
ache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.journaldev.spring</grou
pId>
        <artifactId>spring-dependency-
injection</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <dependencies>
                <dependency>

<groupId>org.springframework</groupId>
                        <artifactId>spring-
context</artifactId>

<version>4.0.0.RELEASE</version>
                </dependency>
                <dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>
```

Current stable version of Spring Framework is *4.0.0.RELEASE* and JUnit current version is *4.8.1*, if you are using any other versions then there might be a small chance that the project will need some change. If you will build the project, you will notice some other jars are also added to maven dependencies because of transitive dependencies, just like above image.

Spring Dependency Injection – Service Classes

Let's say we want to send email message and twitter message to the users. For dependency injection, we need to have a base class for the services. So I have `MessageService` interface with single method declaration for sending message.

```
package com.journaldev.spring.di.services;

public interface MessageService {

        boolean sendMessage(String msg,
String rec);
}
```

Now we will have actual implementation classes to send email and twitter message.

```
package com.journaldev.spring.di.services;

public class EmailService implements
MessageService {

        public boolean sendMessage(String
msg, String rec) {
                System.out.println("Email
Sent to "+rec+ " with Message="+msg);
                return true;
        }

}
```

```
package com.journaldev.spring.di.services;

public class TwitterService implements
MessageService {

        public boolean sendMessage(String
msg, String rec) {
                System.out.println("Twitter
message Sent to "+rec+ " with Message="+msg);
                return true;
        }

}
```

Now that our services are ready, we can move on to Component classes that will consume the service.

Spring Dependency Injection – Component Classes

Let's write a consumer class for above services. We will have two consumer classes – one with Spring annotations for autowiring and another without annotation and wiring configuration will be provided in the XML configuration file.

```
package com.journaldev.spring.di.consumer;

import
org.springframework.beans.factory.annotation
.Autowired;
import
org.springframework.context.annotation.Compo
nentScan;
import
org.springframework.stereotype.Component;

import
com.journaldev.spring.di.services.MessageSer
vice;

@Component
public class MyApplication {

        //field-based dependency injection
        //@Autowired
        private MessageService service;

//      constructor-based dependency
injection
//      @Autowired
//      public MyApplication(MessageService
svc){
//              this.service=svc;
//      }
```

Few important points about MyApplication class:

`@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 and it's retention policy is Runtime. If you are not not familiar with Annotations retention policy, I would suggest you to read java annotations tutorial.

`@Autowired` annotation is used to let Spring know that autowiring is required. This can be applied to field, constructor and methods. This annotation allows us to implement constructor-based, field-based or method-based dependency injection in our components.

For our example, I am using method-based dependency injection. You can uncomment the constructor method to switch to constructor based dependency injection.

Now let's write similar class without annotations.

```
package com.journaldev.spring.di.consumer;

import
com.journaldev.spring.di.services.MessageSer
vice;

public class MyXMLApplication {

        private MessageService service;

        //constructor-based dependency
injection
//      public
```

```
MyXMLApplication(MessageService svc) {
//              this.service = svc;
//      }

        //setter-based dependency injection
        public void
setService(MessageService svc){
                this.service=svc;
        }

        public boolean processMessage(String
msg, String rec) {
                // some magic like
validation, logging etc
                return
```

A simple application class consuming the service. For XML based configuration, we can use implement either constructor-based spring dependency injection or method-based spring dependency injection. Note that method-based and setter-based injection approaches are same, it's just that some prefer calling it setter-based and some call it method-based.

Spring Dependency Injection Configuration with Annotations

For annotation based configuration, we need to write a Configurator class that will be used to inject the actual implementation bean to the component property.

```
package
```

```
com.journaldev.spring.di.configuration;

import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Compo
nentScan;
import
org.springframework.context.annotation.Confi
guration;

import
com.journaldev.spring.di.services.EmailServi
ce;
import
com.journaldev.spring.di.services.MessageSer
vice;

@Configuration
@ComponentScan(value=
{"com.journaldev.spring.di.consumer"})
public class DIConfiguration {

        @Bean
        public MessageService
getMessageService(){
                return new EmailService();
        }
```

Some important points related to above class are:

- `@Configuration` annotation is used to let Spring know that it's a Configuration class.
- `@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.

Let's write a simple program to test our annotation based Spring Dependency Injection example.

```
package com.journaldev.spring.di.test;

import
org.springframework.context.annotation.Annot
ationConfigApplicationContext;

import
com.journaldev.spring.di.configuration.DICon
figuration;
import
com.journaldev.spring.di.consumer.MyApplicat
ion;

public class ClientApplication {

        public static void main(String[]
args) {

AnnotationConfigApplicationContext context =
new
AnnotationConfigApplicationContext(DIConfigu
ration.class);
                MyApplication app =
context.getBean(MyApplication.class);
```

```
                app.processMessage("Hi
Pankaj", "pankaj@abc.com");

                //close the context
```

`AnnotationConfigApplicationContext` is the implementation of `AbstractApplicationContext` abstract class and it's used for autowiring the services to components when annotations are used.
`AnnotationConfigApplicationContext` constructor takes Class as argument that will be used to get the bean implementation to inject in component classes.

getBean(Class) method returns the Component object and uses the configuration for autowiring the objects. Context objects are resource intensive, so we should close them when we are done with it. When we run above program, we get below output.

```
Dec 16, 2013 11:49:20 PM
org.springframework.context.support.AbstractA
pplicationContext prepareRefresh
INFO: Refreshing
org.springframework.context.annotation.Annota
tionConfigApplicationContext@3067ed13:
startup date [Mon Dec 16 23:49:20 PST 2013];
root of context hierarchy
Email Sent to pankaj@abc.com with Message=Hi
Pankaj
Dec 16, 2013 11:49:20 PM
org.springframework.context.support.AbstractA
pplicationContext doClose
INFO: Closing
```

```
org.springframework.context.annotation.Annota
tionConfigApplicationContext@3067ed13:
startup date [Mon Dec 16 23:49:20 PST 2013];
root of context hierarchy
```

Spring Dependency Injection XML Based Configuration

We will create Spring configuration file with below data, file name can be anything.

applicationContext.xml code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
xmlns="https://www.springframework.org/schem
a/beans"
        xmlns:xsi="https://www.w3.org/2001/X
MLSchema-instance"
        xsi:schemaLocation="
https://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans
/spring-beans-4.0.xsd">

<!--
        <bean id="MyXMLApp"
class="com.journaldev.spring.di.consumer.MyX
MLApplication">
                <constructor-arg>
                        <bean
class="com.journaldev.spring.di.services.Twi
tterService" />
```

```
                </constructor-arg>
        </bean>
-->
        <bean id="twitter"
class="com.journaldev.spring.di.services.Twi
tterService"></bean>
        <bean id="MyXMLApp"
class="com.journaldev.spring.di.consumer.MyX
MLApplication">
                <property name="service"
```

Notice that above XML contains configuration for both constructor-based and setter-based spring dependency injection. Since `MyXMLApplication` is using setter method for injection, the bean configuration contains *property* element for injection. For constructor based injection, we have to use *constructor-arg* element.

The configuration XML file is placed in the source directory, so it will be in the classes directory after build.

Let’s see how to use XML based configuration with a simple program.

```
package com.journaldev.spring.di.test;

import
org.springframework.context.support.ClassPat
hXmlApplicationContext;

import
com.journaldev.spring.di.consumer.MyXMLAppli
```

```
cation;

public class ClientXMLApplication {

        public static void main(String[]
args) {

ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(

"applicationContext.xml");
                MyXMLApplication app =
context.getBean(MyXMLApplication.class);

                app.processMessage("Hi
Pankaj", "pankaj@abc.com");

                // close the context
                context.close();
        }
```

`ClassPathXmlApplicationContext` is used to get the ApplicationContext object by providing the configuration files location. It has multiple overloaded constructors and we can provide multiple config files also.

Rest of the code is similar to annotation based configuration test program, the only difference is the way we get the ApplicationContext object based on our configuration choice.

When we run above program, we get following output.

```
Dec 17, 2013 12:01:23 AM
org.springframework.context.support.AbstractA
pplicationContext prepareRefresh
INFO: Refreshing
org.springframework.context.support.ClassPath
XmlApplicationContext@4eeaabad: startup date
[Tue Dec 17 00:01:23 PST 2013]; root of
context hierarchy
Dec 17, 2013 12:01:23 AM
org.springframework.beans.factory.xml.XmlBean
DefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class
path resource [applicationContext.xml]
Twitter message Sent to pankaj@abc.com with
Message=Hi Pankaj
Dec 17, 2013 12:01:23 AM
org.springframework.context.support.AbstractA
pplicationContext doClose
INFO: Closing
org.springframework.context.support.ClassPath
XmlApplicationContext@4eeaabad: startup date
[Tue Dec 17 00:01:23 PST 2013]; root of
context hierarchy
```

Notice that some of the output is written by Spring Framework. Since Spring Framework uses log4j for logging purpose and I have not configured it, the output is getting written to console.

Spring Dependency Injection JUnit Test Case

One of the major benefit of dependency injection in spring is the ease of having mock service classes rather than using actual services. So I have combined all of the learning from above and written everything in a single JUnit 4 test class for dependency injection in spring.

```
package com.journaldev.spring.di.test;

import org.junit.Assert;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import
org.springframework.context.annotation.Annot
ationConfigApplicationContext;
import
org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Compo
nentScan;
import
org.springframework.context.annotation.Confi
guration;

import
com.journaldev.spring.di.consumer.MyApplicat
ion;
import
com.journaldev.spring.di.services.MessageSer
vice;

@Configuration
@ComponentScan(value="com.journaldev.spring.
di.consumer")
public class MyApplicationTest {
```

The class is annotated with `@Configuration` and `@ComponentScan` annotation because *getMessageService()* method returns the `MessageService` mock implementation. That's why *getMessageService()* is annotated with `@Bean` annotation.

Since I am testing `MyApplication` class that is configured with annotation, I am using `AnnotationConfigApplicationContext` and creating it's object in the setUp() method. The context is getting closed in *tearDown()* method. *test()* method code is just getting the component object from context and testing it.