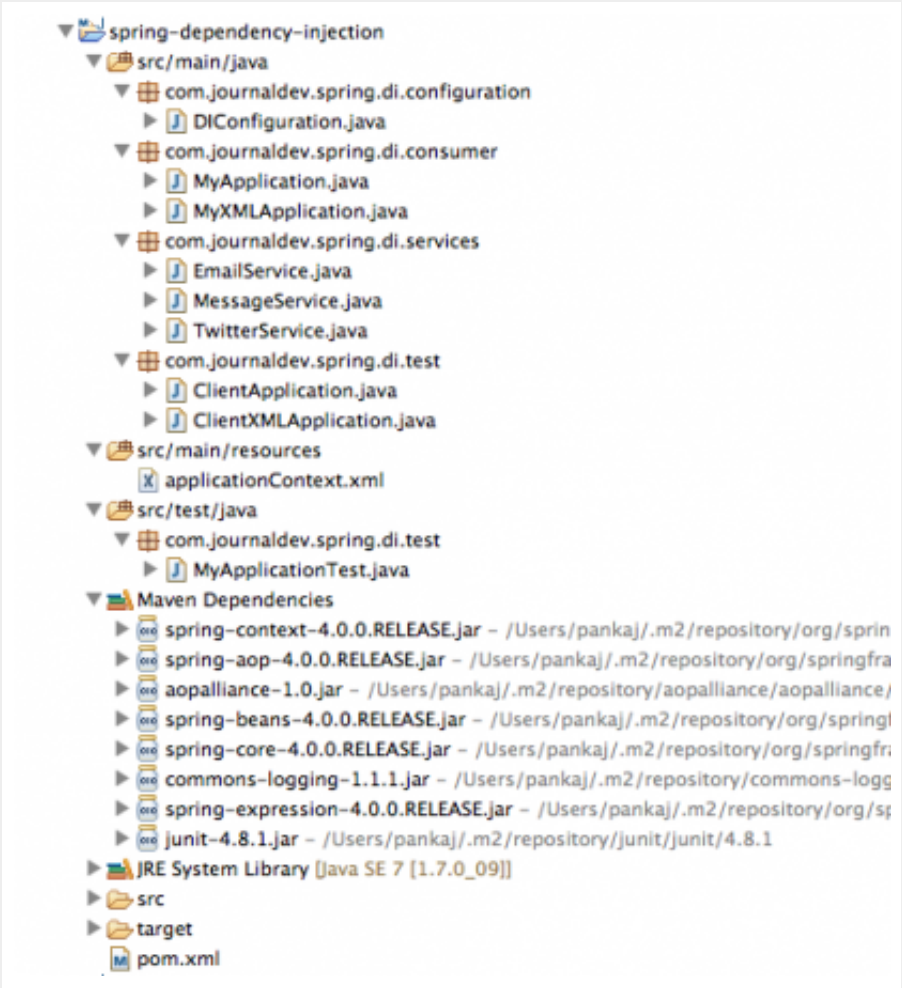
Spring Dependency Injection Example with Annotations and XML Configuration

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

This tutorial is aimed to provide dependency injection example in Spring 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.



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

Spring Maven Dependencies

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

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.journaldev.spring</groupId>
    <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>
            <version>4.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

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.

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.

MessageService.java

```java
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.

EmailService.java

```java
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;
    }

```

```
}
```

TwitterService.java

```
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="+ms
        return true;
    }

}
```

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

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.

MyApplication.java

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

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

import com.journaldev.spring.di.services.MessageService;

@Component
public class MyApplication {

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

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

    @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);
    }
}
```

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.

MyXMLApplication.java

```
1  package com.journaldev.spring.di.consumer;
2
3  import com.journaldev.spring.di.services.MessageService;
4
5  public class MyXMLApplication {
6
7      private MessageService service;
8
9      //constructor-based dependency injection
10 //  public MyXMLApplication(MessageService svc) {
11 //      this.service = svc;
12 //  }
13
14     //setter-based dependency injection
15     public void setService(MessageService svc){
16         this.service=svc;
17     }
18
19     public boolean processMessage(String msg, String rec) {
20         // some magic like validation, logging etc
21         return this.service.sendMessage(msg, rec);
22     }
23 }
```

A simple application class consuming the service. For XML based configuration, we can use implement either constructor-based dependency injection or method-based 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 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.

DIConfiguration.java

```
1  package com.journaldev.spring.di.configuration;
2
3  import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.journaldev.spring.di.services.EmailService;
import com.journaldev.spring.di.services.MessageService;

@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.

ClientApplication.java

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

import org.springframework.context.annotation.AnnotationConfigApplicationConte

import com.journaldev.spring.di.configuration.DIConfiguration;
import com.journaldev.spring.di.consumer.MyApplication;

public class ClientApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigAppli
        MyApplication app = context.getBean(MyApplication.class);

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

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

}
```

`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.AbstractApplicati
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplic
Email Sent to pankaj@abc.com with Message=Hi Pankaj
Dec 16, 2013 11:49:20 PM org.springframework.context.support.AbstractApplicati
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicati
```

Spring XML Based Configuration

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

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/sch

<!--
    <bean id="MyXMLApp" class="com.journaldev.spring.di.consumer.MyXMLApplicat
        <constructor-arg>
            <bean class="com.journaldev.spring.di.services.TwitterService" />
        </constructor-arg>
    </bean>
-->
    <bean id="twitter" class="com.journaldev.spring.di.services.TwitterServic
    <bean id="MyXMLApp" class="com.journaldev.spring.di.consumer.MyXMLApplicat
        <property name="service" ref="twitter"></property>
    </bean>
</beans>
```

Notice that above XML contains configuration for both constructor-based and setter-based 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.

ClientXMLApplication.java

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

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.journaldev.spring.di.consumer.MyXMLApplication;

public class ClientXMLApplication {

```

```
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationC
                "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.AbstractApplicati
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationCo
Dec 17, 2013 12:01:23 AM org.springframework.beans.factory.xml.XmlBeanDefiniti
INFO: Loading XML bean definitions from class path resource [applicationContex
Twitter message Sent to pankaj@abc.com with Message=Hi Pankaj
Dec 17, 2013 12:01:23 AM org.springframework.context.support.AbstractApplicati
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationConte
```

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 JUnit Test Case

One of the major benefit of dependency injection 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.

MyApplicationTest.java

```
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.AnnotationConfigApplicationConte
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.journaldev.spring.di.consumer.MyApplication;
import com.journaldev.spring.di.services.MessageService;
```

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

    private AnnotationConfigApplicationContext context = null;

    @Bean
    public MessageService getMessageService() {
        return new MessageService(){

            public boolean sendMessage(String msg, String rec) {
                System.out.println("Mock Service");
                return true;
            }

        };
    }

    @Before
    public void setUp() throws Exception {
        context = new AnnotationConfigApplicationContext(MyApplicationTest.cla
    }

    @After
    public void tearDown() throws Exception {
        context.close();
    }

    @Test
    public void test() {
        MyApplication app = context.getBean(MyApplication.class);
        Assert.assertTrue(app.processMessage("Hi Pankaj", "pankaj@abc.com"));
    }

}
```

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.

Do you wonder how Spring Framework does the autowiring and calling the methods that are unknown to Spring Framework. It's done with the heavy use of Java Reflection API that we can use to analyze and modify the behaviors of the classes at runtime.

Download Spring Dependency Injection Project
1809 downloads

Download the sample project from above URL and play around with it to learn more.