1 – Introduzione al Framework Spring

Posted on **3 aprile 2013** Views: 1347

Spring è uno dei più famosi Framework per lo sviluppo di applicazioni enterprise in Java. Milioni di sviluppatori nel mondo usano il Framwork Spring per creare codice avente le seguenti caratteristiche:**alte performance**, **facilmente testabile** e **riusabile**.

Il Framework Spring è una piattaforma open sourrce Java, inizialmente scritta da Rod Johnson ed inizialmente rilasciata sotto licenza Apache 2.0 (nel Giugno 2003).

Le fnzionalità appartenenti al core del Framework Spring possono essere usate per lo sviluppo di qualsiasi tipo di applicazione Java, esistono poi estensioni apposite per creare Web Application. Il target del Framework Spring è quello di rendere più semplice lo sviluppo di applicazione J2EE e quello di usare e promuovere buone consuedutini di programmazione usando un modello di programmazione basato sui POJO.

Benefici nell'uso del Framework Spring:

1. Spring consente allo sviluppatore di sviluppare applicazione di tipo enterprise usando i **POJO**. Il vantaggio di usare i POJO è quello di **non aver bisogno di un container EJB** come un application server ma ci basta **usare un robusto servlet container** come **Tomcat** o qualche altro prodotto commerciale.
2. Spring è organizzato in maniera modulare. Anche se il numero dei package che compongono il Framework è elevato dobbiamo preoccuparci solo di ciò che ci serve e possiamo ignorare il resto.
3. La filosofia alla base del Framework Spring è quella di non rinventare la ruota ogni volta, al suo posto fà uso di alcune tecnologie esistente come alcuni Framework ORM, alcuni Framework per effettuare il logging, JEE, Quarts e time JDK, varie tecnologie per implementare le view.
4. Testare un'applicazione scritta usando Spring è semplice perchè usando l'iniezione delle dipendenze possiamo inettare dati di testing.
5. Il Framework Web di Spring è un Framework MVC ben disegnato che fornisce un'ottima alternativa ad altri Framework come Struts.
6. Spring fornisce delle comode API per tradurre le eccezioni di specifiche tecnologie (come ad esempio eccezioni sollevate da JDBC, da Hibernate o da JDO).
7. Lo IoC container tende ad essere leggero, specialmente se comparato con un EJB container. Questo è un grande beneficio anche per il fatto di poter sviluppare le nostre applicazioni su di un computer avente risorse limitate in fatto di memoria e di CPU.
8. Spring fornisce un'interfaccia coerente per le transazioni che può scalare da una transazione locale (utilizzando ad esempio un solo databse) fino a transazioni globali (per esempio usando JTA).

Iniezione delle Dipendenze:

La tecnologia che maggiormente caratterizza Spring è l'**iniezione delle dipendenze** che realizza il pattern **Inversion of Control** che tende a **disaccoppiare i singoli componenti di un sistema**.

L' Inversion of Control (IoC) è un pattern che può essere implementato in vari modi e l'iniezione delle dipendenze è uno delle più note implementazioni.

Maggiori informazioni quì: http://it.wikipedia.org/wiki/Inversion_of_Control

Quando scriviamo complesse applicazioni Java, le classi dell'applicazione dovrebbero essere il più indipendenti possibili dalle altre classi Java per incrementare così la riusabilità di tali classi e per poter testare tali classi in maniera indipendente dalle altre quando eseguiamo gli unit test. L'iniezione della dipendenze ci aiuta a legare insieme queste classi che sono tenute comunque indipendenti le une dalle altre.
Cos'è l'iniezione delle dipendenze esattamente? Guardiamo a queste due parole separatamente:

1. **DIPENDENZA:** Eì un'**associazione tra due classi**. Per esempio, una classe A dipende da una classe B (una classe A usa una classe B).
2. **INIEZIONE:** La classe B viene **iniettata** all'interno della classe A dallo IoC container che si occupa di inserire il riferimento ad una classe B all'interno della classe A (vedremo più avanti come funziona questo meccanismo in pratica).

L'iniezione della dipendenza, ad esempio, può verificarsi quando passiamo dei parametri ad un costruttore o, dopo la costruzione di un oggetto, quando usiamo i metodi setter.
Dal momento che l'Iniezione delle Dipendenze è il cuore del Framework Spring tale argomento sarà spiegato nel dettaglio in un capitolo dedicato con esempi chiarificatori.
Aspect Oriented Programming (AOP):
Una delle componenti chiave di Spring è l'Aspect Oriented Programming (AOP) Framewotk.

Ci sono vari buoni esempi comuni di tali funzionalità che includono. il logging, le transazioni dichiarative, security caching.

Il modulo AOP del Framework Spring fornisce un'implementazione orientata agli aspetti, permettendo così allo sviluppatore di definire metodi che intercettano glii eventi da gestire. In questo modo possiamo disaccoppiare il codice che implementa tali funzionalità dal codice che implementa la logica di business. Tale aspetto sarà discusso in un capitolo separato.

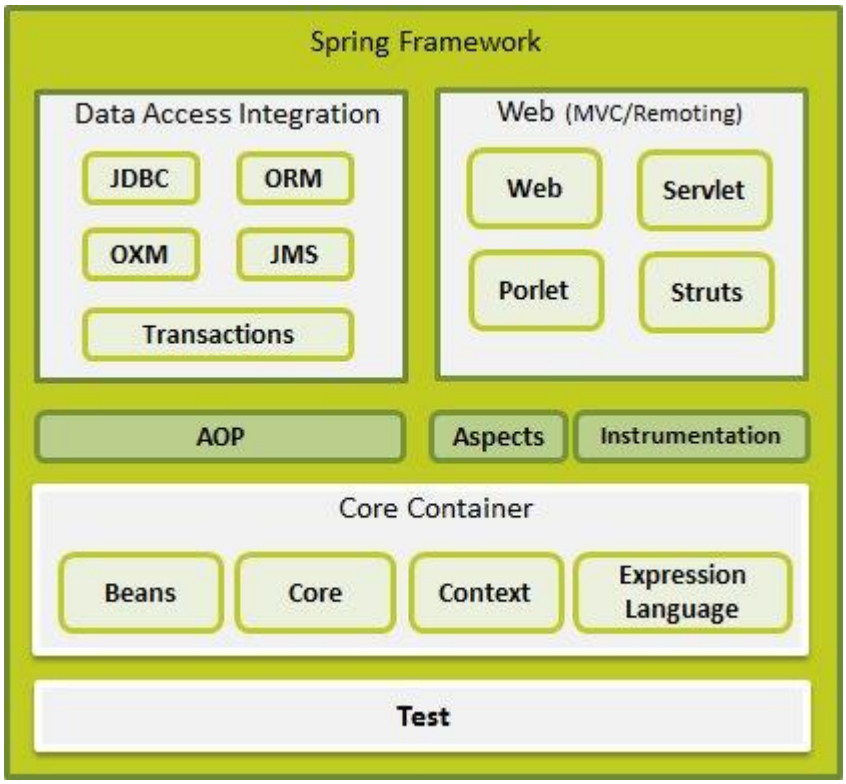
2 – Architettura Framework Spring

Posted on **3 aprile 2013** Views: 1227

In questo articolo andremmo ad esaminare l'architettura del Framework Spring.

Spring è modulare e permette allo sviluppatore di scegliere quali moduli usare nel nostro progetto senza essere costretti a portarsi dietro tutto il resto finendo per appesantire inutilmente il progetto. La seguente sezione fornisce dettagli su tutti i moduli disponibili all'interno del Framework Spring.

Il Framework Spring fornisce circa 20 moduli che possono essere usati in base ai requisiti dell'applicazione che vogliamo sviluppare.



Core Container:

Il Core Container è composto dai seguenti moduli: **Core**, **Beans**, **Context** ed **Expression Language**, ecco i dettagli relativi a tali moduli:

1. **Core**: modulo che fornisce le componenti fondamentali del Framework tra cui lo **IoC** container ed il meccanismo della **Dependency Injection**.
2. **Beans**: modulo che fornisce la **BeanFactory** che rappresenta una sofisticata **implementazione del pattern Factory**.
3. **Context**: E' un modulo costruito sulle solide basi fornite dai moduli Core e Beans ed è un mezzo per accedere a tutti gli oggetti definiti e configurati.
   L'interfaccia ApplicationContext è il punto focare di tale modulo.
4. **Expression Language**: E' un modulo che fornisce un potente expression lanaguage per eseguire query e manipolare oggetti a runtime

Data Access/Integration:

Il livello Data Access/Integration è composto dai seguenti moduli: **JDBC**, **ORM**, **OXM, JMS** e **Transaction**, di seguito i dettagli su tali moduli:

1. **JDBC**: e' un modulo che **fornisce un livello di astrazione a JDBC** che elimina la necessità di implementare a mano tutto il codice necessario a JDBC.
2. **ORM**: modulo che fornisce un livello di integrazione con alcune popolari API che implementano il **mapping tra oggetti Java e tabelle di un database relazionali**.
   Tali API includono il supporto a **JPA**, **JDO**, **Hibernat** ed **iBatis**.
3. **OXM**: modulo che fornisce un livello di astrazione che supporta il **mapping tra oggetti Java e documenti XML**, fornisce implementazioni di **JAXB**, **Castor**, **XMLBeans**, **JiBX** ed **XStream**.

4. **JMS**: modulo che fornisce il **Java Messaging Service**, contiene funzionalità per produrre e consumare messages.
5. **Transaction**: modulo che fornisce un **supporto per la gestione delle transazioni dischiarative** per classi che implementano interfacce speciali e per tutti i POJO.

Web:

Il livello Web è formato dai seguenti moduli: **Web**, **Web-Servlet**, **Web-Struts** e **Web-Portlet**, di seguito i dettagli per ciascun modulo:

1. **Web**: modulo che fornisce le funzioni di integrazione base orientate al web, come ad esempio la funzione **multipart file-upload** e l'**inizializzazione dello IoC container facente uso di servlet listeners** ed un **application context orientato al web**.
2. **Web-Servlet**: modulo che contiene l'implementazione del pattern **Model-View-Contoller** (**MVC**) per sviluppare Web Application.
3. **Web-Struts**: modulo che contiene le **classi di supporto per integrare un'applicazione Struts all'interno di un'applicazione Spring**.
4. **Web-Portlet**: modulo che fornisce un'**implementazione MVC da usare nello sviluppo di una web application basata su portlet**. Tale modulo ripropone in tale modello di sviluppo le funzionalità del modulo Web-Servlet.

Miscellaneous:

Ci sono pochi altri importanti moduli come **AOP**, **Aspects**, **Instrumentation**, **Web** e **Test**, di seguito i dettagli per ciascun modulo:

1. **AOP**: modulo che fornisce l'i**mplementazione per la programmazione orientata agli aspetti** permettendo allo sviluppatore di definire metodi intercettatori per disaccoppiare in maniera pulita codice che implementa funzionalità che dovrebbero essere separate.
2. **Aspects**: modulo che fornisce l'**integrazione con AspectJ** che è un Framework per la programmazione orientata agli asptti potente e maturo.
3. **Instrumentation**: modulo che fornisce classi di **orchestrazione e delle classi caricatore da usare in determinati application server**.
4. **Test**: modulo che fornisce il **supporto di testing** dei componenti Spring facendo uso dei Framework di testing **JUnit** e **TestNG**.

3 – Progetto di Hello World in Spring gestito con Maven

Posted on **3 aprile 2013** Views: 5781

In questo tutorial vedremo come creare una semplicissima applicazione console di Hello World in Spring che fà uso di Maven per il processo di build usando STS\Eclipse come ambiente di sviluppo (STS è la versione di Eclipse fornita dal team di Spring avente i plugin di Spring e di Maven preinstallati, chi usa Eclipse eventualmente dovrà installare i plugin manualmente).

I passi da seguire per creare il progetto sono i seguenti:

1. Aprire STS\Eclipse.
2. Premere "**Ctrl + N**" per creare un **nuovo progetto**.

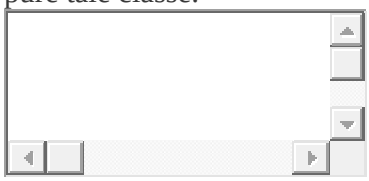
3. Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
4. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.

5. Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.
6. Inserite i seguenti parametri:**Group Id**: org.andrea.myexample
   Artifact Id: myspringhelloworld
   Version: lasciate 0.0.1-SNAPSHOTClickate su **Finish**.
7. Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework: **spring-core**, **spring-bean**, **spring-context**, **spring-context-support**).

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package **org.andrea.myexample.myspringhelloworld** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.

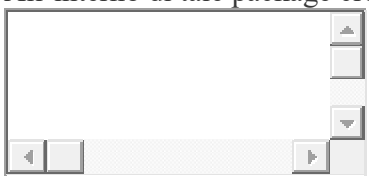


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.andrea.myexample</groupId>
 <artifactId>myspringhelloworld</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>jar</packaging>

 <name>myspringhelloworld</name>
 <url>http://maven.apache.org</url>

 <properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

 <dependencies>
   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>3.8.1</version>
     <scope>test</scope>
   </dependency>
   <dependency>
     <groupId>org.springframework</groupId>
```

```
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28     </dependency>
29     <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33     </dependency>
34     <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38     </dependency>
39     <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43     </dependency>
44   </dependencies>
45 </project>
```

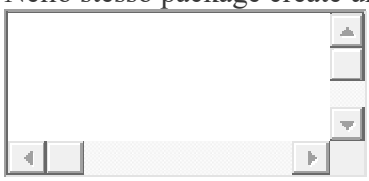
Per completezza di seguito è riportato l'intero codice del file **pom.xml** (potete copiarlo ed incollarlo):

All'interno di tale package creaiamo una classe dandogli nome **HelloWorld.java**:



```
1  package org.andrea.myexample.myspringhelloworld;
2
3  public class HelloWorld {
4
5        // Messaggio da visualizzare:
6        private String message;
7
8        /* Metodo usato per iniettare il valore della variabile message definito
9         * nel file Beans.xml */
10       public void setMessage(String message){
11          this.message  = message;
12       }
13
14       public void getMessage(){
15          System.out.println("Your Message : " + message);
16       }
17 }
```

Nello stesso package create un'altra classe chiamata **MainApp.java**:



```
1  package org.andrea.myexample.myspringhelloworld;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  /* Classe principale contenente il metodo main) */
7  public class MainApp {
8
```

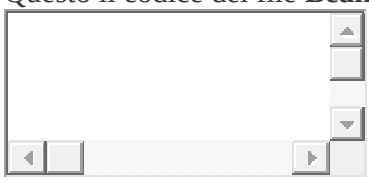
```
 9     public static void main(String[] args) {
10
11         /* Crea il contesto in base alle impostazioni dell'applicazione definite
12          * nel file Beans.xml  */
13         ApplicationContext context =
14             new ClassPathXmlApplicationContext("Beans.xml");
15
16         /* Recupera un bean avente id="helloWorld" nel file di configurazione
17          * Beans.xml */
18         HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
19
20         // Invoco il metodo che stampa il messaggio su tale oggetto
21         obj.getMessage();
22     }
23 }
```

Ci sono due punti importanti da analizzare all'interno nella classe **MainApp.java** che rappresenta la classe contenente il metodo main(), quindi la classe principale che conterrà il metoto eseguito per primo:
Il primo passo è quello in cui viene creato un oggetto **ApplicationContext**.
ApplicationContext è un'**interfaccia che fornisce la configurazione per un'applicazione**. Tale configurazione è in sola lettura fintanto che l'applicazione è in esecuzione ma può essere ricaricata se l'implementazione usata lo permette.
Per maggiori informazioni leggere questo link:
http://static.springsource.org/spring/docs/3.0.x/api/org/springframework/context/ApplicationContext.html
Nel nostro caso per creare tale oggetto stiamo usando la seguente implementazione di tale interfaccia: **ClassPathXmlApplicationContext** Tale API carica un **file di configurazione dei bean usati nel nostro progetto ed eventualmente, basandosi su tale API, si occupa di creare ed inizializzare tutti questi oggetti** (come ad esempio i bean mensionati in tale file di configurazione).

1. Nel nostro progetto tale file di configurazione sarà rappresentato da un file XML chiamato:**Beans.xml**
2. Il secondo passo è quello di **recuperare un bean** corrispondente ad un oggetto **HelloWorld** usando il metodo **getBean()** sul context creato.Tale metodo usa il **bean ID** per ritornare un oggetto generico che alla fine può essere castato all'actual object **HelloWorld**. In pratica stiamo recuperando l'oggetto associato al bean avente id "**helloWorld**" (dichiarato nel file **Beans.xml**).Una volta che abbiamo l'oggetto possiamo usarlo.

Il prossimo passo è quello di **creare questo file di configurazione dei bean**: si tratta di un file **XML**che **agisce come cemento che lega tra loro i bean insieme definendo come interagiscono tra loro**.
Mettiamo tale file nella cartella di STS\Eclipse: "**/src/main/java**" (allo stesso livello del nostro package)
In genere gli sviluppatore chiamano tale file con il nome di **Beans.xml** ma volendo si può dare un nome qualsiasi a tale file.
Il file **Beans.xml** è usato per **assegnare un ID univoco ad ogni bean e per controllare la creazione di oggetti che hanno valori differenti senza impattare sui file sorgente.** Ad esempio nel file di configurazione di tale progetto possiamo passare un qualsiasi valore per la variabile "**message**" potendo così stampare differenti valodri di mesage senza avere nessun impatto sui file **HelloWorld.java** e **MainApp.java**

Questo il codice del file **Beans.xml**:



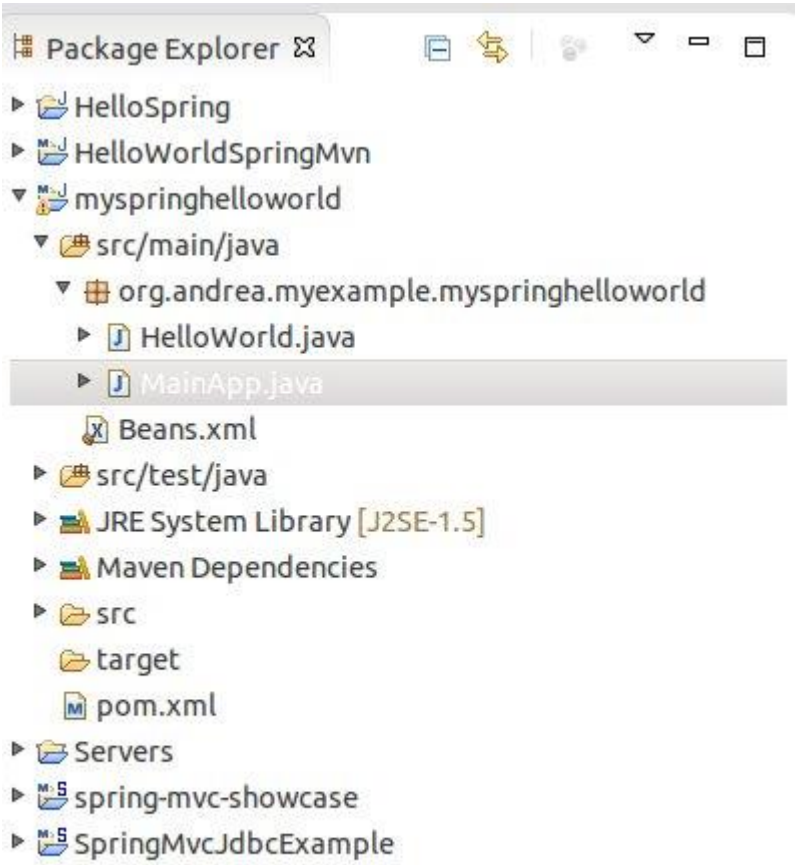
```
1  <beans xmlns="http://www.springframework.org/schema/beans"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://www.springframework.org/schema/beans
4
5  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
6
7     <bean id="helloWorld" class="org.andrea.myexample.myspringhelloworld.HelloWorld">
8        <property name="message" value="Hello World!"/>
9     </bean>
10
11 </beans>
```

Analizziamo il significato di tale file: quando l'applicazione Spring viene caricata in memoria, il Framework fà uso di tale file di configurazione per **creare tutti i bean definiti in tale file e per assegnargli un ID univoco** (come definito all'interno del tag **<bean>**).

Possiamo usare il tag **<property>** per **passare i valori di più variabili nel momento della creazione di tale oggetto**.

Nel nostro progetto ad esempio, quando l'applicazione viene avviata succede che viene creato l'oggetto **HelloWorld** avente id="**helloWorld**" ed iniettiamo il valore: "**Hello World!**" alla stringa **message** in tale classe.

Quì sotto un'immagine dell'organizzazione del progetto:



ESECUZIONE DEL PROGRAMMA:

Ora che abbiamo creato sia i nostrifile sorgenti sia il nostro file di configurazione dei bean, siamo pronti per compilare ed eseguire il programma.Per fare ciò selezionate la classe **MainApp.Java**, **tasto destro del mouse** —> **Runs As** —> **Java Application**

ed otterremo il seguente messaggio di output nel tab Console di STS\Eclipse: **Your Message : Hello World!**

4 – Spring IoC Container

Posted on **3 aprile 2013** Views: 1159

il **container di Spring** è situato nel **modulo core** del Framework Spring.

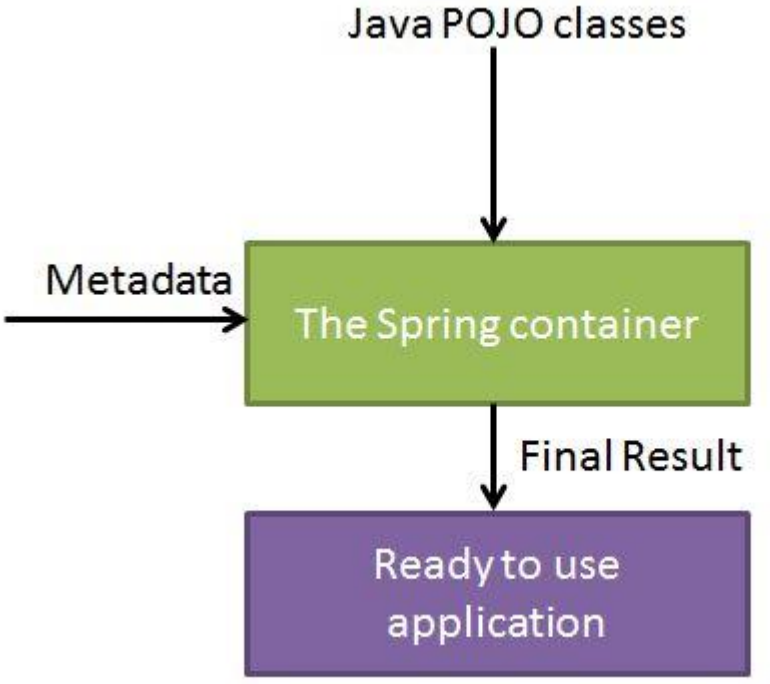
Il **container** si occupa della **creazione degli oggetti**, di **legarli tra loro**, di **configurarli** e dell'**intero ciclo di vita di tali oggetti** (dalla creazione fino alla distruzione).

Il **container di Spring** usa l'**iniezione delle dipendenze** per geristre i componenti che costituiscono l'applicazione.

Tali **oggetti** sono chiamati **Spring Beans** e ne discuteremo in modo più dettagliato nel prossimo articolo.

Il **container** ha **informazioni** su quali **oggetti deve istanziar**e, quali o**ggetti deve configurare** ed **assemblare** in base alla **lettura di metadata di configurazione** che gli vengono forniti. Tali **metadata**

di configurazione possono essere rappresentati nei seguenti modi: usando un **file XML**, usando le **Annotation Java**, attraverso **codice Java**.
Il seguente diagramma mostra ad alto livello come funziona Spring. Lo IoC container di Spring fa uso di classi Java di tipo POJO e metadati di configurazione per la produzione di un sistema completamente configurato ed eseguibile come applicazione.



Spring fornisce i seguenti due differenti tipi di container:

1. **Spring BeanFactory Container:** Si tratta del più semplice container che fornisce un supporto basilare per l'iniezione della dipendenza ed è definito dall'interfaccia ***org.springframework.beans.factory.BeanFactory***
   La **BeanFactory** e le **relative interfacce** (**BeanFactoryAware**, **InizializingBean**, **DisposableBean**) continuano a essere presenti in Spring a scopo di retrocompatibilità con un gran numero di framework di terze parti che si integrano con Spring.

2. **Spring ApplicationContext Container:** Questo container aggiunge funzionalità tipicamente enterprise come la possibilità di risolvere messaggii testuali a partire da file di properties e la capacità di inviare eventi applicativi a specifici listener di eventi. Tale container è definito dall'interfaccia: ***org.springframework.context.ApplicationContext***

Il container ApplicationContext include tutte le funzionalità previste dal container BeanFactory, pertanto è consigliabile usare ApplicationContext. BeanFactory può comunque essere usato nel caso di applicazioni leggere come applicazioni dedicate a device di tipo mobile o applicazioni basate sulle applete in cui potrebbe essere meglio usare una configurazione leggera e minimale.

5 – BeanFactory Vs. ApplicationContext analisi con esempi pratici

Posted on **3 aprile 2013** Views: 1253

Esempio Spring BeanFactory Container:

Si tratta del più semplice container che fornisce un supporto basilare per l'iniezione delle dipendenze. Esso è definito dall'interfaccia: ***org.springframework.beans.factory.BeanFactory***

BeanFactory e le relative interfacce correlate (**BeanFactoryAware**, **InitializingBean**,**DisposableBean**) continuano ad essere presenti all'interno di Spring per permettere la retrocompatibilità con un grande numero di framework di terze parti che possono essere integrati con Spring.

Esistono più implementazioni dell'interfaccia BeanFactory che vengono fornite nativamente insieme a Spring. L'**implementazione più usata** è certamente la classe **XmlBeanFactory**. Tale container **legge i metadati di configurazione da un file XML ed usa tali informazioni per creare un'applicazione completamente configurata**.

L'uso del container BeanFactory è in genere preferito quando le risorse sono limitate, ad esempio nelle applicazioni per dispositivi mobile o per applicazioni basate su applet. In genere è consigliabile usare ApplicationContext al posto di BeanFactory, a meno di non avere valide ragioni per preferire il container minimale

Progetto di esempio che fa uso di BeanFactory:

In questo tutorial vedremo come creare una semplicissima applicazione console che fa uso del container BeanFactory

I passi da seguire per creare il progetto sono i seguenti:

1. Aprire STS\Eclipse.
2. Premere "**Ctrl + N**" per creare un **nuovo progetto**.
3. Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
4. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.

5. Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.
6. Inserite i seguenti parametri:

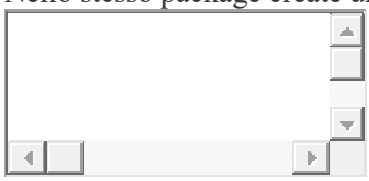
Group Id: org.andrea.myexample
Artifact Id: mybeanfactoryexample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.

Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.
Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.
All'interno della cartella di STS con nome **/src/main/java** troviamo il package **org.andrea.myexample.myspringhelloworld** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
All'interno di tale package creaiamo una classe dandogli nome **HelloWorld.java**:

```
1  package org.andrea.myexample.myspringhelloworld;
2
3  public class HelloWorld {
4
5        // Messaggio da visualizzare:
6        private String message;
7
8        /* Metodo usato per iniettare il valore della variabile message definito
9         * nel file Beans.xml */
10       public void setMessage(String message){
11          this.message  = message;
12       }
13
14       public void getMessage(){
15          System.out.println("Your Message : " + message);
16       }
17 }
```

Nello stesso package create un'altra classe chiamata **MainApp.java**:



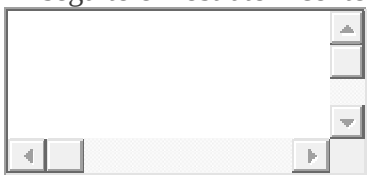
```
1  package org.andrea.myexample.myspringhelloworld;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  /* Classe principale contenente il metodo main) */
7  public class MainApp {
8
9     public static void main(String[] args) {
10
11       /* Crea il contesto in base alle impostazioni dell'applicazione definite
12        * nel file Beans.xml  */
13       ApplicationContext context =
14           new ClassPathXmlApplicationContext("Beans.xml");
15
16       /* Recupera un bean avente id="helloWorld" nel file di configurazione
17        * Beans.xml */
18       HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
19
```

```
20        // Invoco il metodo che stampa il messaggio su tale oggetto
21        obj.getMessage();
22    }
23 }
```

In maniera pressochè analoga a quanto visto nell'esempio di Hello World visto in un precedente articolo troviamo che:

1. Costruiamo la nostra factory che istanzia, inietta e gestisce il ciclo di vita dei nostri bean; questa volta usando l'implementazione BeanFactory.
   Per recuperare il file di configurazione Beans.xml usiamo un oggetto ClassPathResource, maggiori informazioni al seguente link:
   http://static.springsource.org/spring/docs/1.1.x/api/org/springframework/core/io/ClassPathResource.html
2. Usando la factory recuperiamo un bean avente id: "helloWerld" e lo usiamo per ottenere il messaggio di Hello World.
   Notate che il metodo che recupera il bean ottiene un oggetto di tipo generico che viene castato al tipo attuale HelloWorld.

Di seguito è mostrato il contenuto del file di configurazione **Beans.xml**:



```
1  <beans xmlns="http://www.springframework.org/schema/beans"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://www.springframework.org/schema/beans
4
5  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
6
7     <bean id="helloWorld" class="org.andrea.myexample.myspringhelloworld.HelloWorld">
8        <property name="message" value="Hello World!"/>
9     </bean>
10
11 </beans>
```

Come visto nel precedente esempio di HelloWorld, questo file deve essere posizionato all'interno della cartella: "***src/main/java***" del nostro progetto in STS\Eclipse (allo stesso livello del nostro unico package)

ESECUZIONE DEL PROGRAMMA:

Ora che abbiamo creato sia i nostrifile sorgenti sia il nostro file di configurazione dei bean, siamo pronti per compilare ed eseguire il programma.Per fare ciò selezionate la classe **MainApp.Java**, **tasto destro del mouse** —> **Runs As** —> **Java Application**ed otterremo il seguente messaggio di output nel tab Console di STS\Eclipse: **Your Message : Hello World!Abbiamo quindi creato un'applicazione funzionante che fà uso dell'implementazione XmlBeanFactory della factory minimale BeanFactory.**

Esempio Spring ApplicationContext Container:

Andiamo ora ad analizzare un esempio di applicazione Hello World che fà uso della factory più avanzata ApplicationContext.

Application Context è il container più avanzato di Spring. Come BeanFactory esso si occupa di caricare la definizione dei ben, creare tali bean, legarli tra loro ed iniettare i bean quando vengono richiesti. In più rispetto la BeanFactory vengono aggiunte alcune funzioni di stampo tipicamente enterprise come la capacità di risolvere textual messages a partire da un file di properties e la capacità di inoltrare specifici eventi che si verificano a degli specifici event listener che li gestiscono automaticamente.

Tale container è definito dall'interfaccia: ***org.springframework.ApplicationContext***
L'ApplicationContext include tutte le funzionalità fornite da BeanFactory ed è il container raccomandato.
Le implementazioni più comuni dell'interfaccia ApplicationContext sono:

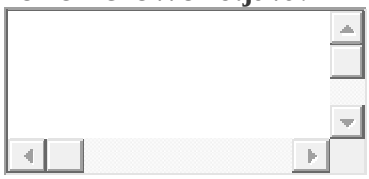
1. **FileSystemXmlApplicationContext:** questo container carica la definizione dei bean da un file XML. In questo caso dobbiamo fornire il path completo di dove si trova il file al costruttore di tale oggetto.
2. **ClassPathXmlApplicationContext:** questo container carica la definizione dei bean da un file XML. In questo caso non abbiamo bisogno di fornire il path completo della locazione del file ma dobbiamo settare correttamente il CLASSPATH in modo tale che questo file XML sia nel CLASSPATH.
3. **WebXmlApplicationContext:** questo container carica il file XML contenente la definizione di tutti i beans da una qweb application.

Andiamo ora ad implementare un **esempio pratico** che fà uso dell'implementazione**FileSystemXmlApplicationContext** dell'interfaccia **ApplicationContext**
La creazione del progetto è esattamente uguale a quanto visto nel precedente esempio quindi eviterò di ripeterla, questi sono i parametri con cui ho creato il mio progetto Maven:

Group Id : org.andrea.myexample
Artifact Id: myapplicationcontextexample
Version: lasciate 0.0.1-SNAPSHOT
Anche in questo caso, dopo aver creato il progetto, aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**, **spring-bean**, **spring-context**, **spring-context-support**).
Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

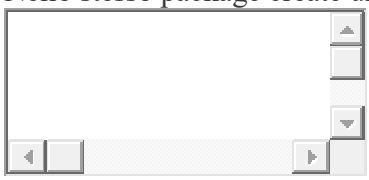
Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.myspringhelloworld** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe. All'interno di tale package creaiamo una classe dandogli nome **HelloWorld.java**:



```
1  package org.andrea.myexample.myspringhelloworld;
2
3  public class HelloWorld {
4
5     // Messaggio da visualizzare:
6     private String message;
7
8     /*
9      * Metodo usato per iniettare il valore della variabile message definito nel
10     * file Beans.xml
11     */
12    public void setMessage(String message) {
13       this.message = message;
14    }
15
16    public void getMessage() {
17       System.out.println("Your Message : " + message);
18    }
19 }
```

Nello stesso package create un'altra classe chiamata **MainApp.java**:

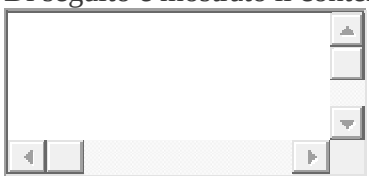


```
1  package org.andrea.myexample.myapplicationcontextexample;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.FileSystemXmlApplicationContext;
5
6  // Classe principale contenente il metodo main()
7  public class MainApp {
8     public static void main(String[] args) {
9        /*
10       * Creo l'ApplicationContext factory usando l'implementazione
11       * FileSystemXmlApplicationContext, pertanto devo passare il path
12       * completo del file di configurazione dei beans al costruttore
13       */
14       ApplicationContext context = new FileSystemXmlApplicationContext(
15             "src/main/java/Beans.xml");
16       /*
17       * Recupera un bean avente id="helloWorld" nel file di configurazione
18       * Beans.xml
19       */
20       HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
21
```

```
22        // Invoco il metodo che stampa il messaggio su tale oggetto:
23        obj.getMessage();
24     }
25 }
```

1. Costruiamo la nostra factory che istanzia, inietta e gestisce il ciclo di vita dei nostri bean; questa volta usando l'implementazione FileSystemXmlApplicationContext.
   Per recuperare il file di configurazione Beans.xml usiamo il percorso completo del file, maggiori informazioni al seguente link: http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html
2. Usando la factory recuperiamo un bean avente id: "helloWerld" e lo usiamo per ottenere il messaggio di Hello World.
   Notate che il metodo che recupera il bean ottiene un oggetto di tipo generico che viene castato al tipo attuale HelloWorld.

Di seguito è mostrato il contenuto del file di configurazione **Beans.xml**:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3     <beans xmlns="http://www.springframework.org/schema/beans"
4        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5        xsi:schemaLocation="http://www.springframework.org/schema/beans
6        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8        <bean id="helloWorld" class="org.andrea.myexample.myapplicationcontextexample.HelloWorld">
9           <property name="message" value="Hello World!"/>
10       </bean>
11
12    </beans>
```

Come visto nel precedente esempio di HelloWorld, questo file deve essere posizionato all'interno della cartella: "***src/main/java***" del nostro progetto in STS\Eclipse (allo stesso livello del nostro unico package)

ESECUZIONE DEL PROGRAMMA:

Ora che abbiamo creato sia i nostrifile sorgenti sia il nostro file di configurazione dei bean, siamo pronti per compilare ed eseguire il programma.

Per fare ciò selezionate la classe **MainApp.Java**, **tasto destro del mouse** —> **Runs As** —> **Java Application**

ed otterremo il seguente messaggio di output nel tab Console di STS\Eclipse: **Your Message : Hello World!**

Abbiamo quindi creato un'applicazione funzionante che fà uso dell'implementazione FileSystemXmlApplicationContext della factory più avanzata ApplicationContext.

6 – Definire i Bean in Spring

Posted on **3 aprile 2013** Views: 9746

Gli oggetti che compongono la spina dorsale di un'applicazione e che sono gestiti dallo IoC Container del Framework Spring sono chiamati beans.

Un **bean** è un'**oggetto che viene istanziato, assemblato e gestito dallo IoC Container di Spring**.Tali bean vengono creati in base ai metadata di configurazione che forniamo al container, ad esempio tramite un file XML definendo ogni bean all'interno di un tag XML <bean/> come abbiamo visto nei capitoli precedenti.
La definizione di un bean contiene delle informazioni chiamata metadati di configurazione, tali informazioni sono necessarie al container per conoscere le seguenti cose:

1. **Come creare un bean**
2. **I dettagli sul cliclo di vita di un bean**
3. **Le dipendenze di un bean**

Tali configurazioni vengono tradotte nella seguente serie di proprietà che compongono la definizione di ogni bean:

PROPIETÀ	DESCRIZIONE
class	Questo attributo è obbligatorio per specificare la classe del bean da utilizzare per crearlo (Specifica il tipo di dato rappresentato dal bean)
name	Questo attributo specifica un identificativo univoco per il bean. Nella configurazione basata su XML si usa l'attributo id e/o name per specificare l'identificativo del bean.
scope	Questo attributo specifica lo scope di un oggetto creato a partire dalla definizione di uno specifico bean (tale argomento sarà discusso in seguito all'interno di un'articolo dedicato allo scope dei bean)
constructor-arg	Questo attributo è usato per iniettare le dipendenze all'interno del costruttore di un oggetto definito da uno specifico bean (sarà discusso nel prossimo articolo)
properties	Questo attributo è usato per iniettare le dipendenze (sarà discusso nel prossimo articolo)
autowiring mode	Questo attributo è usato per iniettare le dipendenze (sarà discusso nel prossimo articolo)
lazy-initialization mode	Attributo che specidica la modalità di inizializzazione lazy di un bean. Tale modalità dice allo IoC Container di creare un'istanza di un bean quando questo viene richiesto per la prima volta anzichè farlo allo startup dell'applicazione (come avviene per default)
initialization method	Attributo che definisce un metodo di callback che viene richiamato subito dopo che tutte le proprietà necessarie di un bean sono state settate dal Container. (Tale argomento verrà discusso nell'articolo dedicato al ciclo di vita dei bean)
destruction	Attributo che definisce un metodo di callback da usare quanto il container che contiene i

method	bean viene distrutto. (Tale argomento verrà discusso nell'articolo dedicato al ciclo di vita dei bean)

Metadati di configurazione di Spring:

Lo IoC Container di Spring è totalmente disaccoppiato dal formato in cui questi metadati di configurazione vengono effettivamente scritti. Ci sono tre metodi per fornire tali metadati di configurazione al Container di Spring:

1. **Configurazione basata su XML**
2. **Configurazione basata sulle Annotation**
3. **Configurazione basata su Java**

Nei precedenti articoli abbiamo già visto come fornire tali metadata di configurazione basati su file XML al Container di Spring, andiamo ora a vedere un altro esempio di file di configurazione XML che contiene le varie definizioni di diversi bean (che includono le proprietà di lazy inizialization, metodo di inizializzazione e metodi di distruzione):

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- A simple bean definition -->
9     <bean id="..." class="...">
10       <!-- collaborators and configuration for this bean go here -->
11    </bean>
12
13    <!-- A bean definition with lazy init set on -->
14    <bean id="..." class="..." lazy-init="true">
15       <!-- collaborators and configuration for this bean go here -->
16    </bean>
17
18    <!-- A bean definition with initialization method -->
19    <bean id="..." class="..." init-method="...">
20       <!-- collaborators and configuration for this bean go here -->
21    </bean>
22
23    <!-- A bean definition with destruction method -->
24    <bean id="..." class="..." destroy-method="...">
25       <!-- collaborators and configuration for this bean go here -->
26    </bean>
27
28    <!-- more bean definitions go here -->
29
30 </beans>
```

Potete rivedere l'articolo relativo al progetto di Hello World in Spring per capire come definire, configurare e creare i beans di Spring.

Per evitare confusione, la configurazione dei bean di Spring mediante il sistema basato sulle Annotation sarà discussa in un altro articolo dopo aver fissato qualche altro concetto fondamentale.

7 – Ciclo di vita dei Bean in Spring: Inizializzazione e Distruzione

Posted on **5 aprile 2013** Views: 3160

Il ciclo di vita dei bean del Framework Spring è pensato per essere facile da capire. Quando un bean viene istanziato potrebbe essere necessario effettuare alcune inizializzazioni in modo da portare tale bean in uno stato utilizzabile.

Similmente, quando un bean non è più necessario viene rimosso dal container e sono richieste alcune operazioni di cleanup.

Anche se ci sono una serie di attività che si pongono tra il momento in cui un bean viene istanziato ed il momento in cui tale bean viene distrutto in questo articolo saranno discusse solamente due importanti metodi di callback appartenenti alla gestione del ciclo di vita dei bean in Spring: quelli relativi al momento di inizializzazione di un bean ed al momento di distruzione di tale bean.

Per definire l'installazione e la distruzzione di un bean, dobbiamo semplicemente dichiarare il tag <bean> con un parametro **init-method** (caso in cui gestiamo la creazione e l'inizializzazione di un bean) e/o con un parametro **destroy-method** (caso in cui gestiamo la distruzione di un bean ed il relativo cleanup).
L'attributo **init-method** specifica un metodo che verrà chiamato sul bean **subito dopo che questo è stato istanziato**.
L'attributo **destroy-method** specifica un metodo che verrà chiamato **subito prima che un bean venga rimosso dal Container**.
Metodo di callback per l'INIZIALIZZAZIONE:

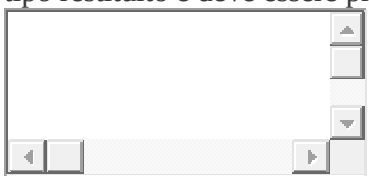
L'interfaccia **org.springframework.beans.factory.InitializingBean** specifica un solo metodo, questo:**void afterPropertiesSet() throws Exception;**

Quindi, basterà implementare tale interfaccia ed il compito relativo all'inizializzazione sarà svolto all'interno del metodo **afterPropertiesSet()** , come mostrato di seguito:

```
1 public class ExampleBean implements InitializingBean {

2     public void afterPropertiesSet() {

3         // do some initialization work

4     }
```

```
5 }
```

Considerando di essere nel caso in cui usiamo metadata di configurazione in formato XML per configurare la nostra applicazione, possiamo usare l'attributo **init-method** per specificare il nome del metodo che gestisce l'inizializzazione di un bean (tale metodo deve avere come signature void come tipo restituito e deve essere privo di argomenti). Per esempio:



```
1 <beanid="exampleBean"
2         class="examples.ExampleBean"init-method="init"/>
```

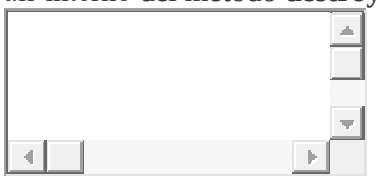
In questo caso stiamo definendo un bean relativo alla classe examples.ExampleBean ed identificato univocamente dall'id=”exampleBean” per cui è definito un metodo di inizializzazione, di seguito la definizione della classe:

```
1 public class ExampleBean {
2     public void init() {
3         // do some initialization work
4     }
5 }
```

Metodo di callback per la DISTRUZIONE:

L'interfaccia ***org.springframework.beans.factory.DisposableBean*** specifica un solo metodo, questo:**void destroy() throws Exception;**

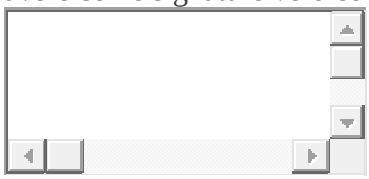
Quindi, basterà implementare tale interfaccia ed il compito relativo all'inizializzazione sarà svolto all'interno del metodo **destroy()** , come mostrato di seguito:



```
1 public class ExampleBean implements DisposableBean {
2     public void destroy() {
3         // do some destruction work
```

```
4     }
5 }
```

Considerando di essere nel caso in cui usiamo metadata di configurazione in formato XML per configurare la nostra applicazione, possiamo usare l'attributo **destroy-method** per specificare il nome del metodo che gestisce il cleanup di un bean che deve essere rimosso dal Container (tale metodo deve avere come signature void come tipo restituito e deve essere privo di argomenti). Per esempio:



```
1 <bean id="exampleBean"
2         class="examples.ExampleBean" destroy-method="destroy"/>
```

In questo caso stiamo definendo un bean relativo alla classe examples.ExampleBean ed identificato univocamente dall'id=”exampleBean” per cui è definito un metodo di distruzione, di seguito la definizione della classe:

```
1 public class ExampleBean {
2     public void destroy() {
3         // do some destruction work
4     }
5 }
```

Se stiamo usando lo IoC Container di Spring in un ambiente che non è quello relativo ad una Web Application (ad esempio in un ambiente desktop) si registrerà uno shutdown hock con la JVM. In questo modo si garantisce un arresto regolare e vengono chiamati i metodo distruttori sui bean che rappresentano dei singletono, così da rilasciare tutte le risorse.

Esempio pratico:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.
Aprire l'elemento Maven e selezionare “Maven Project”. Clickate Next.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione “Use default Workspace location”, lasciare inalterate tali opzioni e clickate su Next.

Nella seconda finestra dello wizard lasciate il default ArtifactId settato su “maven-archetype-quickstart” e clickate su Next.

Inserite i seguenti parametri:

Group Id : org.andrea.myexample
Artifact Id: myInitDestroyExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file **pom.xm** e clickate sul tab “**Dependencies”** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All’interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.myInitDestroyExample** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
All’interno di tale package creaiamo una classe dandogli nome **HelloWorld.java**:

```
1  package org.andrea.myexample.myInitDestroyExample;
2
3  public class HelloWorld {
4
5     private String message; // Messaggio da visualizzare
6
7     /*
8      * Metodo usato per iniettare il valore della variabile message definito nel
9      * file Beans.xml
```

```
10     */
11    public void setMessage(String message) {
12        this.message = message;
13    }
14
15    public void getMessage() {
16        System.out.println("Your Message : " + message);
17    }
18
19    // Metodo di inizializzazione eseguito subito dopo la creazione del bean:
20    public void init() {
21        System.out.println("Bean is going through init.");
22    }
23
24    // Metodo destroy() eseguito subito prima della distruzione del bean:
25    public void destroy() {
26        System.out.println("Bean will destroy now.");
27    }
28 }
```

Di seguito è presentato il codice del file **MainApp.java**. In tale codice dobbiamo registrare lo shutdown hook tramite il metodo **registerShutdownHook()** che è dichiarato nella classe astratta **AbstractApplicationContext**. Tale metotodo assicurerà uno shutdown ottimale e chiamerà tutti gli adeguati metodi distruttori.

La classe **AbstractApplicationContext** è un'implementazione astratta dell'interfaccia **ApplicationContext** e si occupa di implementare alcune funzionalità comuni per il contesto. Usando il designe pattern Template Method richiede che vengano create classi concrete che implementano i suoi metodi astratti.

Quì maggiori informazioni su tale classe astratta: http://static.springsource.org/spring-framework/docs/3.2.0.M2/api/org/springframework/context/support/AbstractApplicationContext.html

Nello stesso package create un'altra classe chiamata **MainApp.java**:

```
1  package org.andrea.myexample.myInitDestroyExample;
2
3  import org.springframework.context.support.AbstractApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  /* Classe principale contenente il metodo main) */
7  public class MainApp {
8     public static void main(String[] args) {
9        /*
10        * Crea il contesto in base alle impostazioni dell'applicazione definite
11        * nel file Beans.xml
12        */
13       AbstractApplicationContext context = new ClassPathXmlApplicationContext(
14             "Beans.xml");
15       /*
16        * Recupera un bean avente id="helloWorld" nel file di configurazione
17        * Beans.xml
18        */
19       HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
20
21       // Invoco il metodo che stampa il messaggio su tale oggetto
22       obj.getMessage();
23
24       /*
25        * Esegue lo shutdown dell'applicazione: registra un "gancio" all'evento
26        * di shutdown della JVM e prima che tale evento si verifichi esegue lo
27        * shudwon del contesto dell'applicazione distruggendo tutti i bean
28        * dell'applicazione e rilasciando lo spazio in memoria prima che si
29        * verifichi lo shudwon della JVM
30        */
31       context.registerShutdownHook();
32    }
```

```
33 }
```

Da notare che il contesto viene dichiarato come **AbstractApplicationContext** (e costruito con il tipo concreto ClassPathXmlApplicationContext) così da poter richiamare il metodo `registerShutdownHook()`che crea un "gancio" all'evento di shutdown della JVM e, prima che tale shutdown avvenga, provvede ad eseguire lo shutdown del contesto dell'applicazione chiamando tutti i metody di cleanup dei vari bean e a rilasciando le risorse in memoria.
Di seguito è mostrato il contenuto del file di configurazione **Beans.xml**:

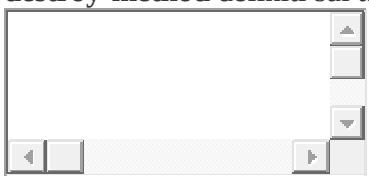
```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8    <bean id="helloWorld"
9        class="org.andrea.myexample.myInitDestroyExample.HelloWorld"
10       init-method="init" destroy-method="destroy">
11       <property name="message" value="Hello World!"/>
12   </bean>
13
14 </beans>
```

Una volta fatto ciò, andiamo ad eseguire l'applicazione (eseguendo la classe MainApp come applicazione Java) e dovremmo ottenere il seguente output:

Bean is going through init.
Your Message : Hello World!
Bean will destroy now.
Per capire cosa sia successo basta guardare i metadati di configurazione all'interno del file di configurazione dell'applicazione Beans.xm**.**

In tale file configuriamo un solo bean usato dalla nostra applicazione. Tale bean corrisponde alla classe HelloWorld ed è identificato univocamente dall'ID "helloWorld".

Per tale metodo definiamo 2 metodi di callback relativi all'inizializzazione da eseguire subito dopo la creazione dell'oggetto da parte della factory (metodo init) e relativo al cleanup dell'applicazione da eseguire subito prima della distruzione dell'oggetto (metodo destroy).
Pertanto nella nostra classe HelloWorld definiamo tali metodi e sarà il framework stesso a chiamare il metodo init() subito dopo aver creato l'oggetto ed il metodo destroy() appena prima della distruzione del bean (che avviene prima dello shutdown della JVM al termine dell'applicazione).
Metodo di default per l'inizilizzazione e la distruzione dei bean:
Nel caso si abbiano molti bean diversi aventi metodi di inizializzazione e\o distruzione che hanno lo stesso nome, non dobbiamo per forza dichiarare gli attributi init-method e\o destroy-method nelle dichiarazioni di ogni singolo bean all"interno del file di configurazione xml. A tale scopo, il framework fornisce la possibilità di configurare tali situazioni usando gli attributi **default-init-method** e\o **default-destroy-method** definiti sul tag **<bean>**, come mostrato di seguito:



```
1  <beans xmlns="http://www.springframework.org/schema/beans"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://www.springframework.org/schema/beans
4        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
5     default-init-method="init" default-destroy-method="destroy">
6
7     <bean id="..." class="...">
8        <!-- collaborators and configuration for this bean go here -->
9     </bean>
10
11 </beans>
```

8 – Bean Post Processor

Posted on **5 aprile 2013** Views: 794

L'interfaccia **BeanPostProcessor** definisce alcuni metodi di callback che possono essere implementati per fornire, ad esempio, una logica di istanziazione ed una logica di risoluzione delle dipendenze

personalizzate. Usando tali metodi possiamo inoltre implementare delle logiche personalizzate su cosa accade dopo che il Container di Spring ha finito di istanziare, configurare ed inizializzare un bean inserendo in esso una o più implementazioni dei metodi dell'interfaccia **BeanPostProcessor**.

I metodi dell'interfaccia BeanPostProcessors operano sulle istanze dei bean (sugli oggetti), il che significa che lo IoC Container di Spring istanzia un bean e poi l'interfaccia BeanPostProcessor fa il suo lavoro.

L'ApplicationContext rileverà automaticamente ogni bean che è definito come un'implementazione dell'interfaccia BeanPostProcessor e registra tali bean come post-processors, per poi essere chiamati in modo appropriato dal Container al momento della creazione del bean.

Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.

Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.

Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.

Inserite i seguenti parametri:

Group Id rg.andrea.myexample

Artifact Id: myBeanPostProcessorExample

Version: lasciate 0.0.1-SNAPSHOT

Clickate su **Finish**.

Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).

Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

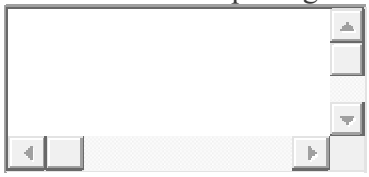
```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myBeanPostProcessorExample</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myBeanPostProcessorExample</name>
11   <url>http://maven.apache.org</url>
```

```
12
13  <properties>
14    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package **org.andrea.myexample.myInitDestroyExample** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
All'interno di tale package creaiamo una classe dandogli nome **HelloWorld.java**:



```
1  package org.andrea.myexample.myBeanPostProcessorExample;
2
3  public class HelloWorld {
4
5     private String message;    // Messaggio da visualizzare
6
7     /* Metodo usato per iniettare il valore della variabile message definito
```

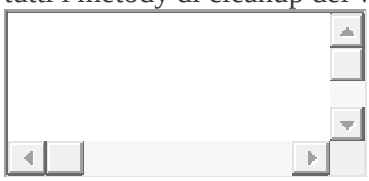
```
8         * nel file Beans.xml */
9      public void setMessage(String message) {
10        this.message = message;
11     }
12
13     public void getMessage() {
14        System.out.println("Your Message : " + message);
15     }
16
17     // Metodo di inizializzazione eseguito subito dopo la creazione del bean:
18     public void init() {
19        System.out.println("Bean is going through init.");
20     }
21
22     // Metodo destroy() eseguito subito prima della distruzione del bean:
23     public void destroy() {
24        System.out.println("Bean will destroy now.");
25     }
26  }
```

Andiamo ora a vedere un esempio veramente basilare di implementazione di **BeanPostProcessor**,**tale implementazione stampa il nome del bean prima e dopo l'inizializzazione di ogni bean**. Possiamo implementare una logica più complessa prima e dopo l'istanziazione di un bean perchè si ha accesso all'oggetto bean in entrambi i metodi di post processor.
Di seguito il contenuto del file **InitHelloWorld.java**:

```
1  package org.andrea.myexample.myBeanPostProcessorExample;
2
3  import org.springframework.beans.factory.config.BeanPostProcessor;
4  import org.springframework.beans.BeansException;
5
6  // Classe che implementa l'interfaccia BeanPostProcessor
7  public class InitHelloWorld implements BeanPostProcessor {
8
9     // Metodo che viene eseguito prima dell'inizializzazione di un bean:
10    public Object postProcessBeforeInitialization(Object bean,
11          String beanName) throws BeansException {
12
13       System.out.println("BeforeInitialization : " + beanName);
14       return bean;  // you can return any other object as well
15    }
16
17    // Metodo che viene eseguito dopo l'inizializzazione di un bean:
18    public Object postProcessAfterInitialization(Object bean,
19          String beanName) throws BeansException {
20
21       System.out.println("AfterInitialization : " + beanName);
22       return bean;  // you can return any other object as well
23    }
24
25 }
```

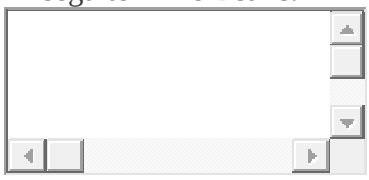
Di seguito viene mostrato il contenuto del file **MainApp.java**.
Da notare che il contesto viene dichiarato come **AbstractApplicationContext** (e costruito con il tipo concreto ClassPathXmlApplicationContext) così da poter richiamare il

metodo **registerShutdownHook**che crea un “gancio” all’evento di shutdown della JVM e, prima che tale shutdown avvenga, provvede ad eseguire lo shutdown del contesto dell’applicazione chiamando tutti i metody di cleanup dei vari bean e a rilasciando le risorse in memoria.



```
1  package org.andrea.myexample.myBeanPostProcessorExample;
2
3  import org.springframework.context.support.AbstractApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  /* Classe principale contenente il metodo main) */
7  public class MainApp {
8
9     public static void main(String[] args) {
10
11        /* Crea il contesto in base alle impostazioni dell'applicazione definite
12         * nel file Beans.xml  */
13        AbstractApplicationContext context = new ClassPathXmlApplicationContext(
14              "Beans.xml");
15
16        /* Recupera un bean avente id="helloWorld" nel file di configurazione
17         * Beans.xml */
18        HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
19
20        // Invoco il metodo che stampa il messaggio su tale oggetto
21        obj.getMessage();
22
23        /* Esegue lo shutdown dell'applicazione: registra un "gancio"
24         * all'evento di shutdown della JVM e prima che tale evento si verifichi
25         * esegue lo shudwon del contesto dell'applicazione distruggendo tutti
26         * i bean dell'applicazione e rilasciando lo spazio in memoria prima che
27         * si verifichi lo shudwon della JVM
28         *
29         */
30        context.registerShutdownHook();
31     }
32 }
```

Di seguito il file **Beans.xml** contenente i metadati di configurazione dell’applicazione:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <bean id="helloWorld" class="org.andrea.myexample.myBeanPostProcessorExample.HelloWorld"
9        init-method="init" destroy-method="destroy">
10       <property name="message" value="Hello World!"/>
11    </bean>
12
13    <bean class="org.andrea.myexample.myBeanPostProcessorExample.InitHelloWorld" />
14
15 </beans>
```

In tale file di configurazione, oltre al consueto bean relativo alla classe HelloWorld, stiamo dichiarando anche di istanziare un bean relativo alla classe **InitHelloWorld** che rappresenta appunto la nostra classe che implementa l'interfaccia **BeanPostProcessor** e l'implementazione dei suoi due metodi di post processamento di un bena: **postProcessBeforeInitialization()** e**postProcessAfterInitialization()**.
Pertanto ciò che avviene è che:

1. Viene istanziato il bean **InitHelloWorld** che rappresenta l'implementazione della classe**BeanPostProcessor** e tale bean viene **riconosciuto automaticamente dal contesto come implmentazione dell'interfaccia BeanPostProcessor e viene registrato come post processor**.
2. Viene istanziato il bean **HelloWorld** e gli viene iniettato il valore "Hello World" all'interno della variabile message.
3. Viene eseguito il metodo **postProcessBeforeInitialization()** della classe **BeanPostProcessor()**che stampa un messaggio.
4. Viene eseguito il metodo **init()** della classe **HelloWorld** che stampa un messaggio.
5. Viene eseguito il metodo **postProcessAfterInitialization()** della classe **BeanPostProcessor()**.
6. Viene eseguito il metodo **destroy()** della classe **HelloWorld**.

Il metodo **postProcessBeforeInitialization()** riceve un bean **prima dell'esecuzione di ogni eventuale suo metodo di callback per l'inizializzazione**.
Le propietà del bean verranno valorizzate. Il bean ritornato può essere un wrapper del bean originale.
Il metodo **postProcessAfterInitialization()** riceve un bean **dopo l'esecuzione di ogni eventuale suo metodo di callback per l'inizializzazione**.
Le propietà del bean verranno valorizzate. Il bean ritornato può essere un wrapper del bean originale.
Pertanto l'output di tale applicazione sarà:

BeforeInitialization : helloWorld
Bean is going through init.
AfterInitialization : helloWorld
Your Message : Hello World!
gen 26, 2013 1:11:47 AM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@11613fe7: startup date [Sat Jan 26 01:11:46 CET 2013]; root of context hierarchy
Bean will destroy now.

9 – Ereditarietà dei Bean di Spring

Posted on **5 aprile 2013** Views: 1004

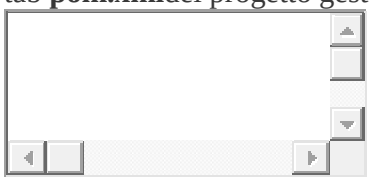
La definizione dei bean può contenere molte informazioni relative alla loro configurazione, alcuni esempi sono: gli argomenti del costruttore, i valori delle proprietà ed informazioni specifiche relative al container come i metodi di inizializzazione e così via.

La definizione di un bean figlio eredita i dati di configurazione dal bean padre. La definizione di un bean figlio può eseguire l'override di alcuni valori o semplicemente aggiungerne altri.

L'ereditarierà della definizione dei Bean di Spring non ha nulla a che fare con l'ereditarietà delle classi Java, ma il concetto di ereditarietà è lo stesso. Possiamo definire un parent bean come un template ed i bean figli possono ereditare le configurazioni richieste dal bean padre.
Quando usiamo metadata di configurazione in formato XML, indichiamo la definizione di un bean figlio usando l'attributo **parent**, il valore di tale attributo specifica così qual'è il bean padre.
Esempio:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.
Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group Id rg.andrea.myexample
Artifact Id: myBeanInheritanceExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myBeanInheritanceExample</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myBeanInheritanceExample</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
```

```
14    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

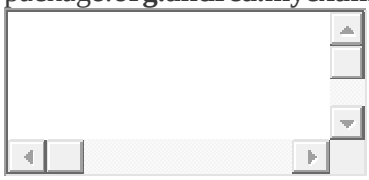
Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.myBeanInheritanceExample** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
Di seguito è presentato il contenuto del file di configurazione **Beans.xml** nel quale è stato definito un **bean** identificato tramite l'**ID=helloWorld** e corrispondente ad una **classe Java HelloWorld**. Tale bean ha due proprietà: **message1** e **message2**.
Inoltre in tale file di configurazione è definito un **secondo bean** identificato tramite l'**ID=helloIndia** e corrispondente ad una **classe Java HelloIndia**.
Tale bean è definito come figlio del bean avente ID=helloWorld e per fare ciò abbiamo usato l'attributo parent.
Possiamo inoltre notare che **il bean figlio eredita la prorietà message2 per come è definita nel bean padre, esegue l'override della proprrietà message1 ed introduce una nuova proprietà message3**.
Tale file di configurazione andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
<?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/schema/beans/spring-beans-3.0.xsd">

   <bean id="helloWorld" class="org.andrea.myexample.myBeanInheritanceExample.HelloWorld">
      <property name="message1" value="Hello World!"/>
      <property name="message2" value="Hello Second World!"/>
   </bean>

   <bean id="helloIndia" class="org.andrea.myexample.myBeanInheritanceExample.HelloIndia"
      parent="helloWorld">
      <property name="message1" value="Hello India!"/>
      <property name="message3" value="Namaste India!"/>
   </bean>

</beans>
```

Di seguito il codice del file **HelloWorld.java** (che andrà inserito nel nostro unico package:**org.andrea.myexample.myBeanInheritanceExample**):



```
package org.andrea.myexample.myBeanInheritanceExample;

public class HelloWorld {

   // Le due proprietà valorizzate nel file Beans.xml per tale bean:
   private String message1;
   private String message2;

   // Metodo getter & setter relativi a tali proprietà:
   public void setMessage1(String message) {
      this.message1 = message;
   }

   public void setMessage2(String message) {
      this.message2 = message;
   }

   public void getMessage1() {
      System.out.println("World Message1 : " + message1);
   }

   public void getMessage2() {
      System.out.println("World Message2 : " + message2);
   }
}
```

Di seguito il codice del file **HelloIndia.java** (che andrà inserito nel nostro unico package:**org.andrea.myexample.myBeanInheritanceExample**):

```
1  package org.andrea.myexample.myBeanInheritanceExample;
2
3  public class HelloIndia {
4
5      /*
6       * Proprietà della classe HelloIndia: Per message1 viene eseguito
7       * l'override, message2 viene ereditata e message3 è stata introdotta come
8       * nuova proprietà
9       */
10     private String message1;
11     private String message2;
12     private String message3;
13
14     // Metodi Getter & Setter relativi a tali proprietà:
15     public void setMessage1(String message) {
16         this.message1 = message;
17     }
18
19     public void setMessage2(String message) {
20         this.message2 = message;
21     }
22
23     public void setMessage3(String message) {
24         this.message3 = message;
25     }
26
27     public void getMessage1() {
28         System.out.println("India Message1 : " + message1);
29     }
30
31     public void getMessage2() {
32         System.out.println("India Message2 : " + message2);
33     }
34
35     public void getMessage3() {
36         System.out.println("India Message3 : " + message3);
37     }
38 }
```

Di seguito il codice della classe principale **MainApp.java** (che andrà inserito nel nostro unico package:**org.andrea.myexample.myBeanInheritanceExample**):

```
1  package org.andrea.myexample.myBeanInheritanceExample;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale contenente il metodo main() che viene eseguito per primo
7  public class MainApp {
8
9      public static void main(String[] args) {
10
11         /* Crea il contesto in base alle impostazioni dell'applicazione definite
12          * nel file Beans.xml  */
```

```
13        ApplicationContext context = new ClassPathXmlApplicationContext(
14                "Beans.xml");
15
16        // Recupera dal contesto un bean avente id="helloWorld":
17        HelloWorld objA = (HelloWorld) context.getBean("helloWorld");
18
19        // Stampa i valori delle proprietà message1 e message2:
20        objA.getMessage1();
21        objA.getMessage2();
22
23        // Recupera dal contesto un bean avente id="helloIndia":
24        HelloIndia objB = (HelloIndia) context.getBean("helloIndia");
25
26        // Stampa i valori delle proprietà message1 e message2 e message3:
27        objB.getMessage1();
28        objB.getMessage2();
29        objB.getMessage3();
30    }
31 }
```

Una volta finita la creazione del codice sorgente dei bean e del file di configurazione Beans.xml, andiamo ad eseguire la nostra applicazione (eseguendo la classe MainApp). L'output ottenuto nel tab **Console** di STS\Eclipse sarà il seguente:

gen 28, 2013 5:12:03 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@792bf678 : startup date [Mon Jan 28 17:12:03 CET 2013]; root of context hierarchy
gen 28, 2013 5:12:03 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [Beans.xml]
gen 28, 2013 5:12:03 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@2c913a58 : defining beans [helloWorld,helloIndia]; root of factory hierarchy
World Message1 : Hello World!
World Message2 : Hello Second World!
India Message1 : Hello India!
India Message2 : Hello Second World!
India Message3 : Namaste India!

Analizzando tale output possiamo notare che è diviso in due parti:

1. **MESSAGGI INFORMATIVI DEL FRAMEWORK:** mostrano il tipo di contesto creato, il caricamento del file Beans.xml contenente le configurazioni dei bean, la creazione dei nostri due bean come singleton.

2. **L'OUTPUT DELL'APPLICAZIONE:** in cui possiamo notare come il valore della proprietà message1 ha subito l'override nel secondo bean, come la prima proprietà sia stata ereditata nel secondo bean e di come la terza proprietà sia stata definita ex novo nel secondo bean.

Bean Definition Template:

Possiamo creare un template di definizione dei bean che può essere usato da altre definizioni di bean figli senza dover fare ulteriori sforzi. Per definire un Template di definizione dei bean, **non dobbiamo specificare l'attributo class ma dobbiamo usare l'attributo abstract settato con valore true**, ecco come:

```
<?xmlversion="1.0"encoding="UTF-8"?>

  <beansxmlns="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/schema/beans/spring-beans-3.0.xsd">

    <beanid="beanTeamplate"abstract="true">
      <propertyname="message1"value="Hello World!"/>
      <propertyname="message2"value="Hello Second World!"/>
      <propertyname="message3"value="Namaste India!"/>
    </bean>

    <beanid="helloIndia"class="com.tutorialspoint.HelloIndia"
      parent="beanTeamplate">
      <propertyname="message1"value="Hello India!"/>
      <propertyname="message3"value="Namaste India!"/>
    </bean>

  </beans>
```

10 – Iniezione delle dipendenze in Spring (Dependency Injection)

Posted on **5 aprile 2013** Views: 1057

Tutte le applicazioni Java sono formate da un insieme di oggetti che lavorano insieme per implementare ciò che l'utende vede come un'applicazione funzionante. Quando scriviamo un'applicazione Java complessa, le classi dell'applicazione dovrebbero risultare il più indipendenti possibili l'una dall'altra al fine di aumentare la possibilità di riusare queste classi e per poterle testare indipendentemente dalle altre classi quando eseguiamo gli unit test. L'Iniezione delle Dipendenze (comunemente nota come Dependency Injection o come Wiring) aiuta ad "incollare" queste classi le une con le altre mantenendole però indipendenti.

Consideriamo di avere un'applicazione dotata di una componente che implementa un editor testuale e supponiamo di voler fornire la funzionalità relativa al controllo ortografico. Un possibile codice standard potrebbe essere:

```
1 public class TextEditor {
2     private SpellChecker spellChecker;
3
4     public TextEditor() {
5         spellChecker = new SpellChecker();
6     }
7 }
```

Ciò che abbiamo fatto in questa classe TextEditor è stato **creare una dipendenza tra la classe TextEditor** (rappresentante il nostro editor di testi) **e la classe SpellChecker** (che rappresenta l'implementazione di un correttore ortografico che viene USATO dalla classe TextEditor).
In uno scenario relativo al pattern Inversion Of Control (IoC) al suo posto avremmo qualcosa del genere:

```
1 public class TextEditor {
2     private SpellChecker spellChecker;
3
4     public TextEditor(SpellChecker spellChecker) {
5         this.spellChecker = spellChecker;
6     }
7 }
```

In questo caso la classe TextEditor non si deve preoccupare della creazione della classe SpellChecker. In questo scenario la classe SpellChecker verrà creata (da una factory) ed implementata in maniera totalmente indipendente e verrà fornita alla classe TextEditor nel momento in cui essa verrà istanziata. L'intera procedura sarà controllota dal Framework Spring.

In questo secondo caso abbiamo rimosso totalmente il controllo dalla classe TextEditor e lo abbiamo riposto da qualche altra parte (ad esempio nel file di configurazione XML visto nei precedenti articoli). La dipendenza (ad esempio la classe SpellChecker) sarà creata da una factory (come tutti i bean facenti parte di un'applicazione Spring) e sarà iniettata all'interno della classe che la usa (ad esempio TextEditor) tramite uno strumento chiamato **Class Constructor**.
Potete notare come il controllo del flusso è stato invertito dalla Dependency Injection poichè abbiamo deletegato la gestione delle dipendenze ad un sistema esterno.

Un secondo metodo per iniettare la dipendenza potrebbe avvenire attraverso i **Metodi Setter** della classe Text Editor in cui creeremo un'istanza della classe SpellChecker e tale istanza sarà usata per chiamare i metodi setter per inizializzare le proprietà dell"oggetto TextEditor (vedremo in seguito un esempio concreto).
Ricapitolando, l'Iniezione delle Dipendenze può essere clasificata nelle seguenti due varianti significative (che analizzeremo nel dettaglio nei prossimi due articoli forniti di esempi concreti):

1. **Iniezione della dipendenza basata sul costruttore:** E' rappresentata dal caso in cui il container invoca il **Class Constructor** passandogli un certo numero di argomenti, ognuno dei quali rappresenta una dipendenza da un'altra classe.
2. **Iniezione della dipendenza basata sui Metodi Setter:** E' rappresentata dal caso in cui il container invoca dei metodi Setter sui nostri bean, dopo aver invocato un costruttore privo di argomenti oppure un metodo statico privo di argomenti appartenente ad una factory per istanziare i nostri bean.

Volendo possiamo usare insieme entrambi i metodi ma è una buona regola usare la versione basata su costruttore per gestire le dipendenze obbligatorie ed usre la versione basata sui metodi setter per gestire le dipendenze opzionali.

Vedremo nei prossimi due articoli dotati di esempi pratici come il codice è più pulito usando la Dependency Injection e creando di conseguenza un maggior disaccoppiando tra le classi (un maggior grado di indipendenza tra le classi). Così facendo accade che un oggetto non si preoccupa di gestire le sue dipendenze e non conosce la locazione o la classe delle dipendenze ma tutto ciò è gestito dal Framework Spring.

11 – Iniezione della Dipendenza basata su Costruttore

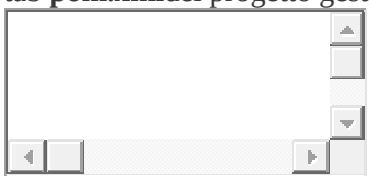
L'iniezione della dipendenza basata sul costruttore viene realizzata quando il container invoca un costruttore di classi con un certo numero di argomenti, ognuno dei quali rappresenta una dipendenza da un'altra classe.

Esempio:
Il seguente esempio mostra una classe **TextEditor** in cui si possono iniettare altre classi solamente usando l'iniezione della dipendenza mediante costruttore.
Lavorando con STS\Eclipse seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.

Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.
Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group Id o rg.andrea.myexample
Artifact Id: myConstructorDependencyInjection
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-beans**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:



```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.andrea.myexample</groupId>
 <artifactId>myConstructorDependencyInjection</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>jar</packaging>

 <name>myConstructorDependencyInjection</name>
 <url>http://maven.apache.org</url>

 <properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

 <dependencies>
   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>3.8.1</version>
     <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.1.1.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>3.1.1.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
```

```
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38     </dependency>
39     <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43     </dependency>
44   </dependencies>
45 </project>
```

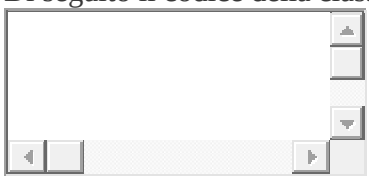
Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package **org.andrea.myexample.myConstructorDependencyInjection** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.

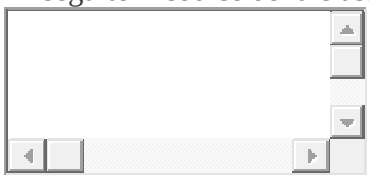
All'interno dell'unico package **org.andrea.myexample.myConstructorDependencyInjection** create le seguenti 3 classi Java: **TextEditor.java**, **SpellChecker.java**, **MainApp.java**

Di seguito il codice della classe **TextEditor.java**:



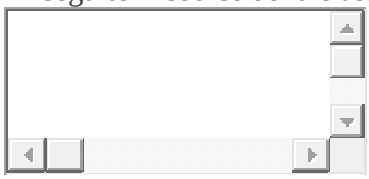
```
1  package org.andrea.myexample.myConstructorDependencyInjection;
2
3  public class TextEditor {
4
5     // Oggeto che rappresenta il correttore ortografico:
6     private SpellChecker spellChecker;
7
8     /* Costruttore della classe TextEditor:
9      * @param: un oggetto che rappresenta il correttore
10    *        ortografico (SpellCheker)
11    * @return: il nuovo oggetto TextEditor
12    */
13    public TextEditor(SpellChecker spellChecker) {
14       System.out.println("Inside TextEditor constructor.");
15       this.spellChecker = spellChecker;
16    }
17
18    /* Metodo chiamato per la correzione ortografica.
19     * Chiama un metodo sull'oggetto SpellChecker
20     */
21    public void spellCheck() {
22       spellChecker.checkSpelling();
23    }
24 }
```

Di seguito il codice della classe subordinata **SpellChecker.java** (TextEditor dipende da tale classe):



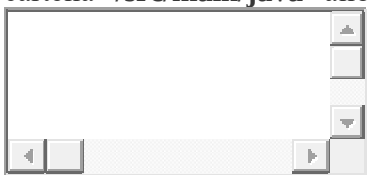
```
1  package org.andrea.myexample.myConstructorDependencyInjection;
2
3  // Classe che rappresenta il correttore ortografico:
4  public class SpellChecker {
5
6     // Costruttore:
7     public SpellChecker() {
8        System.out.println("Inside SpellChecker constructor.");
9     }
10
11    // Metodo che esegue il controllo ortografico:
12    public void checkSpelling() {
13       System.out.println("Inside checkSpelling.");
14    }
15
16 }
```

Di seguito il codice della classe principale **MainApp.java**:



```
1  package org.andrea.myexample.myConstructorDependencyInjection;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale contenente il metodo main()
7  public class MainApp {
8
9     public static void main(String[] args) {
10       /* Crea il contesto in base alle impostazioni
11        * dell'applicazione definite nel file Beans.xml */
12       ApplicationContext context =
13             new ClassPathXmlApplicationContext("Beans.xml");
14
15       /* Recupera un bean avente id="helloWorld" nel file
16        * di configurazione Beans.xml */
17       TextEditor te = (TextEditor) context.getBean("textEditor");
18
19       // Invoca il metodo per la correzione ortografica:
20       te.spellCheck();
21    }
22 }
```

Di seguito il contenuto del file di configurazione **Beans.xml** che come al solito andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- Definition for textEditor bean -->
9     <bean id="textEditor" class="org.andrea.myexample.myConstructorDependencyInjection.TextEditor">
10       <constructor-arg ref="spellChecker" />
```

```
11    </bean>
12
13    <!-- Definition for spellChecker bean -->
14    <bean id="spellChecker" class="org.andrea.myexample.myConstructorDependencyInjection.SpellChecker">
15    </bean>
16
17 </beans>
```

Come si può vedere in questo file di configurazione vengono definiti due bean:

1. Il bean identificato dall'**ID=spellChecker** che corrisponde alla classe **SpellCheker**. Tale bean sarà usato come dipendenza del prossimo bean.
2. Il bean identificato dall'**ID=textEditor** che corrisponde alla classe **TextEditor**. Tale bean prenderà il precedente bean come dipendenza.

Una volta finito di implementare il codice sorgente di tutte le classe e del file di configurazione eseguiamo la classe MainAppp ed otterremo il seguente output nel tab Console di STS\Eclipse:

Inside SpellChecker constructor.
Inside TextEditor constructor.
Inside checkSpelling.

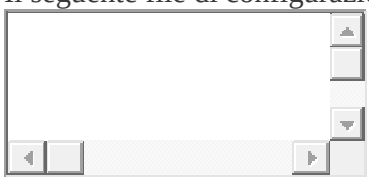
Andando ad analizzare il significato del precedente esempio possiamo vedere come il Framework Spring istanzi prima il bean **SpellChecker** e quando và ad istanziare il bean **TextEditor** inietta la dipendenza (l'indirizzo) del bean **SpellChecker** all'interno del costruttore al momento della costruzione di un oggetto **TextEditor**. Per fare ciò abbiamo usato il tag **<constructor-arg ref="spellChecker"/>** all'interno della definizione del bean **TextEditor**.
Tale tag specifica, all'interno della definizione del bean identificato dall'**ID=textEditor**, gli argomenti da passare al costruttore della relativa classe **TextEditor**. Sarà il framework Spring ad occuparsi di istanziare prima il bean rappresentante la dipendendenza e di andarlo ad iniettare al momento della costruzione del bean che usa tale dipendenza.

Risoluzione degli argomenti del costruttore:
Potrebbe esserci un'ambiguità relativa al passaggio degli argomenti nel caso ci siano più parametri. Per risolvere questa ambiguità, l'ordine in cui gli argomenti del costruttore sono definiti nella definizione dei bean è l'ordine in cui questi argomenti sono forniti allo specifico costruttore. Per capire questo concetto consideriamo la seguente classe:

```
1 package x.y;
2
3 public class Foo {
4     public Foo(Bar bar, Baz baz) {
5         // ...
```

```
6     }
7 }
```

Tale classe ha un costruttore che prende due parametri di input: **Bar bar** e **Baz baz**.
Il seguente file di configurazione funziona bene:



```
1     <beans>
2        <beanid="foo"class="x.y.Foo">
3           <constructor-argref="bar"/>
4           <constructor-argref="baz"/>
5        </bean>
6
7        <beanid="bar"class="x.y.Bar"/>
8        <beanid="baz"class="x.y.Baz"/>
9     </beans>
```

In tale file di configurazione definisce il bean relativo alla classe **Foo**. In tale definizione di bean stiamo dicendo che saranno iniettati come parametri di input del costruttore un bean identificato dall'**ID=bar** ed un bean indentificato dall'**ID=baz**.
Più sotto, nel file di configurazione, **definiamo nello stesso ordine in cui si presentano i parametri iniettati nel precedente costruttore**, i bean che rappresentano le due dipendenze.
Andiamo ora a vedere un altro caso in cui passiamo al costruttore tipi differenti. Consideriamo quindi la seguente classe:

```
1 package x.y;
2
3 public class Foo {
4     public Foo(int year, String name) {
5         // ...
6     }
7 }
```

Nel caso di tipi semplici, il container può anche usare il **matching dei tipi** se nel costruttore specifichiamo esplicitamente il tipo degli argomenti, basta usare l'**attributo type**. Ad esempio:

```
1     <beans>
2
3        <bean id="exampleBean" class="examples.ExampleBean">
4           <constructor-arg type="int" value="2001"/>
5           <constructor-arg type="java.lang.String" value="Zara"/>
6        </bean>
7
```

```
8    </beans>
```

Infine esiste un altro modo per passare gli argomenti al costruttore (e tale metodo risulta essere il migliore): usare l'**attributo index** per **specificare esplicitamente l'indice dell'argomento del costruttore**. L'indice base è 0. Ad esempio:

```
1    <beans>
2
3       <bean id="exampleBean" class="examples.ExampleBean">
4          <constructor-arg index="0" value="2001"/>
5          <constructor-arg index="1" value="Zara"/>
6       </bean>
7
8    </beans>
```

Come si può vedere, in questo caso stiamo esplicitamente dicendo al Framework Spring che deve iniettare il valore 2001 all'interno del primo parametro del costruttore della classe Foo (che appunto è un int), e che deve iniettare il valore "Zara" nel secondo parametro del costruttore della classe Foo (che appunto è una String).

NB: Nel caso stiamo passando un oggetto referenziato (di fatto un indirizzo) dobbiamo usare l'attributo ref all'interno del tag <constructor-arg> mentre se stiamo passando direttamente un valore dobbiamo usare l'attributo value (come nel precedente caso).

12 – Iniezione della Dipendenza basata su metodi Setter

Posted on **5 aprile 2013** Views: 637

L'iniezione della dipendenza basata sui metodi Setter viene realizzata dal container invocando sul nostro bean dei metodi Setter dopo che questo è stato creato mediante un metodo costruttore privo di argomenti della factory che si occupa appunto di istanziare i nostri bean.

Esempio:

Il seguente esempio mostra una classe **TextEditor** in cui si possono iniettare altre classi solamente usando l'iniezione della dipendenza mediante costruttore.

Lavorando con STS\Eclipse s eguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.

Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.

Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.

Inserite i seguenti parametri:

Group Id org.andrea.myexample
Artifact Id: mySetterMethodsInjection
Version: lasciate 0.0.1-SNAPSHOT

Clickate su **Finish**.

Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-beans**, **spring-context**, **spring-context-support**).

Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml** del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>mySetterMethodsInjection</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>mySetterMethodsInjection</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
```

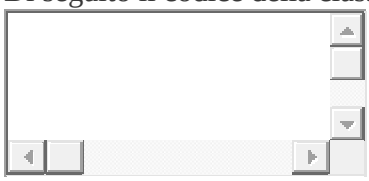
```
17  <dependencies>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All'interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.mySetterMethodsInjection** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
All'interno dell'unico package **org.andrea.myexample.mySetterMethodsInjection** create le seguenti 3 classi Java: **TextEditor.java**, **SpellChecker.java**, **MainApp.java**

Di seguito il codice della classe **TextEditor.java**:



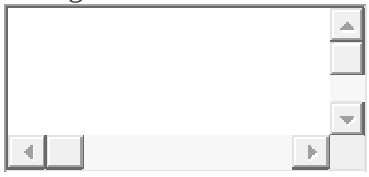
```
1  package org.andrea.myexample.mySetterMethodsInjection;
2
3  public class TextEditor {
4     private SpellChecker spellChecker;
5
6     // Metodo setter per iniettare la dipendenza:
7     public void setSpellChecker(SpellChecker spellChecker) {
8        System.out.println("Inside setSpellChecker.");
9        this.spellChecker = spellChecker;
10    }
11
12    // Costruttore PRIVO DI ARGOMENTI:
13    public SpellChecker getSpellChecker() {
14       return spellChecker;
15    }
16
17    // Metodo che esegue il controllo ortografico:
18    public void spellCheck() {
19       spellChecker.checkSpelling();
20    }
```

```
21 }
```

Quì dobbiamo notare la convenzione usata per nomi dei metodi setter. Per settare la variabbile **SpellChecker spellChecker** stiamo usando il metodo Setter **setSpellChecker()** .
Andiamo ora ad inserire il contenuto della classe subordinata **SpellChecker.java**:

```
1  package org.andrea.myexample.mySetterMethodsInjection;
2
3  // Classe subordinata che rappresenta il correttore ortografico:
4  public class SpellChecker {
5
6      // Costruttore:
7      public SpellChecker() {
8          System.out.println("Inside SpellChecker constructor.");
9      }
10
11     // Metodo che dovrebbe implementare la correzione ortografica:
12     public void checkSpelling() {
13         System.out.println("Inside checkSpelling.");
14     }
15
16 }
```

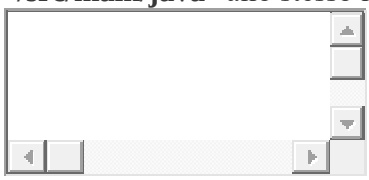
Di seguito il codice della classe principale **MainApp**:



```
1  package org.andrea.myexample.mySetterMethodsInjection;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
```

```
6  // Codice principale contenente il metodo main():
7  public class MainApp {
8      public static void main(String[] args) {
9          /* Crea il contesto in base alle impostazioni
10          * dell'applicazione definite nel file Beans.xml  */
11         ApplicationContext context =
12                 new ClassPathXmlApplicationContext("Beans.xml");
13
14         /* Recupera un bean avente id="textEditor" nel file di
15          * configurazione Beans.xml */
16         TextEditor te = (TextEditor) context.getBean("textEditor");
17
18         // Invoca il metodo che esegue il controllo ortografico:
19         te.spellCheck();
20     }
21 }
```

Di seguito il file di configurazione **Beans.xml** che contiene la configurazione dell'iniezione della dipendenza basata sui metodi Setter, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



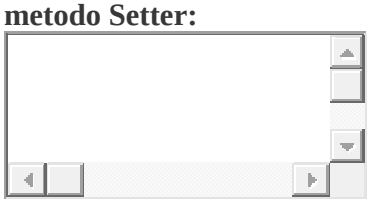
```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5      xsi:schemaLocation="http://www.springframework.org/schema/beans
6      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8      <!-- Definition for textEditor bean -->
9      <bean id="textEditor" class="org.andrea.myexample.mySetterMethodsInjection.TextEditor">
10         <property name="spellChecker" ref="spellChecker" />
```

```
11    </bean>
12
13    <!-- Definition for spellChecker bean -->
14    <bean id="spellChecker" class="org.andrea.myexample.mySetterMethodsInjection.SpellChecker">
15    </bean>
16
17 </beans
```

Potete notare le differenze con il file **Beans.xml** dell'articolo precedente in cui veniva illustrata l'iniezione della dipendenza basata sul costruttore.
La principale differenza si trova all'interno del tag **<bean>** relativo alla definizione del bean relativa alla classe **TextEditor**: in questo caso, invece di usare i tag **<constructor-arg>** per effettuale l'iniezione delle dipendenze mediante costruttore, usiamo il tag **<property>** per **iniettare la dipendenza tramite i metodi setter**.
NB: Nel caso stiamo passando un oggetto referenziato (di fatto un indirizzo) dobbiamo usare l'attributo ref all'interno del tag <property> mentre se stiamo passando direttamente un valore dobbiamo usare l'attributo value (come nel precedente caso).
Una volta finito di implementare il codice sorgente di tutte le classe e del file di configurazione eseguiamo la classe MainAppp ed otterremo il seguente output nel tab Console di STS\Eclipse:

Inside SpellChecker constructor.
Inside setSpellChecker.
Inside checkSpelling.
Configurazione XML che fa uso dei p-namespace:
Nel caso si abbiano molti metodi Setter risulta conveniente usare i **p-namespace** nel file di configurazione XML.
I **p-namespace** consentono di utilizzare gli attributi dell'elemento bean, al posto degli elementi**<property /> nidificati**, per **descrivere i nostri valori delle proprietà e/o i bean che cooperano** (le dipendenze). A partire da Spring 2.0 è fornito il supporto di configurazione tramite namespace che è basato sullo schema di definizione XML.
Per fare chiarezza andiamo a vedere due esempi di file XML di configurazione:

1) File XML standard che usa il tag <property> per specificare le dipendenze da iniettare tramite metodo Setter:



```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <bean id="john-classic" class="com.example.Person">
9       <property name="name" value="John Doe"/>
10      <property name="spouse" ref="jane"/>
11    </bean>
12
13    <bean name="jane" class="com.example.Person">
14      <property name="name" value="John Doe"/>
15    </bean>
16
17  </beans>
```

In questo file stiamo specificando due bean in cui usiamo il tag property per settare i valori delle relative proprietà tramite metodi Setter.

2)File XML che fà uso dei p-namespace per specificare le dipendenze da iniettare tramite metodi Setter (riscrittura della precedente versione):

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xmlns:p="http://www.springframework.org/schema/p"
6    xsi:schemaLocation="http://www.springframework.org/schema/beans
7    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
8
9   <bean id="john-classic" class="com.example.Person"
10    p:name="John Doe"
11    p:spouse-ref="jane"/>
12  </bean>
13
14  <bean name="jane" class="com.example.Person"
15    p:name="John Doe"/>
16  </bean>
```

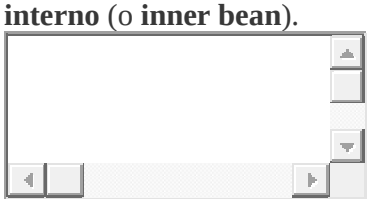
Nell'esempio visto si può notare la differenza tra specificare un valore primitivo ed un riferimento ad un altro bean usando la forma dei **p-namespace**. Nel caso stessimo **iniettando un valore primitivo** usiamo l'attributo **p:name="valore"** mentre nel caso stessimo iniettando il riferimento ad un altro bean usiamo l'attributo **p:property_name-ref="idBean-da-iniettare"**.

13 – Iniettare Bean Interni in Spring (Inner Bean)

Posted on **5 aprile 2013** Views: 1740

Com'è noto le classi Java interne sono definite all'interno del codice di un'altra classe. Eventualmente consiglio di leggere questo pdf sulle classi interne: http://www.federica.unina.it/smfn/linguaggi-di-programmazione-ii/classi-interne-locali-anonime/

Similmente, i **bean interni** sono **bean che sono definiti all'interno di altri bean**. Così un elemento**<bean/>** posto all'interno del tag **<property/>** o del tag **<costructor-arg/>** è chiamato **bean interno** (o **inner bean**).



```
1   <?xmlversion="1.0"encoding="UTF-8"?>
2
3   <beansxmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
```

```
8       <beanid="outerBean"class="...">
9          <propertyname="target">
10            <beanid="innerBean"class="..."/>
11         </property>
12      </bean>
13
14   </beans>
```

In questa configurazione XML abbiamo definiti un bean interno avente **ID=innerBean** all'interno del tag property di un altro bean avente **ID=outerBean**.

Esempio:

Il seguente esempio mostra una classe **TextEditor** in cui si possono iniettare altre classi solamente usando l'iniezione della dipendenza mediante costruttore.

Lavorando con STS\Eclipse s eguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggettp Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.

Nella prima finestra del wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.

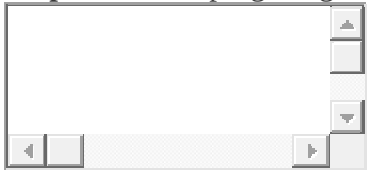
Nella seconda finestra del wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.

Inserite i seguenti parametri:

Group Id: org.andrea.myexample
Artifact Id: myInnerBeanExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.

Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-beans**, **spring-context**, **spring-context-support**).

Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
2  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3  <modelVersion>4.0.0</modelVersion>
4
5  <groupId>org.andrea.myexample</groupId>
6  <artifactId>myInnerBeanExample</artifactId>
7  <version>0.0.1-SNAPSHOT</version>
8  <packaging>jar</packaging>
9
10 <name>myInnerBeanExample</name>
11 <url>http://maven.apache.org</url>
12
13 <properties>
14   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15 </properties>
16
17 <dependencies>
18   <dependency>
19     <groupId>junit</groupId>
20     <artifactId>junit</artifactId>
21     <version>3.8.1</version>
22     <scope>test</scope>
23   </dependency>
24   <dependency>
25       <groupId>org.springframework</groupId>
26       <artifactId>spring-core</artifactId>
27       <version>3.1.1.RELEASE</version>
28   </dependency>
29   <dependency>
30       <groupId>org.springframework</groupId>
31       <artifactId>spring-beans</artifactId>
32       <version>3.1.1.RELEASE</version>
33   </dependency>
```

```
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

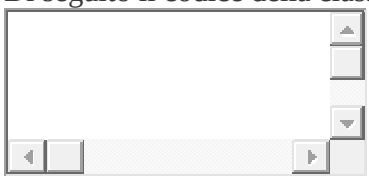
All'interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.myInnerBeanExample** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
All'interno dell'unico package **org.andrea.myexample.mySetterMethodsInjection** create le seguenti 3 classi Java: **TextEditor.java**, **SpellChecker.java**, **MainApp.java**

Di seguito il codice della classe **TextEditor.java**:

```
1  package org.andrea.myexample.myInnerBeanExample;
2
3  // Classe che rappresenta un ipotetico editor di testo:
4  public class TextEditor {
5
```

```
6     // Oggetto che rappresenta il correttore ortografico:
7     private SpellChecker spellChecker;
8
9     // Metodo Setter per effettuare l'iniezione della dipendenza:
10    public void setSpellChecker(SpellChecker spellChecker) {
11        System.out.println("Inside setSpellChecker.");
12        this.spellChecker = spellChecker;
13    }
14
15    // Metodo Getter che ritorna un oggetto Spellchecker:
16    public SpellChecker getSpellChecker() {
17        return spellChecker;
18    }
19
20    // Ipotetico metodo per eseguire il controllo ortogragico:
21    public void spellCheck() {
22        spellChecker.checkSpelling();
23    }
24 }
```

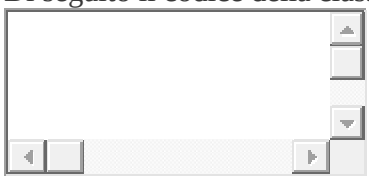
Di seguito il codice della classe **SpellChecker.java:**



```
1  package org.andrea.myexample.myInnerBeanExample;
2
3  // Classe che rappresenta un generico correttore ortografico:
4  public class SpellChecker {
5
6     // Costruttore:
7     public SpellChecker() {
8        System.out.println("Inside SpellChecker constructor.");
9     }
```

```
10
11     // Metodo che esegue la correzione ortografica:
12     public void checkSpelling() {
13         System.out.println("Inside checkSpelling.");
14     }
15
16 }
```

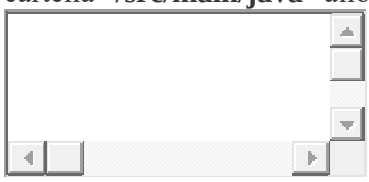
Di seguito il codice della classe principale **MainAppb** contenente il metodo main():



```
 1  package org.andrea.myexample.myInnerBeanExample;
 2
 3  import org.springframework.context.ApplicationContext;
 4  import org.springframework.context.support.ClassPathXmlApplicationContext;
 5
 6  // Classe principale contenente il metodo main():
 7  public class MainApp {
 8
 9     public static void main(String[] args) {
10        /* Crea il contesto in base alle impostazioni
11         * dell'applicazione definite nel file Beans.xml  */
12        ApplicationContext context =
13                new ClassPathXmlApplicationContext("Beans.xml");
14
15        /* Recupera un bean avente id="textEditor" nel file di
16         * configurazione Beans.xml */
17        TextEditor te = (TextEditor) context.getBean("textEditor");
18
19        // Invoca il metodo che esegue il controllo ortografico:
20        te.spellCheck();
21     }
```

```
22 }
```

Di seguito il codice del file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- Definizione del bean textEditor bean che usa un bean interno -->
9     <bean id="textEditor" class="org.andrea.myexample.myInnerBeanExample.TextEditor">
10       <property name="spellChecker">
11          <bean id="spellChecker" class="org.andrea.myexample.myInnerBeanExample.SpellChecker" />
12       </property>
13    </bean>
14
15 </beans>
```

In questo caso anzichè di avere due definizioni di bean separate di cui una dipendenza dell'altra, abbiamo definito il bean **spellChecker** internamente al bean **textEditor** in modo tale che **spellChecker** sia il bean subordinato al bean **textEditor**.

Andando ad eseguire la classe **MainApp** otterremo il seguente output nel tab Console di STS\Eclipse:

Inside SpellChecker constructor.
Inside setSpellChecker.
Inside checkSpelling.

14 – Iniettare Collezioni in Spring

Posted on **5 aprile 2013** Views: 588

Fino ad ora abbiamo visto come configurare l'**iniezione di tipi di dato primitivi** usando l'attributo **value** e di come usare l'**attributo ref del tag <property**> per **iniettare interi bean**.
Vediamo ora come passare valori plurimi come **List**, **Set**, **Map** e **Properties**. Per gestire tali situazioni il Framework Spring offre quattro tipi di elementi per la configurazione di collezioni, essi sono:

ELEMENTO	DESCRIZIONE
<list>	Serve per iniettare liste di valori, permette duplicati.
<set>	Serve per iniettare un insieme di valori, non permette duplicati.
<map>	Può essere usato per iniettare collezioni formate da coppie <nome,valore> in cui gli emenenti nome e valore possono avere qualsiasi tipo.
<props>	Può essere usato per iniettare collezioni formate da coppie <nome,valore> in cui gli elementi nome e valore sono entrambi stringhe.

Possiamo usare sia **<list>** che **<set>** per legare qualsiasi implementazione di **java.util.Collection** o un **array**.
Vedremo due possibili situazioni:

1. **Passaggio di valori diretti alla collezione.**
2. **Passaggio di riferimento ad un bean come un elemento della collezione.**

Esempio:
Lavorando con STS\Eclipse eguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su **Next**.
Nella seconda finestra dello wizard lasciate il default ArtifactId settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group Id rg.andrea.myexample
Artifact Id: myCollectionExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file **pom.xm** e clickate sul tab "**Dependencies"** per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-beans**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4
5   <groupId>org.andrea.myexample</groupId>
6   <artifactId>myCollectionExample</artifactId>
7   <version>0.0.1-SNAPSHOT</version>
8   <packaging>jar</packaging>
9
10  <name>myCollectionExample</name>
11  <url>http://maven.apache.org</url>
12
13  <properties>
14    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

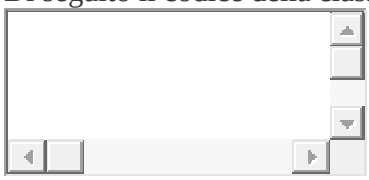
Andiamo ora ad implementare effettivamente il nostro progetto Hello World andando a scrivere il codice.

All’interno della cartella di STS con nome **/src/main/java** troviamo il package**org.andrea.myexample.myCollectionExample** che contiene una classe chiamata **App.java**, eliminiamo pure tale classe.
Creiamo ora le seguenti classi dentro tale package: **JavaCollection** e **MainApp**.
Di seguito il codice da inserire all’interno del file **JavaColletcion.java**:

```
1  package org.andrea.myexample.myCollectionExample;
2
3  import java.util.*;
4
5  // Classe che contiene una serie di classiche collezioni Java:
6  public class JavaCollection {
7
8     List addressList;
9     Set addressSet;
10    Map addressMap;
11    Properties addressProp;
12
13    // Metodo Setter per iniettare una List:
14    public void setAddressList(List addressList) {
15       this.addressList = addressList;
16    }
17
18    // Stampa tutti gli elementi di una lista e ritorna la lista:
19    public List getAddressList() {
20       System.out.println("List Elements :" + addressList);
21       return addressList;
22    }
23
24    // Metodo Setter per iniettare un Set:
25    public void setAddressSet(Set addressSet) {
26       this.addressSet = addressSet;
27    }
28
29    // Stampa tutti gli elementi del Set e ritorna il Set:
30    public Set getAddressSet() {
31       System.out.println("Set Elements :" + addressSet);
32       return addressSet;
33    }
34
35    // Metodo Setter per iniettare una Map:
36    public void setAddressMap(Map addressMap) {
37       this.addressMap = addressMap;
38    }
39
40    // Stampa tutti gli elementi della Map e ritorna la Map:
41    public Map getAddressMap() {
42       System.out.println("Map Elements :" + addressMap);
43       return addressMap;
44    }
45
46    // Metodo Setter per settare un oggetto Properties:
47    public void setAddressProp(Properties addressProp) {
48       this.addressProp = addressProp;
49    }
50
51    /* Stampa tutti gli elementi del Properties e ritorna
52     * l'oggetto Properties: */
53    public Properties getAddressProp() {
```

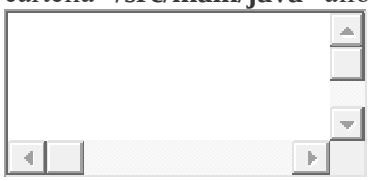
```
54        System.out.println("Property Elements :" + addressProp);
55        return addressProp;
56    }
57 }
```

Di seguito il codice della classe principale **MainApp.java** che contiene il metodo main():



```
 1  package org.andrea.myexample.myCollectionExample;
 2
 3  import org.springframework.context.ApplicationContext;
 4  import org.springframework.context.support.ClassPathXmlApplicationContext;
 5
 6  // Classe principale contenente il metodo main():
 7  public class MainApp {
 8
 9     public static void main(String[] args) {
10
11        /* Crea il contesto in base alle impostazioni
12         * dell'applicazione definite nel file Beans.xml */
13        ApplicationContext context =
14              new ClassPathXmlApplicationContext("Beans.xml");
15
16        /* Recupera un bean avente id="javaCollection" nel file
17         * di configurazione Beans.xml */
18        JavaCollection jc = (JavaCollection) context.getBean("javaCollection");
19
20        // Recupera le collezioni:
21        jc.getAddressList();
22        jc.getAddressSet();
23        jc.getAddressMap();
24        jc.getAddressProp();
25     }
26 }
```

Di seguito il codice del file di configurazione **Beans.xml** che contiene la configurazione per l'iniezione di tutti i tipi di collezioni considerate nell'esempio.Come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2
 3  <beans xmlns="http://www.springframework.org/schema/beans"
 4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 5     xsi:schemaLocation="http://www.springframework.org/schema/beans
 6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
 7
 8     <!-- Definizione del bean javaCollection -->
 9     <bean id="javaCollection" class="org.andrea.myexample.myCollectionExample.JavaCollection">
10
11        <!-- Risulta essere una chiamata a setAddressList(java.util.List) -->
12        <property name="addressList">
13           <list>
14              <value>INDIA</value>
15              <value>Pakistan</value>
16              <value>USA</value>
```

```
17              <value>USA</value>
18           </list>
19        </property>
20
21        <!-- Risulta essere una chiamata a setAddressSet(java.util.Set) -->
22        <property name="addressSet">
23           <set>
24              <value>INDIA</value>
25              <value>Pakistan</value>
26              <value>USA</value>
27              <value>USA</value>
28           </set>
29        </property>
30
31        <!-- Risulta essere una chiamata a setAddressMap(java.util.Map) -->
32        <property name="addressMap">
33           <map>
34              <entry key="1" value="NDIA" />
35              <entry key="2" value="Pakistan" />
36              <entry key="3" value="USA" />
37              <entry key="4" value="USA" />
38           </map>
39        </property>
40
41        <!-- Risulta essere una chiamata a setAddressProp(java.util.Properties) -->
42        <property name="addressProp">
43           <props>
44              <prop key="one">INDIA</prop>
45              <prop key="two">Pakistan</prop>
46              <prop key="three">USA</prop>
47              <prop key="four">USA</prop>
48           </props>
49        </property>
50
51     </bean>
52
53  </beans>
```

Andando ad eseguire la classe **MainApp** otterremo il seguente output nel tab Console di STS\Eclipse:

List Elements :[INDIA, Pakistan, USA, USA]

Set Elements :[INDIA, Pakistan, USA]

Map Elements :{1=NDIA, 2=Pakistan, 3=USA, 4=USA}

Property Elements :{two=Pakistan, one=INDIA, three=USA, four=USA}

Iniettare bean referenziati all'interno di una collezione:

Vediamo ora com'è possibile iniettare dei bean referenziati, anzichè tipi primitivi e valori semplici, all'interno di una collezione. Nell'esempio verrà mostrato che è possibile mischiare bean referenziati e valori semplici.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
 7
 8    <bean id="address1" class="org.andrea.myexample.myCollectionExample.Indirizzo">
 9        <constructor-arg index="0" value="Via Roma" />
10        <constructor-arg index="1" value="1" />
11    </bean>
12
13    <bean id="address2" class="org.andrea.myexample.myCollectionExample.Indirizzo">
14        <constructor-arg index="0" value="Via Milano" />
15        <constructor-arg index="1" value="3" />
16    </bean>
17
18    <!-- Bean Definition to handle references and values -->
19    <bean id="javaCollection"
20        class="org.andrea.myexample.myCollectionExample.JavaCollection2">
21
22        <!-- Passing bean reference for java.util.List -->
23        <property name="addressList">
24            <list>
25                <ref bean="address1" />
26                <ref bean="address2" />
27                <value>Pakistan</value>
28            </list>
29        </property>
30
31    </bean>
32
33 </beans>
```

In tale file di configurazione vediamo che vengono definiti due bean aventi **ID** pari a: "**address1**" ed "**address2**" relativi ad una nuova classe **Indirizzo**, tale classe rappresenta un generico indirizzo e si compone in soli due campi che decrivono un generico indirizzo: via e numero civico.
Nella definizione di tali bean iniettiamo i valori per il campo via e per il campo civico tramite l'iniezione della dipendenza basata sul costruttore (vedere questo articolo).

Ora che il framework ha creato questi due bean valorizzati vogliamo passarli come elementi di una lista.

Per fare ciò definiamo un nuovo bean avente **ID="addressList"** e relativo ad una nuova classe**JavaCollection2**. All'interno della definizione di tale bean vi è un tag **<list/>** che come abbiamo già visto è usato per l'iniezione di liste all'interno del nostro bean.
Andiamo quindi ad iniettare i bean con ID pari a "**address1**" ed "**address2**" precedentemente creati (tramite l'uso del tag **<ref bean\>** ed un valore semplice rappresentante una stringa (tramite l'uso del tag **<value\>**).
Vediamo ora il codice della classe **Indirizzo** (da inserire sempre nel nostro unico package del precedente progetto):
Per mostrare questa cosa ho creato il seguente file di configurazione **Beans2.xml** da affiancare al precedente nell'esempio già visto:

```
1  package org.andrea.myexample.myCollectionExample;
2
```

```
3  public class Indirizzo {
4
5      private String via;
6      private int civico;
7
8      /* Costruttore con relativi parametri di input per effettuare
9       * l'iniezione della dipendenza mediante costruttore: */
10     public Indirizzo(String via, int civico){
11         System.out.println("Entrato nel costruttore di Indirizzo");
12
13         this.via = via;
14         this.civico = civico;
15
16         System.out.println("OGGETTO COSTRUITO");
17         System.out.println("via: " + this.via);
18         System.out.println("civico: " + this.civico);
19
20     }
21
22     // Getter & Setter:
23
24     public String getVia() {
25         return via;
26     }
27
28     public int getCivico() {
29         return civico;
30     }
31
32     public void setVia(String via) {
33         this.via = via;
34     }
35
36     public void setCivico(int civico) {
37         this.civico = civico;
38     }
39
40 }
```

Come abbiamo detto, tale classe implementa un costruttore parametrico per poter effettuare l'iniezione della dipendenza mediante costruttore e poter valorizzare i campi dei relativi bean in fase di creazione.

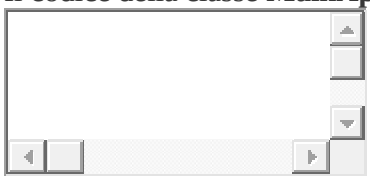
Vediamo ora il codice della classe **JavaCollection2** (da inserire sempre nel nostro unico package del precedente progetto):

```
1  package org.andrea.myexample.myCollectionExample;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Map;
6  import java.util.Properties;
7  import java.util.Set;
8
9  public class JavaCollection2 {
10
11     List addressList;
12
13     // Metodo Setter per iniettare una List:
14     public void setAddressList(List addressList) {
```

```
15        System.out.println("Enter in setterAddressList()");
16
17        this.addressList = addressList;
18
19        System.out.println("List setted");
20    }
21
22    // Stampa tutti gli elementi di una lista e ritorna la lista:
23    public List getAddressList() {
24        System.out.println("List Elements :" + addressList);
25        return addressList;
26    }
27
28 }
```

Come si può vedere nel codice di questa classe abbiamo il solito metodo per iniettare la lista all'interno di un oggetto wrapper **JavaCollection** ma in più troviamo un altro metodo che riceve come parametro di input un oggetto **Indirizzo** e lo aggiungge alla nostra lista.

Il codice della classe **MainApp2** invece è il seguente:



```
 1  package org.andrea.myexample.myCollectionExample;
 2
 3  import org.springframework.context.ApplicationContext;
 4  import org.springframework.context.support.ClassPathXmlApplicationContext;
 5
 6  // Classe principale contenente il metodo main():
 7  public class MainApp2 {
 8
 9     public static void main(String[] args) {
10
11        /* Crea il contesto in base alle impostazioni
12         * dell'applicazione definite nel file Beans.xml */
13        ApplicationContext context =
14              new ClassPathXmlApplicationContext("Beans2.xml");
15
16        /* Recupera un bean avente id="javaCollection" nel file
17         * di configurazione Beans.xml */
18        JavaCollection2 jc = (JavaCollection2) context.getBean("javaCollection");
19
20        jc.getAddressList();
21
22        Indirizzo address1 = (Indirizzo) jc.getAddressList().get(0);
23        System.out.println("Via: " + address1.getVia() + "Numero Civico: " + address1.getCivico());
24     }
25
26 }
```

Andando ad eseguire tale classe otteniamo il seguente output nel tab Console di STS\Eclipse:

Entrato nel costruttore di Indirizzo
OGGETTO COSTRUITO
via: Via Roma
civico: 1
Entrato nel costruttore di Indirizzo
OGGETTO COSTRUITO

via: Via Milano
civico: 3
Enter in setterAddressList()
List setted
List Elements
:[org.andrea.myexample.myCollectionExample.Indirizzo@4b8a4ec6,org.andrea.myexample.myCollectionExample.Indirizzo@6539cfe8, Pakistan]
List Elements
:[org.andrea.myexample.myCollectionExample.Indirizzo@4b8a4ec6,org.andrea.myexample.myCollectionExample.Indirizzo@6539cfe8, Pakistan]
Via: Via RomaNumero Civico: 1

Analizzando anche i messaggi di output inseriti nei vari metodi possiamo vedere che per prima cosa vengono costruiti i due bean relativi alla classe **Indirizzo** e possiamo notare come questi vengano valorizzati direttamente al momento della loro creazione. Vediamo poi come la lista venga settata. Infine un caso in cui viene recuperata la sola lista in cui vengono visualizzati i bean ed il valore semplice all'interno ed un caso più complesso in cui dopo aver recuperato la stringa recuperiamo l'oggetto corrispondente al primo bean inserito nella lista, di cui stampiamo i valori dei suoi campi.

Iniettare oggetti null e stringhe vuote in una lista:

Se abbiamo la necessità di **iniettare stringhe vuote come valori della collezione**, possiamo passarli come segue:

```
1    <bean id="..." class="exampleBean">
2         <property name="email" value=""/>
3    </bean>
```

Tale codice equivale a passare in Java il codice **exampleBean.setEmail("");**

Nel caso invece avessimo la necessità di **iniettare un bean NULL nella collezione**, possiamo passarlo come segue:

```
1    <bean id="..." class="exampleBean">
2      <property name="email"><null/></property>
3    </bean>
```

Tale codice corrisponde a passare in Java il codice **exampleBean.setEmail(null);**

15 – Auto Wiring dei bean in Spring

Posted on **6 aprile 2013** Views: 869

Nei precedenti articoli abbiamo visto come dichiarare i bean, all'interno di un file di configurazione XML, usando il tag <bean> e come iniettare i bean usando i tag <constructor-arg> e <property>.

Il container del Framewrok Spring può legare automaticamente tra loro bean che cooperano senza usare né <constructor-arg> né <property>, ciò aiuta a ridurre drasticamente l'ammontare delle configurazioni scritte all'interno del file XML (che potrebbe diventare eccessivamente grande e complesso nel caso di grandi applicazioni basate su Spring).

Possibili modalità per effettuare l'Autowiring:

Nei prossimi articoli analizzeremo le seguenti modalità per effettuare l'autorwiring. Tali modalità possono essere usate per istruire il container di Spring per usare l'autowiring per effettuare la dependency injection. Useremo l'attributo **autowire** dell'elemento <bean/> per specificare quale specifica modalità sarà usata per una determinata definizione di un bean.

MODALITÀ	DESCRIZIONE
no	Settaggio di default, significa che l'autowiring non è abilitato e che dobbiamo usare esplicitamente il riferimento ad i bean per effettuare il loro wiring. Tale caso si riferisce alla condizione vista nei precedenti articoli in cui esplicitavamo l'iniezione delle dipendenze direttamente nel file di configurazione XML.
byName	Autowiring delle proprietà mediante nome. Il container di Spring guarda le proprietà dei bean, all'interno del file di configurazione XML, aventi l'attributo **autowire** settato con **byName** e prova a matchare e legare tali proprietà con i bean aventi lo stesso nome definiti nello stesso file di configurazione XML.
byType	Autowiring basato sui tipi di dato delle proprietà. Il contariner guarda le proprietà dei bean, all'interno del file di configurazione XML, aventi l'attributo **autowire** settato con **byType** e prova a matchare e legare una proprietà se il suo tipo matcha esattamente con quello di un bean nel file di configurazione XML. Se più di un bean soddisfano tale matching, sarà sollevata una fatal exception.
constructor	Simile a byType, ma il tipo è applicato agli asrgomenti del costruttore. Se non c'è esattamente un bean avente lo stesso tipo dell'argomento del costruttore, sarà sollevata una fatal exception.
autodetected	Spring prima prova ad usare l'autowiring tramite costruttore, se non funziona Spring prova ad effettuare l'autowire tramite byType.

NB: Possiamo usare le modalità byType o constructor per legare array ed altre collezioni tipizzate.

Limitazioni presenti usando autowiring:

L'autowiring lavora bene quando è usato costantemente in un progetto. Se l'autowiring è usato solo per legare pochi bean (e per gli altri si usa la configurazione XML) potrò causare confusione negli sviluppatori.

L'autowiring può ridurre significativamente la necessità di specificare properties o gli argomenti del costruttore ma bisogna considerare le limitazioni e gli svantaggi dell'autowiring prima di usarlo:

1. **Possibilità di overriding:** Possiamo continuare a specificare le dipendenze usando i tag <constructor-arg> e <property> all'interno del file di configurazione XML. Tali configurazioni effettueranno sempre l'override dell'autowiring.
2. **Tipi di dato primitivi:** Non si può effettuare l'autowire dei cosidette proprietà semplici come: tipi primitivi, String e classi.
3. **Confusione:** L'autowiring è meno esatto dei wiring espliciti. Per cui, quando è possibile è consigliabile usare il wiring esplicito.

Nei prossimi articoli analizzeremo, fornendo esempi pratici, i vari tipi di autowiring presentati.

16 – Autowiring byName in Spring

Posted on **6 aprile 2013** Views: 606

Questa modalità specifica l'autowiring mediante nome della proprietà. Il container di Spring guarda i beans, definiti nel file di configurazione XML, aventi l'attributo **auto-wire** è settato con il valore **byName** e prova ad effettuare il macthing e lega tale proprietà con un altro bean, sempre definito nell'XML, avente lo stesso nome speficitato. Se il match viene soddisfatto e viene trovato un bean avente lo stesso nome, questo viene iniettato, altrimenti viene sollevata un'eccezzione.
Per esempio, se la definizione di un bean ha il settaggio dell'autowire configurato come byName e questo contiene una proprietà chiamata **spellChecker** (che corrisponde ad un metodo setSpellChecker), Spring cerca nella definizione dei bean un bean avente nome spellChecker e lo usa per settare la proprietà (lo inietta).
Di seguito è mostrato un esempio che illustra tale concetto:

Esempio:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group Id:org.andrea.myexample
Artifact Id: myAutowiringByName
Version: lasciate 0.0.1-SNAPSHOT

Clickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myAutowiringByName</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myAutowiringByName</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
21       <version>3.8.1</version>
22       <scope>test</scope>
23     </dependency>
```

```
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

All'interno dell'unico package presente create la seguente classe **TextEditor.java**:

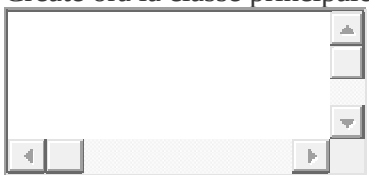
```
1  package org.andrea.myexample.myAutowiringByName;
2
```

```
3  // Classe che rappresenta un editor di testo:
4  public class TextEditor {
5
6      // Correttore ortografico:
7      private SpellChecker spellChecker;
8      private String name;
9
10     // Metodo Setter per iniettare il correttore ortografico:
11     public void setSpellChecker(SpellChecker spellChecker) {
12         this.spellChecker = spellChecker;
13     }
14
15     public SpellChecker getSpellChecker() {
16         return spellChecker;
17     }
18
19     // Metodo Setter per inietare la stringa name
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getName() {
25         return name;
26     }
27
28     // Metodo che esegue la correzione ortografica
29     public void spellCheck() {
30         spellChecker.checkSpelling();
31     }
32 }
```

Create un’altra classe dipendente **SpellChecker.java** ed inseritevi dentro il seguente codice:

```
1  package org.andrea.myexample.myAutowiringByName;
2
3  // Classe dipendente che rappresenta il correttore ortografico:
4  public class SpellChecker {
5
6      // Costruttore:
7      public SpellChecker() {
8          System.out.println("Inside SpellChecker constructor.");
9      }
10
11     // Esegue la correzione ortografica:
12     public void checkSpelling() {
13         System.out.println("Inside checkSpelling.");
14     }
15
16 }
```

Create ora la classe principale **MainApp** contenente il metodo main():



```
1  package org.andrea.myexample.myAutowiringByName;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale contenente il metodo main:
7  public class MainApp {
8      public static void main(String[] args) {
```

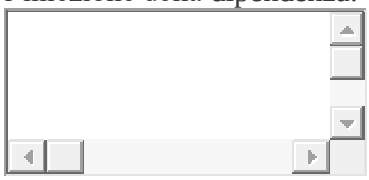
```
9         /* Crea il contesto in base alle impostazioni
10         * dell'applicazione definite nel file Beans.xml  */
11        ApplicationContext context =
12                new ClassPathXmlApplicationContext("Beans.xml");
13
14        /* Recupera un bean avente id="textEditor" nel file di
15         * configurazione Beans.xml */
16        TextEditor te = (TextEditor) context.getBean("textEditor");
17
18        // Invoca il metodo che esegue il controllo ortografico:
19        te.spellCheck();
20    }
21 }
```

Di seguito vediamo la versione del file **Beans.xml** che non fà uso dell'autowiring e che configura il wiring delle dipendenze in maniera esplicita all'interno dell'XML. Creiamo quindi il file di configurazione**Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3     <beans xmlns="http://www.springframework.org/schema/beans"
4           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5           xsi:schemaLocation="http://www.springframework.org/schema/beans
6        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8           <!-- Definition for textEditor bean -->
9           <bean id="textEditor" class="org.andrea.myexample.myAutowiringByName.TextEditor">
10                <property name="spellChecker" ref="spellChecker" />
11                <property name="name" value="Generic Text Editor" />
12          </bean>
```

```
13
14          <!-- Definition for spellChecker bean -->
15          <bean id="spellChecker" class="org.andrea.myexample.myAutowiringByName.SpellChecker"/>
16
17    </beans>
```

Vediamo ora la versione dello stesso file che fà uso dell'**autowiring** settato us **byName** per effettuare l'iniezione della dipendenza:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8    <!-- Definition for textEditor bean -->
9    <bean id="textEditor" class="org.andrea.myexample.myAutowiringByName.TextEditor"
10      autowire="byName">
11      <property name="name" value="Generic Text Editor" />
12   </bean>
13
14   <!-- Definition for spellChecker bean -->
15   <bean id="spellChecker" class="org.andrea.myexample.myAutowiringByName.SpellChecker">
16   </bean>
17
18 </beans>
```

Come si può vedere in tale versione del file di configurazione XML, anzichè iniettare esplicitamente il bean avente ID=spellChecker all'interno del bean avente ID=textEditor lo inietiamo usando l'attributo **autowire="byName"** nella definizione del bean avente ID=textEditor.
Tale attributo dice al container di Spring di andare a vedere la lista delle proprietà all'interno del bean avente ID=textEditor (in questo caso troverà una variabile avente nome spellChecker) e cercherà nella

configurazione XML un bean avente un id con lo stesso nome (in questo caso il bean avente ID=spellChecker), se lo trova lo inietta automaticamente, altrimenti solleva un'eccezione.

Per quanto riguarda invece l'iniezione del valore della proprietà name invece dobbiamo esplicitarla nella configurazione XML perchè si tratta di un valore semplice (una stringa) e non possiamo usare l'autowiring.

Pertanto l'output visualizzato nel tab Console di STS\Eclipse sarà il seguente:

Inside SpellChecker constructor.
Inside checkSpelling.

17 – Autowiring byType in Spring

Posted on **6 aprile 2013** Views: 559

Questa modalità specifica l'autowiring mediante il tipo di una proprietà. Il container di Spring considera i bean, definiti all'interno del file di configurazione XML, in cui è presente l'attributo **autowire** settato su **byType**. Il container cerca di effettuare il matching tra il tipo delle proprietà del bean definito con un altro bean avente lo stesso tipo ed esegue il wire di una property se il tipo dichiarato matcha esattamente con l'ID di uno dei bean definiti nel file di configurazione. Se il match è trovato, il bean viene iniettato altrimenti verrà sollevata un'eccezione.
Per esempio, se la definizione di un bean contiene l'attributo **autowire** settato su **byType** e se questo contiene una property avente ID=spellChecker e **tipo SpellChecker**, Spring cerca la definizione di un bean chiamata Spellchecker e la usa per settare la property.
Di seguito è mostrato un esempio che illustra tale concetto:

Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:**org.andrea.myexample
Artifact **Id:** myAutowiringByType
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myAutowiringByType</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myAutowiringByType</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
```

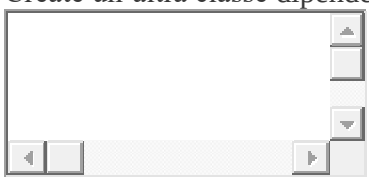
```
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25      <groupId>org.springframework</groupId>
26      <artifactId>spring-core</artifactId>
27      <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30      <groupId>org.springframework</groupId>
31      <artifactId>spring-beans</artifactId>
32      <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35      <groupId>org.springframework</groupId>
36      <artifactId>spring-context</artifactId>
37      <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40      <groupId>org.springframework</groupId>
41      <artifactId>spring-context-support</artifactId>
42      <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

All'interno dell'unico package presente create la seguente classe **TextEditor.java**:

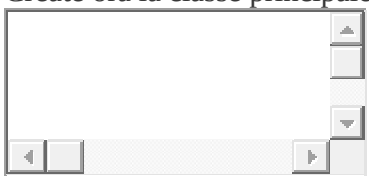
```
1  package org.andrea.myexample.myAutowiringByType;
2
3  //Classe che rappresenta un editor di testo:
4  public class TextEditor {
5
6      // Correttore ortografico:
7      private SpellChecker spellChecker;
8      private String name;
9
10     // Metodo Setter per iniettare il correttore ortografico:
11     public void setSpellChecker(SpellChecker spellChecker) {
12         this.spellChecker = spellChecker;
13     }
14
15     public SpellChecker getSpellChecker() {
16         return spellChecker;
17     }
18
19     // Metodo Setter per inietare la stringa name
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getName() {
25         return name;
26     }
27
28     // Metodo che esegue la correzione ortografica
29     public void spellCheck() {
30         spellChecker.checkSpelling();
31     }
32 }
```

Create un’altra classe dipendente **SpellChecker.java** ed inseritevi dentro il seguente codice:



```
1  package org.andrea.myexample.myAutowiringByType;
2
3  //Classe dipendente che rappresenta il correttore ortografico:
4  public class SpellChecker {
5
6      // Costruttore:
7      public SpellChecker() {
8          System.out.println("Inside SpellChecker constructor.");
9      }
10
11     // Esegue la correzione ortografica:
12     public void checkSpelling() {
13         System.out.println("Inside checkSpelling.");
14     }
15
16 }
```

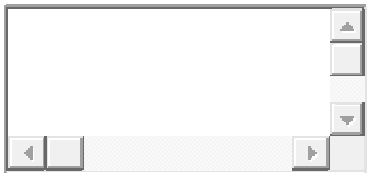
Create ora la classe principale **MainApp** contenente il metodo main():



```
1  package org.andrea.myexample.myAutowiringByType;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  //Classe principale contenente il metodo main:
7  public class MainApp {
```

```
8     public static void main(String[] args) {
9         /* Crea il contesto in base alle impostazioni
10         * dell'applicazione definite nel file Beans.xml  */
11        ApplicationContext context =
12                new ClassPathXmlApplicationContext("Beans.xml");
13
14        /* Recupera un bean avente id="textEditor" nel file di
15         * configurazione Beans.xml */
16        TextEditor te = (TextEditor) context.getBean("textEditor");
17
18        // Invoca il metodo che esegue il controllo ortografico:
19        te.spellCheck();
20    }
21 }
```

Di seguito vediamo la versione del file **Beans.xml** che non fà uso dell'autowiring e che configura il wiring delle dipendenze in maniera esplicita all'interno dell'XML. Creiamo quindi il file di configurazione**Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- Definition for textEditor bean -->
9     <bean id="textEditor" class="org.andrea.myexample.myAutowiringByType.TextEditor">
10       <property name="spellChecker" ref="spellChecker" />
11       <property name="name" value="Generic Text Editor" />
```

```
12    </bean>
13
14    <!-- Definition for spellChecker bean -->
15    <bean id="spellChecker" class="org.andrea.myexample.myAutowiringByType.SpellChecker" />
16
17 </beans>
```

Vediamo ora la versione dello stesso file che fà uso dell'**autowiring** settato su **byType** per effettuare l'iniezione della dipendenza:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- Definition for textEditor bean -->
9     <bean id="textEditor" class="org.andrea.myexample.myAutowiringByType.TextEditor"
10       autowire="byType">
11       <property name="name" value="Generic Text Editor" />
12    </bean>
13
14    <!-- Definition for spellChecker bean -->
15    <bean id="SpellChecker" class="org.andrea.myexample.myAutowiringByType.SpellChecker">
16    </bean>
17
18 </beans>
```

Come si può vedere in tale versione del file di configurazione XML, anzichè iniettare esplicitamente il bean avente ID=spellChecker all'interno del bean avente ID=textEditor lo inietiamo usando l'attributo**autowire="byType"** nella definizione del bean avente ID=textEditor.
Tale attributo dice al container di Spring di andare a vedere la lista delle proprietà all'interno del bean avente ID=textEditor (in questo caso troverà una variabile avente tipo SpellChecker) e cercherà nella configurazione XML un bean avente un id con lo stesso tipo (in questo caso il bean avente ID=spellChecker), se lo trova lo inietta automaticamente, altrimenti solleva un'eccezione.

Per quanto riguarda invece l'iniezione del valore della proprietà name invece dobbiamo esplicitarla nella configurazione XML perchè si tratta di un valore semplice (una stringa) e non possiamo usare l'autowiring.

Pertanto l'output visualizzato nel tab Console di STS\Eclipse sarà il seguente:

Inside SpellChecker constructor.
Inside checkSpelling.

18 – Autowiring mediante Costruttore in Spring

Posted on **6 aprile 2013** Views: 578

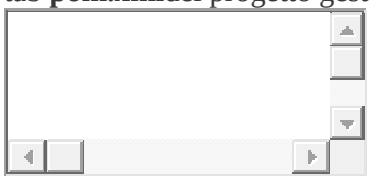
Analizziamo ora la modalità di **autowire constructor**. Tale modalità è molto simile alla modalità**byType** ma a differenza di quest'ultima si applica agli argomenti del costruttore. Il container di Spring considera i bean, definiti all'interno del file di configurazione XML, in cui è presente l'attributo **autowire**settato su **constructor**. Il container di Spring cercherà dunque di eseguire il matching e successivamente cercherà di legare ogni argomento del costruttore con un bean, definito nella configurazione XML, avente lo stesso tipo. Se il match è trovato il bean verrà iniettato, altrimenti sarà sollevata un'eccezione.
Per esempio, se nella configurazione XML è presente la definizione di un bean avente attributo**autowire** settato su **construtctor** e considerando il caso in cui tale bean ha un costruttore che prende in input un solo argomento avente tipo **SpellChecker**, accade che Spring cerca, nella definizione dei bean, un bean avente nome SpellChecker, e lo usa come parametro di input del costruttore.
Di seguito è mostrato un esempio che illustra tale concetto:

Esempio:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace**

location“, lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su “**maven-archetype-quickstart**” e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:**org.andrea.myexample
Artifact **Id:** myAutowiringByConstrucor
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file **pom.xml** e clickate sul tab “**Dependencies**” per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myAutowiringByConstrucor</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myAutowiringByConstrucor</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
```

```
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

All'interno dell'unico package presente create la seguente classe **TextEditor.java**:

```
1  package org.andrea.myexample.myAutowiringByConstrucor;
2
3  // Classe che rappresenta un editor di testo:
4  public class TextEditor {
5
6      // Rappresenta il correttore ortografico:
7      private SpellChecker spellChecker;
8      private String name;
9
10     /* Costruttore:
11        @param il correttore ortografico
12        @param il nome
13      */
14     public TextEditor(SpellChecker spellChecker, String name) {
15         this.spellChecker = spellChecker;
16         this.name = name;
17     }
18
19     // Ritorna il correttore ortografico:
20     public SpellChecker getSpellChecker() {
21         return spellChecker;
22     }
23
24     // Ritorna il nome
25     public String getName() {
26         return name;
27     }
28
29     // Esegue la correzione ortografica:
```

```
30     public void spellCheck() {
31         spellChecker.checkSpelling();
32     }
33 }
```

Create un’altra classe dipendente **SpellChecker.java** ed inseritevi dentro il seguente codice:

```
1  package org.andrea.myexample.myAutowiringByConstrucor;
2
3  //Classe dipendente che rappresenta il correttore ortografico:
4  public class SpellChecker {
5  //Costruttore:
6      public SpellChecker() {
7          System.out.println("Inside SpellChecker constructor.");
8      }
9
10     // Esegue la correzione ortografica:
11     public void checkSpelling() {
12         System.out.println("Inside checkSpelling.");
13     }
14 }
```

Create ora la classe principale **MainApp** contenente il metodo main():

```
1  package org.andrea.myexample.myAutowiringByConstrucor;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
```

```
6  //Classe principale contenente il metodo main:
7  public class MainApp {
8      public static void main(String[] args) {
9          /*
10          * Crea il contesto in base alle impostazioni dell'applicazione definite
11          * nel file Beans.xml
12          */
13         ApplicationContext context = new ClassPathXmlApplicationContext(
14                 "Beans.xml");
15         /*
16          * Recupera un bean avente id="textEditor" nel file di configurazione
17          * Beans.xml
18          */
19         TextEditor te = (TextEditor) context.getBean("textEditor");
20         // Invoca il metodo che esegue il controllo ortografico:
21         te.spellCheck();
22     }
23 }
```

Di seguito vediamo la versione del file **Beans.xml** che non fà uso dell'autowiring e che configura il wiring delle dipendenze in maniera esplicita all'interno dell'XML. Creiamo quindi il file di configurazione**Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3      <beans xmlns="http://www.springframework.org/schema/beans"
4             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5             xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
```

```
8         <!-- Definition for textEditor bean -->
9         <bean id="textEditor" class="org.andrea.myexample.myAutowiringByConstrucor.TextEditor">
10            <constructor-arg ref="spellChecker" />
11            <constructor-arg value="Generic Text Editor" />
12        </bean>
13
14        <!-- Definition for spellChecker bean -->
15        <bean id="spellChecker" class="org.andrea.myexample.myAutowiringByConstrucor.SpellChecker"/>
16
17    </beans>
```

Vediamo ora la versione dello stesso file che fà uso dell'**autowiring** settato su **constructor** per effettuare l'iniezione della dipendenza:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8     <!-- Definition for textEditor bean -->
9     <bean id="textEditor"
10       class="org.andrea.myexample.myAutowiringByConstrucor.TextEditor"
11       autowire="constructor">
12       <constructor-arg value="Generic Text Editor" />
13    </bean>
14
15    <!-- Definition for spellChecker bean -->
16    <bean id="SpellChecker"
17       class="org.andrea.myexample.myAutowiringByConstrucor.SpellChecker">
```

```
18    </bean>
19
20 </beans>
```

Come nei precedenti esempi, il bean avente ID=**SpellChecker** viene legato al parametro di input del costruttore del bean avente ID=textEditor poichè quest'ultimo ha un parametro avente tipo**SpellChecker**.
Per quanto riguarda invece l'iniezione del valore della proprietà name invece dobbiamo esplicitarla nella configurazione XML perchè si tratta di un valore semplice (una stringa) e non possiamo usare l'autowiring.

Pertanto l'output visualizzato nel tab Console di STS\Eclipse sarà il seguente:

Inside SpellChecker constructor.
Inside checkSpelling.

19 – Introduzione alla Configurazione di Spring basata sulle annotazioni

Posted on **7 aprile 2013** Views: 1385

Vediamo ora un argomento che è diventato di fondamentale importanza per la configurazione dell'iniezione delle dipendenze nelle ultime versioni del Framework Spring.A partire dalla versione 2.5 del Framework Spring è diventato possibile configurare l'iniezione delle dipendenze usando le**annotation**.
Così al posto di usare l'XML per descrivere i legami che intercorrono tra i bean, possiamo spostare la configurazione di tali bean all'interno delle classi stesse usando le annotazioni che possono essere usate per annotare classi, metodi e variabili.

L'iniezione delle dipendenze realizzata mediante annotation viene eseguita prima delle eventuali iniezioni delle dipendenze eseguite mediante configurazione XML, pertanto tale seconda configurazione eseguirà l'override delle proprietà legate tra loro mediante l'uso delle annotation.

Il wiring mediante annotation non è abilitato per default dal container. Così, prima di poter usare il wiring mediante annotazioni, dobbiamo abilitarlo nel nostro file di configurazione di Spring. Consideriamo di avere il seguente file di configurazione nel caso volessimo usare le annotazioni in un'applicazione Spring:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config/>
   <!-- bean definitions go here -->

</beans>
```

Una volta che **<context:annotation-config/>** è stato configurato, possiamo iniziare ad annotare il nostro codice per indicare a Spring che dovrebbe automaticamente legare proprietà, metodi e costruttori.
Andiamo a vedere alcune importanti annotazioni per capire come funzionano:

1. **@Required:** Annotazione che può essere usata ai metodi setter delle proprietà dei bean.
2. **@Autowired:** Annotazione che può essere applicata ai metodi setter delle proprietà dei bean, ai metodi che non sono metodi setter, ai metodi costruttore ed alle proprietà.
3. **@Qualified:** Annotazione che insieme ad **@Autowired** può essere usata per rimuovere confusione specificando esattamentequale bean viene legato.
4. **@JSR-250 Annotation:** Spring supporta le annotazioni basate su **JSR-250** che includono annotazioni come **@Resource**, **@PostConstruct** e **@PreDestroy**.

Nei prossimi articoli vedremo un esempio concreto per ognuno di questi casi.

20 – Significato ed uso dell'annotazione @Required in Spring

Posted on **7 aprile 2013** Views: 859

L'annotazione **@Required** si applica ai **metodi setter delle proprietà dei bean** e indica che la proprietà del bean interessata deve essere popolata nel file di configurazione XML in fase di configurazione, altrimenti il container solleverà un'eccezione BeanInitializationException. Di seguito è proposto un esempio che mostro l'uso dell'annotazione **@Required**.

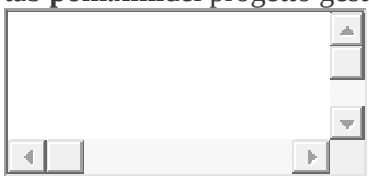
Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace**

location“, lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su “**maven-archetype-quickstart**” e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:**org.andrea.myexample
Artifact **Id:** myRequiredAnnotation
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file pom.xm e clickate sul tab “**Dependencies**” per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

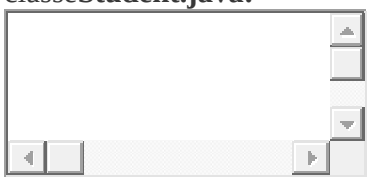


```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myRequiredAnnotation</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myRequiredAnnotation</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
```

```
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento del nostro progetto.

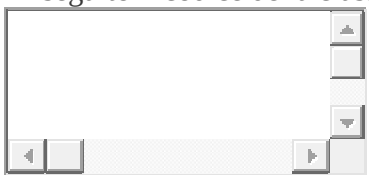
All’interno dell’unico package **org.andrea.myexample.myRequiredAnnotation** creiamo la classe**Student.java:**



```
1  package org.andrea.myexample.myRequiredAnnotation;
2
3  import org.springframework.beans.factory.annotation.Required;
4
5  public class Student {
6
7     // Proprietà: i metodi setter sono annotati con @Required
8     private Integer age;
9     private String name;
10
11    @Required
12    public void setAge(Integer age) {
13        this.age = age;
14    }
15
16    public Integer getAge() {
17        return age;
18    }
19
20    @Required
21    public void setName(String name) {
22        this.name = name;
23    }
24
25    public String getName() {
26        return name;
27    }
```

```
28 }
```

Di seguito il codice della classe principale **MainApp.java**:



```
1  package org.andrea.myexample.myRequiredAnnotation;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Codice principale contenente il metodo main():
7  public class MainApp {
8     public static void main(String[] args) {
9        /*
10        * Crea il contesto in base alle impostazioni dell'applicazione definite
11        * nel file Beans.xml
12        */
13       ApplicationContext context = new ClassPathXmlApplicationContext(
14             "Beans.xml");
15
16       /*
17        * Recupera un bean avente id="student" nel file di configurazione
18        * Beans.xml
19        */
20       Student student = (Student) context.getBean("student");
21
22       System.out.println("Name : " + student.getName());
23       System.out.println("Age : " + student.getAge());
24    }
25 }
```

Di seguito vediamo la versione del file **Beans.xml** che non fà uso dell'autowiring e che configura il wiring delle dipendenze in maniera esplicita all'interno dell'XML. Creiamo quindi il file di

configurazione**Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config />

   <!-- Definition for student bean -->
   <bean id="student" class="org.andrea.myexample.myRequiredAnnotation.Student">
      <property name="name" value="Zara" />
      <property name="age"  value="11"/>
   </bean>

</beans>
```

Andando ad eseguire nuovamente l'applicazione otterremo il seguente output:

Name : Zara
Age : 11

Quello che succede usando l'annotazione @Required sui metodi setter delle proprietà di un bean dovrebbe essere ora chiaro, il valore (iniettato tramite metodi setter all'interno del file di configurazione Beans.xml) deve essere presente, altrimenti si solleva una ***BeanInitializationException***.

21 – Significato ed uso dell'annotazione @Autowired in Spring

Posted on **7 aprile 2013** Views: 1888

L'annotazione **@Autowired** fornisce un controllo più raffinato su cosa e su come l'autowiring deve essere realizzato. L'annotazione **@Autowiring** può essere usata per collegare automaticamente i bean sui metodi Setter (in maniera analoga all'annotazione **@Required**), sui costruttori, alle proprietà di un bean o su metodi con nomi arbitrari e/o più argomenti. Vediamo ora un esempio pratico per ognuno di questi casi:

Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:**myAutowiredAnnotation
Artifact **Id:** myAutowiredAnnotation
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>myAutowiredAnnotation</groupId>
6    <artifactId>myAutowiredAnnotation</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>myAutowiredAnnotation</name>
<url>http://maven.apache.org</url>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.1.1.RELEASE</version>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>3.1.1.RELEASE</version>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.1.1.RELEASE</version>
  </dependency>
```

```
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

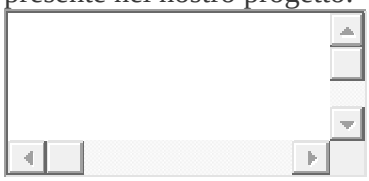
Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento di tale progetto. Andiamo ora ad esaminare i casi precedentemente esposti:

@Autowired sui metodi Setter:

Possiamo usare l'annotazione **@Autowired sui metodi Setter per sbarazzarci dell'elemento <property> all'interno del file di configurazione XML**.

Quando Spring trova che un metodo Setter è annotato con **@Autowired** prova ad eseguire l'autowire**byType** su tale metodo.

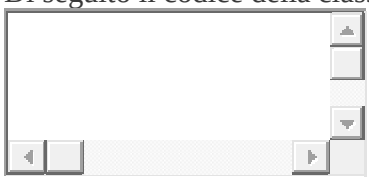
Per capire tale comportamento creiamo una classe **TextEditor1.java** all'interno dell'unico package presente nel nostro progetto:



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4
5  /* Classe che rappresenta un editor di testi e che usa l'annotation
6   * @Autowired sui metodi Setter */
7  public class TextEditor1 {
8
9     // Rappresenta il correttore ortografico
10    private SpellChecker spellChecker;
11
12    // Metodo Setter annotato con @Autowired
```

```
13    @Autowired
14    public void setSpellChecker(SpellChecker spellChecker) {
15        this.spellChecker = spellChecker;
16    }
17
18    public SpellChecker getSpellChecker() {
19        return spellChecker;
20    }
21
22    public void spellCheck() {
23        spellChecker.checkSpelling();
24    }
25 }
```

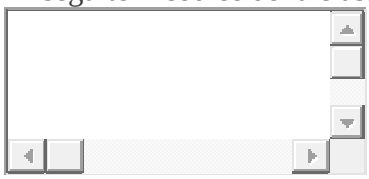
Di seguito il codice della classe dipendente **SpellChecker.java**:



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  // Rappresenta il correttore ortografico
4  public class SpellChecker {
5
6     // Costruttore:
7     public SpellChecker() {
8        System.out.println("Inside SpellChecker constructor.");
9     }
10
11    // Metodo che esegue la correzione ortografica:
12    public void checkSpelling() {
13       System.out.println("Inside checkSpelling.");
14    }
15
```

```
16 }
```

Di seguito il codice della classe principale relativa a tale esempio **MainApp1.java:**



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  //Classe principale contenente il metodo main:
7  public class MainApp1 {
8     public static void main(String[] args) {
9        /*
10        * Crea il contesto in base alle impostazioni dell'applicazione definite
11        * nel file Beans.xml
12        */
13       ApplicationContext context = new ClassPathXmlApplicationContext(
14             "Beans1.xml");
15       /*
16        * Recupera un bean avente id="textEditor" nel file di configurazione
17        * Beans.xml
18        */
19       TextEditor1 te = (TextEditor1) context.getBean("textEditor");
20       // Invoca il metodo che esegue il controllo ortografico:
21       te.spellCheck();
22    }
23 }
```

Vediamo ora il file XML di configurazione relativo a tale esempio, **Beans1.xml**. Creiamo quindi il file di configurazione **Beans1.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:context="http://www.springframework.org/schema/context"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
8     http://www.springframework.org/schema/context
9     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10
11    <context:annotation-config/>
12
13    <!-- Definition for textEditor bean without constructor-arg  -->
14    <bean id="textEditor" class="myAutowiredAnnotation.myAutowiredAnnotation.TextEditor1">
15    </bean>
16
17    <!-- Definition for spellChecker bean -->
18    <bean id="spellChecker" class="myAutowiredAnnotation.myAutowiredAnnotation.SpellChecker">
19    </bean>
20
21 </beans>
```

Andiamo ora ad eseguire la classe **MainApp1** (slelezionate tale classe —> tasto destro del mouse —> Runs As —> Java Application) e nel tab Console di STS\Eclipse otterremo il seguente output:

Inside SpellChecker constructor.
Inside checkSpelling.

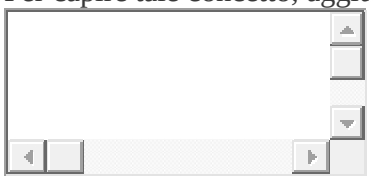
Alla luce di quanto detto in precedenza ciò che accade dovrebbe ora essere chiaro: annotando il metodo ***setSpellChecker(SpellChecker spellChecker)*** con l'annotazione**@Autowired** possiamo eliminare l'elemento <property> che specifica esplicitamente la dipendenza del bean SpellChecker nel file di configurazione XML e sarà il framework Spring stesso ad effettuare un autowire **byType**,

pertanto quando troverà il parametro di input SpellChecker spellChecker andrà a cercare nella definizione dei bean se è presente un bean avente tipo SpellChecker e se lo trova lo inietterà automaticamente, se invece tale bean non sarà trovato sarà sollevata un'eccezione.

@Autowired sulle propietà:

Possiamo usare l'annotazione **@Autowired** direttamente sulle proprietà anzichè usarla sui metodi Setter. Nel caso si passassero valori a delle proprietà annotate con **@Autowired** usando l'elemento**<property>** all'interno del file di configurazione XML, il Framework Spring assegnerà automaticamente a tali proprietà i valori passati (in caso di valori semplici) o i riferimenti (in caso di oggetti).

Per capire tale concetto, aggiungiamo la classe **TextEditor2.java** al package del precedente progetto:



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4
5  /* Classe che rappresenta un editor testuale e che usa l'annotazione
6   * @Autowired sulle proprietà */
7  public class TextEditor2 {
8
9     // Proprietà cannotata
10    @Autowired
11    private SpellChecker spellChecker;
12
13    // Costruttore
14    public TextEditor2() {
15        System.out.println("Inside TextEditor constructor.");
16    }
17
18    public SpellChecker getSpellChecker() {
19        return spellChecker;
```

```
20     }
21
22     public void spellCheck() {
23         spellChecker.checkSpelling();
24     }
25 }
```

Di seguito il codice della classe principale relativa a tale esempio **MainApp2.java:**

```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  //Classe principale contenente il metodo main:
7  public class MainApp2 {
8      public static void main(String[] args) {
9          /*
10          * Crea il contesto in base alle impostazioni dell'applicazione definite
11          * nel file Beans.xml
12          */
13         ApplicationContext context = new ClassPathXmlApplicationContext(
14                 "Beans2.xml");
15         /*
16          * Recupera un bean avente id="textEditor" nel file di configurazione
17          * Beans.xml
18          */
19         TextEditor2 te = (TextEditor2) context.getBean("textEditor");
20         // Invoca il metodo che esegue il controllo ortografico:
```

```
21        te.spellCheck();
22    }
23 }
```

Di seguito il contenuto del file di configurazione **Beans2.xml**:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9
10    <context:annotation-config />
11
12    <!-- Definition for textEditor bean -->
13    <bean id="textEditor" class="myAutowiredAnnotation.myAutowiredAnnotation.TextEditor2">
14    </bean>
15
16    <!-- Definition for spellChecker bean -->
17    <bean id="spellChecker" class="myAutowiredAnnotation.myAutowiredAnnotation.SpellChecker">
18    </bean>
19
20 </beans>
```

Andiamo ora ad eseguire la classe **MainApp2** (slelezionate tale classe —> tasto destro del mouse —> Runs As —> Java Application) e nel tab Console di STS\Eclipse otterremo il seguente output:

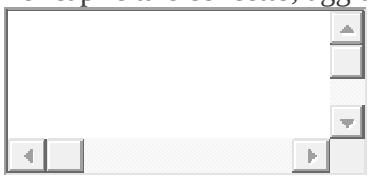
Inside TextEditor constructor.
Inside SpellChecker constructor.
Inside checkSpelling.

Ciò che accade è analogo all'esempio precedente, abbiamo semplicemente spostato l'annotazione **@Autowired** sulla proprietà anzichè sul relativo metoso Setter.

@Autowired sul Costruttore:

Possiamo applicare l'annotazione **@Autowired** anche al costruttore. Annotare un costruttore con**@Autowired** significa dire che quel costruttore deve usare l'autowiring quando crea un bean, anche se non sono stati usati elementi **<constructor-arg>** per specificare esplicitamente le dipendenze nella definizione del bean all'interno del file di configurazione XML. :

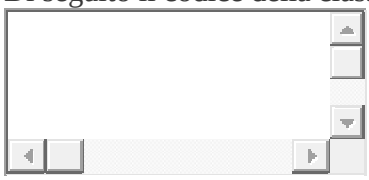
Per capire tale concetto, aggiungiamo la classe **TextEditor3.java** al package del precedente progetto:



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4
5  /* Classe che rappresenta un editor di testi e che usa l'annotazione
6   * @Autowired sul costruttore*/
7  public class TextEditor3 {
8
9     // Rappresenta il correttore ortografico:
10    private SpellChecker spellChecker;
11
12    // Costruttore annotato con @Autowired
13    @Autowired
14    public TextEditor3(SpellChecker spellChecker) {
15        System.out.println("Inside TextEditor constructor.");
16        this.spellChecker = spellChecker;
17    }
18
19    public void spellCheck() {
20        spellChecker.checkSpelling();
```

```
21     }
22 }
```

Di seguito il codice della classe principale relativa a tale esempio **MainApp3.java:**



```
1  package myAutowiredAnnotation.myAutowiredAnnotation;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  //Classe principale contenente il metodo main:
7  public class MainApp3 {
8      public static void main(String[] args) {
9          /*
10          * Crea il contesto in base alle impostazioni dell'applicazione definite
11          * nel file Beans3.xml
12          */
13         ApplicationContext context = new ClassPathXmlApplicationContext(
14                 "Beans3.xml");
15         /*
16          * Recupera un bean avente id="textEditor" nel file di configurazione
17          * Beans.xml
18          */
19         TextEditor3 te = (TextEditor3) context.getBean("textEditor");
20         // Invoca il metodo che esegue il controllo ortografico:
21         te.spellCheck();
22     }
23 }
```

Di seguito il contenuto del file di configurazione **Beans3.xml**:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9
10    <context:annotation-config />
11
12    <!-- Definition for textEditor bean without constructor-arg -->
13    <bean id="textEditor" class="myAutowiredAnnotation.myAutowiredAnnotation.TextEditor3">
14    </bean>
15
16    <!-- Definition for spellChecker bean -->
17    <bean id="spellChecker"
18       class="myAutowiredAnnotation.myAutowiredAnnotation.SpellChecker">
19    </bean>
20
21 </beans>
```

Andiamo ora ad eseguire la classe **MainApp3** (slelezionate tale classe —> tasto destro del mouse —> Runs As —> Java Application) e nel tab Console di STS\Eclipse otterremo il seguente output:

Inside SpellChecker constructor.
Inside TextEditor constructor.
Inside checkSpelling.

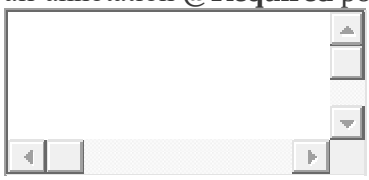
Ciò che accade è che vengono costruiti i bean **SpellChecker** e **TextEditor** (poichè entrambi presenti nella definizione dei bean all'interno del file di configurazione Beans3.xml), il bean principale è**SpellChecker** in cui viene iniettato il bean dipendennte **TextEditor** tramite iniezione della dipendneza effettuata mediante costruttore. Tale costruttore è annotato con l'annotazione **@Autowired** pertanto il framework Spring effettuerà il wiring **byType** e se nella definizione dei bean trova un bean avente lo stesso tipo del parametro del costruttore lo inetta, altrimenti sarà sollevata un'eccezione.

@Autowired con opzione (required=false):

Per default, l'annotazione **@Autowired** implica che la dipendenza sia obbligatoriamente richiesta come nell'annotazione **@Required**, tuttavia, possiamo disattivare tale comportamento usando l'opzione**(required=false)** associata all'annotazione **@Autowired**.

Il seguente esempiofunzionaanche se nonsi passaalcun valoreper la proprietàage ma sarà ancora indispensabile passare un valore alla proprietà name.

Volendo potete provarlo modificando il progetto di esempio relativo all'articolo precedente dedicato all'annotation **@Required** poichè viene cambiata solamente la classe **Student.java** in questo modo:



```
1  package org.andrea.myexample.myRequiredAnnotation;
2  import org.springframework.beans.factory.annotation.Autowired;
3  public class Student {
4     private Integer age;
5     private String name;
6     @Autowired(required=false)
7     public void setAge(Integer age) {
8        this.age = age;
9     }
10    public Integer getAge() {
11       return age;
12    }
13    @Autowired
14    public void setName(String name) {
15       this.name = name;
```

```
16    }
17    public String getName() {
18      return name;
19    }
20 }
```

22 – Significato ed uso dell'annotazione @Qualifier in Spring

Posted on **7 aprile 2013** Views: 993

Possono esserci situazioni in cui creiamo più di un bean di uno stesso tipo ed in cui vogliamo legare solo uno di essi con una proprietà, in questi casi usiamo l'annotazione **@Qualifier** insieme all'annotazione **@Autowired**. Tale annotazione è usata per eliminare confusioni specificando esattamente quale bean sarà legato. Di seguito è mostrato un esempio pratico che mostra l'uso dell'annotazione **@Qualifier**.
Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.

Inserite i seguenti parametri:

Group Id: org.andrea.myexample
Artifact Id: myQualifierAnnotation
Version: lasciate 0.0.1-SNAPSHOT

Clickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).

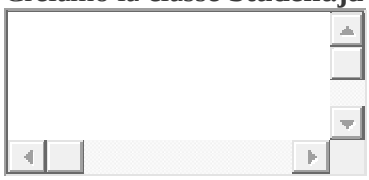
Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>myAutowiredAnnotation</groupId>
6    <artifactId>myQualifierAnnotation</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myQualifierAnnotation</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
21       <version>3.8.1</version>
22       <scope>test</scope>
23     </dependency>
24     <dependency>
25         <groupId>org.springframework</groupId>
26         <artifactId>spring-core</artifactId>
27         <version>3.1.1.RELEASE</version>
28     </dependency>
29     <dependency>
```

```
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento di tale progetto. Andiamo ora ad esaminare i casi precedentemente esposti:

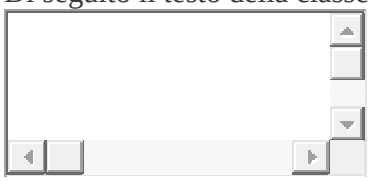
Creiamo la classe **Student.java** all'interno del nostro unico package:



```
1  package org.andrea.myexample.myQualifierAnnotation;
2
3  // Classe che rappresenta uno studente
4  public class Student {
5
6      // Proprietà:
7      private Integer age;
8      private String name;
```

```
9
10    // Metodi Getter e Setter:
11    public void setAge(Integer age) {
12        this.age = age;
13    }
14
15    public Integer getAge() {
16        return age;
17    }
18
19    public void setName(String name) {
20        this.name = name;
21    }
22
23    public String getName() {
24        return name;
25    }
26 }
```

Di seguito il testo della classe **Profile.java**:



```
1  package org.andrea.myexample.myQualifierAnnotation;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.beans.factory.annotation.Qualifier;
5
6  /* Classe in cui viene iniettato un solo specifico bean avente tipo
7   * Student, tra quelli dichirati nel file di configurazione Beans.xml
8   */
9  public class Profile {
10
```

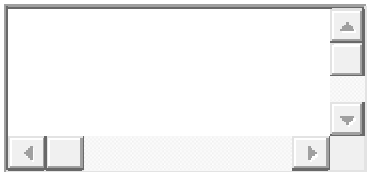
```
11    /* Proprietà in cui inietto uno specifico bean di tipo Student
12     * tramite @Autowire ed uso @Qualifier per specificare l'esatto
13     * bean da iniettare
14     */
15    @Autowired
16    @Qualifier("student1")
17    private Student student;
18
19    // Costruttore:
20    public Profile() {
21        System.out.println("Inside Profile constructor.");
22    }
23
24    // Metodi di output sullo specifico bean iniettato:
25
26    public void printAge() {
27        System.out.println("Age : " + student.getAge());
28    }
29
30    public void printName() {
31        System.out.println("Name : " + student.getName());
32    }
33 }
```

Di seguito il codice della classe principale **MainApp.java**:

```
1  package org.andrea.myexample.myQualifierAnnotation;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
```

```
//Classe principale contenente il metodo main:
public class MainApp {
    public static void main(String[] args) {
        /*
         * Crea il contesto in base alle impostazioni dell'applicazione definite
         * nel file Beans.xml
         */
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "Beans.xml");
        /*
         * Recupera un bean avente id="textEditor" nel file di configurazione
         * Beans.xml
         */
        Profile profile = (Profile) context.getBean("profile");

        profile.printAge();
        profile.printName();
    }
}
```

Vediamo ora il file XML di configurazione relativo a tale esempio, **Beans.xml**. Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
```

```
 8    http://www.springframework.org/schema/context
 9    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
10
11    <context:annotation-config />
12
13    <!-- Definizione del bean profile -->
14    <bean id="profile" class="org.andrea.myexample.myQualifierAnnotation.Profile">
15    </bean>
16
17    <!-- Definizione del student1 -->
18    <bean id="student1" class="org.andrea.myexample.myQualifierAnnotation.Student">
19       <property name="name" value="Zara" />
20       <property name="age" value="11" />
21    </bean>
22
23    <!-- Definizione del bean student2: -->
24    <bean id="student2" class="org.andrea.myexample.myQualifierAnnotation.Student">
25       <property name="name" value="Nuha" />
26       <property name="age" value="2" />
27    </bean>
28
   </beans>
```

Come si può vedere nel file di configurazione **Beans.xml** abbiamo definito due diversi bean aventi tipo Student ed a tali bean sono stati assegnati differenti valori.
Nella classe **Profile** iniettiamo uno specifico bean legandolo ad una proprrietà tramite l'annotazione**@Autowired** e specifichiamo che si tratta del bean istanziato avente **ID=student1** tramite l'annotazione **@Qualifier("student1″)**.
Pertanto l'output generato nel tab Console di STS\Eclipse mostrerà proprio i valori contenuti nelle proprietà del bean selezionato e sarà il seguente:

```
Inside Profile constructor.
Age : 11
Name : Zara
```

23 – Significato ed uso delle annotazioni basate su JSR-250

Posted on **7 aprile 2013** Views: 618

Spring supporta anche le annotazione basate sulla specifica **JSR-250** che includono le seguenti annotazioni: **@PostConstruct**, **@PreDestroy** e **@Resource**.
JSR-250 è una Java Specification Request che ha l'obbiettivo di definire una serie di annotazioni riguardanti comuni concetti semantici che possono essere utilizzati da molti componenti Java EE e Java SE. Maggiori informazioni su tale argomento quì: http://en.wikipedia.org/wiki/JSR_250
Anche se queste annotazioni non sono realmente indispensabili, poichè abbiamo altre alternative che svolgono gli stessi computi, andiamole ad analizzare brevemente per avere un'idea di come funzionino.

Annotazioni **@PostConstruct** e **@PreDestroy**:
Per definire il setup ed il teardown di un bean, in genere dichiariamo il <bean> con un parametro **init-method** e/o il parametro **destroy-method**. L'attributo **init-method** specifica un metodo che viene chiamato sul bean immediatamente dopo la sua istanziazione. Similmente, il parametro **destroy-method** specifica un metodo che viene chiamato appena prima del momento in cui un bean viene rimosso dal container.
Possiamo usare l'annotetion **@PostContruct** come un alternativa a tale metodo di calback per l'inizializzazione e l'annotation **@PreDestroy** come un alternativa al metodo di calback per la distruzione del bean.
Vediamo come funzionano effettivamente nel seguente esempio:

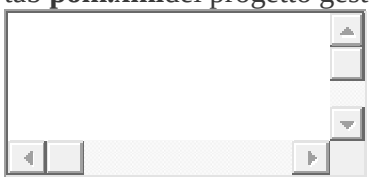
Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group Id: org.andrea.myexample
Artifact Id: myJSR250Annotation
Version: lasciate 0.0.1-SNAPSHOTClickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).

Per completezza ecco il codice del file **pom.xml** che potete inserire direttamente dentro al tab **pom.xml**del progetto gestito da STS\Eclipse:



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myJSR250Annotation</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myJSR250Annotation</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
21       <version>3.8.1</version>
22       <scope>test</scope>
23     </dependency>
24     <dependency>
25         <groupId>org.springframework</groupId>
26         <artifactId>spring-core</artifactId>
27         <version>3.1.1.RELEASE</version>
```

```
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento di tale progetto.

Creiamo la classe **HelloWorld.java** all'interno del nostro unico package:

```
1  package org.andrea.myexample.myJSR250Annotation;
2
3  import javax.annotation.*;
4
5  public class HelloWorld {
6
7     private String message;
```

```
8
9     public void setMessage(String message) {
10       this.message = message;
11    }
12
13    public String getMessage() {
14       System.out.println("Your Message : " + message);
15       return message;
16    }
17
18    // Metodo di callback invocato subito dopo la costruzione del bean
19    @PostConstruct
20    public void init() {
21       System.out.println("Bean is going through init.");
22    }
23
24    // Metodo di callback invocato subito prima della distruzione del bean
25    @PreDestroy
26    public void destroy() {
27       System.out.println("Bean will destroy now.");
28    }
29 }
```

Di seguito è riportato in contenuto della classe principale **MainApp.java**. In questa classe dobbiamo registrare un aggancio all'evento di shotdown tramite il metodo della classe astratta AbstractApplicationContext (come abbiamo fatto in un precedente articolo usando il parametro **destroy-method**). Tale metodo esegue un corretto shutdon dell'applicazione registrando un "gancio" all'evento di shutdoen della JVM ed appena prima che tale evento si verifichi esegue lo shutdown del contesto dell'applicazione distruggendo tutti i bean dell'applicazione e rilasciando lo spazio in memoria.

```
1  package org.andrea.myexample.myJSR250Annotation;
```

```
2
3  import org.springframework.context.support.AbstractApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  public class MainApp {
7     public static void main(String[] args) {
8
9        /*
10        * Crea il contesto in base alle impostazioni dell'applicazione definite
11        * nel file Beans.xml
12        */
13       AbstractApplicationContext context = new ClassPathXmlApplicationContext(
14             "Beans.xml");
15
16       /*
17        * Recupera un bean avente id="helloWorld" nel file di configurazione
18        * Beans.xml
19        */
20       HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
21
22       // Recupera il messaggio contenuto dell'oggetto obj
23       obj.getMessage();
24
25       /*
26        * Esegue lo shutdown dell'applicazione: registra un "gancio" all'evento
27        * di shutdown della JVM e prima che tale evento si verifichi esegue lo
28        * shudwon del contesto dell'applicazione distruggendo tutti i bean
29        * dell'applicazione e rilasciando lo spazio in memoria prima che si
30        * verifichi lo shudwon della JVM
31        */
32       context.registerShutdownHook();
33    }
```

```
34 }
```

Vediamo ora il file XML di configurazione relativo a tale esempio, **Beans.xml**. Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
9
10    <context:annotation-config />
11
12    <bean id="helloWorld" class="org.andrea.myexample.myJSR250Annotation.HelloWorld"
13       init-method="init" destroy-method="destroy">
14       <property name="message" value="Hello World!" />
15    </bean>
16
17 </beans>
```

Andando ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse otterremo il seguente output:

Bean is going through init.
Your Message : Hello World!
Bean will destroy now.

Come si può notare, viene richiamato automaticamente il metodo di callback annotato con**@PostConstruct** subito dopo l'istanziazione del bean, poi viene recuperato il messaggio di Hello World da tale bean ed infine viene richiamato automaticamente il metodo di callback annotato con**@PreDestroy** subito prima della distruzione di tale bean.

Annotazione @Resource:

Possiamo usare l'annotazione **@Resource** sulle proprietà o sui metodi setter dei nostri bean e si comporta esattamente come in Java EE 5. L'annotazione **@Resource** prende un attributo 'name' che sarà interpretato come il nome (l'ID) del bean da iniettare. Come si può notare, ciò segue la semantica dell'autowiring **by-name** come viene dimostrato nel seguente esempio:

```
1  package org.andrea.myexample.myJSR250Annotation;
2  import javax.annotation.Resource;
3  // Classe che rappresenta un editor di testi
4  public class TextEditor {
5      // Rappresenta il correttore ortografico
6      private SpellChecker spellChecker;
7      // Esegue l'autowiring by name implementato mediante metodo setter
8      @Resource(name = "spellChecker")
9      public void setSpellChecker(SpellChecker spellChecker) {
10         this.spellChecker = spellChecker;
11     }
12     public SpellChecker getSpellChecker() {
13         return spellChecker;
14     }
15     public void spellCheck() {
16         spellChecker.checkSpelling();
17     }
18 }
```

Nel caso in cui nessun nome fosse specificato esplicitamente, il nome di default viene derivato dal nome del campo o del metoso setter. Nel caso di un campo viene preso il nome del campo, mentre nel caso del metodo setter viene preso il nome del bean.

24 – Configurazione dei Bean basata su Java in Spring

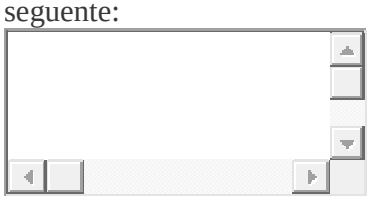
Posted on **9 aprile 2013** Views: 9151

In precedenza abbiamo visto come configurare i bean di Spring usando un file di configurazione XML. Se pensate di trovarvi bene usando la configurazione dei bean mediante XML non è richiesto come effettuare la configurazione dei bean basata su Java poichè tale tipo di configurazione raggiunge gli stessi risultati.

La configurazione dei bean basata su Java ci permette di scrivere la maggior parte delle configurazioni di Spring senza scrivere codice XML che viene sostituido da poche annotazioni che andranno inserite all'interno del codice delle nostre classi Java. Andiamo ad analizzarle:

Annotazioni @Configuration e @Bean:

Annotando una classe con l'annotazione **@Configuration** si indica al framework che quella classe può essere usata dallo IoC Container di Spring come una sorgente di definizioni di bean. L'annotazione**@Bean** invece si applica ai metodi e dice a Spring che un metodo annotato con **@Bean** ritornerà un oggetto che dovrebbe essere registrato come un bean nell'application context di Spring. Un esempio semplice e minimale per l'annotazione **@Configuration** potrebbe essere il seguente:



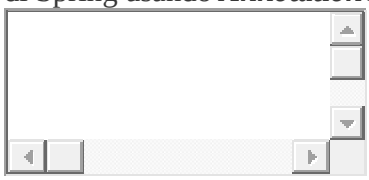
```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.annotation.*;
3  /* Classe annotata con @Configuration: la classe può essere usata dallo
4   * IoC Container di Spring come sorgente di definizione dei bean
5   */
6  @Configuration
7  public class HelloWorldConfig {
8      /*
9       * Metodo annotato coon @Bean che restituisce un oggetto HelloWorld da
10      * inserire nell'application context di Spring
11      */
12     @Bean
13     public HelloWorld helloWorld() {
```

```
14        return new HelloWorld();
15     }
16 }
```

Di seguito la configurazione XML equivalente:

```
1 <beans>
2    <beanid="helloWorld"class="com.tutorialspoint.HelloWorld"/>
3 </beans>
```

In questo caso il metodo annotato mediante **@Bean** si comporta come l'ID del bean e crea e ritorna il bean. La nostra classe di configurazione può anche cotenere dichiarazioni per più di un **@Bean**. Una volta che abbiamo definito le nostre classi di configurazione, possiamo caricarle e fornirle al container di Spring usando ***AnnotationConfigApplicationContext*** come mostrato di seguito:



```
1  public static void main(String[] args) {
2
3     /* Crea il contesto applicativo in base alle impostazioni della
4      * classe annotata HelloWorldConfig.class che abbiamo definito
5      */
6     ApplicationContext ctx =
7           new AnnotationConfigApplicationContext(HelloWorldConfig.class);
8
9     /* Recupera un bean avente tipo HelloWorld */
10    HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
11    helloWorld.setMessage("Hello World!");
12    helloWorld.getMessage();
13 }
```

Possiamo anche caricare più classi di consifurazione, come segue:

```
1  public static void main(String[] args) {
2
3     /* Crea il contesto applicativo in base alle impostazioni di
4      * classi annotate
5      */
6     AnnotationConfigApplicationContext ctx =
7           new AnnotationConfigApplicationContext();
8     // Registra più classi di configurazione
9    ctx.register(AppConfig.class, OtherConfig.class);
10   ctx.register(AdditionalConfig.class);
11
12   /* Carica o segue il refresh di rappresentazioni persistenti di
13    * configurazioni come classi o file XML
14    */
15   ctx.refresh();
16   // Recupera un bean avente tipo MyService
17   MyService myService = ctx.getBean(MyService.class);
18   myService.doStuff();
19 }
```

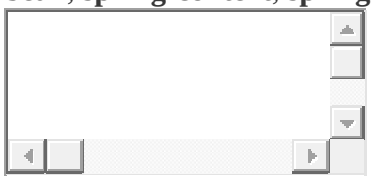
Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample
Artifact **Id:** myJavaConfiguration
Version: lasciate 0.0.1-SNAPSHOT

Clickate su Finish.

Aprite il file pom.xm e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).



```
1 <dependency>
2     <groupId> cglib</groupId>
3     <artifactId> cglib</artifactId>
4     <version> 2.2.2</version>
5 </dependency>
```

Abbiamo così creato un nuovo progetto che fà uso di Maven per la gestione del processo di building e vi abbiamo aggiunto le dipendenze del framework Spring necessarie per il funzionamento di tale progetto.

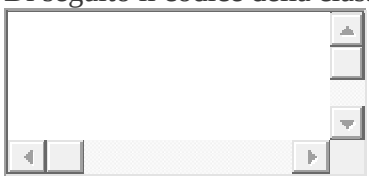
Creiamo la classe **HelloWorldConfig.java** all'interno del nostro unico package:

```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.annotation.*;
3  /* La classe viene usata dal container di Spring come una sorgente di
4   * definizione di bean */
5  @Configuration
6  public class HelloWorldConfig {
7     /* Il metodo ritorna un oggetto che deve essere registrato come un
8      * bean nel contesto di Spring */
9     @Bean
10    public HelloWorld helloWorld() {
11        return new HelloWorld();
12    }
```

```
13 }
```

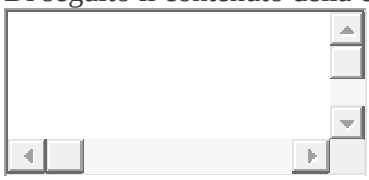
Come si può vedere la classe **HelloWorldConfig** viene usata dal container di Spring come sorgente per la definizione di bean (grazie all'annotazione **@Configuration**). Tali oggetti vengono ritornati dal metodo helloWorld e registrati come bean all'interno del contesto di Spring mediante l'uso dell'annotazione **@Bean**.

Di seguito il codice della classe **HelloWorld.java**:



```
1  package org.andrea.myexample.myJavaConfiguration;
2  /* Classe HelloWorld, gli oggetti istanza di tale classe verranno registrati
3   * come bean nel contesto di Spring grazie alla classe HelloWorldConfig
4   */
5  public class HelloWorld {
6      private String message;
7      public void setMessage(String message) {
8          this.message = message;
9      }
10     public void getMessage() {
11         System.out.println("Your Message : " + message);
12     }
13 }
```

Di seguito il contenuto della classe principale **MainApp.java**:



```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.ApplicationContext;
3  import org.springframework.context.annotation.*;
4  // Classe principale:
5  public class MainApp {
6      public static void main(String[] args) {
```

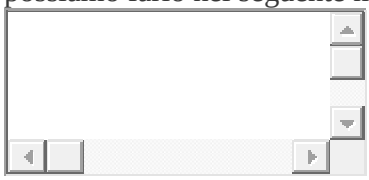
```
7
8         /* Crea il contesto applicativo in base alle impostazioni della
9          * classe annotata HelloWorldConfig.class che abbiamo definito:
10         */
11        ApplicationContext ctx = new AnnotationConfigApplicationContext(
12                HelloWorldConfig.class);
13        // Recupera un bean avente tipo HelloWorld:
14        HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
15        helloWorld.setMessage("Hello World!");
16        helloWorld.getMessage();
17    }
18 }
```

Una volta terminato, andiamo ad eseguire la classe **MainApp** ed, all'interno del tab Console di STS\Eclipse, otteremo il seguente output:

Your Message : Hello World!

Iniettare le dipendenze dei Bean:

Quando un **@Bean** ha delle dipendenze verso altri bean, esprimere tale dipendenza è semplice e possiamo farlo nel seguente modo:



```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.annotation.*;
3  /* La classe viene usata dal container di Spring come una sorgente di
4   * definizione di bean */
5  @Configuration
6  public class AppConfig {
7
8     /* Restituisce un bean che viene inserito nel contesto di Spring con
9      * ID=foo e che prende un altro bean avente ID=bar come dipendenza */
10    @Bean
11    public Foo foo() {
```

```
12        return new Foo(bar());
13    }
14    /* Restituisce un bean che viene inserito nel contesto di Spring con
15     * ID=bar */
16    @Bean
17    public Bar bar() {
18        return new Bar();
19    }
20 }
```

Come si può vedere, questa classe fornisce al container di Spring una sorgente di definizione dei bean. All'interno troviamo due metodi che restituiscono oggetti di tipo Bar e Foo. Nella creazione dell'oggetto di tipo Foo stiamo dicendo che esso dipende dall'oggetto di tipo Bar.

Andiamo ad ampliare il precedente esempio per vedere nel dettaglio tale comportamento:

Creiamo nell'unico package disponibile le seguenti classi Java:

```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.annotation.*;
3  /* La classe viene usata dal container di Spring come una sorgente di
4   * definizione di bean */
5  @Configuration
6  public class TextEditorConfig {
7      /*
8       * Restituisce un oggetto TextEditor che deve essere registrato nel contesto
9       * di Spring come bean e che dipende da un'altro bean, anch'esso nel
10      * contesto di Spring
11      */
12     @Bean
13     public TextEditor textEditor() {
```

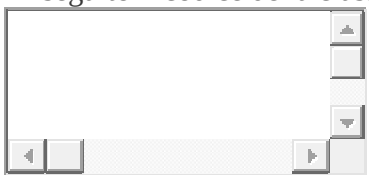
```
14        return new TextEditor(spellChecker());
15    }
16    /*
17     * Restituisce un oggetto SpellChecker che sarà registrato come bean nel
18     * contesto di Spring
19     */
20    @Bean
21    public SpellChecker spellChecker() {
22        return new SpellChecker();
23    }
24 }
```

Codice della classe **TextEditor**:

```
1  package org.andrea.myexample.myJavaConfiguration;
2  // Rappresenta un editor di testo:
3  public class TextEditor {
4
5     // Rappresenta un correttore ortografico:
6     private SpellChecker spellChecker;
7     /* L'oggetto SpellChecker viene iniettato mediante iniezione della
8      * dipendenza realizzata mediante costruttore
9      */
10    public TextEditor(SpellChecker spellChecker) {
11        System.out.println("Inside TextEditor constructor.");
12        this.spellChecker = spellChecker;
13    }
14    // Esegue la correzione ortografica:
15    public void spellCheck() {
16        spellChecker.checkSpelling();
17    }
```

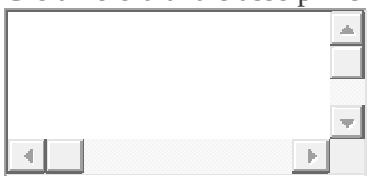
```
18 }
```

Di seguito il codice della classe dipendente **SpellChecker.java**:



```
1  package org.andrea.myexample.myJavaConfiguration;
2  // Classe dipendente:
3  public class SpellChecker {
4
5      // Costruttore:
6      public SpellChecker() {
7          System.out.println("Inside SpellChecker constructor.");
8      }
9      public void checkSpelling() {
10         System.out.println("Inside checkSpelling.");
11     }
12 }
```

Creiamo ora una classe principale chiamata **MainAppTextEditor.java**:



```
1  package org.andrea.myexample.myJavaConfiguration;
2  import org.springframework.context.ApplicationContext;
3  import org.springframework.context.annotation.*;
4  // Classe principale:
5  public class MainAppTextEditor {
6      public static void main(String[] args) {
7
8          /* Crea il contesto applicativo in base alle impostazioni della
9           * classe annotata TextEditorConfig.class che abbiamo definito:
10          */
```

```
11        ApplicationContext ctx = new AnnotationConfigApplicationContext(
12                TextEditorConfig.class);
13        // Recupera un bean avente tipo TextEditor:
14        TextEditor te = ctx.getBean(TextEditor.class);
15        te.spellCheck();
16    }
17 }
```

Possiamo verificarne il corretto funzionamento andando ad eseguire la classe principale**MainAppTextEditor** ed ottenendo il seguente output all'interno del tab Console di STS\Eclipse:

Inside SpellChecker constructor.
Inside TextEditor constructor.
Inside checkSpelling.

Annotazione @Import:

L'annotazione **@Import** permette di caricare le definizioni degli **@Bean** da un'altra classe di configurazione.

Consideriamo quindi la seguente classe di configurazione **ConfigA.java**:

Codice di **TextEditorConfig.java**:

Per poter usare le annotazioni viste in questo tutorial dobbiamo aggiungere anche la libreria CGLIB (si tratta di librerie per estendere classi Java ed usare interfacce in real time): http://cglib.sourceforge.net/

```
1  /* La classe viene usata dal container di Spring come una sorgente di
2   * definizione di bean */
3  @Configuration
4  public class ConfigA {
5
6     /* Il metodo ritorna un oggetto che deve essere registrato come un
7      * bean nel contesto di Spring */
8     @Bean
9     public A a() {
10        return new A();
11    }
```

```
12 }
```

Possiamo quindi importare la dichirazione di tale bean (il bean dichiarato nella precedente classe) nella dichiarazione di un altro bean nel seguente modo:

```
1  /* La classe viene usata dal container di Spring come una sorgente di
2   * definizione di bean.
3   * Tramite l'annotation @Import importo un altra classe sorgente di definizione
4   * dei bean in tale classe */
5  @Configuration
6  @Import(ConfigA.class)
7  public class ConfigB {
8
9      /* Il metodo ritorna un oggetto che deve essere registrato come un
10      * bean nel contesto di Spring, ritorna A */
11     @Bean
12     public B a() {
13         return new A();
14     }
15 }
```

Ora, quando istanziamo il contesto di Spring, piuttosto che specificare sia la classe ConfigA.class che ConfigB.class possiamo fornire solo configB come mostrato di seguito:

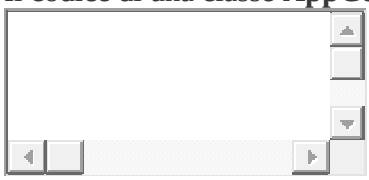
```
1  public static void main(String[] args) {
2
3      /* Crea il contesto applicativo in base alle impostazioni della
4       * sola classe annotata ConfigB.class che abbiamo definito:
```

```
5      */
6     ApplicationContext ctx =
7             new AnnotationConfigApplicationContext(ConfigB.class);
8
9     // Sia il bean A che il bean B saranno disponibili:
10    A a = ctx.getBean(A.class);
11    B b = ctx.getBean(B.class);
12 }
```

Definizione dei metodi di callback mediante configurazione Java:

L'annotazione **@Bean** supporta la specifica di metodi di callback arbitrari per l'inizializzazione e la distruzione molto simili agli attributi **init-method** e **destroy-method** dell'elemento <bean> all'interno della configurazione XML. Per esempio:

Il codice di una classe **AppConfig** che definisce una sorgente di definizione dei bean:



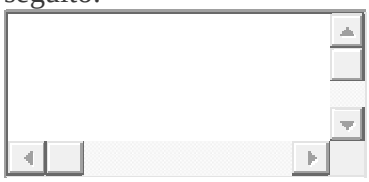
```
1  /* La classe viene usata dal container di Spring come una sorgente di
2   * definizione di bean */
3  @Configuration
4  public class AppConfig { /*
5                     * Restituisce un oggetto TextEditor che deve essere
6                     * registrato nel contesto * di Spring come bean e che
7                     * dipende da un'altro bean, anch'esso nel * contesto di
8                     * Spring. * Per tale bean viene specificato il metodo
9                     * init per l'inizializzazione ed * il metodo cleanup
10                    * per la distruzione
11                    */
12    @Bean(initMethod = "init", destroyMethod = "cleanup")
13    public Foo foo() {
14        return new Foo();
15    }
16 }
```

L’implementazione della classe **Foo** che definisce il bean con i metodi che implementano le logiche di inizializzazione e distruzione:

```
1 public class Foo {
2     public void init() {
3         // Logica di inizializzazione
4     }
5     public void cleanup() {
6         // Logica di distruzione
7     }
8 }
```

Specifica dello scope dei bean mediante la definizione Java:

Lo scope di default è **singleton** (che prevede un solo oggetto istanziato per una specifica definizione di un bean), possiamo effettuarne l’override mediante l’uso dell’annotation **@Scope**, come mostrato di seguito:



```
1  /* La classe viene usata dal container di Spring come una sorgente di
2   * definizione di bean */
3  @Configuration
4  public class AppConfig {
5
6      /* La classe viene usata dal container di Spring come una sorgente di
7       * definizione di bean.
8       * Lo scope di tale bean viene settato su prototype: per tale bean posso
9       * avere un qualsiasi numero di istanze */
10     @Bean
```

```
11    @Scope("prototype")
12    public Foo foo() {
13        return new Foo();
14    }
15 }
```

Tale esempio esegue l'override dello scope di default per la dichiarazione del bean portandolo a **prototype** che specifica la possibilità di avere un numero arbitrario di istanze relativo alla definizione di tale bean.

25 – Gestione degli eventi in Spring

Posted on **9 aprile 2013** Views: 679

Nei precedenti articoli abbiamo visto che il nucleo principale di Spring è l'**ApplicationContext** che gestisce completamente il ciclo di vita dei bean.
L'**ApplicationContext** pubblica determinati tipi di eventi durante il caricamento dei bean. Per esempio, un evento ***ContextStartedEvent*** viene pubblicato quando il **contesto viene avviato** ed un evento***ContextStoppedEvent*** viene pubblicato quando il **contesto viene stoppato**.
La gestione degli eventi all'interno dell'ApplicationContext è fornita usando la **classe *ApplicationEvent***e l'**interfaccia *ApplicationListener***.
Così se un bean implementa l'interfaccia ***ApplicationListener*** allora ogni volta che un***ApplicationEvent*** viene pubblicato nell'ApplicationContext, il bean viene notificato.
Spring fornisce i seguenti **eventi standard**:

S.N.	EVENTI STANDARD DI SPRING & DESCRIZIONE
1	**ContextRefreshedEvent** Questo evento viene pubblicato quando l'**ApplicationContext** viene inizializato o aggiornato. Può inoltre essere generato quando usiamo il metodo **refresh()** dell'interfaccia***ConfigurableApplicationContext***.
2	**ContextStartedEvent** Questo evento viene pubblicato quando l'**ApplicationContext** viene **avviato** usando il metodo**start()** dell'interfaccia ***ConfigurableApplicationContext***. E' possibile interrogare il nostro database o possiamo eseguire\rieseguire ogni applicazione stoppata dopo aver ricevuto tale evento.
3	**ContextStoppedEvent**

	Questo evento viene pubblicatio quando l'**ApplicationContext** viene **stoppato** usando il metodo**stop()** dell'interfaccia ***ConfigurableApplicationContext***. Dopo aver ricevuto tale evento possiamo richiedere dei compiti di gestione.
	ContextClosedEvent Questo evento viene pubblicato quando l'**ApplicationContext** viene**chiuso** usando il metodo**close()**dell'interfaccia ***ConfigurableApplicationContext***.
4	Quando un contesto viene chiuso ha raggiunto la fine del suo ciclo di vita e non può essere ne refreshato ne restartato.
	RequestHandledEvent
5	Questo è un evento specifico per il Web che dice a tutti i bean che una richiesta HTTP è stata servita.

La gestione degli eventi di Spring è **single-threaded** così se un evento viene pubblicato, fintanto e a meno che tutti i riceventi non hanno ottenuto il messaggio, il processo risulta essere bloccato ed il flusso delle operazioni non può continuare. Pertanto, quando si progetta un'applicazione,si deve prestare molta attenzione sela gestione degli eventidebba essere utilizzata.

Ascolto degli eventi del Contesto:

Per ascoltare un evento del contesto, un bean deve implementare l'interfaccia ***ApplicationListener***che è dotata di un solo metodo: **onApplicationEvent()**.

Andiamo a vedere un esempio concreto di come gli eventi si propagano e su come possiamo inserire nel nostro codice dei task basati su specifici eventi.

Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.

Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.

Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.

Inserite i seguenti parametri:

Group Id: org.andrea.myexample

Artifact Id: myEventHandling

Version: lasciate 0.0.1-SNAPSHOT

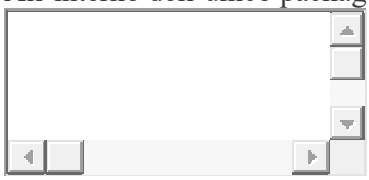
Clickate su Finish.

Aprite il file pom.xml e clickate sul tab “**Dependencies**” per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Qualora non abbiate già scaricato Spring all’interno del vostro repository locale di Maven non avrete la possibilità di scegliere le dipendenze come visto appena sopra, quindi, inserite il seguente codice nel file pom.xml:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myEventHandling</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myEventHandling</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18     <dependency>
19       <groupId>junit</groupId>
20       <artifactId>junit</artifactId>
21       <version>3.8.1</version>
22       <scope>test</scope>
23     </dependency>
```

```
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43    </dependency>
44  </dependencies>
45 </project>
```

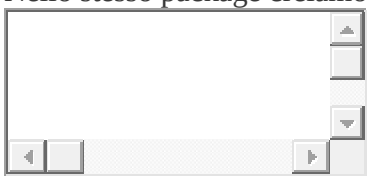
All'interno dell'unico package di tale progetto creiamo ora la seguente classe **HelloWorld.java**:



```
1  package org.andrea.myexample.myEventHandling;
2
3  public class HelloWorld {
4     private String message;
5
6     public void setMessage(String message) {
```

```
7         this.message = message;
8     }
9
10    public void getMessage() {
11        System.out.println("Your Message : " + message);
12    }
13 }
```

Nello stesso package creiamo poi la seguente classe **CStartEventHandler.java**:

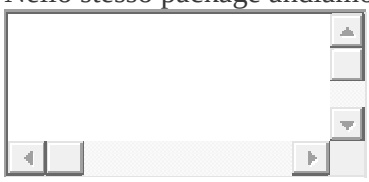


```
1  package org.andrea.myexample.myEventHandling;
2
3  import org.springframework.context.ApplicationListener;
4  import org.springframework.context.event.ContextStartedEvent;
5
6  /* Classe che implementa l'interfaccia ContextStartedEvent per la
7   * gestione dell'evento ContextStartedEvent che si verifica quando
8   * il contesto viene avviato mediante il metodo start()
9  */
10 public class CStartEventHandler implements
11        ApplicationListener<ContextStartedEvent> {
12
13    // Metodo che gestisce l'evento ContextStartedEvent:
14    public void onApplicationEvent(ContextStartedEvent event) {
15        System.out.println("ContextStartedEvent Received");
16    }
17 }
```

Come si può vedere tale classe gestirà l'evento relativo all'avvio del contesto mediante il metodo **start()**dell'interfaccia ***ConfigurableApplicationContext***.
Nello stesso package creiamo ora la classe **CStopEventHandler.java:**

```
1  package org.andrea.myexample.myEventHandling;
2
3  import org.springframework.context.ApplicationListener;
4  import org.springframework.context.event.ContextStoppedEvent;
5
6  /* Classe che implementa l'interfaccia ContextStoppedEvent per la
7   * gestione dell'evento ContextStartedEvent che si verifica quando
8   * il contesto viene avviato mediante il metodo stop()
9  */
10 public class CStopEventHandler implements
11         ApplicationListener<ContextStoppedEvent> {
12
13     // Metodo che gestisce l'evento ContextStoppedEvent:
14     public void onApplicationEvent(ContextStoppedEvent event) {
15         System.out.println("ContextStoppedEvent Received");
16     }
17 }
```

Come si può vedere tale classe gestirà l'evento relativo all'avvio del contesto mediante il metodo **stop()**dell'interfaccia ***ConfigurableApplicationContext***.
Nello stesso package andiamo ora a creare la classe principale **MainApp.java:**



```
1  package org.andrea.myexample.myEventHandling;
2
3  import org.springframework.context.ConfigurableApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale:
```

```
7  public class MainApp {
8     public static void main(String[] args) {
9        /* Crea il contesto in base alle impostazioni
10        * dell'applicazione definite nel file Beans.xml  */
11       ConfigurableApplicationContext context =
12             new ClassPathXmlApplicationContext("Beans.xml");
13
14       // Solleviamo l'evento ContextStartedEvent:
15       context.start();
16
17       /* Recupera un bean avente id="helloWorld" nel file di
18        * configurazione Beans.xml */
19       HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
20
21       obj.getMessage();
22
23       // Solleviamo l'evento ContextStoppedEvent:
24       context.stop();
25    }
26 }
```

NB: Nella classe **MainApp** abbiamo dischiarato il nostro contesto come***ConfigurableApplicationContext*** costruendolo poi con il tipo effettivo***ClassPathXmlApplicationContext***.
L'interfaccia ***ConfigurableApplicationContext*** fornisce dei servizzi per configurare un ApplicationContext che si vanno ad addizionare a quelli messi a disposizione dall'interfaccia***ApplicationContext***.
L'interfaccia ***ConfigurableApplicationContext*** ci mette a disposizione i due metodi usati **start()** e**stop()** ereditandoli a sua volta dall'interfaccia **org.springframework.context.LifeCycle**
Maggiori informazioni quì: http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/context/ConfigurableApplicationContext.html
Di seguito vediamo il contenuto del file **Beans.xml.** Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8    <bean id="helloWorld" class="org.andrea.myexample.myEventHandling.HelloWorld">
9        <property name="message" value="Hello World!"/>
10   </bean>
11
12   <bean id="cStartEventHandler"
13        class="org.andrea.myexample.myEventHandling.CStartEventHandler"/>
14
15   <bean id="cStopEventHandler"
16        class="org.andrea.myexample.myEventHandling.CStopEventHandler"/>
17
18 </beans>
```

Andando ad eseguire la classe principale **MainApp** otterremo il seguente output nel tab Console di STS\Eclipse:

ContextStartedEvent Received
Your Message : Hello World!
ContextStoppedEvent Received

Possiamo vedere che allo scatenarsi degli eventi **ContextStartedEvent** e **ContextStoppedEvent**questi vengono intercettati e gestiti.

26 – Eventi personalizzati in Spring

Posted on **9 aprile 2013** Views: 572

Ci sono un certo numero di passi che devono essere intrapresi per scrivere e pubblicare un nostro evento personalizzato. In questo articolo vedremo come scrivere, pubblicare e gestire un evento personalizzato in Spring.

Esempio:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample
Artifact **Id:** myCustomEvent
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

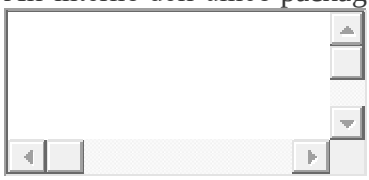
Aprite il file pom.xml e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**).
Qualora non abbiate già scaricato Spring all'interno del vostro repository locale di Maven non avrete la possibilità di scegliere le dipendenze come visto appena sopra, quindi, inserite il seguente codice nel file pom.xml:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myCustomEvent</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
```

```
 9
10  <name>myCustomEvent</name>
11  <url>http://maven.apache.org</url>
12
13  <properties>
14    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18    <dependency>
19      <groupId>junit</groupId>
20      <artifactId>junit</artifactId>
21      <version>3.8.1</version>
22      <scope>test</scope>
23    </dependency>
24    <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28    </dependency>
29    <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33    </dependency>
34    <dependency>
35        <groupId>org.springframework</groupId>
36        <artifactId>spring-context</artifactId>
37        <version>3.1.1.RELEASE</version>
38    </dependency>
39    <dependency>
40        <groupId>org.springframework</groupId>
```

```
41        <artifactId>spring-context-support</artifactId>
42        <version>3.1.1.RELEASE</version>
43     </dependency>
44   </dependencies>
45 </project>
```

All’interno dell’unico package di tale progetto creiamo ora la classe **CustomEvent.java:**



```
1  package org.andrea.myexample.myCustomEvent;
2
3  import org.springframework.context.ApplicationEvent;
4
5  /* Classe che rappresenta un evento personalizzato.
6   * Estente la classe ApplicationEvent
7   */
8  public class CustomEvent extends ApplicationEvent {
9
10    // Costruttore:
11    public CustomEvent(Object source) {
12        super(source);
13    }
14
15    public String toString() {
16        return "My Custom Event";
17    }
18 }
```

Tale classe rappresenta il nostro evento personalizzato e deve estendere la classe astratta**ApplicationEvent** che rappresenta un generico evento applicativo.
Andiamo ora a creare, nel nostro unico package la classe **CustomEventPublisher.java**:

```
1  package org.andrea.myexample.myCustomEvent;
2
3  import org.springframework.context.ApplicationEventPublisher;
4  import org.springframework.context.ApplicationEventPublisherAware;
5
6  /* Classe che si occupa di pubblicare gli eventi custom
7   * Deve implementare l'interfaccia ApplicationEventPublisherAware che
8   * è l'interfaccia che deve essere implementata da ogni oggetto che desidera
9   * essere informato all'ApplicationEventPublisher (tipicamente
10  * l'ApplicationContext) che ci gira
11  */
12 public class CustomEventPublisher implements ApplicationEventPublisherAware {
13
14    /* Interfaccia che incapsula la funzionalità di pubblicazione degli eventi.
15     * Serve come super-interfaccia per ApplicationContext.
16     */
17    private ApplicationEventPublisher publisher;
18
19    /* Inietta l'ApplicationEventPublisher mediante iniezione della dipendenza
20     * realizzata mediante metodo Setter
21     */
22
23    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
24        this.publisher = publisher;
25    }
26
27    // Metodo di pubblicazione dell'evento:
28    public void publish() {
29        CustomEvent ce = new CustomEvent(this);    // Crea un nuovo CustomEvent
```

```
30        publisher.publishEvent(ce);    // Lo fà pubblicare dall'oggetto publisher:
31    }
32 }
```

Un’oggeto istanza di tale classe si occupa di pubblicare i precedenti eventi custom usando un apposito oggetto istanza di ***ApplicationEventPublisher***.
Vediamo ora la classe che gestisce tali eventi personalizzati, create dunque nello stesso package la classe **CustomEventHandler.java**:

```
1  package org.andrea.myexample.myCustomEvent;
2
3  import org.springframework.context.ApplicationListener;
4
5  /* Classe che implementa l'interfaccia ApplicationListener per la
6   * gestione dell'evento custom CustomEvent
7  */
8  public class CustomEventHandler implements ApplicationListener<CustomEvent> {
9
10    // Metodo che gestisce l'evento custom:
11    public void onApplicationEvent(CustomEvent event) {
12        System.out.println(event.toString());
13    }
14
15 }
```

Tale classe si occupa di implementare la logica di gestione del nostro evento custom e verrà eseguito quando tale evento verrà ricevuto.

Creiamo ora, sempre nello stesso package, la classe principale **MainApp.java**:

```
1  package org.andrea.myexample.myCustomEvent;
2
3  import org.springframework.context.ConfigurableApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale:
7  public class MainApp {
8
9      public static void main(String[] args) {
10
11         /* Crea il contesto in base alle impostazioni
12          * dell'applicazione definite nel file Beans.xml  */
13         ConfigurableApplicationContext context = new ClassPathXmlApplicationContext(
14                 "Beans.xml");
15
16         /* Recupera un bean avente id="customEventPublisher" nel file di
17          * configurazione Beans.xml.
18          * Tale bean rappresenta l'oggetto deputato a pubblicare l'evento
19          * custom
20          *  */
21         CustomEventPublisher cvp = (CustomEventPublisher) context
22                 .getBean("customEventPublisher");
23
24         // Pubblica l'evento:
25         cvp.publish();
26         cvp.publish();
27     }
28 }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
7
8    <bean id="customEventHandler"
9       class="org.andrea.myexample.myCustomEvent.CustomEventHandler"/>
10
11   <bean id="customEventPublisher"
12      class="org.andrea.myexample.myCustomEvent.CustomEventPublisher"/>
13
14 </beans>
```

Andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output:

My Custom Event

My Custom Event

27 – Aspect Orienting Programming (AOP) in Spring

Posted on **9 aprile 2013** Views: 1412

Uno dei componenti chiave del Framework Spring è il modulo dedicato all'**Aspect Oriented Programmin (AOP)**

La cosidetta Programmazione Orientata agli Aspetti comporta la disgregazione della logica di programmazione in parti distinte che prendono il nome di concerns (interessi).

Le funzioni che coprono più punti di un'applicazione sono chiamate **cross-cutting concerns** e sono concettualmente separate dalla logica di business dell'aplicazione.

Esistono vari buoni esempi di uso comune che riguardano l'Aspect Oriented Programming, come ad esempio: il logging, l'auditing, le tranfazioni dichiarative, la sechurity, il caching, etc.

L'unità fondamentale della modularità nella programmazione orientata agli oggetti è la classe, mentre nell'AOP l'unità fondamentale della modularità è l'aspetto.
L'iniezione della dipendenza ci aiuta a disaccoppiare gli uni dagli altri gli oggetti della nostra applicazione, mentre l'AOP ci aiuta a disaccoppiare i cross-cutting concerns dagli oggetti su cui lavorano. Possiamo considerare l'AOP come una funzionalità simile ai triggers in linguaggi di programmazione come Perl, .NET, Java ed altri ancora.

Il modulo AOP di Spring fornisce intercettatori per intercettare una richiesta, come la richiesta di esecuzione di un metodo; Potremmo ad esempio aggiungere funzionalità extra prima o dopo l'esecuzione del metodo.

Terminologia relativa ad AOP:
Prima di iniziare a lavorare con il modulo AOP di Spring, iniziamo a vedere i concetti fondamentali e la terminologia relativa ad AOP. Questi termini non sono specifici di Spring ma sono correlati ad AOP.

TERMINE	DESCRIZIONE
Aspect	Un modulo dotato di un insieme di API che provvedono a requisiti cross-cutting. Per esempio, un modulo per il logging dovrebbe essere chiamato AOP aspect per il logging. Un'applicazione può avere un numero arbitrario di aspetti dipendenti dai suoi requisiti.
Joint Point	Rappresente un punto un una nostra applicazione in cui possiamo collegare un aspetto AOP. Possiamo considerarlo come il punto all'interno dell'applicazione in cui un azione sarà accettata usando il modulo AOP di Spring.
Advice	Rappresenta l'azione da eseguire o prima o dopo l'esecuzione del metodo. E' la frazione di codice che viene invocata dal modulo AOP di Spring durante l'esercuzione dell'applicazione.
Pointcut	Rappresenta un insieme di uno o più Joint Point in cui un advice può essere eseguito. Possiamo sepcificare Pointcut usando espressioni o pattern, come vedremo più avanti nell'esempio riguardante AOP.
Introduction	Un'introduction ci permette di aggiungere nuovi metdoi o nuovi attributi ad una classe esistente.
Target	L'oggetto che viene segnalato da uno o più aspetti, tale oggetto sarà sempre un'oggetto

TERMINE	DESCRIZIONE
Object	proxato. Tale oggetto viene anche chiamato come adviced object.
Weaving	Weaving è il processo di collegamento tra gli aspetti ed applicazioni di altro tipo o tra aspetti ed oggetti, al fine di creare un Target Objetct. Ciò può essere fatto a Compile Time, Load Time o a Run Time.

Tipi di Advice:

Gli Aspetti di Spring possono lavorare con 5 tipi di Advice:

ADVICE	DESCRIZIONE
before	Esegue l'advice prima dell'esecuzione del metodo.
after	Esegue l'advice dopo l'esecuzione di un metodo indipendentemente dal suo esito.
after-returning	Esegue l'advice dopo l'esecuzione di un metodo solo nel caso in cui il metodo termina con successo.
after-throving	Esegue l'advice dopo l'esecuzione di un metodo solo nel caso in cui esca dal metodo sollevando un'eccezione.
around	Esegue l'advice prima e dopo l'esecuzione del metodo su cui è applicato l'advice

Implementazioni di Aspetti Personalizzati:

Il Framework Spring supporta lo stile di annotazioni @AspectJ e ll'approccio schema-based che consente di implementare aspetti personalizzati. Entrambi questi approcci saranno spiegati nel dettaglio nei seguenti due capitoli, di seguito una breve descrizione:

Approccio	**DESCRIZIONE**
XML Based Schema	Gli Aspetti vengono implementati medianti normali classi Java insieme ad una configurazione basata su XML.
@AspectJ Annotation	Con @AspectJ gli aspetti vengono dichiarati medianti normali classi Java annotate con annotazioni appartenenti Java 5.

28 – Programmazione Orientata agli Aspetti in Spring, configuraziome mediante XML

Posted on **9 aprile 2013** Views: 524

Per poter usare i namespace tag descritti in questo articolo dobbiamo importare lo schema di spring-aop come descritto quì sotto:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2     <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xmlns:aop="http://www.springframework.org/schema/aop"
5          xsi:schemaLocation="http://www.springframework.org/schema/beans
6          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7          http://www.springframework.org/schema/aop
8          http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
9
10        <!-- bean definition & AOP specific configuration -->
11
12     </beans>
```

Per funzionare avremmo bisogno delle seguenti librerie AspectJ all'interno del classpath della nostra applicazione: aspectjrt.jar, aspectjweaver.jar, aspectj.jar.

Vedremo comunque più avanti come creare un'applicazione che fà uso di tali librerie mediante l'uso di Maven (senza doverle aggiungere manualmente).

Dichiarare gli Aspetti:

Un **Aspetto** viene dichiarato mediante l'uso del tag **<aop:aspect>** ed il backing bean viene referenziato usando l'attributo **ref**, come mostrato di seguito:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
  ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
...
</bean>
```

In questo caso viene configurato un bean avente ID=aBean ed esso viene iniettato come ogni altro bean di Spring visto nei precedenti articoli.

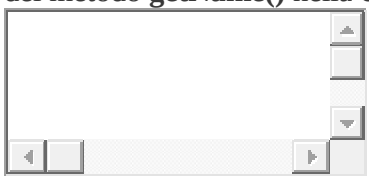
Dichiarare un Pointcut:

Un **pointcut** aiuta a determinare il **joint point** che ci interessa per essere eseguito con **advices**differente. Quando lavoriamo mediante la configurazione basata su XML Schema, i **pointcut** saranno definiti come segue:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">

  <aop:pointcut id="businessService"
    expression="execution(* com.xyz.myapp.service.*.*(..))"/>
  ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
```

```
11  ...
12  </bean>
```

Il seguente esempio definisce un **pointcut** chiamato "**businessService**" che combacia con l'esecuzione del metodo **getName()** nella classe **Student** dek package com.xyz



```
1   <aop:config>
2     <aop:aspect id="myAspect" ref="aBean">
3
4     <aop:pointcut id="businessService"
5       expression="execution(* com.xyz.Student.getName(..))"/>
6     ...
7     </aop:aspect>
8   </aop:config>
9
10  <bean id="aBean" class="...">
11  ...
12  </bean>
```

Dichiarare gli Advice:

Possiamo dichiarare qualsiasi advice, appartenenti ad una delle cinque tipologie viste nel precedente articolo, all'interno del tag **<aop:aspect>** usando il tag **<aop:{ADVICE NAME}>** come mostrato quì sotto:

```
1   <aop:config>
2     <aop:aspect id="myAspect" ref="aBean">
3       <aop:pointcut id="businessService"
4         expression="execution(* com.xyz.myapp.service.*.*(..))"/>
5
6       <!-- a before advice definition -->
```

```
7         <aop:before pointcut-ref="businessService"
8           method="doRequiredTask"/>
9
10        <!-- an after advice definition -->
11        <aop:after pointcut-ref="businessService"
12          method="doRequiredTask"/>
13
14        <!-- an after-returning advice definition -->
15        <!--The doRequiredTask method must have parameter named retVal -->
16        <aop:after-returning pointcut-ref="businessService"
17          returning="retVal"
18          method="doRequiredTask"/>
19
20        <!-- an after-throwing advice definition -->
21        <!--The doRequiredTask method must have parameter named ex -->
22        <aop:after-throwing pointcut-ref="businessService"
23          throwing="ex"
24          method="doRequiredTask"/>
25
26        <!-- an around advice definition -->
27        <aop:around pointcut-ref="businessService"
28          method="doRequiredTask"/>
29      ...
30      </aop:aspect>
31    </aop:config>
32
33    <bean id="aBean" class="...">
34    ...
35    </bean>
```

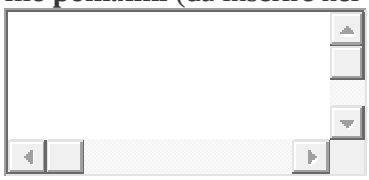
NB: Possiamo sia usare sempre lo stesso metodo per tutti gli advice, in questo caso doRequiredTask(), sia usare differenti metodi per differenti advices. Questi metodi saranno definiti come parte del modulo aspect.

Esempio:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample
Artifact **Id:** mySpringXmlAOP
Version: lasciate 0.0.1-SNAPSHOT
Clickate su Finish.

Aprite il file pom.xml e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**, **spring-aop**.
Dobbiamo anche inserire le dipendenze verso i seguenti pacchetti: **org.aspectj.spectjtools** e**cglib.cglib**
Non avendoli già nel mio repository locale, tali dipendenze saranno inseriti manualmente nel file**pom.xml**
Qualora non abbiate già scaricato Spring all'interno del vostro repository locale di Maven non avrete la possibilità di scegliere le dipendenze come visto appena sopra, quindi, inserite il seguente codice nel file **pom.xml** (da inserire nel tab pom.xml del wizard visualizato):

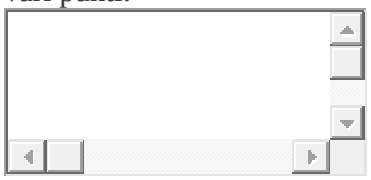


```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>mySpringXmlAOP</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>mySpringXmlAOP</name>
11   <url>http://maven.apache.org</url>
```

```
12
13  <properties>
14      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18      <dependency>
19          <groupId>junit</groupId>
20          <artifactId>junit</artifactId>
21          <version>3.8.1</version>
22          <scope>test</scope>
23      </dependency>
24      <dependency>
25          <groupId>org.springframework</groupId>
26          <artifactId>spring-core</artifactId>
27          <version>3.1.1.RELEASE</version>
28      </dependency>
29      <dependency>
30          <groupId>org.springframework</groupId>
31          <artifactId>spring-beans</artifactId>
32          <version>3.1.1.RELEASE</version>
33      </dependency>
34      <dependency>
35          <groupId>org.springframework</groupId>
36          <artifactId>spring-context</artifactId>
37          <version>3.1.1.RELEASE</version>
38      </dependency>
39      <dependency>
40          <groupId>org.springframework</groupId>
41          <artifactId>spring-context-support</artifactId>
42          <version>3.1.1.RELEASE</version>
43      </dependency>
```

```
44
45        <dependency>
46            <groupId>org.springframework</groupId>
47            <artifactId>spring-aop</artifactId>
48            <version>3.1.1.RELEASE</version>
49        </dependency>
50
51        <dependency>
52            <groupId>org.aspectj</groupId>
53            <artifactId>aspectjtools</artifactId>
54            <version>1.6.2</version>
55        </dependency>
56
57        <dependency>
58            <groupId>cglib</groupId>
59            <artifactId>cglib</artifactId>
60            <version>2.2</version>
61        </dependency>
62    </dependencies>
63 </project>
```

Di seguito il contenuto della classe **Logging.java** (da inserire all'interno del nostro unico package). Tale classe rappresenta un esempio di modulo aspect che definisce i metodi che devono essere chiamati nei vari punti:

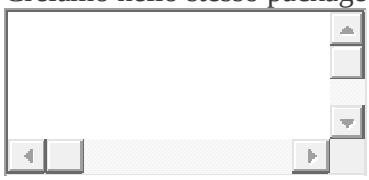


```
1  package org.andrea.myexample.mySpringXmlAOP;
2
3  /** Classe che rappresenta un modulo Aspect: definisce i metodi che verranno
4   *  chiamati nei vari punti
5   */
6  public class Logging {
```

```
7
8   /**
9    * Metodo che deve essere eseguito prima dell'esecuzione di un determinato
10   * metodo selezionato
11   */
12  public void beforeAdvice() {
13      System.out.println("Going to setup student profile.");
14  }
15
16  /**
17   * Metodo che deve essere eseguito dopo dell'esecuzione di un determinato
18   * metodo selezionato
19   */
20  public void afterAdvice() {
21      System.out.println("Student profile has been setup.");
22  }
23
24  /**
25   * Metodo che deve essere eseguito dopo l'esecuzione di un metodo
26   * selezionato, solo nel caso in cui questo ritorna un valore con successo
27   */
28  public void afterReturningAdvice(Object retVal) {
29      System.out.println("Returning:" + retVal.toString());
30  }
31
32  /**
33   * Metodo che deve essere eseguito se, durante l'esecuzione du un metodo
34   * selezionato, viene sollevata un'eccezione
35   */
36  public void AfterThrowingAdvice(IllegalArgumentException ex) {
37      System.out.println("There has been an exception: " + ex.toString());
38  }
```

```
39
40 }
```

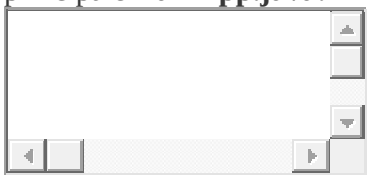
Creiamo nello stesso package la classe **Student.java**



```
1  package org.andrea.myexample.mySpringXmlAOP;
2
3  public class Student {
4        private Integer age;
5        private String name;
6
7        public void setAge(Integer age) {
8           this.age = age;
9        }
10
11       public Integer getAge() {
12          System.out.println("Age : " + age );
13          return age;
14       }
15
16       public void setName(String name) {
17          this.name = name;
18       }
19       public String getName() {
20          System.out.println("Name : " + name );
21          return name;
22       }
23
24       public void printThrowException(){
25           System.out.println("Exception raised");
26           throw new IllegalArgumentException();
```

```
27      }
28
29 }
```

Infine creiamo, sempre nell'unico package presente nel nostro progetto, la classe principale**MainApp.java:**



```
1  package org.andrea.myexample.mySpringXmlAOP;
2
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  // Classe principale:
7  public class MainApp {
8
9     public static void main(String[] args) {
10
11       /* Crea il contesto in base alle impostazioni definite nel
12        * file Beans.xmll
13        */
14       ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
15
16       // Recupera dal contesto il bean avente ID=student
17       Student student = (Student) context.getBean("student");
18
19       // Chiamate ai metodi per cui sono stati definiti gli aspetti:
20       student.getName();
21       student.getAge();
22
23       student.printThrowException();
24    }
```

```
25 }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6     http://www.springframework.org/schema/aop
7     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
8
9     <aop:config>
10       <aop:aspect id="log" ref="logging">
11          <aop:pointcut id="selectAll"
12             expression="execution(* org.andrea.myexample.mySpringXmlAOP.*.*(..))" />
13          <aop:before pointcut-ref="selectAll" method="beforeAdvice" />
14          <aop:after pointcut-ref="selectAll" method="afterAdvice" />
15          <aop:after-returning pointcut-ref="selectAll"
16             returning="retVal" method="afterReturningAdvice" />
17          <aop:after-throwing pointcut-ref="selectAll"
18             throwing="ex" method="AfterThrowingAdvice" />
19       </aop:aspect>
20    </aop:config>
21
22    <!-- Definizione del bean student -->
23    <bean id="student" class="org.andrea.myexample.mySpringXmlAOP.Student">
24       <property name="name" value="Zara" />
```

```
25        <property name="age" value="11" />
26     </bean>
27
28     <!-- Definizione per l'aspect per il logging -->
29     <bean id="logging" class="org.andrea.myexample.mySpringXmlAOP.Logging" />
30
31 </beans>
```

A questo punto andiamo ad eseguire la classe **MainApp** e dovremmo ottenere il seguente messaggio di output:

```
1  <strong>Going to setup student profile.
2  Name : Zara
3  Student profile has been setup.
4  Returning:Zara
5  Going to setup student profile.
6  Age : 11
7  Student profile has been setup.
8  Returning:11
9  Going to setup student profile.
10 Exception raised
11 Student profile has been setup.
12 There has been an exception: java.lang.IllegalArgumentException
13 .....
14 ALTRO CONTENUTO DELL'ECCEZIONE CHE È STATA SOLLEVATA</strong>
```

Andiamo ora ad analizare il file di configurazione preso in considerazione.

Il tag **<aop:pointcut>** definisce appunto il **Pointcut** ovvero un insieme di punti (i **JointPoint**) in cui possiamo collegare gli **Advice** (le azioni che possono essere eseguite, i nostri metodi). In questo caso stiamo usando un’espressione regolare che seleziona tutti i metodi definiti all’interno del package:**org.andrea.myexample.mySpringXmlAOP**

L’espressione regolare di tale caso è appunto: *** org.andrea.myexample.mySpringXmlAOP.*.*(..))**
Nel caso invece volessimo eseguire il nostro **Advice** prima e dopo l’esecuzione di un singolo specifico metodo, potremmo definire il nostro **Pointcut** in modo da restringere l’insieme dei **JointPoint** in cui devono eseguiti gli **Advice**. Per fare ciò, nella definizione dei **Pointcut** dobbiamo rimpiazzare il simbolo star (*****) con il nome reale della classe e del metodo. Di seguito un esempio di tale concetto costruito modificando il file di configurazione **Beans.xml** visto in precedenza:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:aop="http://www.springframework.org/schema/aop"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/aop
8     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
9
10   <aop:config>
11   <aop:aspect id="log" ref="logging">
12      <aop:pointcut id="selectAll"
13      expression="execution(* org.andrea.myexample.mySpringXmlAOP.Student.getName(..))"/>
14      <aop:before pointcut-ref="selectAll" method="beforeAdvice"/>
15      <aop:after pointcut-ref="selectAll" method="afterAdvice"/>
16   </aop:aspect>
17   </aop:config>
18
19   <!-- Definition for student bean -->
20   <bean id="student" class="org.andrea.myexample.mySpringXmlAOP.Student">
21      <property name="name"  value="Zara" />
22      <property name="age"  value="11"/>
23   </bean>
24
```

```
25    <!-- Definition for logging aspect -->

26    <bean id="logging" class="org.andrea.myexample.mySpringXmlAOP.Logging"/>

27

28 </beans>
```

Andando ora ad eseguire la classe principale **MainApp** usando tale file di configurazione (al posto dell'originale) otterremo quindi il seguente output che mostra come l'**Advice** viene eseguito solo per il**JointPoint** relativo al metodo **getName()**:

```
1 <strong>Going to setup student profile.

2 Name : Zara

3 Student profile has been setup.

4 Age : 11

5 Exception raised

6 .....

7 other exception content</strong>
```

29 – Programmazione Orientata agli Aspetti (AOP), configurazione mediante @AspectJ

Posted on **10 aprile 2013** Views: 651

@AspectJ si riferisce alla modalità di dichiarere gli aspetti mediante normali classi Java annotate mediante le annotazioni di Java 5. Il supporto ad **@AspectJ** può essere abilitato includendo il seguente elemento all'interno del nostro file di configurazione XML:

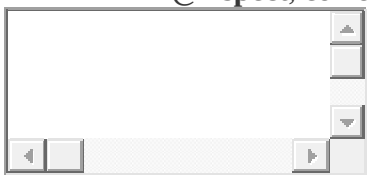
```
1 <aop:aspectj-autoproxy/>
```

Inoltre bisogna inserire le seguenti librerie di **AspectJ** nel classpath della nostra applicazione:**aspectjrt.jar**, **aspectjweaver.jar**, **aspectj.jar**

Nell'esempio proposto in questo articolo vedremo come creare un progetto che usi Maven per il processo di build, così da evitare di dover inserire manualmente tali librerie all'interno del nostro progetto.

Dichiarare un Aspetto:

Le classi che rapprsentano gli aspetti sono come ogni altro bean e possono avere metodi e campi proprio come ogni altra classe, l'unica differenza è che devono essere annotate mediante l'annotazione**@Aspect**, come mostrato di seguito:



```
1 package org.xyz;
2
3 import org.aspectj.lang.annotation.Aspect;
4
5 @Aspect
6 public class AspectModule {
7
8 }
```

Tale classe dovrà poi essere configurata all'interno del file di configurazione XML come ogni altro bean:

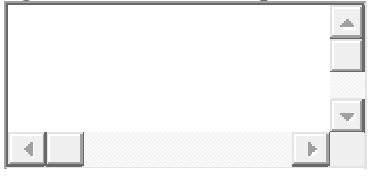
```
1 <bean id="myAspect" class="org.xyz.AspectModule">
2     <!-- Configuriamo le proprietà di un aspetto come un normale bean  -->
3 </bean>
```

Dichiarare un PointCut:

Un **PointCut** aiuta a determinare i **JointPoint** (vale a dire metodi) di interesse che devono essere eseguite con diversi **Advice**. Quando lavoriamo usando la configurazione basata su **@AspectJ** la dichiarazione di un **Pointcut** è composta da due parti:

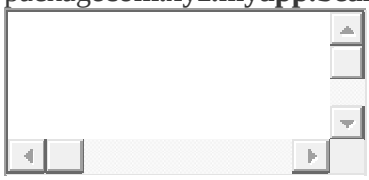
1. **Un'espressione pointcut che determina esattamente quali esecuzioni di metodi ci interessano.**
2. **Una signature del pointcut comprendente un nome e un numero qualsiasi di parametri. Il corpo effettivo del metodo è irrilevante e in realtà dovrebbe essere vuoto.**

Il seguente esempio definisce un **PointCut** chiamato "**businessServiceb** che matcha con l'esecuzuine di ogni metodo reso disponibile all'interno delle classi che si trovano nel package**com.xyz.myapp.service**



```
1 import org.aspectj.lang.annotation.Pointcut;
2
3 @Pointcut("execution(* com.xyz.myapp.service.*.*(..))") // expression
4 private void businessService() {}  // signature
```

Il seguente esempio definisce un **PointCut** chiamato "**getName**" che matcha con l'eseguzione del metodo **getName()** reso disponibile dalla della classe **Student** che si trova all'interno del package**com.xyz.myapp.beans**



```
1 import org.aspectj.lang.annotation.Pointcut;
2
3 @Pointcut("execution(* com.xyz.myapp.beans.Student.getName(..))")
4 private void getname() {}
```

Dichiarare gli Advice:

Possiamo dichiarare un qualsiasi tipo di **Adivice**, appartenente ad una delle cinque tipologie di Advice rese disponibili da Spring, usando le annotazioni **@{ADVICE-NAME}** come mostrato di seguito. Nel seguente esempio stiamo assumendo di aver già definito la signature di un metodo **PointCut** chiamato**businessService()**.

```
 1  @Before("businessService()")
 2  public void doBeforeTask(){
 3    ...
 4  }
 5
 6  @After("businessService()")
 7  public void doAfterTask(){
 8    ...
 9  }
10
11 @AfterReturning(pointcut = "businessService()", returning="retVal")
12 public void doAfterReturnningTask(Object retVal){
13    // you can intercept retVal here.
14    ...
15 }
16
17 @AfterThrowing(pointcut = "businessService()", throwing="ex")
18 public void doAfterThrowingTask(Exception ex){
19    // you can intercept thrown exception here.
20    ...
21 }
22
23 @Around("businessService()")
24 public void doAroundTask(){
25   ...
26 }
```

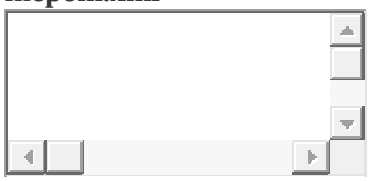
Possiamo anche definire un **PointCut inline** per un qualsiasi specifico b. Di seguito un esempio di come definire un **PointCut per l'Advice before**:

```
1 @Before("execution(* com.xyz.myapp.service.*.*(..))")
2 public doBeforeTask(){
3  ...
4 }
```

Esempio progetto AOP mediante @AspectJ:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven.
Aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**.
Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

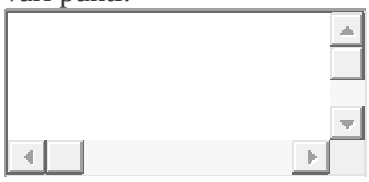
Group **Id:** org.andrea.myexample
Artifact **Id:** mySpringAspectJAOP
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file pom.xml e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support**.
Dobbiamo anche inserire le dipendenze verso i seguenti pacchetti: **org.org.aspectj.aspectjweaver**
Non avendoli già nel mio repository locale, tali dipendenze saranno inseriti manualmente nel file**pom.xml**



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>mySpringAspectJAOP</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>mySpringAspectJAOP</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
```

```
14      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15    </properties>
16
17    <dependencies>
18      <dependency>
19        <groupId>junit</groupId>
20        <artifactId>junit</artifactId>
21        <version>3.8.1</version>
22        <scope>test</scope>
23      </dependency>
24      <dependency>
25        <groupId>org.springframework</groupId>
26        <artifactId>spring-core</artifactId>
27        <version>3.1.1.RELEASE</version>
28      </dependency>
29      <dependency>
30        <groupId>org.springframework</groupId>
31        <artifactId>spring-beans</artifactId>
32        <version>3.1.1.RELEASE</version>
33      </dependency>
34
35      <dependency>
36        <groupId>org.springframework</groupId>
37        <artifactId>spring-context</artifactId>
38        <version>3.1.1.RELEASE</version>
39      </dependency>
40      <dependency>
41        <groupId>org.springframework</groupId>
42        <artifactId>spring-context-support</artifactId>
43        <version>3.1.1.RELEASE</version>
44      </dependency>
45
46      <dependency>
47        <groupId>org.aspectj</groupId>
48        <artifactId>aspectjweaver</artifactId>
49        <version>1.6.8</version>
50      </dependency>
51
52      <dependency>
53        <groupId>cglib</groupId>
54        <artifactId>cglib</artifactId>
55        <version>2.2</version>
56      </dependency>
57
58    </dependencies>
59  </project>
```

Di seguito il contenuto della classe **Logging.java** (da inserire all'interno del nostro unico package). Tale classe rappresenta un esempio di modulo aspect che definisce i metodi che devono essere chiamati nei vari punti:



```
1  package org.andrea.myexample.mySpringAspectJAOP;
2
3  import org.aspectj.lang.annotation.Aspect;
4  import org.aspectj.lang.annotation.Pointcut;
5  import org.aspectj.lang.annotation.Before;
6  import org.aspectj.lang.annotation.After;
7  import org.aspectj.lang.annotation.AfterThrowing;
8  import org.aspectj.lang.annotation.AfterReturning;
9  import org.aspectj.lang.annotation.Around;
```

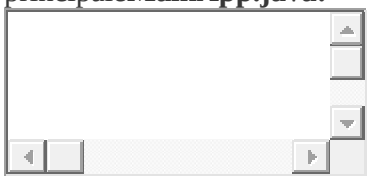
```
10
11 // Classe che definisce gli aspetti:
12 @Aspect
13 public class Logging {
14
15    /**
16     * Di seguito la definizione per un PointCut che seleziona tutti i metodi
17     * disponibili nelle classi presenti nell'unico package del progetto- Così
18     * facendo, gli Advices saranno chiamati per tutti i metodi.
19     */
20    @Pointcut("execution(* org.andrea.myexample.mySpringAspectJAOP.*.*(..))")
21    private void selectAll() {
22    }
23
24    /**
25     * Metodo che deve essere eseguito PRIMA dell'esecuzione di uno dei metodi
26     * selezionati.
27     */
28    @Before("selectAll()")
29    public void beforeAdvice() {
30       System.out.println("Going to setup student profile.");
31    }
32
33    /**
34     * Metodo che deve essere eseguito DOPO l'esecuzione di uno dei metodi
35     * selezionati.
36     */
37    @After("selectAll()")
38    public void afterAdvice() {
39       System.out.println("Student profile has been setup.");
40    }
41
42    /**
43     * Metodo che deve essere eseguito quando un metodo selezionato TERMINA la
44     * sua esecuzione RITORNANDO UN VALORE e senza sollevare un'eccezione.
45     */
46    @AfterReturning(pointcut = "selectAll()", returning = "retVal")
47    public void afterReturningAdvice(Object retVal) {
48       System.out.println("Returning:" + retVal.toString());
49    }
50
51    /**
52     * Metodo che deve essere eseguito quando un metodo selezionato TERMINA la
53     * sua esecuzione SOLLEVANDO UN'ECCEZIONE.
54     */
55    @AfterThrowing(pointcut = "selectAll()", throwing = "ex")
56    public void AfterThrowingAdvice(IllegalArgumentException ex) {
57       System.out.println("There has been an exception: " + ex.toString());
58    }
59
60 }
```

Creiamo nello stesso package la classe **Student.java**:

```
1  package org.andrea.myexample.mySpringAspectJAOP;
2
3  public class Student {
4     private Integer age;
5     private String name;
6
7     public void setAge(Integer age) {
8        this.age = age;
```

```
 9     }
10
11     public Integer getAge() {
12        System.out.println("Age : " + age);
13        return age;
14     }
15
16     public void setName(String name) {
17        this.name = name;
18     }
19
20     public String getName() {
21        System.out.println("Name : " + name);
22        return name;
23     }
24
25     public void printThrowException() {
26        System.out.println("Exception raised");
27        throw new IllegalArgumentException();
28     }
29 }
```

Infine creiamo, sempre nell'unico package presente nel nostro progetto, la classe principale**MainApp.java:**



```
 1  package org.andrea.myexample.mySpringAspectJAOP;
 2
 3  import org.springframework.context.ApplicationContext;
 4  import org.springframework.context.support.ClassPathXmlApplicationContext;
 5
 6  // Classe principale:
 7  public class MainApp {
 8
 9     public static void main(String[] args) {
10
11        /* Crea il contesto in base alle impostazioni definite nel
12         * file Beans.xmll
13         */
14        ApplicationContext context = new ClassPathXmlApplicationContext(
15              "Beans.xml");
16
17        // Recupera dal contesto il bean avente ID=student
18        Student student = (Student) context.getBean("student");
19
20        // Chiamate ai metodi per cui sono stati definiti gli aspetti:
21        student.getName();
22        student.getAge();
23
24        student.printThrowException();
25     }
26  }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:aop="http://www.springframework.org/schema/aop"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7     http://www.springframework.org/schema/aop
8     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
9
10    <aop:aspectj-autoproxy/>
11
12    <!-- Definition for student bean -->
13    <bean id="student" class="org.andrea.myexample.mySpringAspectJAOP.Student">
14       <property name="name"  value="Zara" />
15       <property name="age"  value="11"/>
16    </bean>
17
18    <!-- Definition for logging aspect -->
19    <bean id="logging" class="org.andrea.myexample.mySpringAspectJAOP.Logging"/>
20
21 </beans>
```

Notate come, a differenza del file di configurazione del precedente articolo in cui dichiaravamo tutta la configurazione degli aspetti all'interno della configurazione XML, in questo caso nel file di configurazione XML oltre alla configurazione dei bean troviamo solamente l'elemento **<aop:aspectJ-autoproxy/>** che abilita l'uso delle annotazioni **@AspectJ**.

Pertanto tutta la configurazione di AOP viene racchiusa direttamente nelle classi Java e delegata alle annotazioni **@AspectJ**.

A questo punto andiamo ad eseguire la classe **MainApp** e dovremmo ottenere il seguente messaggio di output:

Going to setup student profile.
Name : Zara
Student profile has been setup.
Returning:Zara
Going to setup student profile.
Age : 11
Student profile has been setup.
Returning:11
Going to setup student profile.
Exception raised
Student profile has been setup.
There has been an exception: java.lang.IllegalArgumentException
Exception in thread "main" java.lang.IllegalArgumentException
…..
ALTRO CONTENUTO DELL'ECCEZIONE CHE È STATA SOLLEVATA

30 – Introduzione a JDBC in Spring

Posted on **15 aprile 2013** Views: 1063

Lavorando sui database usando il buon vecchio **JDBC** diventa scomodo perchè bisogna scrivere il codice della gestione delle eccezioni, quello riguardante l'apertura e la chiusura delle connessioni al datbase, etc. Fortunatamente il **Framework JDBC Spring si prende cura di tutti i dettagli a basso livello** a partire dall'apertura della connessione, alla preparazione ed esecuzione delle istruzioni SQL, al trattamento delle eccezione, alla gestione delle transazioni, fino alla chiusura della connessione.

Così ciò che dobbiamo fare è solo definire i parametri di connessione e specificare le istruzioni SQL che devono essere eseguite e svolgere il lavoro richiesto per ogni iterazione durante il recupero dei dati dal database.

Spring JDBC fornisce diversi approcci (e corrispondenti diverse classi) per interfacciarsi con il database. In questo articolo andremo a vedere l'approccio più comune che prevede l'uso della classe**JdbcTemplate** appartenente a tale Framework. Questa è la classe centrale del framework che gestisce tutte le comunicazioni con il database e la gestione delle eccezioni.

La classe JdbcTemplate:

La classe JdbcTemplate si occupa di eseguire le query SQL, eseguire le operazioni di update, eseguire le chiamate alle stored procedure, eseguire le iterazioni sul ResultSet ed eseguire l'estrazione dei valori dei parametri di ritorno. Inoltre cattura le ecezioni di JDBC e le traduce in eccezioni generiche maggiormente informative che vengono definite in modo gerarchico all'interno del package org.springframework.dao

Le istanze della classe **JdbcTemplate**, una volta configurate, sono **threadsafe**. Possiamo così configurare una singola istanza di **JdbcTemplate** e poi iniettarvi in maniera sicura un riferimento condiviso a più oggetti DAO.

Una pratica tipica quando utilizziamo la classe **JdbcTemplate** è quella di configurare un **DataSource**all'interno del nostro file di configurazione di Spring, e poi iniettare la dipendenza che condivide il bean**DataSoruce** all'interno delle nostro classi **DAO**.

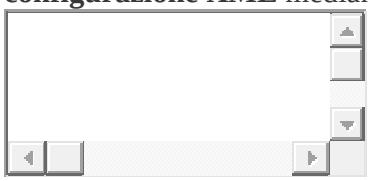
Configuriamo il DataSource:

Per prima cosa andiamo a creare una tabella **Student** all'interno di un nostro database chiamato**TEST**.

In questo articolo si assume che si stia lavorando usando **MySql** come database, se lavorate con un altro database allora dovrete cambiare di conseguenza le vostre DDL e le vostre query SQL.

```
1 CREATE TABLE Student(
2    ID   INT NOT NULL AUTO_INCREMENT,
3    NAME VARCHAR(20) NOT NULL,
4    AGE  INT NOT NULL,
5    PRIMARY KEY (ID)
6 );
```

Ora abbiamo bisogno di fornire un DataSource nel JdbcTemplate in modo che possa configurarsi così da ottenere l'accesso al database. Possiamo configurare il **DataSoruce** all'interno del nostro **file di configurazione XML** mediante il seguente frammento di codice:



```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

Definiamo il DAO (Data Access Object):

La sigla **DAO** stà per **Data Access Object**, si tratta di un oggetto usato comunemente per **gestire le interazioni con il database**. I **DAO** forniscono un mezzo per leggere e scrivere dati sul database e devono esporre questi funzionalità tramite un interfaccia attraverso la quale il resto dell'applicazione accede a tale componente.

I **Data Access Object** (**DAO**) supportati in Spring rendono facile lavorare con le varie tecnologie di accesso ai dati come **JDBC**, **Hibernate**, **JPA** o **JDO** in modo consistente.

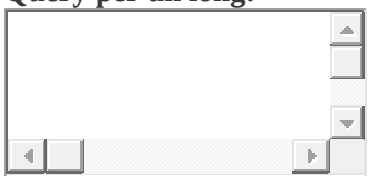
Eseguire le istruzioni SQL:

Andiamo a vedere come possiamo eseguire le operazioni di **CRUD** (**Create**, **Read**, **Destroy** ed**Update**) sulle tabelle del database usando l'oggetto **JdbcTemplate**.

Query per un intero:

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Query per un long:



```
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

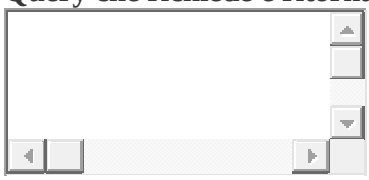
Una semplice query che usa una variabile di bind:

```
1 String SQL = "select age from Student where id = ?";
2 int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Query per una String:

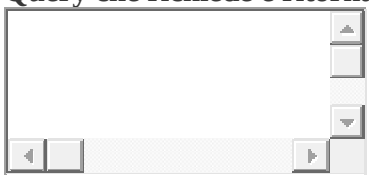
```
1 String SQL = "select name from Student where id = ?";
2 String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class);
```

Query che richiede e ritorna un oggetto:



```
 1  String SQL = "select * from Student where id = ?";
 2  Student student = jdbcTemplateObject.queryForObject(SQL,
 3                  new Object[]{10}, new StudentMapper());
 4
 5  public class StudentMapper implements RowMapper<student> {
 6     public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
 7        Student student = new Student();
 8        student.setID(rs.getInt("id"));
 9        student.setName(rs.getString("name"));
10        student.setAge(rs.getInt("age"));
11        return student;
12     }
13  }
```

Query che richiede e ritorna più oggetti:



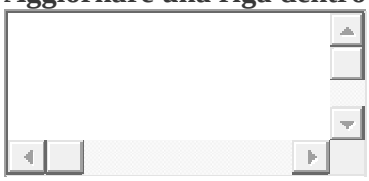
```
 1  String SQL = "select * from Student";
 2  List<student> students = jdbcTemplateObject.query(SQL,
 3                        new StudentMapper());
 4
 5  public class StudentMapper implements RowMapper<student> {
 6     public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
 7        Student student = new Student();
 8        student.setID(rs.getInt("id"));
 9        student.setName(rs.getString("name"));
10        student.setAge(rs.getInt("age"));
11        return student;
12     }
13  }
```

Inserire una riga all'interno di una tabella:

```
1 String SQL = "insert into Student (name, age) values (?, ?)";
```

```
2 jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Aggiornare una riga dentro la tabella:



```
1 String SQL = "update Student set name = ? where id = ?";
2 jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Eliminare una riga da una tabella:

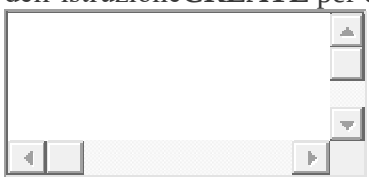
```
1 String SQL = "delete Student where id = ?";
2 jdbcTemplateObject.update( SQL, new Object[]{20} );
```

Eseguire istruzioni DDL (Data Definition Language):

Il **Data Definition Language** (**DDL**) è un linguaggio che permette di **creare**, **modificare** o **eliminare** gli oggetti **in un database** ovvero agire sullo schema di database. Sono i **comandi DDL** a **definire la struttura del database** e quindi l'**organizzazione logica dei dati in esso contenuti**, ma n**on fornisce gli strumenti per modificare i valori assunti dai dati o per interrogare i dati stessi** per il quale si usano rispettivamente il **Data Manipulation Language** e il **Data Query Language**.

Maggiori informazioni su DDL sono reperibili al seguente link: http://it.wikipedia.org/wiki/Data_Definition_Language

La classe **JdbcTemplate** mette a disposizione il metodo **execute(…)** che permette di **eseguire ogni istruzione SQL o le istruzioni DDL**. Di seguito è presentato un esempio dell'uso dell'istruzione**CREATE** per creare una tabella tramite **DDL**.



```
1 String SQL = "CREATE TABLE Student( " +
2   "ID   INT NOT NULL AUTO_INCREMENT, " +
3   "NAME VARCHAR(20) NOT NULL, " +
4   "AGE  INT NOT NULL, " +
5   "PRIMARY KEY (ID));"
6
7 jdbcTemplateObject.execute( SQL );
```

Nei prossimi due articoli presenteremo due **esempi pratic**i di uso del **Framework JDBC**:

1. **Esempio pratico JDBC in Spring:** Esempio che spiega come scrivere una semplice applicazione Spring basata sull'uso di JDBC per effettuare l'accesso ai dati.
2. **Stored Procedure SQL in Spring**: Esempio per imparare a chiamare le SQL Stored Procedure usando JDBC in un'applicazione Spring.

31 – Esempio pratico JDBC in Spring

Posted on **15 aprile 2013** Views: 886

Per capire i concetti legati all'uso di **JDBC**, all'interno di un'applicazione Spring, facendo uso della classe **JdbcTemplate** andiamo ad esaminare un semplice esempio che implementa tutte le operazioni di **CRUD** (**Create**, **Read**, **Destroy** ed **Update**) sulla seguente tabella **Student**:
In questo articolo si assume che si stia lavorando usando **MySql** come database, se lavorate con un altro database allora dovrete cambiare di conseguenza le vostre DDL e le vostre query SQL.

Creiamo il Database:

Per prima cosa accediamo al nostro database nel modo che riteniamo opportuno (attraverso riga di comando o usando qualche programma per interfacciarsi con il DB). Ad esempio considerando di usare Linux e di accedere al DB MySql dalla shell potremmo fare:

mysql -u root -p (ed inserire la password)
Potremmo poi creare un database dedicato a tale esempio tramite il comando MySql: **create database SpringTestDb;**
Accediamo ora al database appena creato lanciando il seguente comando MySql: **use SpringTestDb;**
Infine creiamo la tabella lanciando il seguente comando nella shell:

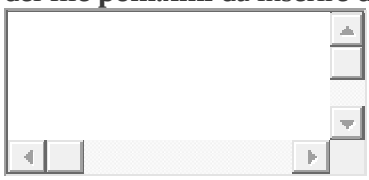
```
1 CREATE TABLE Student(
2    ID   INT NOT NULL AUTO_INCREMENT,
3    NAME VARCHAR(20) NOT NULL,
4    AGE  INT NOT NULL,
5    PRIMARY KEY (ID)
6 );
```

Creiamo il progetto in STS\Eclipse:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven usando STS\Eclipse.

All’interno di STS\Eclipse aprire l’elemento **Maven** e selezionare “**Maven Project**“. Clickate **Next**. Nella prima finestra dello wizard troveremo spuntato solo l’opzione “**Use default Workspace location**“, lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su “**maven-archetype-quickstart**” e clickate su **Next**.
Inserite i seguenti parametri:

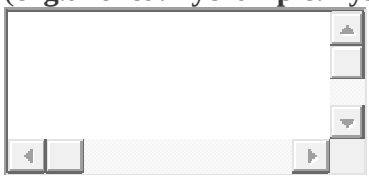
Group Id: org.andrea.myexample
Artifact Id: myJdbcSpringExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file pom.xml e clickate sul tab “**Dependencies**” per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support, spring-jdbc**).
Dovremmo inoltre aggiungere la seguente dipendenza relativa al driver MySql necessario per effettuare la connessione: **mysql-connector-java**
Qualora non riusciste a scaricare automaticamente le dipendenze attraverso il wizard, questo è il codice del file **pom.xml** da inserire direttamente nel file tramite il tab **pom.xml** della view relativa a tale file:



```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myJdbcSpringExample</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
9
10   <name>myJdbcSpringExample</name>
11   <url>http://maven.apache.org</url>
12
13   <properties>
14      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15   </properties>
16
17   <dependencies>
18
19   <dependency>
20     <groupId>junit</groupId>
21     <artifactId>junit</artifactId>
22     <version>3.8.1</version>
23     <scope>test</scope>
24   </dependency>
25   <dependency>
26      <groupId>org.springframework</groupId>
27      <artifactId>spring-core</artifactId>
28      <version>3.1.1.RELEASE</version>
29   </dependency>
30   <dependency>
31      <groupId>org.springframework</groupId>
32      <artifactId>spring-beans</artifactId>
```

```
33        <version>3.1.1.RELEASE</version>
34    </dependency>
35    <dependency>
36        <groupId>org.springframework</groupId>
37        <artifactId>spring-context</artifactId>
38        <version>3.1.1.RELEASE</version>
39    </dependency>
40    <dependency>
41        <groupId>org.springframework</groupId>
42        <artifactId>spring-context-support</artifactId>
43        <version>3.1.1.RELEASE</version>
44    </dependency>
45
46       <dependency>
47    <groupId>org.springframework</groupId>
48    <artifactId>spring-jdbc</artifactId>
49    <version>3.2.1.RELEASE</version>
50 </dependency>
51
52 <dependency>
53    <groupId>mysql</groupId>
54    <artifactId>mysql-connector-java</artifactId>
55    <version>5.1.23</version>
56 </dependency>
57
58   </dependencies>
59 </project>
```

All'interno dell'unico package presente nel nostro progetto (**org.andrea.myexample.myJdbcSpringExample**) andiamo a creare la classe **StudentDAO.java**:

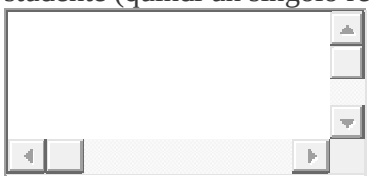


```
1  package org.andrea.myexample.myJdbcSpringExample;
2
3  import java.util.List;
4  import javax.sql.DataSource;
5
6  /* Interfaccia che rappresenta il nostro DAO per la tabella Student.
7   * Tale interfaccia definisce tutte le operazioni di CRUD per tale tabella
8   */
9  public interface StudentDAO {
10    /*
11     * Metodo usato per inizializzare le risorse del database (la connessione):
12     */
13    public void setDataSource(DataSource ds);
14
15    /*
16     * Metodo usato per creare un record nella tabella Student:
17     */
18    public void create(String name, Integer age);
19
20    /*
21     * Metodo usato per elencare un record, corrispondente ad un certo id
22     * passato, appartenente alla tabella Student:
23     */
24    public Student getStudent(Integer id);
25
26    /*
27     * Metodo che viene usato per elencare tutti i recordo appartenenti alla
28     * tabella Student:
29     */
30    public List<Student> listStudents();
```

```
31
32     /*
33      * Metodo che viene usato per eliminare un record, corrispondente ad un
34      * certo id passato, dalla tabella Student:
35      */
36     public void delete(Integer id);
37
38     /*
39      * Metodo che viene usato per eseguire l'update di un recordo che si trova
40      * all'interno della tabella Student:
41      */
42     public void update(Integer id, Integer age);
43 }
```

Tale classe rappresenta il nostro **DAO** per la tabella **Student**. Tale classe implementa tutte le operazioni di **CRUD** (**Create**, **Read**, **Destroy** ed **Update**) per operare su tale tabella.

Sempre nello stesso package andiamo ora a creare la seguente classe Java che rappresenta un generico studente (quindi un singolo record della nostra tabella): **Student.java**



```
 1  package org.andrea.myexample.myJdbcSpringExample;
 2
 3  // Classe che rappresenta un generico studente:
 4  public class Student {
 5
 6      // Proprietà:
 7      private Integer age;
 8      private String name;
 9      private Integer id;
10
11      // Metodi Getter & Setter:
12      public void setAge(Integer age) {
13          this.age = age;
14      }
15
16      public Integer getAge() {
17          return age;
18      }
19
20      public void setName(String name) {
21          this.name = name;
22      }
23
24      public String getName() {
25          return name;
26      }
27
28      public void setId(Integer id) {
29          this.id = id;
30      }
31
32      public Integer getId() {
33          return id;
34      }
35  }
```

Sempre nello stesso package andiamo ora a creare la seguente classe Java che rappresenta**StudentMapper.java**:

```
1  package org.andrea.myexample.myJdbcSpringExample;
2
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import org.springframework.jdbc.core.RowMapper;
6
7  /* Classe che implementa l'interfaccia RowMapper. Si tratta di un'interfaccia
8   * usata da JdbcTemplate per mappare le righe di un ResultSet (oggetto che
9   * contiene l'insieme delle righe restituite da una query SQL) riga per riga.
10  * Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto
11  * risultante senza doversi preoccupare della gestione delle eccezioni poichè
12  * le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.
13  */
14
15 public class StudentMapper implements RowMapper<Student> {
16
17     /* Implementazione del metodo dell'interfaccia RowMapper che mappa una
18      * specifica riga della tabella su di un oggetto Student
19      *
20      * @param Un oggetto ResultSet contenente l'insieme di tutte le righe
21      *        restituite dalla query
22      *
23      * @param L'indice che indentifica una specifica riga
24      *
25      * @return Un nuovo oggetto Student rappresentante la riga selezionata
26      *         all'interno dell'oggetto ResultSet
27      *
28      * @see org.springframework.jdbc.core.RowMapper#mapRow(java.sql.ResultSet, int)
29      */
30     public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
31         Student student = new Student();
32         student.setId(rs.getInt("id"));
33         student.setName(rs.getString("name"));
34         student.setAge(rs.getInt("age"));
35         return student;
36     }
37 }
```

Tale Classe che implementa l'interfaccia RowMapper che è un'interfaccia messa a disposizione da Spring usata da JdbcTemplate per mappare le righe di un ResultSet (l'oggetto che contiene l'insieme delle righe restituite da una query SQL) riga per riga.

Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto risultante senza doversi preoccupare della gestione delle eccezioni poich le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.

Gli oggetti RowMapper sono tipicamente stateless e quindi riutilizzabili.

Vediamo come la classe **StudentMapper** implementa il metodo **mapRow(ResultSet rs, int rowNum)**che prende in input l'oggetto ResultSet contenente l'insieme delle righe della tabella restituite dalla query eseguita ed un parametro intero che rappresenta l'indice di una specifica riga. Per tale

specifica riga viene creato e restituito un **oggetto Student** che contiene le informazioni all'interno di tale riga nell'oggetto **RowMapper**.

Maggiori informazioni sul **ResultSet** quì: http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html

Maggiori informazioni su **RowMapper** quì: http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jdbc/core/RowMapper.html

Di seguito è presentata la classe **StudentJDBCTemlpate.java** che costituisce un'**implementazione**per il nostro **DAO** le cui funzionalità di **CRUD** sono state definite tramite l'**interfaccia StudentDAO**:

```
1  package org.andrea.myexample.myJdbcSpringExample;
2
3  import java.util.List;
4  import javax.sql.DataSource;
5  import org.springframework.jdbc.core.JdbcTemplate;
6
7  /* Classe che fornisce l'implementazione per il nostro DAO le cui funzionalità
8   * di CRUD sono state definite tramite l'interfaccia StudentDAO
9   */
10 public class StudentJDBCTemplate implements StudentDAO {
11
12    // Proprietà:
13    private DataSource dataSource;
14    private JdbcTemplate jdbcTemplateObject;
15
16    /* Metodo Setter per l'Injection della dipendenza relativa alla sorgente
17     * dati.
18     * Tale metodo inoltre costruisce anche l'oggetto istanza di JdbcTemplate
19     * usato per interagire con i dati nel database.
20     *
21     * @param la sorgente dati
22     * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#setDataSource(javax.sql.DataSource)
23     */
24    public void setDataSource(DataSource dataSource) {
25       this.dataSource = dataSource;
26       this.jdbcTemplateObject = new JdbcTemplate(dataSource);
27    }
28
29    /* Metodo che inserisce un nuovo record, rappresentante uno studente,
30     * dentro la tabella.
31     *
32       @param nome studente
33       @param età studente
34     * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#create(java.lang.String, java.lang.Integer)
35     */
36    public void create(String name, Integer age) {
37       // Rappresenta la query:
38       String SQL = "insert into Student (name, age) values (?, ?)";
39
40       // Esegue la query passandogli anche i valori effettivi da inserire:
41       jdbcTemplateObject.update(SQL, name, age);
42
43       System.out.println("Created Record Name = " + name + " Age = " + age);
44       return;
45    }
46
47    /* Metodo che recupera un record, rappresentante uno studente, avente uno
```

```
48      * specifico id dalla tabella.
49      *
50      * @param L'id dello studente
51      * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#getStudent(java.lang.Integer)
52      */
53     public Student getStudent(Integer id) {
54        // Rappresenta la query che seleziona una specifica riga della tabella:
55        String SQL = "select * from Student where id = ?";
56
57        /*
58         * Ottengo l'oggetto Student invocando il metodo queryForObject
59         * sull'oggetto JdbcTemplate passandogli i seguenti parametri.
60         *
61         * @param La query per creare il preparated statement
62         * @param Una lista di argomenti contenente il singolo id
63         * @param Un oggetto che implementa RowMapper che viene usato per
64         *        mappare una singola riga della tabella su di un oggetto Java
65         */
66        Student student = jdbcTemplateObject.queryForObject(SQL,
67                        new Object[] { id }, new StudentMapper());
68        return student;
69     }
70
71     /* Metodo che recupera la lista di tutti gli studenti rappresentanti dalle
72      * righe della tabella Student
73      *
74      * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#listStudents()
75      */
76     public List<Student> listStudents() {
77        // Rappresenta la query che seleziona TUTTE le righe della tabella
78        String SQL = "select * from Student";
79
80        /* Ottengo la lista degli oggetti Student corrispondenti a tutte le
81         * righe della tabella invocando il metodo queryForObject sull'oggetto
82         * JdbcTemplate passandogli i seguenti parametri.
83         *
84         * @param La query per creare il preparated statement
85         * @param Un oggetto che implementa RowMapper che viene usato per
86         *        mappare una singola riga della tabella su di un oggetto Java
87         */
88        List<Student> students = jdbcTemplateObject.query(SQL,
89                                        new StudentMapper());
90        return students;
91     }
92
93     /* Metodo che elimina dalla tabella Student la riga avente uno specifico id
94      * (non-Javadoc)
95      * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#delete(java.lang.Integer)
96      */
97     public void delete(Integer id) {
98        // Rappresenta la Query SQL che elimina una specifica riga dalla tabella
99        String SQL = "delete from Student where id = ?";
100
101       /* Esegue la query invocando sull'oggetto JdbcTemplate il metodo
102        * update() e passandogli la query SQL e l'id della riga da eliminare
103        */
104       jdbcTemplateObject.update(SQL, id);
105       System.out.println("Deleted Record with ID = " + id);
106       return;
107    }
108
109    /* Metodo che aggiorna il campo age di una riga della tabella Student avente
110     * uno specifico id
111     * @param l'id della riga da aggiornare
112     * @param il valore del campo age da aggiornare
113     * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#update(java.lang.Integer, java.lang.Integer)
114     */
115    public void update(Integer id, Integer age) {
```

```
116         // Rappresenta la query:
117         String SQL = "update Student set age = ? where id = ?";
118
119         /* Esegue la query invocando sull'oggetto JdbcTemplate il metodo
120          * update() e passandogli la query SQL, il nuovo valore del campo age
121          * e l'id della specifica riga da aggiornare
122          */
123         jdbcTemplateObject.update(SQL, age, id);
124         System.out.println("Updated Record with ID = " + id);
125         return;
126     }
127
128 }
```

Andiamo ad analizzare brevemente questa classe per capirne il significato.

La classe **StudentJDBCDao** fornisce un'implementazione dell'interfaccia **StudentDAO** fornendo quindi un'implementazione di tutti i metodi dichiarati in tale interfaccia preposti ad effettuale le operazioni di**CRUD**.
Tale classe è dotata di due proprietà corrispondenti alle classe **DataSource** che rappresenta la nostra sorgente dati ed alla classe **JdbcTemplate** che rappresenta l'oggetto che useremo per interagire con il nostro database.
Il primo metodo che troviamo è il metodo **setDataSource()** che esegue l'injection della sorgente dati e che si occupa di creare un nuovo oggetto di tipo **JdbcTemplate** che useremo per interaggire con i dati della tabbella **Student**.
Gli altri metodi che troviamo sono i metodi preposti per effettuare le operazioni di CRUD ed il loro comportamento è documentato in maniera abbastanza dettagliata tramite commenti.

Per fare maggior chiarezza sulla logica di JDBC in un'applicazione Spring, consideriamo ad esempio il metodo **create()** che si occupa di creare una nuova riga (rappresentante uno studente) all'interno della tabella **Student**.
Il metodo ha il seguente codice:

```
 1  /* Metodo che inserisce un nuovo record, rappresentante uno studente,
 2   * dentro la tabella.
 3   *
 4     @param nome studente
 5     @param età studente
 6   * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#create(java.lang.String, java.lang.Integer)
 7   */
 8  public void create(String name, Integer age) {
 9      // Rappresenta la query:
10      String SQL = "insert into Student (name, age) values (?, ?)";
11
12      // Esegue la query passandogli anche i valori effettivi da inserire:
13      jdbcTemplateObject.update(SQL, name, age);
14
15      System.out.println("Created Record Name = " + name + " Age = " + age);
```

```
16     return;
17 }
```

Tale metodo prende due parametri di input rappresentanti i valori del campo **NAME** ed **AGE** della tabella **Student** (il campo **ID** non è specificato in quanto è stato impostato su **AUTO_INCREMENT** in fase di creazione della tabella).

Nel corpo del metodo troviamo una stringa **SQL** che di fatto rappresenta la nostra **query SQL** che sarà interpretata ed eseguita da **JDBC**, la cosa che potrebbe sembrare strana è che questa stringa ha la forma:

```
1 "insert into Student (name, age) values (?, ?)"
```

in cui sono presenti i due **placeholder** indicati dal simbolo **?** (vedremo a breve a cosa servono)

A questo punto, per esseguire la query, viene invocato il metodo **update()** sul nostro oggetto istanza della classe **JdbcTemplate** passandogli la **stringa SQL** rappresentante la nostra query ed i valori dei**due parametri di input name** ed **age** che rappresentano rispettivamente i campi NAME ed AGE della nostra nuova riga.

A questo punto possiamo vedere cosa succede. Semplicemente tali valori vengono sostituiti ai due placeholder rappresentati dal simobolo **?** e **JDBC** eseguirà la seguente query:

Consideriamo il caso in cui questi sono i valori dei due parametri di input:

name = Pippo

age = 32

insert into Student (name, age) values (Pippo, 32)

Pertanto, all'interno della tabella Student sarà creata una nuova riga avente campo NAME = Pippo e campo AGE = 32

Il meccanismo che fà uso dei placeholder viene usata principalmente per evitare di dover prencerci carico di eseguire manualmente l'escaping di ogni valore che andiamo ad inserire nel database per evitare problemi relativi a vulnerabilità di sicurezza di tipo SQL Injection.

Come ultima cosa possiamo dire che l'esecuzione di una query in JDBC è fortemente collegata al concetto di esecuzione di un oggetto di tipo **PreparateStatment** che di fatto è un oggetto che contiene delle query da inviare al DBMS precompilate in modo tale che quando il DBMS le riceve può eseguirle direttamente senza doverle ricompilare.

Usando JDBC all'interno di un'applicazione Spring possiamo sia usare oggetti aventi tipo**PreparateStatment** o fare come abbiamo fatto nel precedente esempio, il concetto non cambia visto che sarà il Framework a creare tale tipo di oggetto per noi in base alla stringa contenente la query SQL ed i parametri.

Per avere maggiori informazioni su tale tipo di oggetti potete leggere quì: http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

Come ultima analisi relativa a tale classe possiamo vedere che ci sono alcuni metodi che eseguono operazioni di **CRUD** che **non usano il metodo upate()** per eseguire la query ma, ad esempio, usano il metodo **queryForObject()**

Consideriamo il metodo **getStudent(Integer id)** che restituisce un oggetto **Student** corrispondente alla riga identificata dal suo parametro di input.

Tale metodo esegue la query in questo modo:

```
1 Student student = jdbcTemplateObject.queryForObject(SQL, new Object[] { id }, new StudentMapper());
```

L'oggetto Student relativo a tale specifica riga della tabella viene recuperato attraverso l'uso del metodo**queryForObject()** che in questo caso prende come parametr di input:

1. La stringa **SQL** contenente la nostra query SQL.
2. Una **lista di argomenti usati dalla query** che in questo specifico caso corrispondono al singolo valore Integer id rappresentante l'id della riga.
3. Un nuovo oggetto che implementa l'interfaccia **RowMapper** (nel nostro caso **StudentMapper**) che è usato per **mappare una singola riga della tabella Student su di un singolo oggetto istanza della classe Student**.

I metodi che possiamo usare per interagire con i dati nel DB e per poter eseguire le query sono svariati e potete trovare la loro descrizione nella documentazione della classe **JdbcTemplate** al seguente link: http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

Così come è possibile trovare una descrizione dettagliata dell'interfaccia **RowMapper** quì: http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jdbc/core/RowMapper.html

Andiamo infine a creare, sempre nello stesso unico package, la classe principale **MainApp.java**:

```
 1  package org.andrea.myexample.myJdbcSpringExample;
 2
 3  import java.util.List;
 4
 5  import org.springframework.context.ApplicationContext;
 6  import org.springframework.context.support.ClassPathXmlApplicationContext;
 7
 8  // Classe principale
 9  public class MainApp {
10
11     public static void main(String[] args) {
```

```
12
13      /* Crea il contesto in base alle impostazioni dell'applicazione definite
14       * nel file Beans.xml */
15      ApplicationContext context = new ClassPathXmlApplicationContext(
16              "Beans.xml");
17
18      /* Recupera un bean avente id="textEditor" nel file di configurazione
19       * Beans.xml */
20      StudentJDBCTemplate studentJDBCTemplate = (StudentJDBCTemplate) context
21              .getBean("studentJDBCTemplate");
22
23      // CREAZIONE DEI RECORD NEL DB:
24      System.out.println("------Creazione dei record--------");
25      studentJDBCTemplate.create("Zara", 11);
26      studentJDBCTemplate.create("Nuha", 2);
27      studentJDBCTemplate.create("Ayan", 15);
28
29      // ELENCA RECORD MULTIPLI:
30      System.out.println("------Elencando record multipli--------");
31
32      // Recupera la lista di tutti gli studenti:
33      List<Student> students = studentJDBCTemplate.listStudents();
34
35      /* Per ogni oggetto Student recuperato stampa il valore di tutti
36       * i suoi campi:
37       */
38      for (Student record : students) {
39         System.out.print("ID : " + record.getId());
40         System.out.print(", Name : " + record.getName());
41         System.out.println(", Age : " + record.getAge());
42      }
43
44      // AGGIORNA IL CAMPO AGE DEL RECORD AVENTE ID=2
45      System.out.println("----Aggiornamento del record avente ID = 2 -----");
46      studentJDBCTemplate.update(2, 20);
47
48      System.out.println("----Recupera il record avente ID = 2 -----");
49
50      // Recupera il record avente ID=2
51      Student student = studentJDBCTemplate.getStudent(2);
52
53      // Ne stampa i valori di tutti i campi:
54      System.out.print("ID : " + student.getId());
55      System.out.print(", Name : " + student.getName());
56      System.out.println(", Age : " + student.getAge());
57
58   }
59 }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">
6
7    <!-- Initialization for data source -->
```

```
 8    <bean id="dataSource"
 9      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
11      <property name="url" value="jdbc:mysql://localhost:3306/SpringTestDb"/>
12      <property name="username" value="root"/>
13      <property name="password" value="YOUR_PSWD_DB"/>
14    </bean>
15
16    <!-- Definition for studentJDBCTemplate bean -->
17    <bean id="studentJDBCTemplate"
18      class="org.andrea.myexample.myJdbcSpringExample.StudentJDBCTemplate">
19      <property name="dataSource"  ref="dataSource" />
20    </bean>
21
22  </beans>
```

Come si può vedere in tale file di configurazione provvediamo a settare le proprietà per effettuare le connessioni al database iniettandole nella definizione del bean avente **ID=dataSource** e definiamo il bean avente **ID=”studentJDBCTemplate”** che usiamo per interagire con i dati nella tabella Student del database. Notate come in tale secondo bean iniettiamo il riferimento al primo bean rappresentante la sorgente dati.

Andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output:

——Creazione dei record———
Created Record Name = Zara Age = 11
Created Record Name = Nuha Age = 2
Created Record Name = Ayan Age = 15
——Elencando record multipli———
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 2
ID : 3, Name : Ayan, Age : 15
—-Aggiornamento del record avente ID = 2 ——
Updated Record with ID = 2
—-Recupera il record avente ID = 2 ——
ID : 2, Name : Nuha, Age : 20

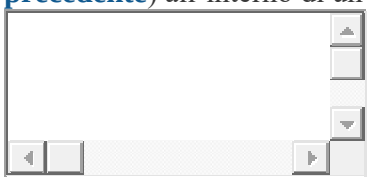
Potete provare l’operazione di delete da soli (non l’ho usata in questo esempio). Avete ora in mano un’applicazione Spring che usa JDBC per accedere ad i dati. Potete provare ad estenderla aggiungendogli funzionalità più sofisticate basate sui requisiti del vostro progetto.

Esistoni anche altri approcci per accedere ai database nei quali potremmo usare le classi**NamedParameterJdbcTemplate** e **SimpleJdbcTemplate** ma tali argomenti esulano dallo scopo di questo articolo.

32 – Stored Procedure SQL in Spring

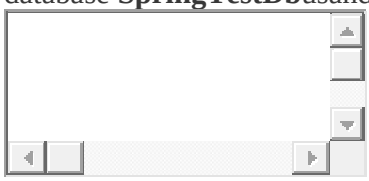
Posted on **15 aprile 2013** Views: 847

La classe **SimpleJdbcCall** può essere usata per chiamare una **Stored Procedure** dotata di parametri di input e parametri di output. Possiamo usare questo approccio quando lavoriamo con un **RDBMS**(**Datbase Relazionale**) come ad esempio: MySQL, Apache Derby, DB2, Microsoft SQL Server, Oracle e Sybase.
Per capire tale approccio consideriamo la nostra tabella **Student** che abbiamo creato (vedere **articolo precedente**) all'interno di un database **SpringTestDb** usando i seguenti comandi DDL:



```
1 CREATE TABLE Student(
2    ID   INT NOT NULL AUTO_INCREMENT,
3    NAME VARCHAR(20) NOT NULL,
4    AGE  INT NOT NULL,
5    PRIMARY KEY (ID)
6 );
```

Consideriamo ora la seguente **Stored Procedure MySql** che si occupa di **prendere l'Id di uno studente (come parametro di input) e di ritornare nei parametri di output i valori relativi al nome ed all'età di tale studente**.
Andiamo quindi a creare questa **Stored Procedure** all'interno del nostro database **SpringTestDb**usando il prompt di comandi di **MySQL**:



```
1  DELIMITER $$
2
3  DROP PROCEDURE IF EXISTS `SpringTestDb`.`getRecord` $$
4  CREATE PROCEDURE `SpringTestDb`.`getRecord` (
5  IN in_id INTEGER,
6  OUT out_name VARCHAR(20),
7  OUT out_age  INTEGER)
8  BEGIN
9    SELECT name, age
10   INTO out_name, out_age
```

```
11    FROM Student where id = in_id;
12 END $$
13
14 DELIMITER ;
```

Creiamo il progetto in STS\Eclipse:

Come al solito lavoriamo usando **STS\Eclipse**, seguiamo i seguenti passi per creare il nostro progetto:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven usando STS\Eclipse.

All'interno di STS\Eclipse aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

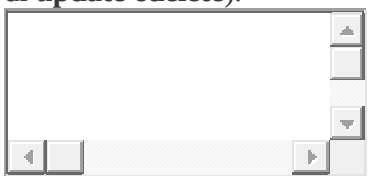
Group **Id:** org.andrea.myexample
Artifact **Id:** myStoredProcedureSpringExample
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file pom.xml e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support, spring-jdbc**).
Dovremmo inoltre aggiungere la seguente dipendenza relativa al driver MySql necessario per effettuare la connessione: **mysql-connector-java**
Qualora non riusciste a scaricare automaticamente le dipendenze attraverso il wizard, questo è il codice del file **pom.xml** da inserire direttamente nel file tramite il tab **pom.xml** della view relativa a tale file:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myStoredProcedureSpringExample</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>myStoredProcedureSpringExample</name>
<url>http://maven.apache.org</url>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>3.2.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>3.2.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.2.1.RELEASE</version>
    </dependency>
```

```
39      <dependency>
40          <groupId>org.springframework</groupId>
41          <artifactId>spring-context-support</artifactId>
42          <version>3.2.1.RELEASE</version>
43      </dependency>
44
45      <dependency>
46          <groupId>org.springframework</groupId>
47          <artifactId>spring-jdbc</artifactId>
48          <version>3.2.1.RELEASE</version>
49      </dependency>
50
51      <dependency>
52          <groupId>mysql</groupId>
53          <artifactId>mysql-connector-java</artifactId>
54          <version>5.1.23</version>
55      </dependency>
56
57    </dependencies>
58 </project>
```

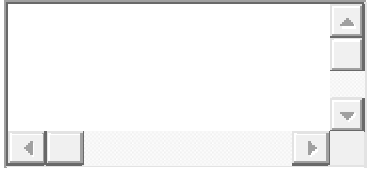
All'interno dell'unico package presente nel nostro progetto (**org.andrea.myexample.myStoredProcedureSpringExample**) creiamo l'interfaccia **StudentDAO**che rappresenta le operazioni di **CRUD** definite per la classe **Student.java** (in questo specifico caso, per semplicità, definiamo solo le operazioni di **create** e di **read** tralasciando le operazioni di **update** e**delete**).



```
1  package org.andrea.myexample.myStoredProcedureSpringExample;
2
3  import java.util.List;
4  import javax.sql.DataSource;
```

```
5
6  /** Interfaccia che rappresenta il nostro DAO per la tabella Student.
7   * Tale interfaccia definisce tutte le operazioni di CRUD per tale tabella
8   */
9  public interface StudentDAO {
10     /**
11      * Metodo usato per inizializzare le risorse del database (la connessione):
12      */
13     public void setDataSource(DataSource ds);
14
15     /**
16      * Metodo usato per creare un record nella tabella Student:
17      */
18     public void create(String name, Integer age);
19
20     /**
21      * Metodo usato per elencare un record, corrispondente ad un certo id
22      * passato, appartenente alla tabella Student:
23      */
24     public Student getStudent(Integer id);
25
26     /**
27      * Metodo che viene usato per elencare tutti i recordo appartenenti alla
28      * tabella Student:
29      */
30     public List<Student> listStudents();
31 }
```

Sempre nello stesso package andiamo ora a creare la seguente classe Java che rappresenta un generico studente (quindi un singolo record della nostra tabella): **Student.java**:



```
1  package org.andrea.myexample.myStoredProcedureSpringExample;
2
3  // Classe che rappresenta un generico studente:
4  public class Student {
5
6      // Proprietà:
7      private Integer age;
8      private String name;
9      private Integer id;
10
11     // Metodi Getter & Setter:
12     public void setAge(Integer age) {
13         this.age = age;
14     }
15
16     public Integer getAge() {
17         return age;
18     }
19
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getName() {
25         return name;
26     }
27
28     public void setId(Integer id) {
29         this.id = id;
30     }
31
32     public Integer getId() {
```

```
33         return id;
34     }
35 }
```

Sempre nello stesso package andiamo ora a creare la seguente classe Java **StudentMapper.java**:

```
1  package org.andrea.myexample.myStoredProcedureSpringExample;
2
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import org.springframework.jdbc.core.RowMapper;
6
7  /* Classe che implementa l'interfaccia RowMapper. Si tratta di un'interfaccia
8   * usata da JdbcTemplate per mappare le righe di un ResultSet (oggetto che
9   * contiene l'insieme delle righe restituite da una query SQL) riga per riga.
10  * Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto
11  * risultante senza doversi preoccupare della gestione delle eccezioni poichè
12  * le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.
13  */
14
15 public class StudentMapper implements RowMapper<Student> {
16
17     /* Implementazione del metodo dell'interfaccia RowMapper che mappa una
18      * specifica riga della tabella su di un oggetto Student
19      *
20      * @param Un oggetto ResultSet contenente l'insieme di tutte le righe
21      *         restituite dalla query
22      *
23      * @param L'indice che indentifica una specifica riga
24      *
25      * @return Un nuovo oggetto Student rappresentante la riga selezionata
```

```
26     *         all'interno dell'oggetto ResultSet
27     *
28     * @see org.springframework.jdbc.core.RowMapper#mapRow(java.sql.ResultSet, int)
29     */
30    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
31        Student student = new Student();
32        student.setId(rs.getInt("id"));
33        student.setName(rs.getString("name"));
34        student.setAge(rs.getInt("age"));
35        return student;
36    }
37 }
```

Tale Classe che implementa l'interfaccia **RowMapper** che è un'interfaccia messa a disposizione da Spring usata da **JdbcTemplate** per **mappare le righe di un ResultSet (l'oggetto che contiene l'insieme delle righe restituite da una query SQL) riga per riga**.
Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto risultante senza doversi preoccupare della gestione delle eccezioni poich le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.

Gli oggetti **RowMapper** sono tipicamente **stateless** e quindi **riutilizzabili**.
Vediamo come la classe **StudentMapper** implementa il metodo **mapRow(ResultSet rs, int rowNum)**che prende in input l'oggetto ResultSet contenente l'insieme delle righe della tabella restituite dalla query eseguita ed un parametro intero che rappresenta l'indice di una specifica riga. Per tale specifica riga viene creato e restituito un **oggetto Student** che contiene le informazioni all'interno di tale riga nell'oggetto **RowMapper**.
Maggiori informazioni sul **ResultSet** quì: http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html
Maggiori informazioni su **RowMapper** quì: http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jdbc/core/RowMapper.html
Di seguito è presentata la classe **StudentJDBCTemplate** che implementa l'interfaccia **StudentDAO**andando ad implementare i metodi di **CRUD** che avevamo definito in tale interfaccia:

```
1  package org.andrea.myexample.myStoredProcedureSpringExample;
2
3  import java.util.List;
4  import java.util.Map;
5
6  import javax.sql.DataSource;
7  import org.springframework.jdbc.core.JdbcTemplate;
8  import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
9  import org.springframework.jdbc.core.namedparam.SqlParameterSource;
10 import org.springframework.jdbc.core.simple.SimpleJdbcCall;
11
12 /** Classe che fornisce l'implementazione per il nostro DAO le cui funzionalità
13  *  di CRUD sono state definite tramite l'interfaccia StudentDAO
14  */
15 public class StudentJDBCTemplate implements StudentDAO {
16
17     // Proprietà:
18     private DataSource dataSource;
19     private SimpleJdbcCall jdbcCall;
20     private JdbcTemplate jdbcTemplateObject;
21
22     /** Metodo Setter per l'Injection della dipendenza relativa alla sorgente
23      *  dati.
24      *  Tale metodo inoltre costruisce anche l'oggetto istanza di SimpleJdbcCall
25      *  usato per rappresentare una chiamata ad una Stored Procedure (nel caso
26      *  specifico la procedura avente nome "getRecord" definita nel database.
27      *  Inoltre costruisce l'oggetto istanza di JdbcTemplate per operare sul
28      *  database.
29      *
30      *  @param la sorgente dati
31      *  @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#setDataSource(javax.sql.DataSource)
32      */
```

```
33  public void setDataSource(DataSource dataSource) {
34     this.dataSource = dataSource;    // Inietta la sorgente dati
35     this.jdbcTemplateObject = new JdbcTemplate(dataSource);
36
37     // Riferimento alla progedura "getRecord" definita nel database:
38     this.jdbcCall = new SimpleJdbcCall(dataSource)
39                         .withProcedureName("getRecord");
40  }
41
42  /** Metodo che inserisce un nuovo record, rappresentante uno studente,
43   *  dentro la tabella.
44   *
45     @param nome studente
46     @param età studente
47   *  @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#create(java.lang.String, java.lang.Integer)
48   */
49  public void create(String name, Integer age) {
50
51     // Rappresenta la query SQL:
52     String SQL = "insert into Student (name, age) values (?, ?)";
53
54     // Esegue la query SQL parametrizzata:
55     jdbcTemplateObject.update(SQL, name, age);
56     System.out.println("Created Record Name = " + name + " Age = " + age);
57     return;
58  }
59
60  /* Metodo che recupera un record, rappresentante uno studente, avente uno
61   * specifico id dalla tabella.
62   *
63   * @param L'id dello studente
64   * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#getStudent(java.lang.Integer)
```

```
65      */
66     public Student getStudent(Integer id) {
67
68         /** Crea un nuovo oggetto instanza di MapSqlParameterSource (che è
69          *  implementazione dell'interfaccia SqlParameterSource che definisce
70          *  delle funzionalità comuni ad oggetti in grado di offrire valori
71          *  parametrizzati per parametri di query o stored procedure SQL
72          *  aventi nomi).
73          *
74          *  MapSqlParameterSource mette in una mappa i parametri di input della
75          *  query o della Stored Procedure, in questo caso il solo parametro
76          *  id avente key: "in_id"
77          */
78         SqlParameterSource in = new MapSqlParameterSource().addValue("in_id",
79                 id);
80
81         /** Esegue il metodo execute sull'oggetto SimpleJdbcCall jdbcCall
82          *  (l'oggetto che rappresenta la StoredProcedure che in base all'id
83          *  restituisce i valori del campo name ed age).
84          *
85          *  Tale chiamata ritorna una mappa contenente i parametri di output
86          *  nella forma: <key, oggetto>
87          */
88         Map<String, Object> out = jdbcCall.execute(in);
89
90         Student student = new Student();    // Crea nuovo oggetto Student
91
92         /** Setta i campi con i valori contenuti nella mappa dei parametri
93          *  di output
94          */
95         student.setId(id);
96         student.setName((String) out.get("out_name"));
```

```
97          student.setAge((Integer) out.get("out_age"));
98
99          return student;
100     }
101
102     public List<Student> listStudents() {
103         // Rappresenta la query:
104         String SQL = "select * from Student";
105
106         /** Ottengo la lista degli oggetti Student corrispondenti a tutte le
107          *  righe della tabella invocando il metodo queryForObject sull'oggetto
108          *  JdbcTemplate passandogli i seguenti parametri.
109          *
110          *  @param La query per creare il preparated statement
111          *  @param Un oggetto che implementa RowMapper che viene usato per
112          *         mappare una singola riga della tabella su di un oggetto Java
113          */
114         List<Student> students = jdbcTemplateObject.query(SQL,
115                 new StudentMapper());
116
117         return students;
118     }
119
120 }
```

A differenza del precedente articolo, in questo caso, anzichè usare una normale query **per ottenere i valori dei campi name ed age relativi ad uno studente avente un certo ID abbiamo usato una Stored Procedure**. Il codice che abbiamo scritto per l'esecuzione di tale Stored Procedure prevede la creazione di un oggetto **SqlParameterSource** contenente i **parametri di input** (nel caso specifico abbiamo usato l'implementazione **MapSqlParameterSource** che prevede di mettere i **parametri di input all'interno di una mappa**).

NB: E' importante che il nome fornito per tali parametri di input match con il nome del parametro dichiarato nella definizione della Stored Procedure.

Il metodo execute() prende il parametro di input e ritorna un ogetto Map contenente tutti i parametri di output avente per chiave il nome del parametro di output specificcato nella definizione della Stored Procedure.

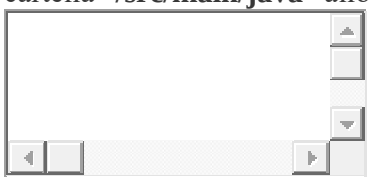
Andiamo infine a creare, sempre nello stesso unico package, la classe principale **MainApp.java**:

```
1  package org.andrea.myexample.myStoredProcedureSpringExample;
2
3  import java.util.List;
4  import org.springframework.context.ApplicationContext;
5  import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7  //Classe principale:
8  public class MainApp {
9
10    public static void main(String[] args) {
11
12       /* Crea il contesto in base alle impostazioni dell'applicazione definite
13        * nel file Beans.xml */
14       ApplicationContext context = new ClassPathXmlApplicationContext(
15                                    "Beans.xml");
16
17       /* Recupera un bean avente id="studentJDBCTemplate" nel file di configurazione
18        * Beans.xml */
19       StudentJDBCTemplate studentJDBCTemplate = (StudentJDBCTemplate) context
20             .                getBean("studentJDBCTemplate");
21
22       // CREAZIONE DEI RECORD NEL DB:
23       System.out.println("------RCreazione dei Record--------");
24       studentJDBCTemplate.create("Zara", 11);
25       studentJDBCTemplate.create("Nuha", 2);
```

```
26        studentJDBCTemplate.create("Ayan", 15);
27
28        // ELENCA RECORD MULTIPLI:
29        System.out.println("------Elencando record multipli--------");
30
31        // Recupera la lista di tutti gli studenti:
32        List<Student> students = studentJDBCTemplate.listStudents();
33
34        /* Per ogni oggetto Student recuperato stampa il valore di tutti
35         * i suoi campi:
36         */
37        for (Student record : students) {
38            System.out.print("ID : " + record.getId());
39            System.out.print(", Name : " + record.getName());
40            System.out.println(", Age : " + record.getAge());
41        }
42
43        // Recupera il record avente ID=2
44        System.out.println("----Recupera il record avente ID = 2 -----");
45
46        Student student = studentJDBCTemplate.getStudent(2);
47        System.out.print("ID : " + student.getId());
48        System.out.print(", Name : " + student.getName());
49        System.out.println(", Age : " + student.getAge());
50
51    }
52 }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">
6
7    <!-- Inizializzazione della sorgente dati -->
8    <bean id="dataSource"
9       class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
11      <property name="url" value="jdbc:mysql://localhost:3306/SpringTestDb"/>
12      <property name="username" value="root"/>
13      <property name="password" value="aprile12"/>
14   </bean>
15
16   <!-- Definitione del bean studentJDBCTemplate  -->
17   <bean id="studentJDBCTemplate"
18      class="org.andrea.myexample.myStoredProcedureSpringExample.StudentJDBCTemplate">
19      <property name="dataSource"  ref="dataSource" />
20   </bean>
21
22 </beans>
```

Come si può vedere in tale file di configurazione provvediamo a settare le proprietà per effettuare le connessioni al database iniettandole nella definizione del bean avente **ID=dataSource** e definiamo il bean avente **ID=”studentJDBCTemplate”** che usiamo per interagire con i dati nella tabella Student del database. Notate come in tale secondo bean iniettiamo il riferimento al primo bean rappresentante la sorgente dati.

Andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output:

——RCreazione dei Record———
Created Record Name = Zara Age = 11
Created Record Name = Nuha Age = 2
Created Record Name = Ayan Age = 15

——Elencando record multipli———
ID : 1, Name : Zara, Age : 11
ID : 2, Name : Nuha, Age : 20
ID : 3, Name : Ayan, Age : 15
ID : 4, Name : Zara, Age : 11
ID : 5, Name : Nuha, Age : 2
ID : 6, Name : Ayan, Age : 15
—-Recupera il record avente ID = 2 ——
feb 20, 2013 5:31:16 PM org.springframework.jdbc.core.JdbcTemplate extractReturnedResults
INFO: Added default SqlReturnUpdateCount parameter named #update-count-1
ID : 2, Name : Nuha, Age : 20

33 – Gestione delle Transazioni in Spring (Introduzione)

Posted on **16 aprile 2013** Views: 3366

Una transazione sul database è una sequenza di azioni che può essere trattata come una singola unità di lavoro. Tali azioni dovrebbero o completare interamente o in caso contrario non avere alcun effetto.

La **gestione delle transazioni** è una parte importante delle applicazioni enterprise orientate ai RDBMS (Reletional Database Managment System) che **assicura l'integrità e la consistenza dei dati**. I concetti relativi alle transazioni possono essere descritti dalle seguenti **quattro proprietà fondamentali** chiamate **ACID**:

1. **Atomicità:** Una transazione dovrebbe essere trattata come una singola unità di operazioni sia che l'intera sequenza di operazioni abbia successo sia che l'intera sequenza di operazioni non lo abbia (ciò significa che se anche una sola operazione della transazione non ha successo allora l'intera transazione non ha successo e di conseguenza non ha alcun effetto sul database).
2. **Consistenza:** Ciò rappresenta la consistenza dell'integrità referenziale del database, chiave primaria univoca per le tabelle, etc.
3. **Isolamento:** Ci possono essere più transazioni che operano su di uno stesso insieme di dati allo stesso tempo, ogni transazione dovrebbe essere isolata dalle altre per prevenire la corruzione di tali dati.
4. **Durabilità:** Una volta che una transazione viene ultimata, il risultato di tale transazione deve essere reso permanente sul database e non può essere cancellato dal database a casusa di un errore di sistema.

Un vero e proprio sistema di database RDBMS garantirà tutte le quattro proprietà per ogni transazione. La versione semplicistica di cosa sia una transazione rilasciata da un database che fà uso di SQL è la seguente:

1. La transazione ha inizio usando il comando **begin transaction**.
2. Realiza le varie operazione di delete, update o insert usando **query SQL**.
3. Se tutte le operazioni hanno successo allora esegue l'operazione di **commit** (salvando il risultato in maniera persistente sul database), altrimenti esegue l'operazione di **rollback** (tornando alla situazione che aveva trovato nel momento precedente all'inizio di tale transazione).

Il framework Spring fornisce un livello astratto al di sopra delle sottostanti API per la gestione delle transazioni. Il supporto alle transazioni di Spring punta a fornire un'alternativa alla gestione delle transazioni mediante l'uso di EJB aggiungendo funzionalità di transazione ai POJO. Spring fornisce sia un supporto programmatico sia un supporto dichiarativo per la gestione delle transazioni. Ricordiamo che gli EJB richiedono un application server mentre la gestione delle transazioni in Spring può essere implementata senza dover ricorrere ad un applicazion server.

Transazioni Locali Vs. Transazioni Globali:

Le **transazioni locali** sono specifiche di una singola risorsa transazionale come ad esempio una connessione JDBC mentre le **transazioni globali** possono interessare risorse transazionali multimple come nel caso di sistemi distrubuiti.

La gestione delle transazioni locali può essere può essere centralizzata in un singolo ambiente di calcolo in cui le compnenti applicative e le risorse sono localizzate in una singola locazione e la gestione delle transazione coinvolge solo un data manager locale su di una singola macchina. Le transazioni locali sono più semploci da implmenentare.

Le gestione delle transazioni globali è richiestra in ambienti di calcolo distribuiti in cui tutte le risorse sono distribuite tra più sistemi. In questi casi la gestione delle transazioni ha bisogno sia di un livello locale che di un livello globale. Una trandazione distribuita (o transazione globale) viene eseguita attraverso più sistemi, e la sua esecuzione richiede la coordinazione tra un sistema globale di gestione delle transazioni e e di tutti i data manager locali coinvolti nel sistema distribuito.

Supporto Programmatico Vs. Supporto Dichiarativo:

Spring fornisce due tipi di gestione delle transazioni:

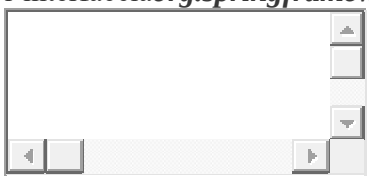
1. **Supporto Programmatico alla gestione delle transazioni:** Significa che dobbiamo gestire le transazioni con l'aiuto della programmazione. Fornisce estrema flessibilità ma di contro è più difficile da mantenere.
2. **Supporto Dichiarativo alla gestione delle transazioni:** Significa che la gestione delle transazioni viene separata dal codice di business. Possiamo solo urare delle annotation oppure una configurazione basata su XML per gestire le transazioni.

E' preferibile usare la gestione dichiarativa delle transazioni anzichè la gestione programmatica anche se risulta essere meno flessibile di quest'ultima.

Inoltre la gestione delle transazioni dichiarativa può essere modularizzata mediante l'approccio AOP. Spring fornisce il supporto dichiarativo alle transazioni mediante il modulo Spring AOP.

Spring Transaction Abstractions:

La chiave dell'astrazione alle transazioni in Spring è definita tramite l'interfaccia***org.springframework.transaction.PlatformTransactionManager*** che è la seguente:



```
1 public interface PlatformTransactionManager {
2    TransactionStatus getTransaction(TransactionDefinition definition);
3    throws TransactionException;
4    void commit(TransactionStatus status) throws TransactionException;
5    void rollback(TransactionStatus status) throws TransactionException;
6 }
```

Analisi dei metodi definiti:

1. **TransactionStatus getTransaction(TransactionDefinition definition):** Questo metodo ritorna la transazione correntemente attiva o ne crea una nuova secondo il comportamento di propagazione specificato.
2. **void commit(TrasactionStatus status):** Questo metodo esegue l'operazione di commit della transazione proposta, per quanto riguarda il suo status.
3. **void rollback(TransactionStatus status):** Questo metodo esegue l'operazione di rollback della transazione data.

L'interfaccia ***TransactionDefinition*** è il nucleo fondamentale del supporto alle transazioni in Spring ed è definita nel seguente modo:

```
1 public interface TransactionDefinition {
2    int getPropagationBehavior();
3    int getIsolationLevel();
4    String getName();
5    int getTimeout();
6    boolean isReadOnly();
7 }
```

Analisi dei metodi definiti:

1. **int getPropagationBehavior():** Questo metodo ritorna il comportamento di propagazione. Spring offre tutte le opzioni di propagazione delle transazioni da EJB CMT.
2. **int getIsolationLevel():** Questo metodo ritorna il grado in cui la transazione corrente è isolata dal lavoro di altre transazioni.

3. **int getName():** Questo metodo ritorna il nome della transazione corrente.
4. **int getTimeout():** Questo metodo ritorna il tempo in secondi in cui la transazione deve essere completata.
5. **boolean isReadOnly():** Questo metodo restituisce se la transazione è di sola lettura.

Di seguito sono riportati i possibili valori relativi al livello di isolamento:

1. **TransactionDefinition.ISOLATION_DEFAULT:** E' il valore di default del livello di isolamento.
2. **TransactionDefinition.ISOLATION_READ_COMMITTED:** Indica che vengono impedite le letture sporche ma che le letture non ripetibili e letture fantasma possono verificarsi.
3. **TransactionDefinition.ISOLATION_READ_UNCOMMITTED:** India che le letture sporche, le letture non ripetibili e le letture fantasma possono verificarsi.
4. **TransactionDefinition.ISOLATION_REPEATABLE_READ:** Indica che le letture sporche e le lettura non ripetibili sono impedite ma che le letture fantasma possono verificarsi.
5. **TransactionDefinition.ISOLATION_SERIALIZABLE:** Indica che le letture sporche, le letture non ripetibili e le letture fantasma non possono verificarsi.

Di seguito sono riportati i possibili valori relativi ai possibili tipi di propagazione:

1. **TransactionDefinition.PROPAGATION_MANDATORY:** Supporta una transazione in corso; solleva un'eccezione se non esiste transazione corrente.
2. **TransactionDefinition.PROPAGATION_NESTED:** Eseguita all'interndo di una transazione annidata se la transazione corrente esiste.
3. **TransactionDefinition.PROPAGATION_NEVER:** Non supporta la transazione corrente; solleva un'eccezione se la transazione corrente esiste.
4. **TransactionDefinition.PROPAGATION_NOT_SUPPORTED:** Non supporta una transazione corrente, anzi sempre eseguire non transazionale.
5. **TransactionDefinition.PROPAGATION_REQUIRED:** Supporta la transazione corrente; crea una nuova transazione se questa non esiste.
6. **TransactionDefinition.PROPAGATION_REQUIRES_NEW:** Crea una nuova transazione, sospende la transazione corrente se questa esiste.
7. **TransactionDefinition.PROPAGATION_SUPPORTS:** Supporta la transazione corrente; La esegue in maniera non transazionale se non esiste.
8. **TransactionDefinition.TIMEOUT_DEFAULT:** Utilizza il timeout di defaul del sistema di transazione sottostante o non lo usa se il timeout non è supportato.

L'intarfaccia ***TransactionStatus*** fornisce un modo semplice per il codice transazionale che controlla l'esecuzione delle transazione e dello status delle query transazionali:

```
1 public interface TransactionStatus extends SavepointManager {
2    boolean isNewTransaction();
3    boolean hasSavepoint();
```

```
4    void setRollbackOnly();
5    boolean isRollbackOnly();
6    boolean isCompleted();
7 }
```

Analisi dei metodi definiti:

1. **boolean hasSavepoint():** Questo metodo ritorna se questa transazione internamente esegue un punto di salvataggio, cioè che è stata creato come una transazione nidificata basata su un punto di salvataggio.
2. **boolean isCompleted():** Questo metodo ritorna se questa transazione è stata completata, cioè se è stata già eseguita l'operatazione di commit o l'operazione di rollback.
3. **boolean isNewTransaction():** Questo metodo ritorna true nel caso che la transazione corrente è nuova.
4. **boolean isRollbackOnly():** Questo metodo ritorna true se la transazione è stata contrassegnata come rollback-only.
5. **void setRollbackOnly():** Questo metodo setta la transazione a rollback only.

34 – Gestione delle Transazioni in maniera programmatica in Spring

Posted on **16 aprile 2013** Views: 967

L'approccio alla gestione programmatica delle transazioni ci consente di gestire le transazioni mediante l'aiuto della programmazione all'interno del nostro codice sorgente. Tale approccio risulta essere estremamente flessibile ma più difficile da mantenere rispetto alla gestione dichiarativa.

Prima di iniziare è importante avere almeno due tabelle all'interno del nostro database sulle quali è possibile eseguire diverse operazioni di CRUD che sfruttano l'aiuto delle transazioni.

Per prima cosa andiamo a creare la tabella **Student** all'interno di un database che io ho chiamato**SpringTestDb**.

Creiamo il Database:

Per prima cosa accediamo al nostro database nel modo che riteniamo opportuno (attraverso riga di comando o usando qualche programma per interfacciarsi con il DB). Ad esempio considerando di usare Linux e di accedere al DB MySql dalla shell potremmo fare:

mysql -u root -p (ed inserire la password)

Potremmo poi creare un database dedicato a tale esempio tramite il comando MySql: **create database SpringTestDb;**

Accediamo ora al database appena creato lanciando il seguente comando MySql: **use SpringTestDb;**

Infine creiamo la tabella lanciando il seguente comando nella shell:

```
1 CREATE TABLE Student(
2    ID   INT NOT NULL AUTO_INCREMENT,
3    NAME VARCHAR(20) NOT NULL,
4    AGE  INT NOT NULL,
5    PRIMARY KEY (ID)
6 );
```

La seconda tabella da creare all'intdeno del database **SpringTestDb** è la tabella **Marks** che conterrà i voti degli studenti basandosi sugli anni. In questa tabella troviamo il campo **SID** che rappresenta la**foreign key** che lega i record della tabella **Marks** ad un record della tabella **Student**.

```
1 CREATE TABLE Marks(
2    SID INT NOT NULL,
3    MARKS  INT NOT NULL,
4    YEAR   INT NOT NULL
5 );
```

Andiamo ad usare l'interfaccia ***PlatformTransactionManager*** (presentata nel precedente articolo) per realizzare l'approccio programmatico che implementa le transazioni. Per iniziare una nuova transazione abbiamo bisogno di un'istanza di un'implementazione concreta dell'interfaccia ***TransactionDefinition***con gli appropriati attributi di transazione. In questo esempio creeremo semplicemente un'istanza di***DefaultTransactionDefinition*** *in modo da usare gli attributi di transazione standard.*

Una volta che un'istanza di ***TransactionDefinition*** è stata creata, possiamo eseguire la nostra transazione invocando il metodo ***getTransaction()*** (sull'istanza che implementa***PlatformTransactionManager***) che ritorna un'istanza di ***TransactionStatus***.

L'oggetto ***TransactionStatus*** contribuisce al monitoraggio del tracciamento dello status corrente della transazione ed alla fine, se tutto è andato a buon fine, possiamo usare il metodo ***commit()*** di***PlatformTransactionManager*** per eseguire la commit della transazione e renderla persistente sul database, altrimenti possiamo usare il metodo ***rollback()*** per ripristina la situazione antecedente all'inizio della transazione.

Andiamo ora ad implementare un'applicazione Spring che usa JDBC per l'accesso ai dati e che implementa alcune semplici operazioni di CRUD sulle tabelle Student e Marks. Come al solito lavoreremo usando STS\Eclipse come ambiente di sviluppo.

Creiamo il progetto in STS\Eclipse:
Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven usando STS\Eclipse.

All'interno di STS\Eclipse aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample
Artifact **Id:** myProgrammaticTransactionSpring
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file **pom.xml** e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support, spring-jdbc**).
Dovremmo inoltre aggiungere la seguente dipendenza relativa al driver MySql necessario per effettuare la connessione: **mysql-connector-java**
Qualora non riusciste a scaricare automaticamente le dipendenze attraverso il wizard, questo è il codice del file **pom.xml** da inserire direttamente nel file tramite il tab **pom.xml** della view relativa a tale file:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3    <modelVersion>4.0.0</modelVersion>
4
5    <groupId>org.andrea.myexample</groupId>
6    <artifactId>myProgrammaticTransactionSpring</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <packaging>jar</packaging>
```

```
9
10  <name>myProgrammaticTransactionSpring</name>
11  <url>http://maven.apache.org</url>
12
13  <properties>
14      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15  </properties>
16
17  <dependencies>
18      <dependency>
19          <groupId>junit</groupId>
20          <artifactId>junit</artifactId>
21          <version>3.8.1</version>
22          <scope>test</scope>
23      </dependency>
24      <dependency>
25          <groupId>org.springframework</groupId>
26          <artifactId>spring-core</artifactId>
27          <version>3.2.1.RELEASE</version>
28      </dependency>
29      <dependency>
30          <groupId>org.springframework</groupId>
31          <artifactId>spring-beans</artifactId>
32          <version>3.2.1.RELEASE</version>
33      </dependency>
34      <dependency>
35          <groupId>org.springframework</groupId>
36          <artifactId>spring-context</artifactId>
37          <version>3.2.1.RELEASE</version>
38      </dependency>
39      <dependency>
40          <groupId>org.springframework</groupId>
```

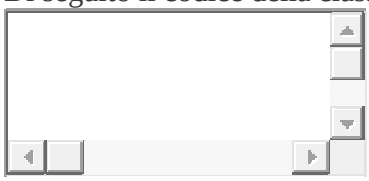
```
41            <artifactId>spring-context-support</artifactId>
42            <version>3.2.1.RELEASE</version>
43        </dependency>
44
45        <dependency>
46            <groupId>org.springframework</groupId>
47            <artifactId>spring-jdbc</artifactId>
48            <version>3.2.1.RELEASE</version>
49        </dependency>
50
51        <dependency>
52            <groupId>mysql</groupId>
53            <artifactId>mysql-connector-java</artifactId>
54            <version>5.1.23</version>
55        </dependency>
56    </dependencies>
57 </project>
```

Per prima cosa andiamo a creare dell'unico package presente nel nostro progetto (**org.andrea.myexample.myJdbcSpringExample**) creare la classe **StudentDAO** che elenca la lista di tutti i metodi di **CRUD** che saranno messi a disposizione dal nostro DAO.

```
1  package org.andrea.myexample.myProgrammaticTransactionSpring;
2
3  import java.util.List;
4
5  import javax.sql.DataSource;
6
7  /** Interfaccia che definisce i metodi che implementano le operazioni di CRUD
8   *  che vogliamo implementare nel nostro DAO:
9   */
```

```
10 public interface StudentDAO {
11
12     /**
13      * Questo metodo viene usato per inizializzare le risorse del database cioè
14      * la connessione al database:
15      */
16     public void setDataSource(DataSource ds);
17
18     /**
19      * Questo metodo serve a creare un record nella tabella Student e nella
20      * tabella Marks:
21      */
22     public void create(String name, Integer age, Integer marks, Integer year);
23
24     /**
25      * Questo metodo serve ad elencare tutti i record all'interno della tabella
26      * Studend e della tabella Marks
27      */
28     public List<StudentMarks> listStudents();
29 }
```

Di seguito il codice della classe **StudentMarks** che rappresenta la nostra entity:

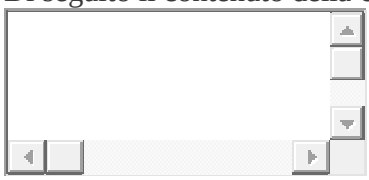


```
1  package org.andrea.myexample.myProgrammaticTransactionSpring;
2
3  // Rappresenta l'entity:
4  public class StudentMarks {
5
6      // Proprietà:
7      private Integer age;
8      private String name;
```

```
 9  private Integer id;
10  private Integer marks;
11  private Integer year;
12  private Integer sid;
13
14  // Metodi Getter & Setter:
15  public void setAge(Integer age) {
16      this.age = age;
17  }
18
19  public Integer getAge() {
20      return age;
21  }
22
23  public void setName(String name) {
24      this.name = name;
25  }
26
27  public String getName() {
28      return name;
29  }
30
31  public void setId(Integer id) {
32      this.id = id;
33  }
34
35  public Integer getId() {
36      return id;
37  }
38
39  public void setMarks(Integer marks) {
40      this.marks = marks;
```

```
41     }
42
43     public Integer getMarks() {
44         return marks;
45     }
46
47     public void setYear(Integer year) {
48         this.year = year;
49     }
50
51     public Integer getYear() {
52         return year;
53     }
54
55     public void setSid(Integer sid) {
56         this.sid = sid;
57     }
58
59     public Integer getSid() {
60         return sid;
61     }
62 }
```

Di seguito il contenuto della classe **StudentMarksMapper:**



```
1  package org.andrea.myexample.myProgrammaticTransactionSpring;
2
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import org.springframework.jdbc.core.RowMapper;
6
```

```
7  /** Classe che implementa l'interfaccia RowMapper. Si tratta di un'interfaccia
8   *  usata da JdbcTemplate per mappare le righe di un ResultSet (oggetto che
9   *  contiene l'insieme delle righe restituite da una query SQL) riga per riga.
10  *  Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto
11  *  risultante senza doversi preoccupare della gestione delle eccezioni poichè
12  *  le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.
13  */
14 public class StudentMarksMapper implements RowMapper<StudentMarks> {
15
16    /** Implementazione del metodo dell'interfaccia RowMapper che mappa una
17     *  specifica riga della tabella su di un oggetto Student
18     *
19     *  @param Un oggetto ResultSet contenente l'insieme di tutte le righe
20     *         restituite dalla query
21     *
22     *  @param L'indice che indentifica una specifica riga
23     *
24     *  @return Un nuovo oggetto Student rappresentante la riga selezionata
25     *          all'interno dell'oggetto ResultSet
26     *
27     *  @see org.springframework.jdbc.core.RowMapper#mapRow(java.sql.ResultSet, int)
28     */
29    public StudentMarks mapRow(ResultSet rs, int rowNum) throws SQLException {
30
31       StudentMarks studentMarks = new StudentMarks();
32
33       studentMarks.setId(rs.getInt("id"));
34       studentMarks.setName(rs.getString("name"));
35       studentMarks.setAge(rs.getInt("age"));
36       studentMarks.setSid(rs.getInt("sid"));
37       studentMarks.setMarks(rs.getInt("marks"));
38       studentMarks.setYear(rs.getInt("year"));
```

```
39
40        return studentMarks;
41     }
42 }
```

Di seguito ecco l'implementazione della classe **StudentJDBCTemplate** che implementa i metodi di**CRUD** definiti nell'interfaccia **StudentDAO:**

```
1  package org.andrea.myexample.myProgrammaticTransactionSpring;
2
3  import java.util.List;
4
5  import javax.sql.DataSource;
6  import org.springframework.dao.DataAccessException;
7  import org.springframework.jdbc.core.JdbcTemplate;
8  import org.springframework.transaction.PlatformTransactionManager;
9  import org.springframework.transaction.TransactionDefinition;
10 import org.springframework.transaction.TransactionStatus;
11 import org.springframework.transaction.support.DefaultTransactionDefinition;
12
13 /**
14  * Classe che fornisce l'implementazione per il nostro DAO le cui funzionalità
15  * di CRUD sono state definite tramite l'interfaccia StudentDAO
16  */
17 public class StudentJDBCTemplate implements StudentDAO {
18
19     // Proprietà:
20     private DataSource dataSource; // Utility per l'accesso alla sorgente dati
21     // Oggetto usato per interagire con i dati sul database:
22     private JdbcTemplate jdbcTemplateObject;
23     // Interfaccia centrale per la gestione delle transazioni in Spring:
```

```
24  private PlatformTransactionManager transactionManager;
25
26  /**
27   * Metodo Setter per l'Injection della dipendenza relativa alla sorgente
28   * dati. Tale metodo inoltre costruisce anche l'oggetto istanza di
29   * JdbcTemplate usato per interagire con i dati nel database.
30   *
31   * @param la
32   *            sorgente dati
33   * @see org.andrea.myexample.myJdbcSpringExample.StudentDAO#setDataSource(javax.sql.DataSource)
34   */
35  public void setDataSource(DataSource dataSource) {
36      this.dataSource = dataSource;
37      this.jdbcTemplateObject = new JdbcTemplate(dataSource);
38  }
39
40  /**
41   * Metodo Setter per l'Injection della dipendenza relativa all'interfaccia
42   * PlatformTransactionManager per la gestione delle transazioni
43   *
44   * @param transactionManager
45   */
46  public void setTransactionManager(
47          PlatformTransactionManager transactionManager) {
48      this.transactionManager = transactionManager;
49  }
50
51  /**
52   * Metodo relativo all'operazione di CREATE che inserisce un nuovo record
53   * all'interno della tabella Student ed un correlato nuovo record nella
54   * tabella Marks.
55   */
```

```
56  public void create(String name, Integer age, Integer marks, Integer year) {
57
58      /**
59       * Rappresenta la definizione della transazione creata con i valori di
60       * default: (PROPAGATION_REQUIRED, ISOLATION_DEFAULT, TIMEOUT_DEFAULT,
61       * readOnly=false)
62       */
63      TransactionDefinition def = new DefaultTransactionDefinition();
64
65      // Rappresenta lo status della transazione appena creata:
66      TransactionStatus status = transactionManager.getTransaction(def);
67
68      try {
69          // Query che inserisce nome ed età nella tabella Student:
70          String SQL1 = "insert into Student (name, age) values (?, ?)";
71          // Esegue la query passandogli anche i valori effettivi da inserire:
72          jdbcTemplateObject.update(SQL1, name, age);
73
74          // Seleziona l'ultimo studente inserito nella tabella Marks:
75          String SQL2 = "select max(id) from Student";
76          // Esegue la query e mette il risultato (l'ID) in sid:
77          int sid = jdbcTemplateObject.queryForInt(SQL2);
78
79          /**
80           * Query che inserisce un nuovo record nella tabella Marks. Il
81           * record rappresenta il voto per l'ultimo studente inserito nella
82           * tabella Student:
83           */
84          String SQL3 = "insert into Marks(sid, marks, year) "
85                  + "values (?, ?, ?)";
86          // Esegue la query passandogli anche i valori effettivi da inserire:
87          jdbcTemplateObject.update(SQL3, sid, marks, year);
```

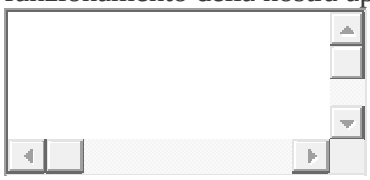
```
88
89              System.out.println("Created Name = " + name + ", Age = " + age);
90              transactionManager.commit(status);
91          } catch (DataAccessException e) {
92              System.out.println("Error in creating record, rolling back");
93              transactionManager.rollback(status);
94              throw e;
95          }
96          return;
97      }
98
99      /**
100      * Metodo relativo all'operazione di READ che recupera la lista degli
101      * studenti e dei relativi voti
102      *
103      * @return La lista di oggetti che rappresentano uno studente ed i suoi voti
104      *         correlati
105      */
106     public List<StudentMarks> listStudents() {
107
108         /**
109          * Query che estrae la lista di tutti i record nella tabella Student e
110          * che per ogni record in tale tabella estrae i relativi record
111          * correlati nella tabella Marks
112          */
113         String SQL = "select * from Student, Marks where Student.id=Marks.sid";
114
115         /**
116          * Ottengo la lista degli oggetti StudentMarks, corrispondenti ognuno ad
117          * un record della tabella Student con i correlati vori rappresentati
118          * dai record della tabella Marks, invocando il metodo query
119          * sull'oggetto JdbcTemplate passandogli i seguenti parametri.
```

```
120         *
121         * @param La query per creare il preparated statement
122         * @param Un oggetto che implementa RowMapper che viene usato per
123         *        mappare una singola riga della tabella su di un oggetto Java
124         */
125        List<StudentMarks> studentMarks = jdbcTemplateObject.query(SQL,
126                                                new StudentMarksMapper());
127        return studentMarks;
128     }
129 }
```

Come si può vedere tale classe implementa il metodo **create()** che si occupa di inserire un nuovo record nella tabella **Student** ed il corrispondente nuovo record nella tabella **Marks** mantenendo l'**integrità referenziale tra tali record** ed il metodo **listStudents()** che ritorna una lista di oggetti**StudentMarks** che rappresentano un oggetto aggragato contenente i dati di un determinato studente (presi dal relativo record nella tabella **Student**) e dei suoi voti (preso dal record correlato nella tabella**Marks**).
Notiamo inoltre che in tale classe, oltre ad iniettare l'oggetto **DataSource** contenente le configurazioni per l'accesso di JDBC al database, iniettiamo anche un oggetto **PlatformTransactionManager** che come abbiamo visto rappresenta l'**interfaccia centrale per la gestione delle transazioni in Spring**. Come si può notare all'interno del file di configurazione **Beans.xml** tale oggetto sarà costruito usando uno dei suoi tipi concreti di default, in questo caso come tipo concreto di default viene usato**DataSourceTransactionManager** che rappresenta un'**implementazione adatta nel caso si usi una singola sorgente dati JDBC**.
Maggiori informazioni sull'interfaccia **PlatformTransactionManager**:
http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html

Maggiori informazioni sulla sua implementazione **DataSourceTransactionManager**:
http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jdbc/datasource/DataSourceTransactionManager.html

Di seguito è presentato il codince della classe principale **MainApp** che usiamo per testare il corretto funzionamento della nostra applicazione:



```
1  package org.andrea.myexample.myProgrammaticTransactionSpring;
2
3  import java.util.List;
4  import org.springframework.context.ApplicationContext;
5  import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7  // Classe principale:
8  public class MainApp {
9
10    public static void main(String[] args) {
11
12       /**
13        * Crea il contesto in base alle impostazioni dell'applicazione definite
14        * nel file Beans.xml
15        */
16       ApplicationContext context = new ClassPathXmlApplicationContext(
17             "Beans.xml");
18
19       /**
20        * Recupera un bean avente id="studentJDBCTemplate" nel file di
21        * configurazione Beans.xml
22        */
23       StudentJDBCTemplate studentJDBCTemplate = (StudentJDBCTemplate) context
24             .getBean("studentJDBCTemplate");
25
26       System.out.println("------Creazione dei record--------");
27       // Creo i record nelle tabelle Studend e Marks:
28       studentJDBCTemplate.create("Zara", 11, 99, 2010);
29       studentJDBCTemplate.create("Nuha", 20, 97, 2010);
30       studentJDBCTemplate.create("Ayan", 25, 100, 2011);
31
32       System.out.println("------Elenca tutti i record--------");
```

```
33        // Recupera la lista degli studenti con i voti ad essi associati:
34        List<StudentMarks> studentMarks = studentJDBCTemplate.listStudents();
35
36        for (StudentMarks record : studentMarks) {        // e li stampa
37            System.out.print("ID : " + record.getId());
38            System.out.print(", Name : " + record.getName());
39            System.out.print(", Marks : " + record.getMarks());
40            System.out.print(", Year : " + record.getYear());
41            System.out.println(", Age : " + record.getAge());
42        }
43    }
44 }
```

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">
6
7     <!-- Initizializzazione della sorgente dati: -->
8     <bean id="dataSource"
9        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
10       <property name="driverClassName" value="com.mysql.jdbc.Driver" />
11       <property name="url" value="jdbc:mysql://localhost:3306/SpringTestDb" />
12       <property name="username" value="root" />
13       <property name="password" value="aprile12" />
```

```
14    </bean>
15
16    <!-- Initialization for TransactionManager -->
17    <bean id="transactionManager"
18        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
19        <property name="dataSource" ref="dataSource" />
20    </bean>
21
22    <!-- Definition for studentJDBCTemplate bean -->
23    <bean id="studentJDBCTemplate"
    class="org.andrea.myexample.myProgrammaticTransactionSpring.StudentJDBCTemplate">
24
          <property name="dataSource" ref="dataSource" />
25
          <property name="transactionManager" ref="transactionManager" />
26
      </bean>
27
28
    </beans>
```

Come si può vedere in tale file di configurazione provvediamo a settare le proprietà per effettuare le connessioni al database iniettandole nella definizione del bean avente **ID=dataSource** e definiamo il bean avente **ID=”studentJDBCTemplate”** che usiamo per interagire con i dati nella tabella Student del database. Notate come in tale secondo bean iniettiamo il riferimento al primo bean rappresentante la sorgente dati. Si può inoltre notare che all’interno della definzione del bean avente **ID=”studentJDBCTemplate”** andiamo anche ad iniettare il bean che gestisce la gestione delle transazione avente **ID=”transactionManager”** e che anche in tale bean viene a sua volta iniettata la sorgente dati.

Andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output:

```
——Creazione dei record——
Created Name = Zara, Age = 11
Created Name = Nuha, Age = 20
Created Name = Ayan, Age = 25
——Elenca tutti i record——
ID : 1, Name : Zara, Marks : 99, Year : 2010, Age : 11
ID : 2, Name : Nuha, Marks : 97, Year : 2010, Age : 20
ID : 3, Name : Ayan, Marks : 100, Year : 2011, Age : 25
ID : 4, Name : Zara, Marks : 99, Year : 2010, Age : 11
```

ID : 5, Name : Nuha, Marks : 97, Year : 2010, Age : 20
ID : 6, Name : Ayan, Marks : 100, Year : 2011, Age : 25
ID : 7, Name : Zara, Marks : 99, Year : 2010, Age : 11
ID : 8, Name : Nuha, Marks : 97, Year : 2010, Age : 20
ID : 9, Name : Ayan, Marks : 100, Year : 2011, Age : 25

In questo articolo abbiamo quindi visto come gestire una transazione (una sequenza di istruzioni viste come un'unica istruzione SQL che può concludersi con un successo se tutte le istruzioni della sequenza vanno a buon fine o in caso contrario con un insuccesso) in modo tale da garantire le proprietà ACID che le contraddistinguono.

35 – Gestione delle transazioni in maniera dichiarativa in Spring

Posted on **16 aprile 2013** Views: 884

L'**approccio dichiarativo alla gestione delle transazioni** in Spring ci permette di gestire le transazioni mediante l'aiuto di apposite configurazioni anzichè di codificare tali configurazioni dentro al codice sorgente (come visto nell'articolo precedente riguardante la gestione delle transazioni in maniera programmatica). Ciò significa che possiamo tenere nettamente **separate la gestione delle transazioni dal codice di business**. Per definire tali configurazioni relative alle transazioni possiamo usare le**annotation** oppure definirle in un **file di configurazione XML**. La configurazione dei bean specificherà quindi quali metodi che devono essere transazionali.
Di seguito sono presentati i passi associati alla definizione dichiarativa delle transazioni:

1. Usiamo il tag **<tx:advice />** che si occupa di creare una gestione degli **advice** (rappresenta l'azione da eseguire o prima o dopo l'esecuzione del metodo. E' la frazione di codice che viene invocata dal modulo AOP di Spring durante l'esercuzione dell'applicazione) delle transazioni ed allo stesso tempo definiamo un pointcut che matcha con tutti i metodi che vogliamo rendere transazionali e che fanno riferimento all'advice trasazionale. I valori di default di **<tx:advice />**saranno:
   Propagation Setting: REQUIRED
   Isolation Level: DEFAULT
   Transaction: READ/WRITE
   Transaction Timeout: corrisponde al timeout di default del sistema di transazioni sottostante, oppure nessuno se il timout non è supportato
   Ogni volta che si verifica una RuntimeException ciò innesca un'operazione di rollback mentre le eventuali checked Exception non la innescheranno

2. Se il nome di un metodo è stato incluso nella configurazione transazionale allora, creato un advice, la transazione inizierà prima della chiamata di tale metodo.

3. Il metodo target sarà eseguito all'interno di un blocco **try/catch**.

4. Se il metodo termina normalmente, l'advice AOP eseguira l'operazione di commit della transazione che avrà così successo rendendo così i dati persistenti sul database, altrimenti sarà eseguita un'operazione di rollback che riporterà lo stato all'istante prima dell'inizio di tale transazione

Prima di iniziare è importante avere almeno due tabelle all'interno del nostro database sulle quali è possibile eseguire diverse operazioni di CRUD che sfruttano l'aiuto delle transazioni.

Per prima cosa andiamo a creare la tabella **Student** all'interno di un database che io ho chiamato**SpringTestDb**.

Creiamo il Database:

Per prima cosa accediamo al nostro database nel modo che riteniamo opportuno (attraverso riga di comando o usando qualche programma per interfacciarsi con il DB). Ad esempio considerando di usare Linux e di accedere al DB MySql dalla shell potremmo fare:

mysql -u root -p (ed inserire la password)

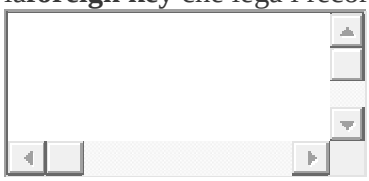
Potremmo poi creare un database dedicato a tale esempio tramite il comando MySql: **create database SpringTestDb;**

Accediamo ora al database appena creato lanciando il seguente comando MySql: **use SpringTestDb;**

Infine creiamo la tabella lanciando il seguente comando nella shell:

```
1 CREATE TABLE Student(
2    ID   INT NOT NULL AUTO_INCREMENT,
3    NAME VARCHAR(20) NOT NULL,
4    AGE  INT NOT NULL,
5    PRIMARY KEY (ID)
6 );
```

La seconda tabella da creare all'intdeno del database **SpringTestDb** è la tabella **Marks** che conterrà i voti degli studenti basandosi sugli anni. In questa tabella troviamo il campo **SID** che rappresenta la**foreign key** che lega i record della tabella **Marks** ad un record della tabella **Student**.



```
1 CREATE TABLE Marks(
2    SID INT NOT NULL,
3    MARKS  INT NOT NULL,
4    YEAR   INT NOT NULL
5 );
```

Andiamo ora ad implementare un'applicazione Spring che usa JDBC per l'accesso ai dati e che implementa alcune semplici operazioni di CRUD sulle tabelle Student e Marks. Come al solito lavoreremo usando STS\Eclipse come ambiente di sviluppo.

Creiamo il progetto in STS\Eclipse:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven usando STS\Eclipse.

All'interno di STS\Eclipse aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample
Artifact **Id:** myDeclarativeTransactionSpring
Version: lasciate 0.0.1-SNAPSHOT
Clickate su **Finish**.
Aprite il file **pom.xml** e clickate sul tab "**Dependencies**" per aggiungere le seguenti dipendenze di org.springframework (nel riquadro Dependencies, non Dependecies Management): **spring-core**,**spring-bean**, **spring-context**, **spring-context-support, spring-jdbc**).
Dovremmo inoltre aggiungere la seguente dipendenza relativa al driver MySql necessario per effettuare la connessione: **mysql-connector-java**
e le dipendenze relative ad: **org.aspecjt** e **cglib**
Qualora non riusciste a scaricare automaticamente le dipendenze attraverso il wizard, questo è il codice del file **pom.xml** da inserire direttamente nel file tramite il tab **pom.xml** della view relativa a tale file:

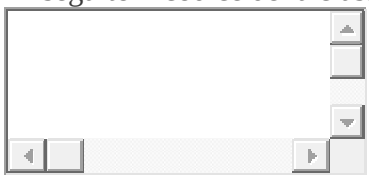
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.andrea.myexample</groupId>
  <artifactId>myDeclarativeTransactionSpring</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>myDeclarativeTransactionSpring</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.2.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>3.2.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.2.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context-support</artifactId>
      <version>3.2.1.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>3.2.1.RELEASE</version>
    </dependency>

    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.23</version>
    </dependency>

    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.6.8</version>
    </dependency>

    <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib</artifactId>
      <version>2.2</version>
    </dependency>
  </dependencies>
```

```
69 </project>
```

Per prima cosa andiamo a creare dell'unico package presente nel nostro progetto (**org.andrea.myexample.myJdbcSpringExample**) creare la classe **StudentDAO** che elenca la lista di tutti i metodi di **CRUD** che saranno messi a disposizione dal nostro DAO.

```
 1  package org.andrea.myexample.myDeclarativeTransactionSpring;
 2
 3  import java.util.List;
 4
 5  import javax.sql.DataSource;
 6
 7  /** Interfaccia che definisce i metodi che implementano le operazioni di CRUD
 8   *  che vogliamo implementare nel nostro DAO:
 9   */
10  public interface StudentDAO {
11
12      /**
13       * Questo metodo viene usato per inizializzare le risorse del database cioè
14       * la connessione al database:
15       */
16      public void setDataSource(DataSource ds);
17
18      /**
19       * Questo metodo serve a creare un record nella tabella Student e nella
20       * tabella Marks:
21       */
22      public void create(String name, Integer age, Integer marks, Integer year);
23
24      /**
25       * Questo metodo serve ad elencare tutti i record all'interno della tabella
26       * Studend e della tabella Marks
27       */
28      public List<StudentMarks> listStudents();
29  }
```

Di seguito il codice della classe **StudentMarks** che rappresenta la nostra entity:



```
 1  package org.andrea.myexample.myDeclarativeTransactionSpring;
 2
 3  // Rappresenta l'entity:
 4  public class StudentMarks {
 5
 6      // Proprietà:
 7      private Integer age;
 8      private String name;
 9      private Integer id;
10      private Integer marks;
11      private Integer year;
12      private Integer sid;
13
14      // Metodi Getter & Setter:
15      public void setAge(Integer age) {
16          this.age = age;
```

```
17    }
18
19    public Integer getAge() {
20        return age;
21    }
22
23    public void setName(String name) {
24        this.name = name;
25    }
26
27    public String getName() {
28        return name;
29    }
30
31    public void setId(Integer id) {
32        this.id = id;
33    }
34
35    public Integer getId() {
36        return id;
37    }
38
39    public void setMarks(Integer marks) {
40        this.marks = marks;
41    }
42
43    public Integer getMarks() {
44        return marks;
45    }
46
47    public void setYear(Integer year) {
48        this.year = year;
49    }
50
51    public Integer getYear() {
52        return year;
53    }
54
55    public void setSid(Integer sid) {
56        this.sid = sid;
57    }
58
59    public Integer getSid() {
60        return sid;
61    }
62 }
```

Di seguito il contenuto della classe **StudentMarksMapper** che si occupa di mappare i campi informativi di un record rappresentante della tabella Student rappresentante uno studente ed i campi informativi del corrispettivo record della tabella Marks rappresentanti i voti di tale studente in un oggetto**StudentMark**

```
1  package org.andrea.myexample.myDeclarativeTransactionSpring;
2
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import org.springframework.jdbc.core.RowMapper;
6
7  /** Classe che implementa l'interfaccia RowMapper. Si tratta di un'interfaccia
8   *  usata da JdbcTemplate per mappare le righe di un ResultSet (oggetto che
```

```
 *  contiene l'insieme delle righe restituite da una query SQL) riga per riga.
 *  Le implementazioni di questa interfaccia mappano ogni riga su di un oggetto
 *  risultante senza doversi preoccupare della gestione delle eccezioni poichè
 *  le SQLException saranno catturate e gestite dalla chiamata a JdbcTemplate.
 */
public class StudentMarksMapper implements RowMapper<StudentMarks> {

    /** Implementazione del metodo dell'interfaccia RowMapper che mappa una
     *  specifica riga della tabella su di un oggetto Student
     *
     *  @param Un oggetto ResultSet contenente l'insieme di tutte le righe
     *         restituite dalla query
     *
     *  @param L'indice che indentifica una specifica riga
     *
     *  @return Un nuovo oggetto Student rappresentante la riga selezionata
     *          all'interno dell'oggetto ResultSet
     *
     *  @see org.springframework.jdbc.core.RowMapper#mapRow(java.sql.ResultSet, int)
     */
    public StudentMarks mapRow(ResultSet rs, int rowNum) throws SQLException {

        StudentMarks studentMarks = new StudentMarks();

        studentMarks.setId(rs.getInt("id"));
        studentMarks.setName(rs.getString("name"));
        studentMarks.setAge(rs.getInt("age"));
        studentMarks.setSid(rs.getInt("sid"));
        studentMarks.setMarks(rs.getInt("marks"));
        studentMarks.setYear(rs.getInt("year"));

        return studentMarks;
    }
}
```

Di seguito ecco l'implementazione della classe **StudentJDBCTemplate** che implementa i metodi di**CRUD** definiti nell'interfaccia **StudentDAO:**

```
package org.andrea.myexample.myDeclarativeTransactionSpring;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;

/**
 * Classe che fornisce l'implementazione per il nostro DAO le cui funzionalità
 * di CRUD sono state definite tramite l'interfaccia StudentDAO
 */
public class StudentJDBCTemplate implements StudentDAO {

    // Utility per l'accesso alla sorgente dati
    private JdbcTemplate jdbcTemplateObject;

    /**
     * Metodo Setter per l'Injection della dipendenza relativa alla sorgente
     * dati. Tale metodo inoltre costruisce anche l'oggetto istanza di
     * JdbcTemplate usato per interagire con i dati nel database.
     *
     * @param la sorgente dati
     */
```

```
24  public void setDataSource(DataSource dataSource) {
25     this.jdbcTemplateObject = new JdbcTemplate(dataSource);
26  }
27
28  /**
29   * Metodo relativo all'operazione di CREATE che inserisce un nuovo record
30   * all'interno della tabella Student ed un correlato nuovo record nella
31   * tabella Marks.
32   */
33  public void create(String name, Integer age, Integer marks, Integer year) {
34
35     try {
36        // Query che inserisce nome ed età nella tabella Student:
37        String SQL1 = "insert into Student (name, age) values (?, ?)";
38        // Esegue la query passandogli anche i valori effettivi da inserire:
39        jdbcTemplateObject.update(SQL1, name, age);
40
41        // Seleziona l'ultimo studente inserito nella tabella Marks:
42        String SQL2 = "select max(id) from Student";
43        // Esegue la query e mette il risultato (l'ID) in sid:
44        int sid = jdbcTemplateObject.queryForInt(SQL2);
45
46        /**
47         * Query che inserisce un nuovo record nella tabella Marks. Il
48         * record rappresenta il voto per l'ultimo studente inserito nella
49         * tabella Student:
50         */
51        String SQL3 = "insert into Marks(sid, marks, year) "
52              + "values (?, ?, ?)";
53        // Esegue la query passandogli anche i valori effettivi da inserire:
54        jdbcTemplateObject.update(SQL3, sid, marks, year);
55
56        System.out.println("Created Name = " + name + ", Age = " + age);
57
58        // SIMULA UNA RuntimeException:
59        //throw new RuntimeException("Simulazione di una condizione d'errore");
60     } catch (DataAccessException e) {        // GESTIONE DELL'ECCEZIONE
61        System.out.println("Errore nella creazione dei record, esegue rollback");
62        throw e;
63     }
64  }
65
66  /**
67   * Metodo relativo all'operazione di READ che recupera la lista degli
68   * studenti e dei relativi voti
69   *
70   * @return La lista di oggetti che rappresentano uno studente ed i suoi voti
71   *         correlati
72   */
73  public List<StudentMarks> listStudents() {
74
75     /**
76      * Query che estrae la lista di tutti i record nella tabella Student e
77      * che per ogni record in tale tabella estrae i relativi record
78      * correlati nella tabella Marks
79      */
80     String SQL = "select * from Student, Marks where Student.id=Marks.sid";
81
82     /**
83      * Ottengo la lista degli oggetti StudentMarks, corrispondenti ognuno ad
84      * un record della tabella Student con i correlati vori rappresentati
85      * dai record della tabella Marks, invocando il metodo query
86      * sull'oggetto JdbcTemplate passandogli i seguenti parametri.
87      *
88      * @param La query per creare il preparated statement
89      * @param Un oggetto che implementa RowMapper che viene usato per
90      *         mappare una singola riga della tabella su di un oggetto Java
91      */
```

```
92        List<StudentMarks> studentMarks = jdbcTemplateObject.query(SQL,
93                                          new StudentMarksMapper());
94        return studentMarks;
95     }
96 }
```

NB: Come potete notare, a differenza dell'esempio proposto nell'articolo precedente, in questo caso non abbiamo iniettato un'oggetto instanza di *PlatformTransactionManager* nella classe che implementa il nostro DAO poichè la gestione delle transazioni sarà di tipo dichiarativo e la sua configurazione avverrà mediante XML.

Ed infine è presentato il codice della classe **MainApp.java**

```
1  package org.andrea.myexample.myDeclarativeTransactionSpring;
2
3  import java.util.List;
4  import org.springframework.context.ApplicationContext;
5  import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7  // Classe principale:
8  public class MainApp {
9
10    public static void main(String[] args) {
11
12       /**
13        * Crea il contesto in base alle impostazioni dell'applicazione definite
14        * nel file Beans.xml
15        */
16       ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
17
18       /**
19        * Recupera un bean avente id="studentJDBCTemplate" nel file di
20        * configurazione Beans.xml
21        */
22       //StudentJDBCTemplate studentJDBCTemplate = (StudentJDBCTemplate) context.getBean("studentJDBCTemplate");
23       StudentDAO studentJDBCTemplate = (StudentDAO) context.getBean("studentJDBCTemplate");
24       System.out.println("DAO Type: " + studentJDBCTemplate.getClass());
25
26       System.out.println("------Creazione dei record--------");
27       // Creo i record nelle tabelle Studend e Marks:
28       studentJDBCTemplate.create("Zara", 11, 99, 2010);
29       studentJDBCTemplate.create("Nuha", 20, 97, 2010);
30       studentJDBCTemplate.create("Ayan", 25, 100, 2011);
31
32       System.out.println("------Elenca tutti i record--------");
33       // Recupera la lista degli studenti con i voti ad essi associati:
34       List<StudentMarks> studentMarks = studentJDBCTemplate.listStudents();
35
36       for (StudentMarks record : studentMarks) {        // e li stampa
37          System.out.print("ID : " + record.getId());
38          System.out.print(", Name : " + record.getName());
39          System.out.print(", Marks : " + record.getMarks());
40          System.out.print(", Year : " + record.getYear());
41          System.out.println(", Age : " + record.getAge());
42       }
43    }
44 }
```

NB: In questa classe **MainApp** siamo vincolati a creare ed eseguire il cast dell'oggetto DAO con il tipo della sua interfaccia **StudentDAO** e non possiamo dichiararlo con il tipo concreto poichè così dovrà

essere dichiarato nella definizione del pointcut nel seguente file di configurazione **Beans.xml** altrimenti otterremo la seguente eccezione:

```
Exception in thread "main" java.lang.ClassCastException: com.sun.proxy.$Proxy0 cannot be cast to
org.andrea.myexample.myDeclarativeTransactionSpring.StudentJDBCTemplate
    at org.andrea.myexample.myDeclarativeTransactionSpring.MainApp.main(MainApp.java:22)
```

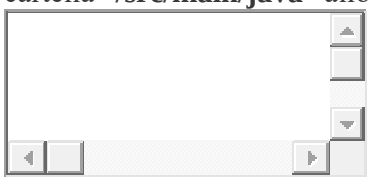
Questo comportamento è dovuto al fatto che Spring supporta AOP tramite proxy e che, di default, tale proxy è basato sull'interfaccia (quindi in questo caso l'interfaccia StudentDAO), volendo potremmo cambiare tale impostazione facendo basare tale meccanismo sulle sottoclassi che implementano tale interfaccia andando a cambiare nel seguente modo la definizione della configurazione di AOP nel nostro file di configurazione XML:

```
<aop:config proxy-target-class="true">
    ...
</aop:config>
```

Tuttavia il sistema di default basato sulle interfacce risulta essere più pulito dell'andare a modificare le impostazioni per basarsi sulle classi concrete che le implementano.

Vediamo infine il file di configurazione **Beans.xml**

Creiamo quindi il file di configurazione **Beans.xml**, come al solito tale file andrà posizionato nella cartella "**/src/main/java**" allo stesso livello del nostro unico package:



```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/tx
   http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
   http://www.springframework.org/schema/aop
   http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <!-- Initializzazione della sorgente dati: -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/SpringTestDb" />
```

```
17      <property name="username" value="root" />
18      <property name="password" value="aprile12" />
19   </bean>
20
21   <tx:advice id="txAdvice" transaction-manager="transactionManager">
22      <tx:attributes>
23         <tx:method name="create" />
24      </tx:attributes>
25   </tx:advice>
26
27   <aop:config>
28      <aop:pointcut id="createOperation"
29         expression="execution(* org.andrea.myexample.myDeclarativeTransactionSpring.StudentDAO.create(..))" />
30      <aop:advisor advice-ref="txAdvice" pointcut-ref="createOperation" />
31   </aop:config>
32
33   <!-- Inizializzazione del Transaction Manager: -->
34   <bean id="transactionManager"
35      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
36      <property name="dataSource" ref="dataSource" />
37   </bean>
38
39   <!-- Definizione del bean che rappresenta il DAO studentJDBCTemplate: -->
40   <bean id="studentJDBCTemplate" class="org.andrea.myexample.myDeclarativeTransactionSpring.StudentJDBCTemplate">
41      <property name="dataSource" ref="dataSource" />
42   </bean>
43
44 </beans>
```

Come si può vedere nel precedente file di configurazione la sematinca delle transazioni è incapsulata all'interno del tag `<tx:advice/>` che in questo caso specifica semplicemente che il metodo **create()**definito nell'interfaccia **StudentDAO** usa le impostazioni transazionali di default. Per quanto riguarda il significato del contenuto del tag **<aop:config>** si può dire che ciò assicura che il precedente advice transazionale sarà eseguito per ogni esecuzione del metodo**com.tutorialspoint.StudentJDBCTemplate.create()** in modo tale da assicurare che il suo comportamento sarà transazionale.

Andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output corrispondente alla simulazione del sollevamente dell'eccezione:

DAO Type: class com.sun.proxy.$Proxy0
——Creazione dei record———
Created Name = Zara, Age = 11
Exception in thread "main" java.lang.RuntimeException: Simulazione di una condizione d'errore

In questo caso viene eseguita l'operazione di **rollback** e **nessun record viene salvato sul database**.

Andando ad eliminare l'istruzione che simula l'eccezione nella classe **StudentJDBCTemplate**:

throw new RuntimeException("Simulazione di una condizione d'errore");

ed andando a rieseguire la classe **MainApp** otterremo invece il seguente output corrispondente all'operazione di **commit** e quindi al fatto che i dati saranno resi persistenti nelle tabelle **Student** e**Marks** del nostro database:

DAO Type: class com.sun.proxy.$Proxy0
——Creazione dei record———
Created Name = Zara, Age = 11
Created Name = Nuha, Age = 20

Created Name = Ayan, Age = 25
——Elenca tutti i record———
ID : 15, Name : Zara, Marks : 99, Year : 2010, Age : 11
ID : 16, Name : Nuha, Marks : 97, Year : 2010, Age : 20
ID : 17, Name : Ayan, Marks : 100, Year : 2011, Age : 25

36 – Creaiamo un DAO integrando Spring ed Hibernate

Posted on **16 aprile 2013** Views: 1233

Introduzione:

Nel corso di questa serie di articoli abbiamo già visto come creare un DAO in Spring per l'interazione con il database usando **JDBC**, in questo **articolo**

Andiamo ora ad analizzare l'intergrazione tra **Spring** (in questo esempio stò usando la versione Spring 3.2 del framework), **JPA** ed **Hibernate**.

Per prima cosa ci tengo a chiarire brevemente il rapporto tra **JPA** ed **Hibernate**, almeno per quanto riguarda questo esempio. In maniera molto semplicistica possiamo dire che **JPA** è usato come una**specifica** mentre **Hibernate** è visto come un'**implementazione di tale specifica**. In questo progetto andremo ad usare le annotazioni JPA per annotare la classe entity da persistere mentre Hibernate ci fornirà la logica per eseguire le operazioni di **CRUD** sul database.

Il Framework Spring ci consente di configurare in maniera facile ed agevole un Hibernate Datasource. Quando usiamo un Framework ORM da solo (ad esempio in un'applicazione Java che non fà uso di Framework come Spring) dobbiamo configurare la sua resource factory mediante le sue API. Ed in questo caso per Hibernate e JPA dobbiamo costruire un oggetto session factory ed un entity manger a partire dalle API nativi di Hibernate e di JPA come illustrato in questo articolo: 10 – Esempio concreto progetto DAO Hibernate

Senza il supporto fornito da Spring non c'è altra scelta se non quella di gestire manualmente tali ogetti.

Spring fornisce alcuni **factory bean** pensati per creare la **session factory di Hibernate** o l'**entity manager factory di JPA**. Tali oggetti saranno visti come **singleton** (oggetti unici) all'interno dello IoC container di Spring e vengono messi a disposizione per facilitare il lavoro del programmatore.

Queste **factory** fornite dal framework possono essere **condivise da più bean** attraverso il noto**meccanismo dell'iniezione della dipendenza**. Inoltre ciò permette alla **session factory di Hibernate**ed all'**entity manager factory di JPA** di **integrarsi** agevolmente con **altri servizzi di accesso ai dati forniti da Spring**, come ad esempio i **data soruces** ed il **transaction manager**.

In questo tutorial vedremo come persistere un oggetto usando **Hibernate 4** (versione esatta usata al momento della scrittura dell'articolo Hibernate 4.1.9**)** e **Spring 3.2** (versione esatta usata al momento della scrittura dell'articolo Spring 3.2.1).

Creazione della tabella all'interno del database:

Per tale esempio sarà usato il DBMS **MySql**. In tale DBMS è stato creato un database chiamato**SpringTestDb** all'interno del quale deve essere creata la tabella **Person** attraverso il seguente comando **DDL**:

```
1 CREATE TABLE person (
2   pid int(11) NOT NULL AUTO_INCREMENT,
3   firstname varchar(255) DEFAULT NULL,
4   lastname varchar(255) DEFAULT NULL,
5   PRIMARY KEY (pid)
6 )
```

Creiamo il progetto in STS\Eclipse:

Seguiamo sempre la stessa procedura vista negli articoli precedenti per creare un nuovo proggetto Spring gestito tramite Maven usando STS\Eclipse.

All'interno di STS\Eclipse aprire l'elemento **Maven** e selezionare "**Maven Project**". Clickate **Next**. Nella prima finestra dello wizard troveremo spuntato solo l'opzione "**Use default Workspace location**", lasciare inalterate tali opzioni e clickate su Next. Nella seconda finestra dello wizard lasciate il default**ArtifactId** settato su "**maven-archetype-quickstart**" e clickate su **Next**.
Inserite i seguenti parametri:

Group **Id:** org.andrea.myexample

Artifact **Id:** HibernateOnSpring

Version: lasciate 0.0.1-SNAPSHOT

Clickate su **Finish**.

Questo è il codice del file **pom.xml** da inserire direttamente nel file tramite il tab **pom.xml** della view relativa a tale file:

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
3     <modelVersion>4.0.0</modelVersion>
4
```

```
<groupId>org.andrea.myexample</groupId>
<artifactId>HibernateOnSpring</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>HibernateOnSpring</name>
<url>http://maven.apache.org</url>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
   <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
   </dependency>

   <!-- Dipendenze di Spring Framework -->
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context-support</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>

   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>

   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>

   <dependency>   <!-- Usata da Hibernate 4 per LocalSessionFactoryBean -->
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>3.2.1.RELEASE</version>
   </dependency>

   <!-- Dipendenze per AOP -->
   <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib</artifactId>
      <version>2.2.2</version>
   </dependency>

   <!-- Dipendenze per Persistence Managment -->
```

```
73
74      <dependency>    <!-- Apache BasicDataSource -->
75          <groupId>commons-dbcp</groupId>
76          <artifactId>commons-dbcp</artifactId>
77          <version>1.4</version>
78      </dependency>
79
80      <dependency>    <!-- MySQL database driver -->
81          <groupId>mysql</groupId>
82          <artifactId>mysql-connector-java</artifactId>
83          <version>5.1.23</version>
84      </dependency>
85
86      <dependency>    <!-- Hibernate -->
87          <groupId>org.hibernate</groupId>
88          <artifactId>hibernate-core</artifactId>
89          <version>4.1.9.Final</version>
90      </dependency>
91
92    </dependencies>
93 </project>
```

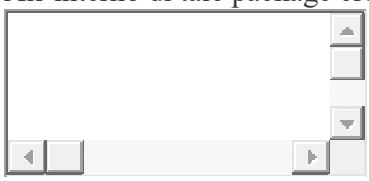
Creiamo l'Entity Class:

Creiamo ora la classe **Person** che rappresenta la nostra **Entity Class** ovvero quella classe le cui istanze rappresentano gli oggetti che devono essere resi persistenti sul database.

Per mantenere il progetto il più ordinato possibile create il seguente package all'interno del percorso "**/src/main/java**" di STS\Eclipse:

org.andrea.myexample.HibernateOnSpring.entity (come regola di buon senso tale package è adibito a contenere tutte le entity class le cui istanze dovrebbero essere persistite sul database, anche se il nostro esempio ne conterrà solo una).

All'interno di tale package create quindi la classe **Person**:



```
 1  package org.andrea.myexample.HibernateOnSpring.entity;
 2
 3  import javax.persistence.Entity;
 4  import javax.persistence.GeneratedValue;
 5  import javax.persistence.GenerationType;
 6  import javax.persistence.Id;
 7  import javax.persistence.Table;
 8
 9  @Entity
10  @Table(name="Person")
11  public class Person {
12
13      @Id
14      @GeneratedValue(strategy=GenerationType.AUTO)
15      private int pid;
16
17      private String firstname;
18
19      private String lastname;
20
21      public int getPid() {
22          return pid;
23      }
24
```

```
25    public void setPid(int pid) {
26        this.pid = pid;
27    }
28
29    public String getFirstname() {
30        return firstname;
31    }
32
33    public void setFirstname(String firstname) {
34        this.firstname = firstname;
35    }
36
37    public String getLastname() {
38        return lastname;
39    }
40
41    public void setLastname(String lastname) {
42        this.lastname = lastname;
43    }
44 }
```

Come detto questa classe rappresenta l'oggetto da persistere sul database ed i suoi capi sono in corrispondenza 1 ad 1 con le colonne della tabella Person creata nel DB.
Come potete notare questa classe e le sue proprietà sono state annotate mediante l'uso di**annotazioni** appartenenti a **JPA2**, andiamo ad analizzarle nel dettaglio:

1. **@Entity:** Si tratta di un'annotation usata per annotare classi e specifica che la classe in questione è un'**Entity Class** ovvero una classe le cui istanze rappresentano record in una tabella del database.

2. **@Table(name="Person"):** Si tratta di un'annotation usata per annotare classi e specifica il mapping tra l'**Entity Class** a cui è applicata ed una tabella nel database. In questo caso specifica che la **classe Person** viene mappata con la **tabella Person** del database: **ogni istanza della classe Person rappresenterà un record della tabella Person sul database**.

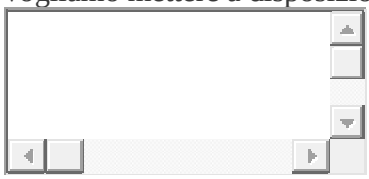
3. **@Id:** Specifica che il campo dell'Entity Class annotato con tale annotation mappa la colonna che rappresenta primary key sulla tabella. Il tipo del campo della classe Java deve essere o un tipo primitivo Java o un qualsiasi wrapper di tipi primativi come: `String`; `java.util.Date`;`java.sql.Date`; `java.math.BigDecimal`; `java.math.BigInteger`.
   Se non è specificato il nome di nessuna colonna, allora si assume per default che il nome della colonna nella tabella sia i nome della proprietà annotata con **@Id** nella classe Java.
4. **@GeneratedValue(strategy=GenerationType.AUTO):** Stabilisce le specifiche del strategia di generazione dei valori della chiave primaria. L'annotazione **@GeneratedValue** può essere applicata ad una proprietà che mappa la chiave primaria di una tabella. Il valore della strategy settato a **GenerationType.AUTO** indica che il provider di persistenza sceglie automaticamente la strategia adeguata per lo specifico database in uso. Quì maggiori informazioni circa tale annotazione ed i valori delle possibili strategie per generare la chiave primaria:

http://docs.oracle.com/javaee/5/api/javax/persistence/GenerationType.html

Notate inoltre che questa classe è un **POJO** come dovrebbero essere tutte le clasi che rappresentano oggetti persistiti o da persistere su di una tabella di un database.
Maggiori informazioni quì: 7 – Persistent Class in Hibernate

Creiamo l'interfaccia che rappresenta il nostro DAO:

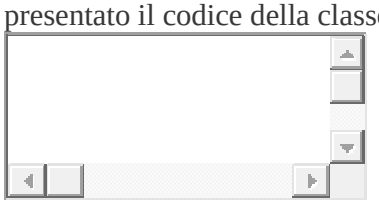
Per mantenere il progetto il più ordinato possibile create il seguente package all'interno del percorso "**/src/main/java**" di STS\Eclipse:
org.andrea.myexample.HibernateOnSpring.dao (come regola di buon senso tale package è adibito a contenere tutte le interfacce e le classi concrete che le implementano correlate agli ogetti DAO per interagire con il database. Qualora il progetto avesse grandi dimensioni potremmo prevedere 2 package: uno per le interfacce dei DAO ed uno per le rispettive implementazioni).
All'interno di tale package creiamo quindi la classe **PersonDAO** che definisce i metodi di **CRUD** che vogliamo mettere a disposizione per interagire con il database:



```
1  package org.andrea.myexample.HibernateOnSpring.dao;
2
3  import java.util.List;
4
5  import org.andrea.myexample.HibernateOnSpring.entity.Person;
6
7  public interface PersonDAO {
8
9      public Integer addPerson(Person p);
10
11     public Person getById(int id);
12
13     public List<Person> getPersonsList();
14
15     public void delete(int id);
16
17     public void update(Person personToUpdate);
18
19 }
```

Creazione della classe concreta che implementa il DAO:

Sempre all'interno dello stesso package che contiene l'interfaccia **PersonDao** andiamo a creare la classe concreta che la implementa ed in cui andremmo ad implementare i metodi in essa definita. Di seguito è presentato il codice della classe **PersonDAOImpl**:



```
1  package org.andrea.myexample.HibernateOnSpring.dao;
2
3  import java.util.List;
4
```

```java
import org.andrea.myexample.HibernateOnSpring.entity.Person;
import org.hibernate.Criteria;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
public class PersonDAOImpl implements PersonDAO {

    // Factory per la creazione delle sessioni di Hibernate:
    private SessionFactory sessionFactory;

        // Metodo Setter per l'iniezione della dipendenza della SessionFactory:
        public void setSessionFactory(SessionFactory sessionFactory) {
            this.sessionFactory = sessionFactory;
            System.out.println("Ho iniettato la SessionFactory: " + sessionFactory);

        }

        /** CREATE CRUD Operation:
         * Aggiunge un nuovo record rappresentato nella tabella rappresentato
         * da un oggetto Person
         */
        @Transactional(readOnly = false)
        public Integer addPerson(Person p) {
            System.out.println("Inside addPerson()");

            /** Recupera la sessione correntemente aperta e persiste
             * l'oggetto sul DB:
             */
            Integer personID = (Integer) sessionFactory.getCurrentSession().save(p);
            return personID;

        }

        // READ CRUD Operation (legge un singolo record avente uno specifico id):
        @Transactional
        public Person getById(int id) {

            Person retrievedPerson = null;

            System.out.println("Inside getById()");

            /** Recupera la sessione correntemente aperta e ritorna l'oggetto
             *  avente l'id passato al metodo come parametro di input
             */
            retrievedPerson = (Person) sessionFactory.getCurrentSession().get(Person.class, id);

            return retrievedPerson;

        }

        // READ CRUD Operation (recupera la lista di tutti i record nella tabella):
        @SuppressWarnings("unchecked")
        @Transactional
        public List<Person> getPersonsList() {

            System.out.println("Inside getPersonsList()");
            List<Person> personList = null;

            /** Recupera la sessione correntemente aperta e crea un oggetto
             *  Criteria rappresentante la query
             */
            Criteria criteria = sessionFactory.getCurrentSession().createCriteria(Person.class);
            personList = criteria.list();  // Ottiene la lista di tutti i record

            return personList;
```

```
73      }
74
75      // DELETE CRUD Operation (elimina un singolo record avente uno specifico id):
76      @Transactional
77      public void delete(int id) {
78         System.out.println("Inside delete()");
79
80         Person personToDelete = getById(id);
81
82         // Recupera la sessione correntemente aperta ed elimina l'oggetto passato:
83         sessionFactory.getCurrentSession().delete(personToDelete);
84      }
85
86      // UPDATE CRUD OPERATION (aggiorna un determinato record rappresentato da un oggetto)
87      @Transactional
88      public void update(Person personToUpdate) {
89
90         System.out.println("Inside update()");
91
92         // Recupera la sessione correntemente aperta ed aggiorna un record:
93         sessionFactory.getCurrentSession().update(personToUpdate);
94      }
95
96 }
```

Come si può vedere in tale classe sono stati implementati tutti i metodi di **CRUD** definiti nell'interfaccia**SpringDAO**.
Le cose più importanti da notare sono due:

1. **Tutti i metodi sono annotati mediante l'annotazione @Transactional:** Annotare un metodo con **@Transactional** significa dire a Spring che quel metodo deve avere un comportamento transazionale, ovvero che tutte le istruzioni svolte al suo interno devono essere viste come un'unica istruzione che può andare a buon fine e terminare di conseguenza con una commit sul database o terminare con un fallimento che comporterà un'operazione di rollback che riporterà il database nella condizione trovata appena prima dell'invocazione del metodo in questione.
   Possiamo annotare con **@Transactional** sia i singoli metodi di una classe concreta, sia l'intera classe (in questo secondo caso ciò implicherà che tutti i metodi al suo interno saranno considerati a loro volta transazionali).
   Bisogna inoltre dire che la presenza di **@Transactional** su un metodo o su una classe non è sufficiente per attivare il comportamento transazionale poichè tale annotation è un solo un metadato che può essere consumato da qualche infrastruttura che è in grado di utilizzare tali metadati di configurazione a runtime. Per attivare il comportamento transazionale si è dovuto inserire l'elemento **<tx:annotation-driven transaction-manager="transactionManager"/>**all'interno del file di configurazione **datasource.xml**.
   Magiori informazioni su **@Transacional** quì:
   http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/transaction.html#transaction-declarative-annotations

2. **Non dobbiamo gestire l'apertura e la chiusura delle Session e la gestione delle transazioni a mano:** Un'altra cosa importante da notare è il fatto di non dover gestire manualmente l'apertura e la chiusura dell'oggetto **Session** rispettivamente quando entriamo in un metodo di CRUD e quando ne stiamo per uscire. Ne tantomeno dobbiamo gestire manualmente le transazioni come dovremmo invece fare se stessimo sviluppando un'applicazione Hibernate standalone (ovvero in questo caso che non preveda l'uso di Spring) come si è fatto in questo articolo: **10 – Esempio concreto progetto DAO Hibernate**
   Ciò è dovuto al fatto che sarà Spring stesso a gestire i metodi in maniera transazionale considerandoli come un'unica operazione che potrà andare a buon fine o fallire (come detto al punto precedente). Inoltre sarà sempre Spring ad occuparsi di aprire la sessione quando entra in un metodo annotato mediante **@Transactional** e a chiuderla nel momento prima di uscirne.
   Come potete notare confrontando la classe concreta che implementa il DAO sviluppato in questo articolo e quella che implementa il DAO nell'articolo linkato dedicato ad un'esempio di applicazione Hibernate Standalone il risparmio di linee di codice è consistente e tende a diverntare veramente considerevole in progetti di grandi dimensioni in cui vi sono tante tabelle da mappare con relativi POJO da persistere.

Per quanto riguarda l'annotazione **@SuppressWarnings("unchecked")** con cui è stato annotato il metodo **getPersonList()** si tratta di un'annotazione Java che **dice al compilatore di ignorare i warning relativi ad operazioni generiche unchecked come i cast (non ignora le eccezioni)**. La usiamo perchè, a volte, i tipi generics di Java non ci fanno fare ciò che vorremmo fare (o meglio ci mostrano dei warning), così facendo stiamo dicendo al compilatore che le operazioni eseguite sono da considerare perfettamente legali
In questa implementazione del DAO abbiamo creato query usando:

1. **HQL (Hibernate Query Language):** quì la documentazione: http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html

2. **Oggetti Criteria:** quì la documentazione: http://docs.jboss.org/hibernate/orm/4.1/devguide/en-US/html/ch12.html

Creaione della classe principale MainApp:
Per mantenere il progetto il più ordinato possibile create il seguente package all'interno del percorso "**/src/main/java**" di STS\Eclipse:
org.andrea.myexample.HibernateOnSpring
All'interno di tale package creiamo la classe principale **MainApp** usata per testare il corretto funzionamento della nostra applicazione:

```
1  package org.andrea.myexample.HibernateOnSpring;
2
3  import java.util.List;
4  import java.util.Iterator;
5
6  import org.andrea.myexample.HibernateOnSpring.dao.PersonDAO;
7  import org.andrea.myexample.HibernateOnSpring.entity.Person;
8  import org.springframework.context.ApplicationContext;
9  import org.springframework.context.support.ClassPathXmlApplicationContext;
10
11 public class MainApp {
12
13    public static void main(String[] args) {
14
15       ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
16       System.out.println("Contesto recuperato: " + context);
17
18       Person persona1 = new Person();
19
20       persona1.setFirstname("Pippo");
21       persona1.setLastname("Blabla");
22
23       System.out.println("Creato persona1: " + persona1);
24
25       PersonDAO dao = (PersonDAO) context.getBean("personDAOImpl");
26
27       System.out.println("Creato dao object: " + dao);
28
29       dao.addPerson(persona1);
30
31       System.out.println("persona1 salvata nel database");
32       System.out.println("persona1 ha id: " + persona1.getPid());
33
34       System.out.println("Recupero l'oggetto Person avente id: "
35             + persona1.getPid());
36
37       Person personRetrieved = dao.getById(persona1.getPid());
38
39       System.out.println("nome: " + personRetrieved.getFirstname());
40       System.out.println("cognome: " + personRetrieved.getLastname());
41       System.out.println("ID: " + personRetrieved.getPid());
42
43       System.out.println("Aggiungo altri 2 record nella tabella: ");
44
45       Person persona2 = new Person();
46       Person persona3 = new Person();
47
48       persona2.setFirstname("Mario");
49       persona2.setLastname("Rossi");
50       persona3.setFirstname("Paolino");
51       persona3.setLastname("Paperino");
52
53       dao.addPerson(persona2);
54       dao.addPerson(persona3);
55
56       List<Person> listaPersone = dao.getPersonsList();
57
58       for (Iterator iterator = listaPersone.iterator(); iterator.hasNext();) {
59          Person persona = (Person) iterator.next();
60          System.out.print("Nome: " + persona.getFirstname());
61          System.out.print(" Cognome: " + persona.getLastname());
```

```
62            System.out.println("  ID: " + persona.getPid());
63        }
64
65        System.out.println("Elimina il primo record dalla tabella");
66
67        Person firstPerson = listaPersone.get(0);
68        System.out.println("Persona da eliminare: " + firstPerson);
69
70        dao.delete(firstPerson.getPid());
71
72        listaPersone = dao.getPersonsList();
73
74        for (Iterator iterator = listaPersone.iterator(); iterator.hasNext();) {
75            Person persona = (Person) iterator.next();
76            System.out.print("Nome: " + persona.getFirstname());
77            System.out.print(" Cognome: " + persona.getLastname());
78            System.out.println("  ID: " + persona.getPid());
79        }
80
81        System.out.println("Aggiorna il primo record della tabella:");
82
83        firstPerson = listaPersone.get(0);
84
85        System.out.println("Persona da aggiornare: " + firstPerson);
86        System.out.print("Nome: " + firstPerson.getFirstname());
87        System.out.print(" Cognome: " + firstPerson.getLastname());
88        System.out.println("  ID: " + firstPerson.getPid());
89
90        System.out.println("CAMBIO DEI DATI:");
91
92        firstPerson.setFirstname("Gatto");
93        firstPerson.setLastname("Silvestro");
94
95        System.out.println("Nuovi dati: " + firstPerson);
96        System.out.print("Nome: " + firstPerson.getFirstname());
97        System.out.print(" Cognome: " + firstPerson.getLastname());
98        System.out.println("  ID: " + firstPerson.getPid());
99
100       System.out.println("UPDATING !!!");
101
102       dao.update(firstPerson);
103
104       listaPersone = dao.getPersonsList();
105
106       for (Iterator iterator = listaPersone.iterator(); iterator.hasNext();) {
107           Person persona = (Person) iterator.next();
108           System.out.print("Nome: " + persona.getFirstname());
109           System.out.print(" Cognome: " + persona.getLastname());
110           System.out.println("  ID: " + persona.getPid());
111       }
112
113   }
114
115 }
```

Come è facile intuire questa classe viene usata solamente allo scopo di provare il corretto funzionamento di tutte le operazioni di CRUD che sono stae definite.

Creazione dei file di configurazione XML:

Questo step consiste nel creare due file XML di configurazione: tali file devono essere inseriti nella root del classpath dell'applicazione (nel nostro caso dentro "**src/main/java**"), allo stesso livello dei package definiti:

Creiamo il file Beans.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/schema/beans/spring-beans-3.1.xsd">

    <import resource="classpath:datasource.xml"/>

    <!-- Register @Autowired annotation -->
    <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />
</beans>
```

Creiamo il file datasource.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"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.1.xsd ">

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/SpringTestDb" />
        <property name="username" value="root" />
        <property name="password" value="aprile12" />
        <property name="initialSize" value="2" />
        <property name="maxActive" value="5" />
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="packagesToScan" value="org.andrea.myexample.HibernateOnSpring.entity" />
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop key="hibernate.show_sql">false</prop>
            </props>
        </property>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="personDAOImpl" class="org.andrea.myexample.HibernateOnSpring.dao.PersonDAOImpl" >
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- Register @Autowired annotation -->
```

```
42     <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />
43 </beans>
```

Tale file è il file principale in cui si defiscono e si configgurano i bean relativi alla sorgente dati JDBC (che gestisce la connessione con il DB), il bean relativo alla SessionFactory di Hibernate (per la creazione degli oggetti Session che consentono al DAO di interagire con il DB), il bean relativo alla gestione delle transazioni di Spring ed il bean relativoal nostro DAO.

Per concludere andando ora ad eseguire la classe principale **MainApp**, nel tab Console di STS\Eclipse, otterremo il seguente output:

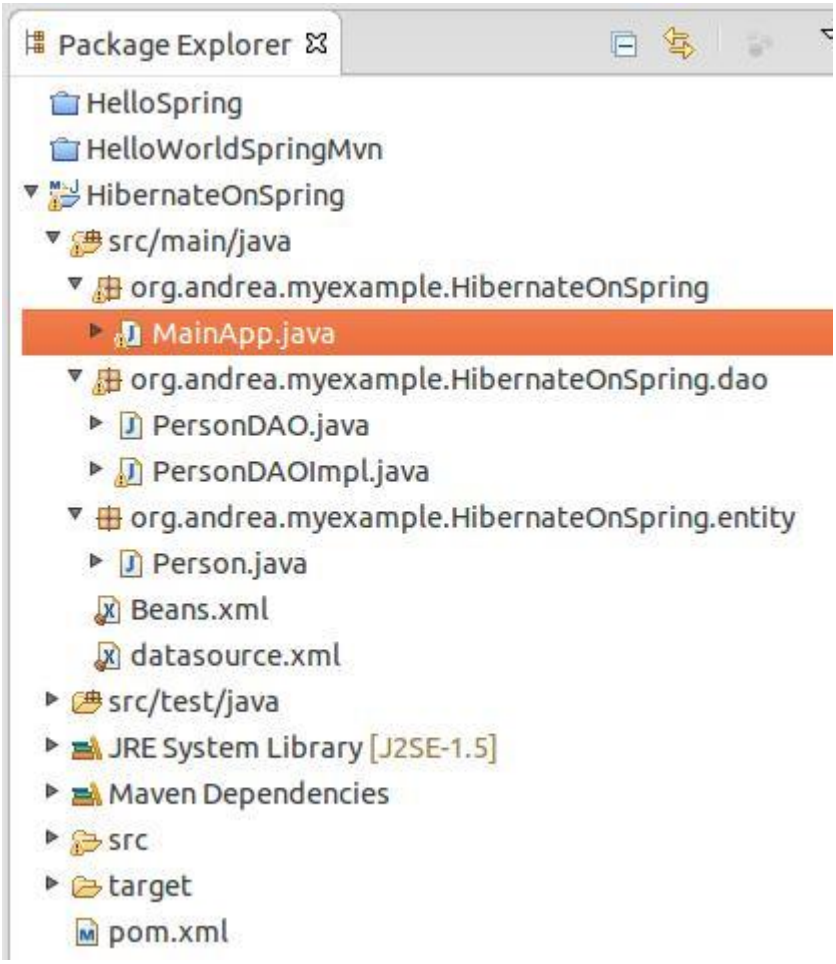
Ho iniettato la SessionFactory: org.hibernate.internal.SessionFactoryImpl@13793140
Contesto recuperato: org.springframework.context.support.ClassPathXmlApplicationContext@70501e4e: startup date [Sat Mar 09 20:05:33 CET 2013]; root of context hierarchy
Creato persona1: org.andrea.myexample.HibernateOnSpring.entity.Person@30eed69a
Creato dao object: org.andrea.myexample.HibernateOnSpring.dao.PersonDAOImpl@73da4e1e
Inside addPerson()
persona1 salvata nel database
persona1 ha id: 187
Recupero l'oggetto Person avente id: 187
Inside getById()
nome: Pippo
cognome: Blabla
ID: 187
Aggiungo altri 2 record nella tabella:
Inside addPerson()
Inside addPerson()
Inside getPersonsList()
Nome: Pippo Cognome: Blabla ID: 187
Nome: Mario Cognome: Rossi ID: 188
Nome: Paolino Cognome: Paperino ID: 189
Elimina il primo record dalla tabella
Persona da eliminare: org.andrea.myexample.HibernateOnSpring.entity.Person@7a92a968
Inside delete()
Inside getById()
Inside getPersonsList()
Nome: Mario Cognome: Rossi ID: 188
Nome: Paolino Cognome: Paperino ID: 189
Aggiorna il primo record della tabella:
Persona da aggiornare: org.andrea.myexample.HibernateOnSpring.entity.Person@354f672e
Nome: Mario Cognome: Rossi ID: 188

CAMBIO DEI DATI:
Nuovi dati: org.andrea.myexample.HibernateOnSpring.entity.Person@354f672e
Nome: Gatto Cognome: Silvestro ID: 188
UPDATING !!!
Inside update()
Inside getPersonsList()
Nome: Gatto Cognome: Silvestro ID: 188
Nome: Paolino Cognome: Paperino ID: 189

Accedendo al nostro database dalla console ed eseguendo la seguente query otterremo il contenuto che ci aspettiamo della tabella:

```
1 mysql&gt; select * from Person;
2 +-----+-----------+-----------+
3 | pid | firstname | lastname  |
4 +-----+-----------+-----------+
5 | 188 | Gatto     | Silvestro |
6 | 189 | Paolino   | Paperino  |
7 +-----+-----------+-----------+
8 2 rows in set (0.00 sec)
```

Per maggiore chiarezza allego uno screenshot dell'organizzazione del progetto all'interno di STS\Eclipse:



Abbiamo quindi visto come creare un semplice DAO che sfrutti Hibernate all'interno di Spring. Ci sarebbero anche altre possibili soluzioni più articolate che sfruttano il modulo AOP di Spring per implementare DAO all'interno di Spring ma saranno argomento di futuri articoli.

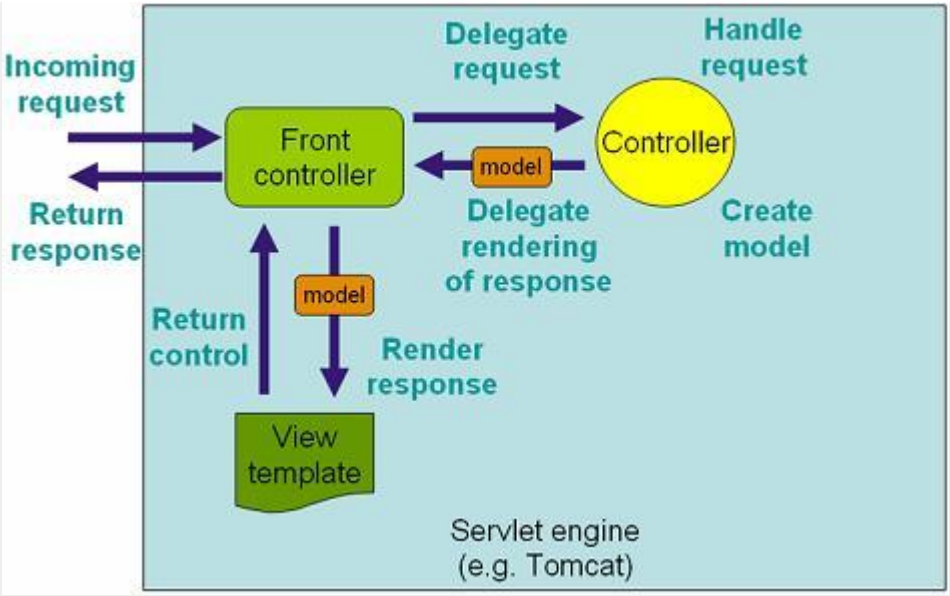
Introduzione a Spring 3.X MVC

Posted on **17 aprile 2013** Views: 4805

Spring MVC è la componente web del framwork Spring. Esso fornisce un ricco insieme di funzionalità per costruire robuste Web Application. Il framework Spring MVC è architetturato e disegnato in modo tale che ogni sin gola parte di logica e di funzionalità risulti essere altamente configurabile.

Inoltre Spring può integrarsi facilmente con altri popolari Framework rivolti alle Web Application come ad esempio Struts, WebWork, Java Server Faces e Tapestry. Ciò significa che si può dire a Spring di usare un qualsiasi di questi framework. Inoltre Spring non è strettamente accoppiato con le Servlet o le JSP per disegnare le View al client. L'intergrazione con altre tecnologie per l'implementazione di View come Velocity, Freemarker, Excell o PDF è comunque possibile.

In Spring Web MVC possiamo usare ogni oggetto come un **oggetto command** o come un **oggetto di form**; Non dobbiamo implementare un'interfaccia specifica del framework o una classe base.
Il binding dei dati in Spring è altamente flessibile: per esempio tratta la mancanta corrispondenza di dati come errori di validazione che possono essere valutati dall'applicazione e non come errori di sistema.
Richiesta di elaborazione del ciclo di vita:



Ciclo di vita di Spring MVC

Il framework Spring Web MVC è, come molti altri web framework MVC, guidato dalle richieste e disegnato intorno ad una **servlet centrale** che dispaccia le richieste ai controllers e che offre altre funzionalità che facilitano lo sviluppo di Web Application. La **DispatcherServlet** di Spring è completamente integrata con lo IoC container di Spring e ci permette di usare ogni altro aspetto (features) di Spring.
Di seguito è mostrata la richesta di elaborazione del ciclo di vita di Spring 3.0 MVC:

1. Il client manda una richiesta al web container tramite una richiesta http.
2. Questa richiesta in entrata viene intercettata dal Front Controller (la DispatcherServlet) e questo cercherà di trovare un appropriato Handler Mappings.
3. Con l'aiuto dell'Handler Mappings, la DispatcherServlet invierà la richiesta al controller appropriato.
4. Il controller proverà a processare la richiesta e ritornerà un oggetto Model and View sotto forma di istanza ModelAndView al front controller.
5. Il Front Controller allora proverà a risolvere la View (che può essere una pagina JSP, Freemarker, Velocity etc) consultando l'oggetto View Resolver.
6. La View selezionata dall'oggetto ViewResolver viene allora resa indietro al client.

CARATTESTISTICHE DI SPRING 3.1\3.2:

Il framework Spring 3.1\3.2 supporta Java 5. Esso fornisce un sistema di configurazione basato sulle annotazioni.
Le caratteristiche di Java 5 come i generics, le annotazioni, varargs, etc possono essere usati in Spring.

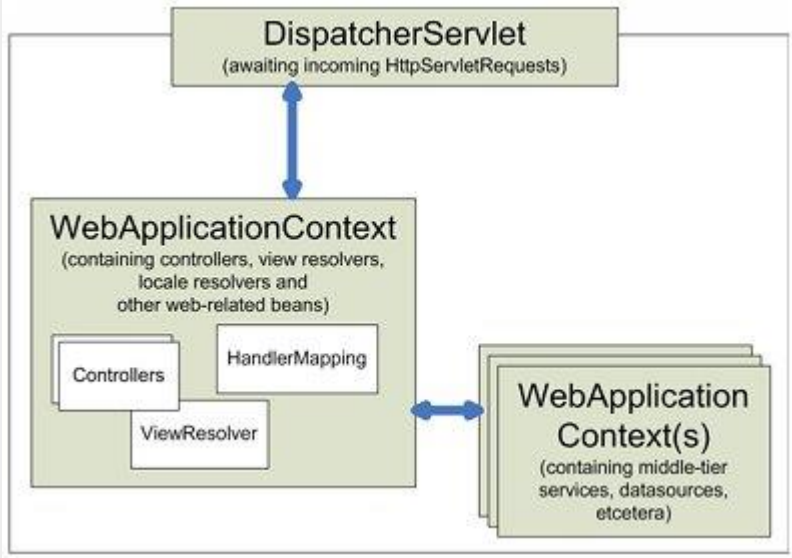
- E' stato introdotto un nuovo expression Language chiamato SpEL (Spring Expression Language) e può essere utilizzato durante la definizione della definizione XML e annotazione di based bean definition.
- Spring 3.0 supporta Web Services di tipo REST.
- La formattazione dei dati non può essere resa più facile di così. Spring 3.0 supporta la formattazione basata sulle annotazione. Ora ad esempio possiamo usare le annotazioni @DateFimeFormat(iso=ISO.DATE) e @NumberFormat(style=Style.CURRENCY) le date ed i formati currency.
- Da Spring 3.0 è supportato JPA 2.0.

CONFIGURAZIONE DI SPRING 3 MVC:

L'entry point di Spring 3.0 MVC è la **DispatcherServlet**. **DispatcherServlet** è una normale classe servlet che implementa la classe base **HttpServlet**. Pertanto abbiamo bisogno di configurarlo in**web.xml**.

```
1  <web-app>
2     <servlet>
3        <servlet-name>example</servlet-name>
4        <servlet-class>org.springframework.web.servlet.DispatcherServlet
5        </servlet-class>
6        <load-on-startup>1</load-on-startup>
7     </servlet>
8
9     <servlet-mapping>
10       <servlet-name>example</servlet-name>
11       <url-pattern>*.html</url-pattern>
12    </servlet-mapping>
13 </web-app>
```

Nel frammento di codice sopra, abbiamo configurato la **DispatcherServlet** nel file **web.xml**. Da notare che abbiamo mappato l'url pattern *.html con l'example DispatcherServlet. Così facendo ogni url avente il pattern .html chiamerà automaticamente lo Spring MVC Front Controller (appunto la nostra DispatcherServlet). Quindi tutte le richieste a file .html saranno gestite da tale example Dispatcher Servlet.



Spring DispatcherServlet

Una volta che la DispatcherServlet è inizializzata, questa cercherà un file di nome [servlet-name]-servlet.xml nel percorso WEB-INF della Web Application. Nell'esempio di sopra, il framework cercherà un file chiamato example-servlet.xml.

Si guardi il precedente diagramma dell'architettura. La WebApplicationContext specificata nel precedente diagramma è un'estensione della comune ApplicationContext con alcune caratteristiche extra necessarie per le Web Application. La WebApplicationContext è capace di risolvere temi ed è anche associrata ad una servlet corrispondente.
Nel prossimo articolo affronteremo la creazione di un primo progetto di Hello World che fà uso di Spring MVC.

2 – Hello World in Spring MVC

Posted on **18 aprile 2013** Views: 13088

Spring 3 MVC: Creare un'applicazione Hello World tramite Spring 3 MVC:
Benvvenuti nella Parte 2 della serie di tutorial dedicata a Spring 3 MVC. Nel precedente articolo abbiamo visto l'introduzione al framework Spring 3, il ciclo di vita del processo di richieste ed il diagramma dell'architettura. In questo articolo creeremo una semplice applicazione di Hello World in Spring MVC 3
A differenza del tutorial originale ho preferito usare come ambiente di sviluppo Spring Tool Suite (STS) al posto di Eclipse, di fatto no cambia nulla poiché si tratta di una versione modificata di Eclipse dotata di tutti i plugin utili nello sviluppo di un'applicazione Spring preinstallati.

COSE DI CUI ABBIAMO BISOGNO:

Prima di iniziare con lo sviluppo dell'applicazione di Hello World dobbiamo procurarci alcuni tool:

1. **JDK 1.5 o superiore2.Tomcat 5.x o superiore o un qualsiasi altro contenitore (Glassfish, JBoss, Websphere, Weblogic etc)**
2. **Tomcat 5.x o superiore o un qualsiasi altro container (Glassfish, JBoss, Websphere, Weblogic etc)**
3. **L'ultima versione disponibile di Spring Tool Suite (STS)**
4. **Spring 3.2.0 MVC JAR files:(download). Following are the list of JAR files required for this application.**

- commons-logging-1.0.4.jar
- jstl-1.2.jar
- org.springframework.asm-3.1.2.RELEASE-A.jar
- org.springframework.beans-3.1.2.RELEASE-A.jar
- org.springframework.context-3.1.2.RELEASE-A.jar
- org.springframework.core-3.1.2.RELEASE-A.jar
- org.springframework.expression-3.1.2.RELEASE-A.jar
- org.springframework.web.servlet-3.1.2.RELEASE-A.jar
- org.springframework.web-3.1.2.RELEASE-A.jar

Il nostro obbiettivo:

Il nostro obbbiettivo è di creare un'applicazione base Spring MVC. Ci sarà una pagina index che visualizzerà: "Say Hello" all'utente. Clickando su tale link l'utente verrà rediretto ad un'altra pagina di hello che visualizzerò il messaggio "Hello World, Spring 3.2.0 !!!"

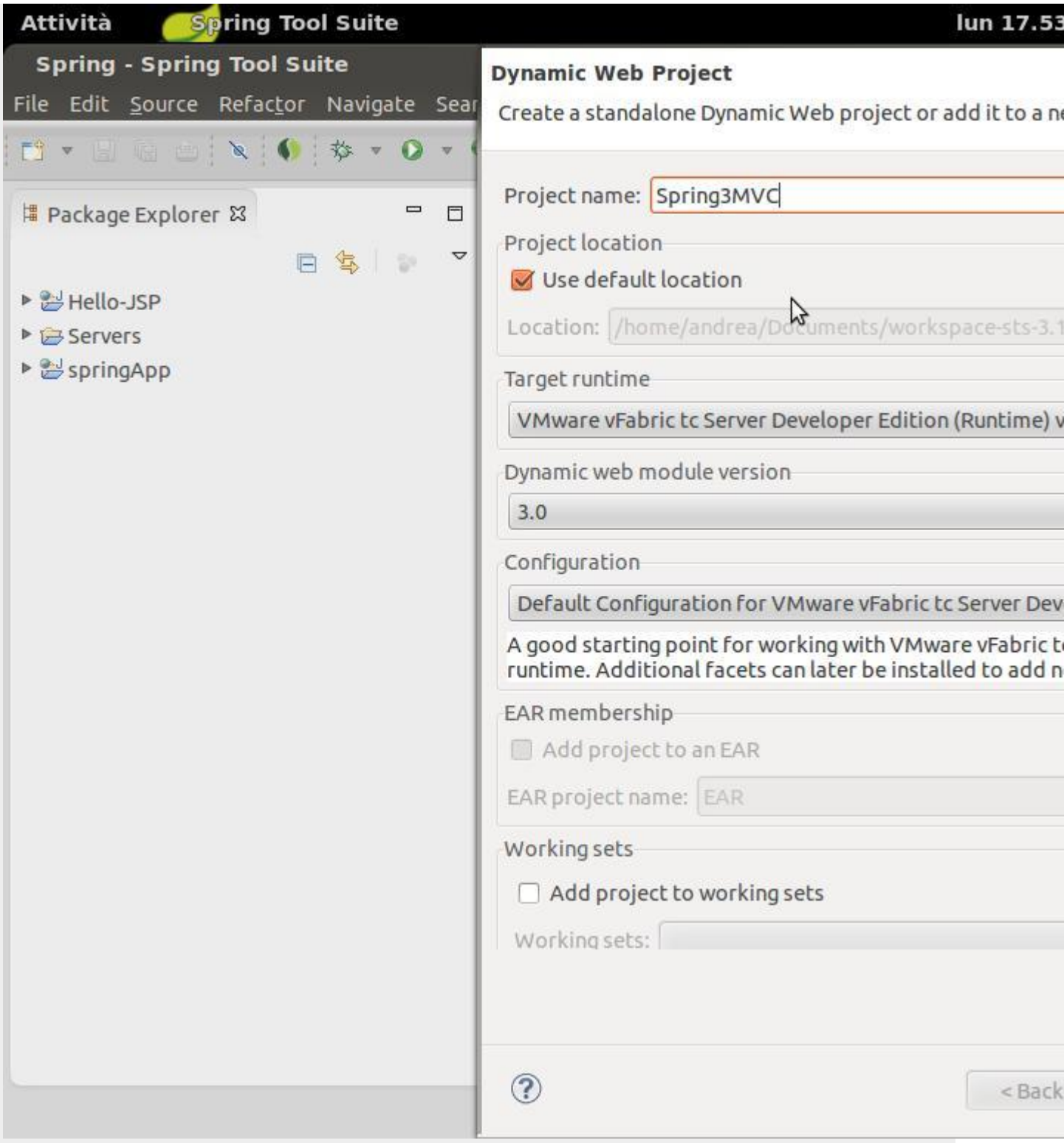
Iniziamo:

Iniziamo con la nostra prima applicazione base Spring 3.2.0 MVC

Aprite STS ed andate in **File —> New —> Project e selezionate Dynamic Web Project** nel wizard chiamato **New Project**.

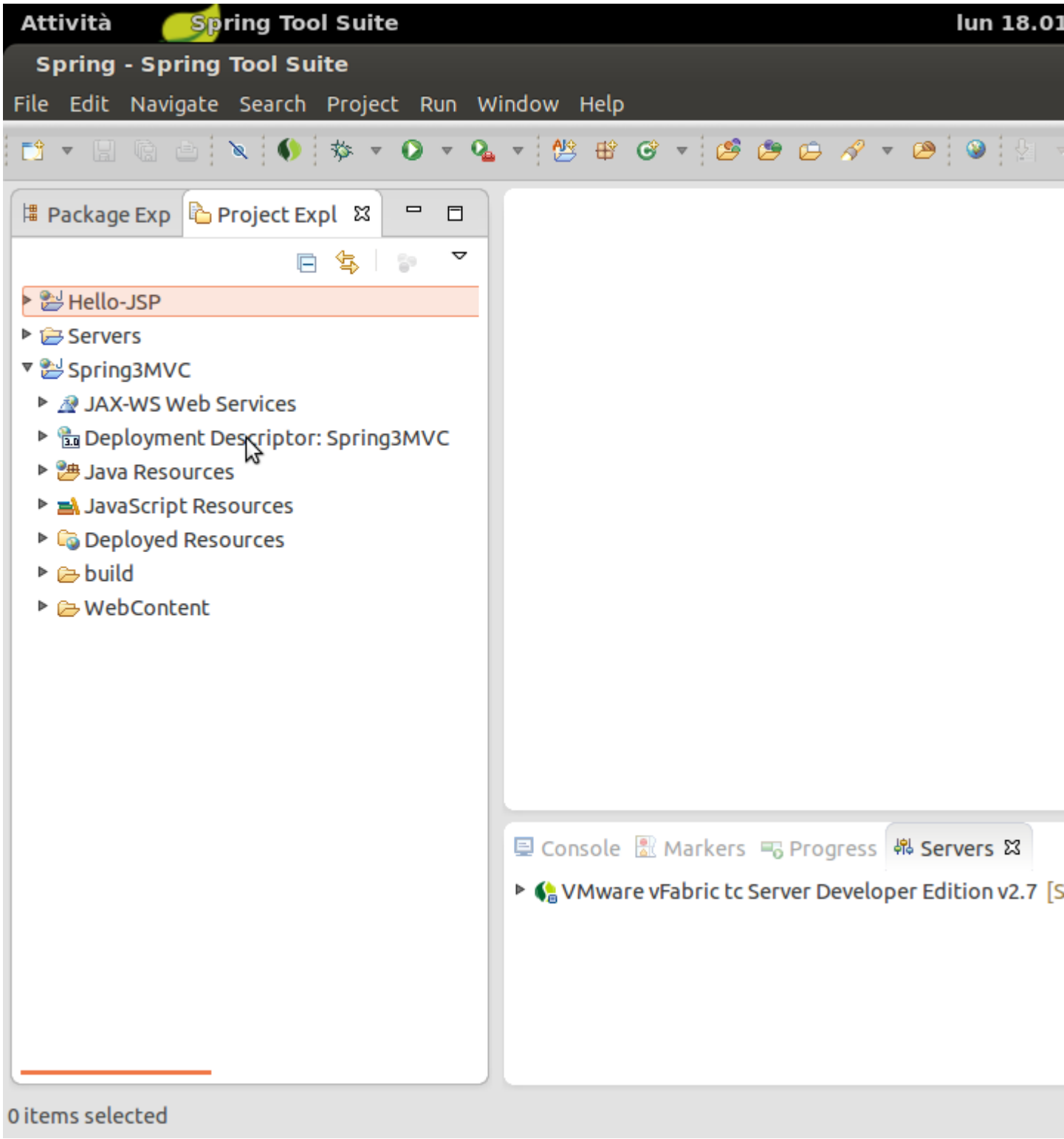
Dopo aver selezionato **Dynamic Web Project** cliclare su **Next** scrivete il nome del progetto. Ad esempio **Spring3MVC**.

Una volta fatto ciò, selezionate il target runtime environment (ad esempio Apache Tomcat v6.0). Ciò serve a far girare il progetto dentro l'ambiente di Eclipse. Alla fine clickate su **Finish**.

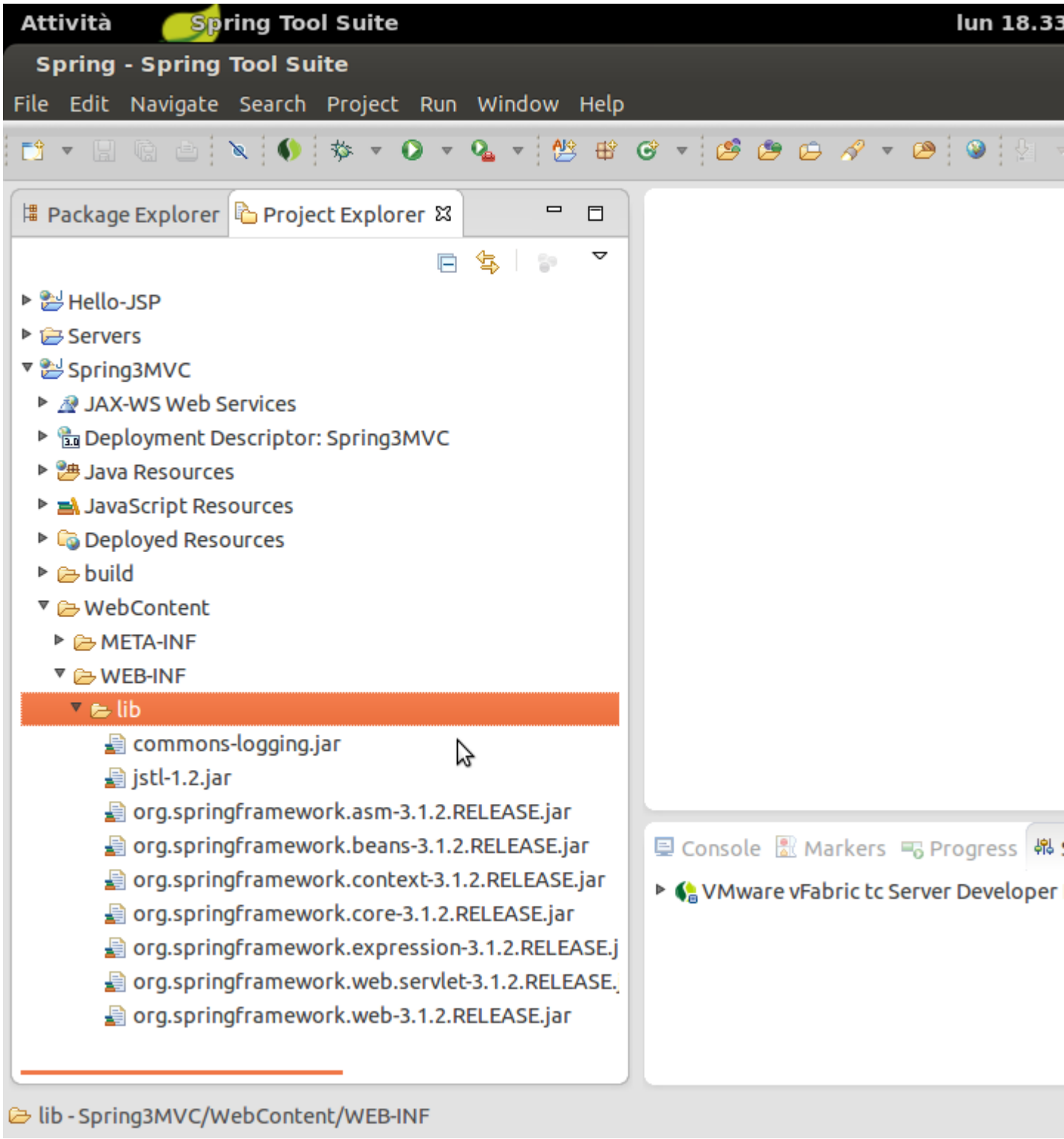


Creazione progetto Spring MVC

Una volta che il progetto è stato creato potrai vedere la struttura all'interno della view Project Explorer di STS\Eclipse.



Ora copia tutti i JAR richiesti nel percorso WebContent > WEB-INF > lib. Create questo percorso nel caso non esistesse.



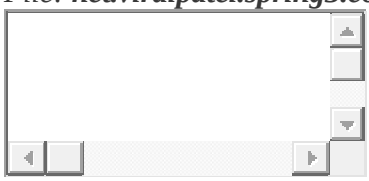
La classe Controller di Spring:

Abbiamo ora bisogno di una classe che implementi il controller MVC di Spring che in pratica processi le richiesti e mostri il messaggio di "Hello World" all'utente.

Per fare ciò creeremo un nuovo package chiamato **net.viralpatel.spring3.controller** all'interno del percorso **source**. Tale package conterrà le classi che implementeranno i controller.

Create una classe chiamata **HelloWorldController all'interno del package net.viralpatel.spring3.controller e copiate il seguente codice dentro tale classe**

File: ***net.viralpatel.spring3.controller.HelloWorldController***



```
1  package net.viralpatel.spring3.controller;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.servlet.ModelAndView;
6
7  @Controller
8  public class HelloWorldController {
9
10    @RequestMapping("/hello")
11    public ModelAndView helloWorld() {
12
13        String message = "Hello World, Spring 3.0!";
14        return new ModelAndView("hello", "message", message);
15    }
16 }
```

Notate che abbiamo annotato la classe HelloWorldController con le annotazioni **@Controller** e**@RequestMapping("/hello")** alla linea 7 ed alla linea 10. Quando Spring esegue la scansione dei nostri package, riconoscerà questo bean come un Controller Bean per processare richieste.

L'annotazione **@RequestMapping** dice a Spring che questo controller deve elaborare tutte le richieste che iniziano con il pattern /hello nel percorso del path. Ciò include sia percorsi del tipo: /hello/* sia percorsi del tipo /hello.html.

Il metodo **helloWorld()** (che come abbiamo detto gestisce tutte le richieste del client che iniziano con il pattern /hello) ritorna al chiamante (al Front Controller, quindi al nostro oggetto istanza di DispatcherServlet) un oggetto **ModelAndView**.

Tale oggetto ModelAndView prova a risolvere ad una view chiamata "hello" ed il data model viene passato indietro al brroswer così da poter accedere ai dati all'interno della JSP.

Il nome logico della view ("hello") viene risolto concretamente in "**/WEB-INF/jsp/hello.jsp**".

In seguito discuteremo di come il nome logico della view ("hello") che è ritornato dal nostro Controller Bean all'interno dell'oggetto ModelAndView viene mappato verso il path /WEB-INF/jsp/hello.jsp.

L'oggetto ModelAndView contiene anche un messaggio avente chiave "message" e valore "Hello World, Spring 3!". Questo è il dato che verrà passato alla nostra view. Normalmente questo è un oggetto valore (ad esempio una stringa) in un form di un java bean che conterrà dati da mostrare nella nostra view. In questo caso stiamo semplicemente passando alla nostra View una stringa.

La View: creiamo una JSP:

Per mostrare il messaggio di Hello World creeremo una pagina JSP. Notate che questa pagina JSP verrà

creata dentro al percorso /WEB-INF/jsp/. Create la pagina hello.jsp all'interno della directory /WEB-INF/jsp e copiate il seguente codice dentro di essa:

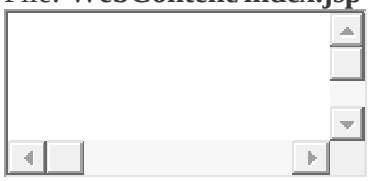
File: **WEB-INF/jsp/hello.jsp**

```
1 <title>Spring 3.0 MVC Series: Hello World - ViralPatel.net</title>
2     ${message}
```

Questa JSP mostra semplicemente un messaggio usando l'espressione **${message}**. Notate che il nome "message" è quello che abbiamo settato dentro all'oggetto **ModelAndView** con la stringa di messaggio.

A questo punto abbiamo anche bisogno di un file **index.jsp** che sarà l'**entry point** della nostra applicazione. Create un file sotto la cartella WebContent nel vostro progetto e copiate il seguente codice all'interno di tale file

File: **WebContent/index.jsp**



```
1 <title>Spring 3.0 MVC Series: Index - ViralPatel.net</title>
2     <a href="hello.html">Say Hello</a>
```

Specifichiamo il Mapping di Spring MVC all'interno del file web.xml:
Come discusso nel precedente articolo (Introduzione a Spring 3 MVC), l'entry point di un'applicazione Spring MVC sarà la Servlet definita nel descrittore di deploy (ovvero il file web.xml). Quindi verrà definita la seguente classe di entry nel file web.xml: **org.springframework.web.servlet.DispatcherServlet**.
Aprite quindi il file web.xml sotto la directory WEB-INF e copiate il seguente codice:

File: **WEB-INF/web.xml**

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/javaee"
4     xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6     id="WebApp_ID" version="2.5">
```

```
 7    <display-name>Spring3MVC</display-name>
 8    <welcome-file-list>
 9        <welcome-file>index.jsp</welcome-file>
10    </welcome-file-list>
11
12    <servlet>
13        <servlet-name>spring</servlet-name>
14        <servlet-class>
15            org.springframework.web.servlet.DispatcherServlet
16        </servlet-class>
17        <load-on-startup>1</load-on-startup>
18    </servlet>
19    <servlet-mapping>
20        <servlet-name>spring</servlet-name>
21        <url-pattern>*.html</url-pattern>
22    </servlet-mapping>
23 </web-app>
```

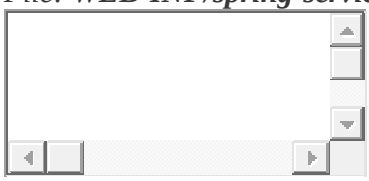
Il codice all'interno del file web.xml mapperà la DispatcherServlet con tutti gli url aventi pattern *.html (quindi la DispatcherServlet avente nome "spring" si occuperà di gestire tutte le richieste http di indirizzi che terminano con .html).

Una cosa da notare è il nome della servlet racchiuso all'interno dei tag **<servlet-name>** nel file**web.xml**. Una volta che la DispatcherServlet è inizializzata, questa cercherà un file chiamato **[servlet-name]-servlet.xml** all'interno della cartella WEB-INF della Web Application. In questo esempio, il framework cercherà un file chiamato **spring-servlet.xml**

File di configurazione di Spring:

Create un file chiamato spring-servlet.xml dentro la cartella WEB-INF e copiate il seguente codice dentro di esso.

*File: **WEB-INF/spring-servlet.xml***



```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <beans xmlns="http://www.springframework.org/schema/beans"
 3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4      xmlns:p="http://www.springframework.org/schema/p"
 5      xmlns:context="http://www.springframework.org/schema/context"
 6      xsi:schemaLocation="http://www.springframework.org/schema/beans
 7
 8  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 9
10  http://www.springframework.org/schema/context
11
12  http://www.springframework.org/schema/context/spring-context-3.0.xsd">
13
14      <context:component-scan
15          base-package="net.viralpatel.spring3.controller" />
16
17      <bean id="viewResolver"
18          class="org.springframework.web.servlet.view.UrlBasedViewResolver">
19          <property name="viewClass"
20              value="org.springframework.web.servlet.view.JstlView" />
21          <property name="prefix" value="/WEB-INF/jsp/" />
```

```
22         <property name="suffix" value=".jsp" />
23     </bean>
24 </beans>
```

In tale file xml di configurazione abbiamo definito un tag <context:component-scan>. Tale tag permette a Spring di chiamare tutti i componenti dal package net.viralpatel.spring3.controller e da tutti i suoi sottopacchetti. Ciò caricherà la nostra classe controller HelloWorldController

Abbiamo anche definito un bean viewResolver. Tale bean si occuperò di risolvere la view e di aggiungere la stringa di prefisso /WEB-INF/jsp/ ed il suffisso .jsp alla view logica nel ModelAndView.
Nota che nella nostra classe HelloWorldController, ritorniamo un oggetto ModelAndView con una view avente nome logico "hello". Il bean viewResolver la risolverà quindi nel path /WEB-INF/jsp/hello.jsp.

Conclusioni:

Ora potete eseguire l'applicazione evedere il risultato. Assumendo che abbiate già configurato il server vFacric (o in alternativa TomCat) all'interno di STS\Eclipse.

Tutto ciò che dovete fare ora è questo:

Aprite la view Server qualora non fosse ancora aperta e visibile all'interno del vostro ambiente di sviluppo (Windows —> Show View —> Server). Per eseguire il progetto fate click destro sul nome del progetto dalla view Project Explorer o dalla view Package Explorer e selezionate:

Run as > Run on Server (Shortcut: Alt+Shift+X, R)

In questo tutorial abbiamo creato una semplice applicazione di Hello World usando il framework Spring 3.1.2 MVC. Abbiamo anche imparato la configurazione di Spring e le 2 annotazioni @Controller e @RequestMapping. Nel prossimo articolo vedremo quanto è facile gestire i form usando Spring MVC.

3 – Spring MVC 3: Gestione Form in Spring 3.X MVC, esempio pratico

Posted on **9 novembre 2013** Views: 4848

Benvenuti nella terza parte della serie di tutorial dedicati a Spring 3 MVC. Abbiamo imparato a configurare Spring MVC nel file di configurazione web.xml e come usare le 2 annotazione @Controller e @RequestMapping. In questo articolo andremo a vedere come gestire i form in Spring 3 MVC.

Useremo il progetto che abbiamo creato nel precedente articolo come base di riferimento ed aggiungeremo le funzionalità del form all'interno di essa. Anche l'applicazione che creeremo sarà una applicazione Contact Manager.

Obbiettivo:
Il nostro obbiettivo è di creare un'applicazione Contact Manager basilare. Questa applicazione avrà un form che prende le informazioni di un contatto dall'utente. Per il momento ci limiteremo a scrivere le informazioni del contatto in un file di log. Impareremo come catturare i dati all'interno del form tramite Spring 3 MVC.

Spring MVC Form

Partenza:

Andiamo ad aggiungere il contact form alla nostra applicazione di Hello World. Aprire il file index.jsp e cambiatelo nel seguente modo:

File: WebContent/index.jsp

```
1  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2  <html>
3  <head>
4      <title>Spring 3.0 MVC Series: Index - ViralPatel.net</title>
5  </head>
6  <body>
7      <div>
8          <a href="hello.html">Say Hello</a>
```

```
 9     </div>
10 </body>
11 </html>
```

Tale codice rindirizza semplicemente l'utente verso la pagina contacts.html

La View – contact.jsp:

Create un file contact.jsp che visualizzerà all'utente il form di inserimento dei dati del nuovo contatto:

File: /WebContent/WEB-INF/jsp/contact.jsp

```
 1 <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
 2 <html>
 3 <head>
 4    <title>Spring 3 MVC Series - Contact Manager</title>
 5 </head>
 6 <body>
 7 <h2>Contact Manager</h2>
 8 <form:form method="post" action="addContact.html">
 9
10    <table>
11    <tr>
12        <td><form:label path="firstname">First Name</form:label></td>
13        <td><form:input path="firstname" /></td>
14    </tr>
15    <tr>
16        <td><form:label path="lastname">Last Name</form:label></td>
17        <td><form:input path="lastname" /></td>
18    </tr>
19    <tr>
20        <td><form:label path="lastname">Email</form:label></td>
21        <td><form:input path="email" /></td>
22    </tr>
23    <tr>
24        <td><form:label path="lastname">Telephone</form:label></td>
25        <td><form:input path="telephone" /></td>
26    </tr>
27    <tr>
28        <td colspan="2">
29            <input type="submit" value="Add Contact"/>
30        </td>
31    </tr>
32 </table>
33
34 </form:form>
35 </body>
36 </html>
```

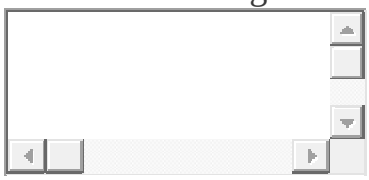
Quì sopra la pagina contact.jsp che visualizza il form. Notate che il form viene inviato ad una pagina **addContact.html**

Aggiungere il Form ed il Controller in Spring 3:

Dobbiamo ora aggiungere la logica in Spring 3 per visualizzare il form ed andare a prendere i valori inseriti dall'utente da questo. Per fare ciò creeremo 2 classi Java.
La prima: **Contact.java** che non è altro che il form per visualizzare/recuperare i dati da schermo.
La seconda: **ContactController.java** che implementa il controller di Spring.

Per manterene il progetto ordinato e separare logicamente le classi che implementano i controler da quelle che implementano la rappresentazione dei nostri dati (quello che generalmente viene chiamato il model, in questo caso una classe che rappresenta i campi del form) creiamo il seguente nuovo package dentro alla cartella src: **net.viralpatel.spring3.form**

All'interno del package appena costruito andiamo a creare la classe Java Contact.java che conterrà il seguente codice:

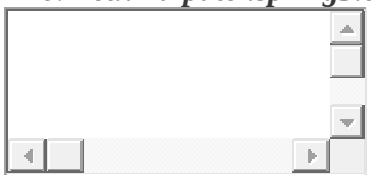


```
 1  package net.viralpatel.spring3.form;
 2
 3  public class Contact {
 4      private String firstname;
 5      private String lastname;
 6      private String email;
 7      private String telephone;
 8      public String getFirstname() {
 9          return firstname;
10      }
11      public void setFirstname(String firstname) {
12          this.firstname = firstname;
13      }
14      public String getLastname() {
15          return lastname;
16      }
17      public void setLastname(String lastname) {
18          this.lastname = lastname;
19      }
20      public String getEmail() {
21          return email;
22      }
23      public void setEmail(String email) {
24          this.email = email;
25      }
26      public String getTelephone() {
27          return telephone;
28      }
29      public void setTelephone(String telephone) {
30          this.telephone = telephone;
31      }
32
33  }
```

Nota bene: ricordate di generare tutti i metodi getter e setter (non inseriti qui dentro per non allungare troppo il codice), per evitare di farlo a mano potete sfruttare le potenzialità

del vostro ambiente IDE, in STS\Eclipse: **Source —> Generate Getter and Setter**, selezionate tutte le proprietà mostrate nel wizard, confermate ed il gioco è fatto.
Creiamo ora la classe Java **ContactController** che implementerà il controller nel package**net.viralpatel.spring3.controller**

*File: **net.viralpatel.spring3.controller.ContactController**:*



```
1  package net.viralpatel.spring3.controller;
2
3  import net.viralpatel.spring3.form.Contact;
4
5  import org.springframework.stereotype.Controller;
6  import org.springframework.validation.BindingResult;
7  import org.springframework.web.bind.annotation.ModelAttribute;
8  import org.springframework.web.bind.annotation.RequestMapping;
9  import org.springframework.web.bind.annotation.RequestMethod;
10 import org.springframework.web.bind.annotation.SessionAttributes;
11 import org.springframework.web.servlet.ModelAndView;
12
13 @Controller
14 @SessionAttributes
15 public class ContactController {
16
17    /* L'annotazione @RequestMapping dice a Spring che questo controller deve elaborare tutte le richieste che iniziano
18       con il pattern /addContact nel percorso del path (quindi gestisce percorsi del tipo /addContact.html).
19       L'attributo method=”RequestMethod.POST” significa che IL METODO addContact() VIENE CHIAMATO SOLO
20 QUANDO L'UTENTE
21       GENERA UNA RICHIESTA DI TIPO POST AD UNA PAGINA /addContact.html
22
23       L'annotazione @ModeLAttribute del parametro di input eseguirà il binding tra i dati della richiesta POST e l'oggetto
24       di tipo Contact usato come parametro di input del metodo
25     */
26    @RequestMapping(value = "/addContact", method = RequestMethod.POST)
27    public String addContact(@ModelAttribute("contact")
28                             Contact contact, BindingResult result) {
29
30       System.out.println("First Name:" + contact.getFirstname() +
31                "Last Name:" + contact.getLastname());
32
33       return "redirect:contacts.html";
34    }
35
36    /* L'annotazione @RequestMapping dice a Spring che questo controller deve elaborare tutte le richieste che iniziano
37     * con il pattern /contacts nel percorso del path. */
38    @RequestMapping("/contacts")
39    public ModelAndView showContacts() {
40
41       return new ModelAndView("contact", "command", new Contact());
42    }
}
```

Nella classe ContactController abbiamo creato 2 metodi con 2 differenti annotazioni Request Mapping:

1. Il metodo **showContacts()** ha l'annotazione **@RequestMapping("/contacts")**. Tale metodo viene chiamato quando l'utente richiede un url che termina con contacts.html.
   Tale metodo renderizzerà un modello con name "contact". Notate che nell'oggetto ModelAndView abbiamo passato un oggetto Contact vuoto avente nome "command". Il framework Sprimg si aspetta un oggetto con un nome command se stare usando il tag **<form:form>** nella vostra pagina JSP.
2. Notate inolte che il metodo il metodo **addContact()** è stato annotato con l'annotazione:**@RequestMapping(value = "/addContact", method = RequestMethod.POST)**.
   Così il metodo verrà chiamato solamente quando l'utente genera una richiesta ad un metodo post dentro l'url /addContact.html.
   Abbiamo inoltre annotato anche l'oggetto di tipo Contact passato come argomento al metodo addContact() usando l'annotazione **@ModelAttribute**. Questa annotazione eseguirà il binding tra i dati della richiesta POST e l'oggetto Contact.

   Nel corpo di questo metodo stampiamo semplicemente la concatenazione dei valori del campo Firstname e Lastname ed alla fine eseguiamo una redirezione verso la view contacts.html

Conclusioni:

Il tutorial sull'uso dei form è ora completo. Eseguite l'applicazione: **Run as > Run on Server (Shortcut: Alt+Shift+X, R)** e verrà mostrato il form dei contatti. Inserite dei valori e premete il bottone Add. Una volta premuto il bottone verrà stampata la concatenazione dei campi firstname e lastname nei sysout logs.
In questo articolo abbiamo imparato come creare un form usando Spring 3 MVC e come visualizzarlo all'interno di una pagina JSP. Abbiamo anche imparato come recuperare i valori inseriti nel form usando l'annotazione ModelAttribute. Nella proissima sezione analizzeremo la validazione dei form ed alcuni diversi metodi di conversione in Spring 3 MVC.

4 – Dividiamo il layout in più sezioni con Tiles

Posted on **9 novembre 2013** Views: 3396

Spring 3 MVC: Tiles Plugin Tutorial con esempio in STS\Eclipse:

Benevenuti nella quarta parte dei tutorial dedicati a Spring 3 MVC. Nel precedente articolo abbiamo visto come creare un form usando Spring 3 MVC e come mostrarlo all'interno di una pagina JSP. Abbiamo anche imparato ad usare l'annotazione **@ModelAttribute.**

In questo tutorial discuteremo di Tiles Framework e della sua integrazione con Spring 3 MVC.

Aggiungeremo il supporto a Tiles al progetto che abbiamo creato nel precedente articolo. Vi raccomando fortemente di procedere seguendo il precedente articolo e di scaricare il codice sorgente della stessa applicazione.

Introduzione a Tiles 2:

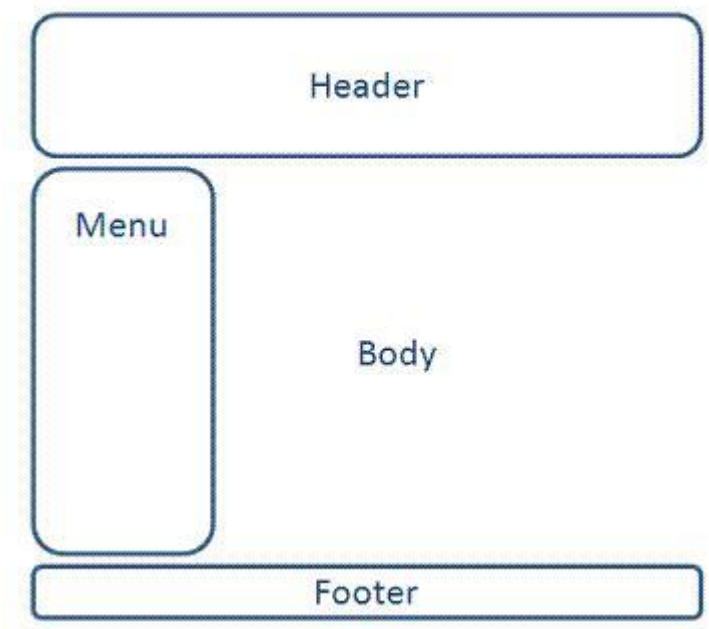
Al giorno d'oggi i siti web sono genealmente divisi in più parti di template riusabili che vengono renderizzati attraverso differenti pagine web. Per esempio un sito contenente un header, un footer, un menù, etcetc. Questi elementi compongono il sito web e gli danno un aspetto comune. Risulta essere molto difficile codificare tutto in ogni pagina web ed inoltre vi è anche la problematica che se in futuro sarà necessaria una modifica ad una di queste parti sarebbe necessario modificarla in ogni pagina. Quindi useremo un meccanismo di "templetizzazione". Creeremo quindi un generico header, un generico footer, un generico menù e li includeremo in ogni pagina.

Il Tiles Plugin permette sia il templating che la componentizzazione. Infatti entrambi i meccanismo sono simili: tu definisci parte di una pagina (una "Tile", una piastrella) che assempli per costruire un'altra parte di una pagina completa. Una parte può prendere parametri, permettendo così contenuti dimanici e può essere vista come un metodo del linguaggio Java. Tiles è un sistema di templating usato per mantenere un aspetto coerente tra tutte le pagine web della nostra Web Application. Esso incrementa la riusabilità del template e riduce la duplicazione del codice.

Un generico layout di un sito web è generalmente definito in un file di configurazione centrale e tale layout può essere esteso in tutte le pagine web della Web Application.

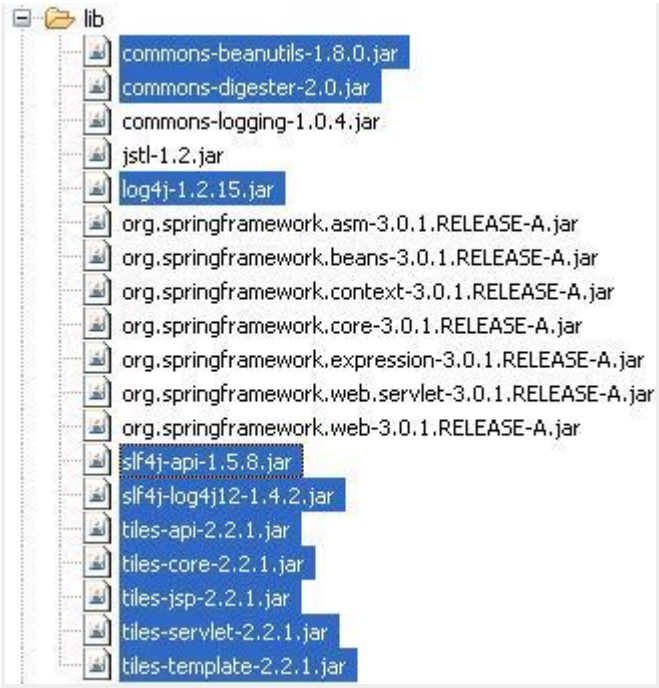
Il nostro Application Layout:

Il nostro obbiettivo è di aggiungere un Header, un Footer ed un Menù all'applicazione implementata nel precedente tutorial. Ecco la schematizzazione di come si presenterà tale layout:



JAR richiesti:

Al fine di aggiungere il supporto a Tiles alla nostra applicazione Spring 3 MVC abbiamo bisogno di aggiungere qualche file jar alle nostre librerie. Quì sotto la lista dei file JAR del nostro esempio.
Aggiungete questi file JAR nella cartella WEB-INF/lib (**NB: Qualora esistessero versioni più recenti di tali file Jar usate pure quelle**).

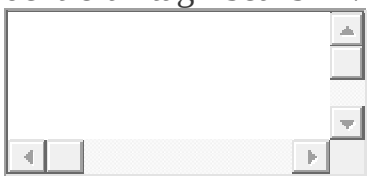


jar tiles framework

Configurazione del framework Tiles all'interno di Sprinv MVC:

Per configurare Tiles bisogna aggiungere un'entry per il bean TilesConfigure nel nostro file spring-servlet.xml

Aprite il file spring-servlet.xml nella cartella WEB-INF ed aggiungete il seguente codice dentro un tag <beans> </beans>

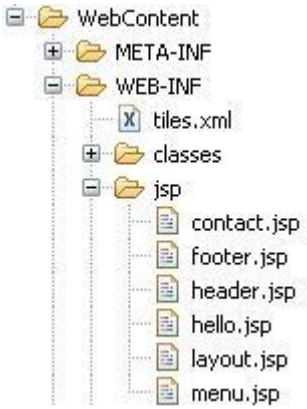


```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE tiles-definitions PUBLIC
3        "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
4        "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
5  <tiles-definitions>
6     <definition name="base.definition" template="/WEB-INF/jsp/layout.jsp">
7        <put-attribute name="title" value="" />
8        <put-attribute name="header" value="/WEB-INF/jsp/header.jsp" />
9        <put-attribute name="menu" value="/WEB-INF/jsp/menu.jsp" />
10       <put-attribute name="body" value="" />
```

```
11      <put-attribute name="footer" value="/WEB-INF/jsp/footer.jsp" />
12    </definition>
13
14    <definition name="contact" extends="base.definition">
15      <put-attribute name="title" value="Contact Manager" />
16      <put-attribute name="body" value="/WEB-INF/jsp/contact.jsp" />
17    </definition>
18
19 </tiles-definitions>
```

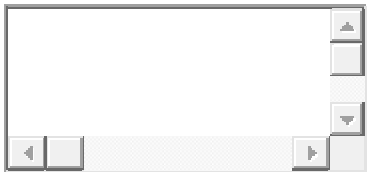
Nel file **tiles.xml** abbiamo definito un template **base.definition**, questo layout contiene degli attributi come **title**, **header**, **menu**, **body**, **footer**. Il layout è poi esteso con nuove definizioni per la pagina Contact. Abbiamo così fatto l'override del layout di default ed abbiamo cambiato il contenuto delle sezioni Title e Body per la pagina Contact.

Creazione delle view: le pagine JSP:



Dobbiamo ora definire il **template generale** per la nostra Web Application in un file JSP chiamato **layout.jsp**. Questo template conterrà i differenti segmenti di una generica pagina web della nostra Web Application (**title**, **header**, **menu**, **body**, **footer**).

File: WebContent/WEB-INF/jsp/layout.jsp



```
1  <%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
2  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
3  "http://www.w3.org/TR/html4/loose.dtd">
4  <html>
5  <head>
```

```
 6 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 7 <title><tiles:insertAttribute name="title" ignore="true" /></title>
 8 </head>
 9 <body>
10 <table border="1" cellpadding="2" cellspacing="2" align="center">
11   <tr>
12     <td height="30" colspan="2"><tiles:insertAttribute name="header" />
13     </td>
14   </tr>
15   <tr>
16     <td height="250"><tiles:insertAttribute name="menu" /></td>
17     <td width="350"><tiles:insertAttribute name="body" /></td>
18   </tr>
19   <tr>
20     <td height="30" colspan="2"><tiles:insertAttribute name="footer" />
21     </td>
22   </tr>
23 </table>
24 </body>
25 </html>
```

Create 3 nuovi file JSP: **header.jsp**, **menu.jsp** e **footer.jsp** e copiateci dentro i seguenti contentuti

*File: **WebContent/WEB-INF/jsp/header.jsp***

```
1 <h1>Header</h1>
```

*File: **WebContent/WEB-INF/jsp/menu.jsp***

```
1 <p>Menu</p>
```

File: ***WebContent/WEB-INF/jsp/footer.jsp***

```
1 <p>Copyright &copy; ViralPatel.net</p>
```

Notate che per quanto riguarda la sezione dei contenuti, questa è stata assegnata alla view contact.jsp direttamente all'interno del file di configurazione tiles.xml

Conclusioni:

Compilate ed eseguite l'applicazione in STS\Eclipse e vedrete come l'header, il menu ed il footervengono applicati.

Abbiamo visto come configurare il framework Tiles all'interno di Web Application sviluppate tramite Spring 3 MVC. Per fare ciò abbiamo usato la classe**org.springframework.web.servlet.view.tiles2.TilesConfigurer** nella definizione dei

nostri bean per definire il file di configurazione di Tiles (passandoglielo come parametro di input)-

Nel prossimo articolo discuteremo circa l'Internazionalizzazione\Localizzazione di una Web Application e di come aggiungere tale supporto all'interno di applicazioni implementate mediante Spring 3 MVC.

5 – Spring MVC Internaziolaizzazione e Localizzazione tutorial con esempio

Posted on **9 novembre 2013** Views: 1711

Nel precedente articolo abbiamo visto come configurare il framework Tiles in una Web Application Spring 3 MVC.

In questo articolo discuteremo a proposito dell'**Internazionalizzazione** (**I18N**) e della**Localizzazione** (**L10N**) in un'applicazione Spring 3 MVC. Dovremmo così aggiunger il supporto I18N seguito dal supporto L10N all'applicazione che abbiamo creato nei precedenti tutorial.
Vi raccomando fortemente di procedere seguendo il precedente articolo e di scaricare il codice sorgente della stessa applicazione.

Cosa sono I18N e L10N ?

In Informatica, l'internazionalizzazione e la localizzazione sono mezzi di adattamento di un programma per diverse lingue e per differenze regionali.
L'internazionalizzazione è il processo di progettazione di un'applicazione in modo tale che questa possa adattarsi a diverse lingue e regioni senza modifiche tecniche.
La localizzazione è il processo di adattamento di un software internazionalizzato per una specificata regione o lingua locale aggiungendo specifici componenti e traducento il testo.

Differenze tra Internazionalizzazione e Localizzazione:

La differenza tra internazionalizzazione e localizzazione è sottile ma fondamentale. L'internazionalizzazione è l'adattamento di prodotti per un potenziale utilizzo al di fuori del mercato o ambiente in cui sono stati progettati i prodotti stessi, mentre la localizzazione è l'aggiunta ai prodotti di caratteristiche speciali che permettano di utilizzare tali prodotti in specifici mercati o ambienti cosiddetti locali. I processi sono evidentemente complementari e devono essere combinati e sinergici per raggiungere l'obiettivo di un prodotto che funzioni (o possa avere lo stesso successo) su un piano globale. Ecco alcuni elementi specifici della localizzazione: Traduzione linguistica; supporto speciale per alcune lingue, come ad esempio le lingue dell'estremo oriente; usi locali; contenuti locali; simboli, metodi di ordinamento degli elenchi; estetica; valori culturali e contesto sociale.
Nello sviluppo software, dopo che un prodotto è stato internazionalizzato, il termine

localizzazione fa riferimento al processo necessario a renderlo pronto per uno specifico mercato.
Per questo motivo ci si può riferire ad un prodotto come “internazionalizzato” se è stato sviluppato per soddisfare i requisiti di una comunità internazionale, ma non ancora pronto per un mercato specifico. La preparazione per uno specifico mercato viene chiamata “localizzazione”.

Obbiettivo:

Il nostro obbiettivo è quello di aggiungere il supporto all’Internazionalizzazione ed alla Localizzazione alla nostra Web Application sviluppata usando Spring MVC nei precedenti tutorial.
Una volta finito la nostra applicazione dovrebbe avere il seguente aspetto:

Aggiungeremo il supporto a 2 lingue alla nostra applicazione: Inglese e Tedesco. La lingua sarà automaticamente scelta in base al local setting del browser dell’utente. Inoltre l’utente sarà in grado di selezionare manualmente la lingua che desidera usare tramite un apposito menù posizionato in alto a destra.

Message Resouces File:

Dobbiamo creare 2 file properties che dovranno contenere tutti i messaggi visualizzati nell'applicazione. Questi file dovranno essere conservati all'interno di una cartella sorgente chiamata "**resources**".
Create una cartella sorgente nel vostro progetto:

Click destro su **Project name** > **New** > **Source Folder** e chiamatela **resources**

Dentro tale cartella creiamo ora 2 file messages_en.properties e messages_de.properties e copiate i seguenti contenuti al loro interno:

File: ***resources/messages_en.properties***

```
1 label.firstname=First Name
2 label.lastname=Last Name
3 label.email=Email
4 label.telephone=Telephone
5 label.addcontact=Add Contact
6 label.title=Contact Manager
7 label.menu=Menu
8 label.footer=&copy; ViralPatel.net
```

File: ***resources/messages_de.properties***

```
1 label.firstname=Vorname
2 label.lastname=Familiename
3 label.email=Email
```

```
4 label.telephone=Telefon

5 label.addcontact=Addieren Kontakt

6 label.title=Kontakt Manager

7 label.menu=Men&#252;

8 label.footer=&copy; ViralPatel.net
```

Configurazione dell'Internazionalizzazione (I18N) e della Localizzazione (L10N) all'interno di Spring MVC:

Ora che abbiamo creato i file message resource properties per la nostra applicazione dobbiamo dichiarare tali file allìinterno del file di configurazione di Spring.

Useremo la classe**org.springframework.context.support.ReloadableResourceBundleMessageSource** per determinare i file message resources.
Notate che dobbiamo anche fornire una funzionalità mediante la quale l'utente può essere in grado di selezionare la lingua manualmente. Tale funzionalità viene implementata dalla classe:**org.springframework.web.servlet.i18n.LocaleChangeInterceptor**
La classe **LocaleChangeInterceptor** intercetterà ogni cambiamento nella località. Tali cambiamenti sono salvati nei cookies per le richieste future perciò bisognerà usare anche la seguente classe per memorizzare i cambiamenti di località all'interno dei cookies:**org.springframework.web.servlet.i18n.CookieLocaleResolver**
Aggiungete il seguente codice nel file **spring-servlet.xml**

*File:**WebContent/WEB-INF/spring-servlet.xml***

```
1 <?xml version="1.0" encoding="UTF-8"?>

2 <beans xmlns="http://www.springframework.org/schema/beans"

3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

4    xmlns:p="http://www.springframework.org/schema/p"

5    xmlns:context="http://www.springframework.org/schema/context"

6    xsi:schemaLocation="http://www.springframework.org/schema/beans

7       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

8       http://www.springframework.org/schema/context

9       http://www.springframework.org/schema/context/spring-context-3.0.xsd">

10
```

```
11  <context:component-scan
12      base-package="net.viralpatel.spring3.controller" />
13
14  <bean id="viewResolver"
15      class="org.springframework.web.servlet.view.UrlBasedViewResolver">
16      <property name="viewClass">
17          <value>
18              org.springframework.web.servlet.view.tiles2.TilesView
19          </value>
20      </property>
21  </bean>
22
23  <!-- CONFIGURAZIONE DEL FRAMEWORK TILES -->
24  <bean id="tilesConfigurer"
25      class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
26      <property name="definitions">
27          <list>
28              <value>/WEB-INF/tiles.xml</value>
29          </list>
30      </property>
31  </bean>
32
33  <!-- INTERNAZIONALIZZAZIONE & LOCALIZZAZIONE -->
34
35  <!-- Classe per determinare i file message resources contenenti i contenuti nelle varie lingue -->
36  <bean id="messageSource"
37      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
38      <property name="basename" value="classpath:messages" />
39      <property name="defaultEncoding" value="UTF-8"/>
40  </bean>
41
42  <!-- Classe che implementa la funzionalità per cui l'utente può cambiare lingua manualmente -->
```

```
43    <bean id="localeChangeInterceptor"
44        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
45        <property name="paramName" value="lang" />
46    </bean>
47
48    <!-- Classe per memorizzare i cambiamenti di località nei cookies -->
49    <bean id="localeResolver"
50        class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
51        <property name="defaultLocale" value="en"/>
52    </bean>
53
54    <bean id="handlerMapping"
55        class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
56        <property name="interceptors">
57            <ref bean="localeChangeInterceptor" />
58        </property>
59    </bean>
60
61 </beans>
```

Notate che in questa configurazione abbiamo definito la proprietò basename come parametro di input del bean messageSource settandola con **classpath:messages**. Tramite tale parametro, Spring identificherà i message resource message_ saranno usati nell'applicazione.

Cambiamo le View – Modifichiamo le pagine JSP:

Ora che abbiamo creato i 2 file message resources e che li abbiamo configurati all'interno di Spring MVC dobbiamo usare questi file all'interno delle pagine JSP.

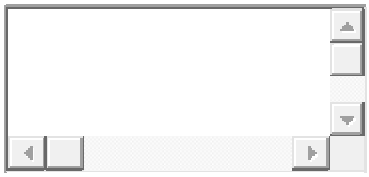
Aprite tutti i file JSP della nostra applicazione dimostrativa ed aggiornatele con i seguenti codici:

*File:**WebContent/WEB-INF/jsp/header.jsp***

```
1 <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
```

```
2
3 <h3><spring:message code="label.title"/></h3>
4
5 <span style="float: right">
6     <a href="?lang=en">en</a>
7     |
8     <a href="?lang=de">de</a>
9 </span>
```

*File:**WebContent/WEB-INF/jsp/menu.jsp***



```
1 <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
2
3 <p><spring:message code="label.menu"/></p>
```

*File:**WebContent/WEB-INF/jsp/footer.jsp***

```
1 <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
2
3 <spring:message code="label.footer"/>
```

*File:**WebContent/WEB-INF/jsp/contact.jsp***

```
 1  <%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
 2  <%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
 3  <html>
 4  <head>
 5      <title>Spring 3 MVC Series - Contact Manager</title>
```

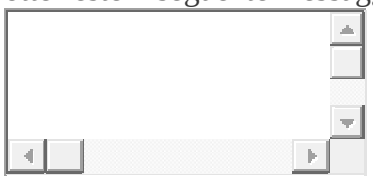
```
</head>
<body>

<form:form method="post" action="addContact.html">

  <table>
  <tr>
      <td><form:label path="firstname"><spring:message code="label.firstname"/></form:label></td>
      <td><form:input path="firstname" /></td>
  </tr>
  <tr>
      <td><form:label path="lastname"><spring:message code="label.lastname"/></form:label></td>
      <td><form:input path="lastname" /></td>
  </tr>
  <tr>
      <td><form:label path="lastname"><spring:message code="label.email"/></form:label></td>
      <td><form:input path="email" /></td>
  </tr>
  <tr>
      <td><form:label path="lastname"><spring:message code="label.telephone"/></form:label></td>
      <td><form:input path="telephone" /></td>
  </tr>
  <tr>
      <td colspan="2">
          <input type="submit" value="<spring:message code="label.addcontact"/>"/>
      </td>
  </tr>
</table>

</form:form>
</body>
</html>
```

Notate nelle precedenti pagine JSP che abbiamo usato il tag **<spring:message>** per visualizzare il messaggio preso dal resource boundle.
Una cosa che dobbiamo notare è che in header.jsp abbiamo specificato 2 link per scegliere la lingua. I link settano un parametro di richiesta **?lang=** quando l'utente clicka su uno di questi link. Notate che Spring identifica questo parametro di richiesta usando l'intercettatore LocaleChangeInterceptor e cambia la località di conseguenza.
Notate anche che quando abbiamo configurato il bean **LocaleChangeInterceptor** all'interno del file di configurazione spring-servlet.xml, abbiamo specificato la property (il parametro di input di tale bean) "paramName" con valore "lang".

<property name="paramName" value="lang" />

Così il framework Spring cercherà un parametro chiamato "lang" nella richiesta

ATTENZIONE: Se eseguiste l'applicazione questa andrebbe in errore e nel vosto stack trace otterreste il seguente messaggio come ultimo messaggio di errore nello stack:



```
javax.servlet.jsp.JspTagException: No message found under code 'label.title' for locale 'en'.
org.springframework.web.servlet.tags.MessageTag.doStartTagInternal(MessageTag.java:184)
org.springframework.web.servlet.tags.RequestContextAwareTag.doStartTag(RequestContextAwareTag.java:79)
org.apache.jsp.WEB_002dINF.jsp.header_jsp._jspx_meth_spring_005fmessage_005f0(header_jsp.java:103)
org.apache.jsp.WEB_002dINF.jsp.header_jsp._jspService(header_jsp.java:69)
org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:432)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:390)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:334) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.jasper.runtime.JspRuntimeLibrary.include(JspRuntimeLibrary.java:954)

org.apache.jasper.runtime.PageContextImpl.doInclude(PageContextImpl.java:684)


org.apache.jasper.runtime.PageContextImpl.include(PageContextImpl.java:678)
org.apache.tiles.jsp.context.JspTilesRequestContext.include(JspTilesRequestContext.java:103)
org.apache.tiles.jsp.context.JspTilesRequestContext.dispatch(JspTilesRequestContext.java:96)
org.apache.tiles.renderer.impl.TemplateAttributeRenderer.write(TemplateAttributeRenderer.java:44)
org.apache.tiles.renderer.impl.AbstractBaseAttributeRenderer.render(AbstractBaseAttributeRenderer.java:106)
org.apache.tiles.renderer.impl.ChainedDelegateAttributeRenderer.write(ChainedDelegateAttributeRenderer.java:76)
org.apache.tiles.renderer.impl.AbstractBaseAttributeRenderer.render(AbstractBaseAttributeRenderer.java:106)
org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:670)
org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:336)
org.apache.tiles.template.InsertAttributeModel.renderAttribute(InsertAttributeModel.java:210)
org.apache.tiles.template.InsertAttributeModel.end(InsertAttributeModel.java:126)
org.apache.tiles.jsp.taglib.InsertAttributeTag.doTag(InsertAttributeTag.java:311)
org.apache.jsp.WEB_002dINF.jsp.layout_jsp._jspx_meth_tiles_005finsertAttribute_005f1(layout_jsp.java:142)
org.apache.jsp.WEB_002dINF.jsp.layout_jsp._jspService(layout_jsp.java:77)
org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:432)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:390)
```

org.apache.jasper.servlet.JspServlet.service(JspServlet.java:334) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.tiles.servlet.context.ServletTilesRequestContext.forward(ServletTilesRequestContext.java:241)
org.apache.tiles.servlet.context.ServletTilesRequestContext.dispatch(ServletTilesRequestContext.java:222)
org.apache.tiles.renderer.impl.TemplateAttributeRenderer.write(TemplateAttributeRenderer.java:44)
org.apache.tiles.renderer.impl.AbstractBaseAttributeRenderer.render(AbstractBaseAttributeRenderer.java:106)

org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:670)

org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:690)
org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:644)
org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:627)
org.apache.tiles.impl.BasicTilesContainer.render(BasicTilesContainer.java:321)
org.springframework.web.servlet.view.tiles2.TilesView.renderMergedOutputModel(TilesView.java:124)
org.springframework.web.servlet.view.AbstractView.render(AbstractView.java:262)
org.springframework.web.servlet.DispatcherServlet.render(DispatcherServlet.java:1180)
org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:950)
org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:852)
org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:882)
org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:778)
javax.servlet.http.HttpServlet.service(HttpServlet.java:621) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.jasper.runtime.PageContextImpl.doForward(PageContextImpl.java:746)
org.apache.jasper.runtime.PageContextImpl.forward(PageContextImpl.java:716)
org.apache.jsp.index_jsp._jspService(index_jsp.java:63) org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:432)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:390)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:334) javax.servlet.http.HttpServlet.service(HttpServlet.java:722)

Analizzando tale messaggio di errore si può intuire che l'applicazione non riesce a vedere i file di message properties delle lingue.
In realtà il messaggio di errore ci stà dicendo solo che non riesce a vedere il messaggio associato a **label.title** per la lingua locale inglese ma il fatto che tale etichetta è presente nel file messages_en.properties e che vi è associato il corretto messaggio "Contact Manager" associato al fatto che molto probabilmente anandosi a schiantare proprio sulla **label.title** (che rappresenta il titolo della pagina web) significa che si stè andando a schiantare proprio sulla prima label che conterrà il primo messaggio da visualizzare sulla pagina web che si stà tentando di renderizzare (non si stà schiantando su un'altra label, dopo che è riuscito ad accedere ad altre in precedenza ma proprio sulla prima) dovrebbe far pensare al fatto che molto probabilmente l'applicazione non è in grado di accedere a questi file…

Per capire come risolvere questo problema dobbiamo capire da cosa dipende, quindi andiamo a vedere come stiamo caricando questi file all'interno di Spring quindi andiamo a vedere il file di configurazione di Spring 3 MVC: **spring.servlet.xml** e troviamo che avevamo configurato l'accesso a tali file mediante la dichiarazione del seguente bean:

```
1 <!-- Classe per determinare i file message resources contenenti i contenuti nelle varie lingue -->
2 <bean id="messageSource"
3      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
4           <property name="basename" value="classpath:messages" />
5           <property name="defaultEncoding" value="UTF-8"/>
6 </bean>
```

Come si può notare dalla definizione del bean in questione, la classe **ReloadableResourceBundleMessageSource** riceve 2 parametri di input di cui uno rappresenta proprio il folder che contiene proprio i file di properties relativi alle varie lingue. Tale percorso viene definito mediante il valore **classpath:messages**, in pratica significa che stiamo tentando di accedere ai file **messages_** di properties nel folder **resources** attraverso il classpath.

Quindi per risolvere il problema dobbiamo inserire il folder resources nel classpath dell'applicazione. Per fare questa cosa dobbiamo semplicemente fare:
Click destro sul nome del nostro progetto —> Properties —> Java Build Path —> rimanere nel tab Source (il primo, aperto di default) —> click su add Folders —> (apparirà una lista con tutte le cartelle del progetto) spuntare resources e premere ok.
Abbiamo così inserito il folder resources all'interno del class path ed abbiamo così reso accessibili i file di properties relativi alle lingue.

Conclusioni:

Questo è tutto. Abbiamo visto come aggiungere il supporto all'Internazionalizzazione ed alla Localizzazione alla nostra applicazione dimostrativa Spring 3 MVC. Eseguendo l'appl dovreste ottenere il seguente risultato:

Spring MVC Form

Abbiamo visto quindi come aggiungere il supporto all'Internazionalizzazione **I18N** ed alla Localizzazione **L10N** in un'applicazione sviluppata mediante Spring 3 MVV. Abbiamo usato la classe interceptor **LocaleChangeInterceptor** per intercettare i cambiamenti nella località e la classe **ReloadableResourceBundleMessageSource** per aggiungere le properits message resources disponibili nell'applicazione.
Nella prossimo articolo discuteremo i Themes in Spring 3 MVC e come implementarli.