

Java Spring

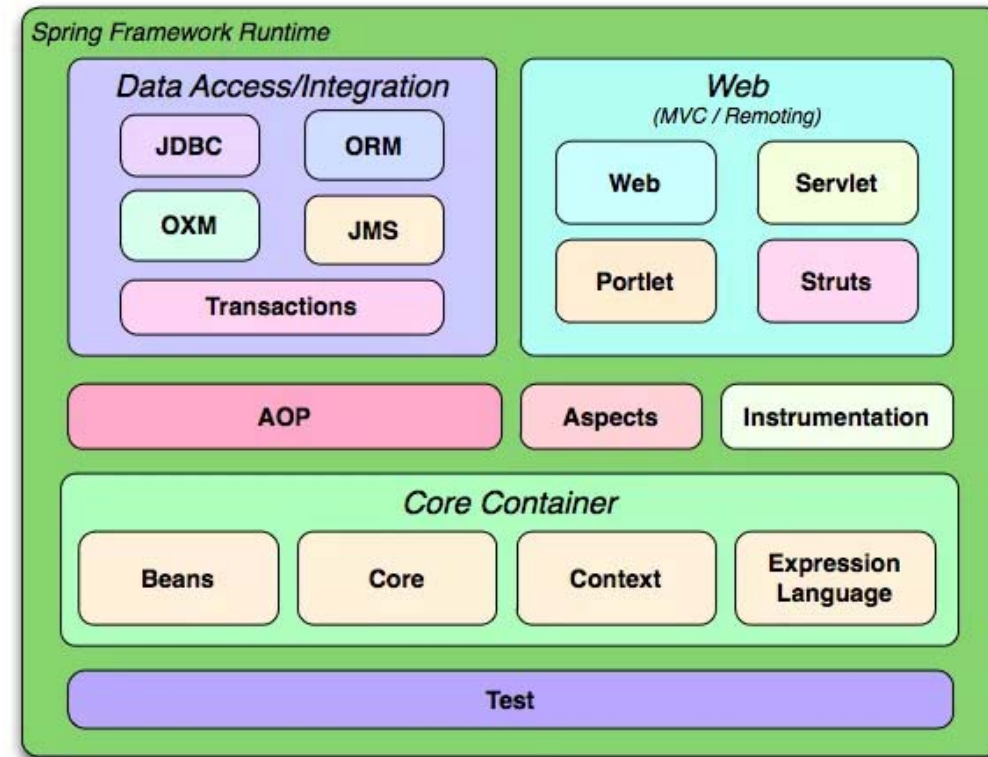


Spring

- Spring è un framework “**leggero**” e grazie alla sua architettura estremamente modulare é possibile utilizzarlo nella sua interezza o solo in parte. L’adozione di Spring in un progetto è molto semplice, può avvenire in maniera incrementale e non ne sconvolge l’architettura esistente. Questa sua peculiarità ne permette anche una facile integrazione con altri framework esistenti, come ad esempio Struts.
- Spring è un lightweight container e si propone come alternativa/complemento a J2EE. A differenza di quest’ultimo, Spring propone un modello più semplice e leggero (soprattutto rispetto ad EJB) per lo sviluppo di entità di business. Tale semplicità è rafforzata dall’utilizzo di tecnologie come l’Inversion of Control e l’Aspect Oriented che danno maggiore spessore al framework e favoriscono la focalizzazione dello sviluppatore sulle logica applicativa essenziale.
- A differenza di molti framework che si concentrano maggiormente nel fornire soluzioni a problemi specifici, Spring mette a disposizione una serie completa di strumenti atti a gestire l’intera complessità di un progetto software.
- Fornisce un approccio semplificato alla maggior parte dei problemi ricorrenti nello sviluppo software (accesso al database, gestione delle dipendenze, testing, etc.).
- Spring è un framework nato con la concezione che il codice di qualità debba essere facilmente testato. Questa filosofia fa sì che, con Spring, sia molto facile testare il codice e, grazie a questa peculiarità, il framework si è ritagliato uno spazio importante in quegli ambiti dove il testing è considerato parte fondamentale del progetto software.
- Intro: <https://www.youtube.com/watch?v=rMLP-NEPgnM>

Spring

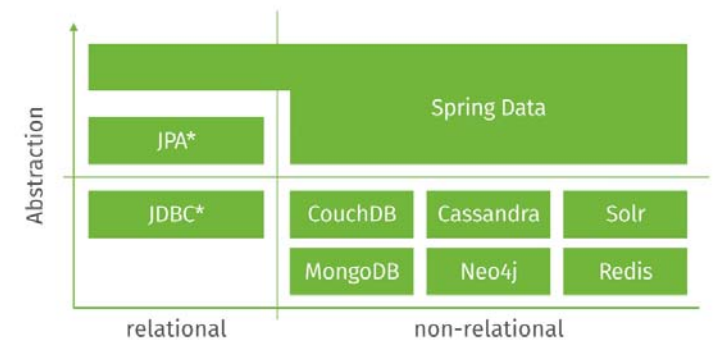
- architettura in 20 moduli (100 packages)
- I moduli di Core e Beans sono responsabili delle funzionalità di **Inversion Of Control (IoC)** e **Dependency Injection** ed hanno come compito principale la creazione, gestione e manipolazione di oggetti di qualsiasi natura che, in Spring, vengono detti beans.
- Il modulo **context** estende i servizi basilari del **Core** aggiungendo le funzionalità tipiche di un moderno framework. Tra queste troviamo JNDI, EJB, JMX, internazionalizzazione(I18N) e supporto agli eventi.
- Infine il modulo Expression Language fornisce un potente linguaggio per interrogare e modificare oggetti a runtime.



Data access, integration

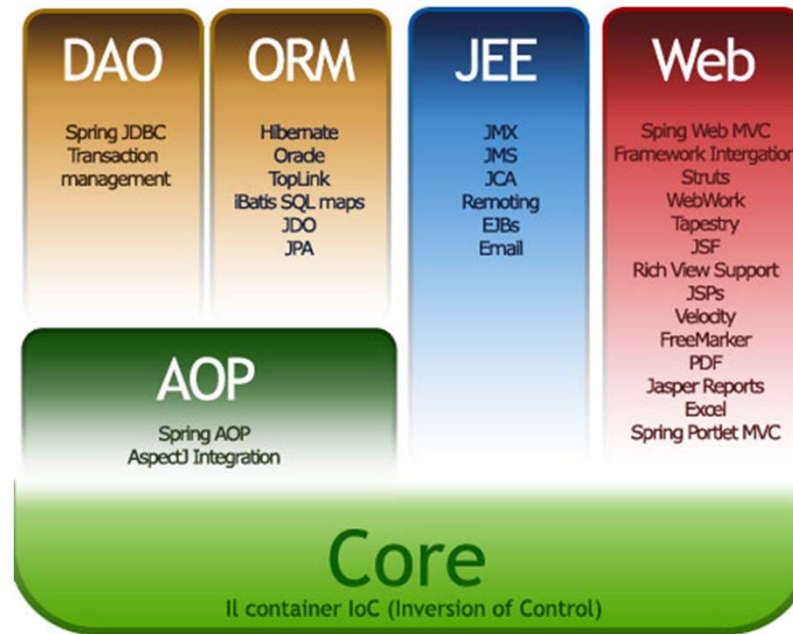
- Fornisce un **livello di astrazione per l'accesso ai dati** mediante tecnologie eterogenee tra loro come ad esempio JDBC, Hibernate o JDO. Questo modulo tende a nascondere la complessità delle API di accesso ai dati, semplificando ed uniformando quelle che sono le problematiche legate alla gestione delle connessioni, delle transazioni e delle eccezioni.
- Notevole attenzione è stata data all'integrazione del framework con i principali ORM in circolazione compresi JPA, JDO, Hibernate, e iBatis. Oltre a questo il Data Access si occupa di fornire supporto per il Java Message Service e per svariate implementazioni di Object/XML Mapper come JAXB, Castor e XMLBean.

Spring Data



AOP

- Aggiunge al framework la funzionalità della **programmazione Aspect Oriented**. AOP offre un nuovo modo di programmare che come vedremo in seguito porterà notevoli vantaggi in tutte quelle operazioni che sono trasversali tra più oggetti. In Spring l'utilizzo di AOP offre il meglio di sé nella **gestione delle transazioni**, permettendo di evitare l'utilizzo degli EJB per tale scopo..

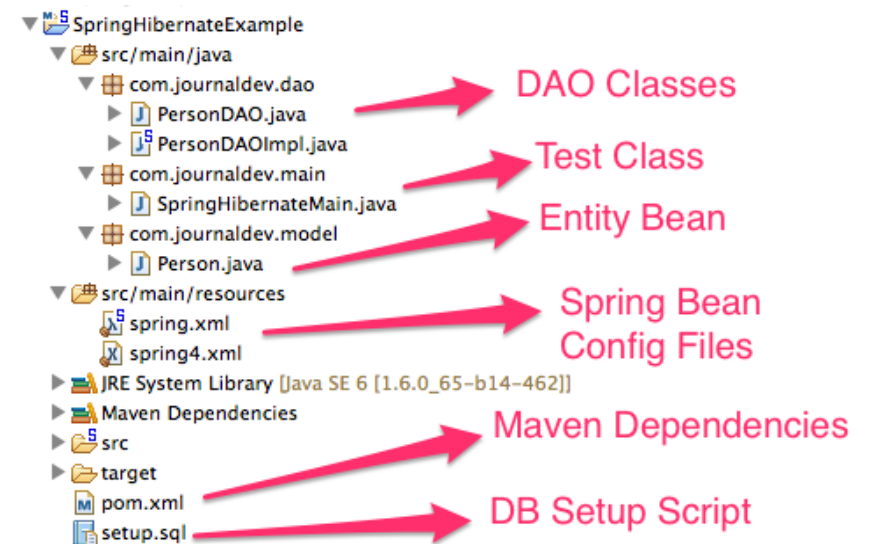


Web

- Come si intuisce dal nome, questo livello è responsabile delle caratteristiche Web del framework. Oltre alle funzionalità basilari, per la creazione di applicazioni Web, questo livello mette a disposizione un'implementazione del pattern **MVC** realizzando lo Spring MVC Framework (moduli Web-Servlet e Web-Portlet).
- Il framework MVC è ampiamente configurabile attraverso delle Strategy, può operare sia in contesti servlet che portlet e può essere utilizzato con numerose tecnologie di view comprese JSP e Velocity.

Test

- Un'altra delle caratteristiche per il quale Spring si contraddistingue è il supporto al testing. Questo livello mette a disposizione un ambiente molto potente per il test delle componenti Spring, grazie anche alla sua integrazione con **JUnit** e TestNG e alla presenza di Mock objects per il testing del codice in isolamento.



Ambiente

- In particolare i componenti software utilizzati sono:
- Java SDK 1,8
- Eclipse IDE
- Spring Framework 5
- Apache Derby
- Apache Tomcat 10
- Spring <https://spring.io/projects/>

Installazione Spring

- Scaricare ed estrarre la cartella spring oppure con **Maven** o con Gradle
- In particolare l'archivio include tra le altre
 - la directory **dist** dove è presente il framework vero e proprio e
 - la directory **lib** dove sono contenute le dipendenze.
- Per utilizzare Spring, basterà semplicemente linkare nei propri progetti il file **distspring.jar** e le opportune dipendenze necessarie di volta in volta.
- Per Eclipse abbiamo un plugin dal marketplace:
<http://marketplace.eclipse.org/content/spring-ide>
- STS spring tools(IDE su base eclipse): <https://spring.io/tools/sts/all>

Installazione Apache Derby

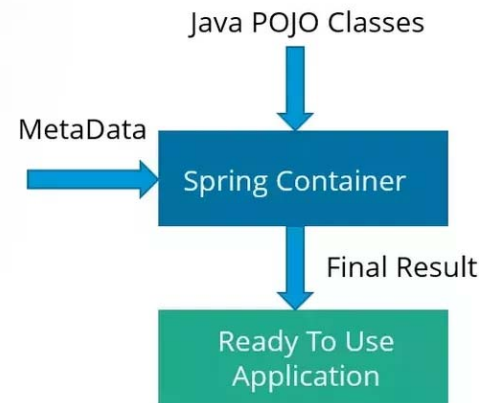
- Basandosi su JDBC per l'accesso ai dati, Spring può essere utilizzato con la maggior parte dei database presenti sul mercato e in particolare con tutti quelli che forniscono un driver per JDBC.
- http://db.apache.org/derby/derby_downloads.html
- L'installazione è molto semplice, una volta scaricato l'archivio zip contenente l'ultima release del database dal sito del progetto, basterà scompattarlo in una directory a scelta (ad esempio c:db-derby). Dopodiché sarà sufficiente creare la variabile d'ambiente DERBY_HOME impostando come valore il percorso della directory di Derby e aggiungere alla variabile d'ambiente PATH il valore %DERBY_HOME%bin.

Installazione Apache Tomcat

- Per poter testare la parte della guida inerente Spring MVC è necessario installare un web container. Grazie alla sua semplicità Apache Tomcat è l'ideale per questo scopo, anche se per applicazioni più complesse si suggerisce l'utilizzo di un application server come ad esempio JBoss AP.
- Per installare Apache Tomcat è sufficiente eseguire il file d'installazione reperibile sul sito ufficiale. <http://tomcat.apache.org/>

IoC

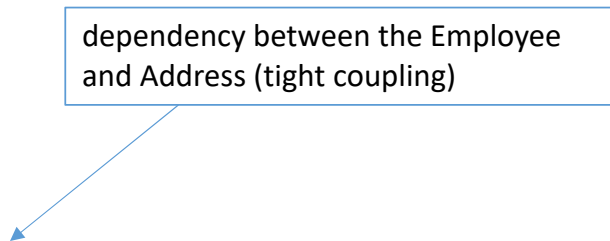
- **L’Inversion of Control** è un principio architetturale nato alla fine degli anni ottanta, basato sul concetto di invertire il controllo del flusso di sistema (Control Flow) rispetto alla programmazione tradizionale. Questo principio è molto utilizzato nei framework e ne rappresenta una delle caratteristiche basilari che li distingue dalle API.
- Nella programmazione tradizionale la logica di tale flusso è definita esplicitamente dallo sviluppatore, che si occupa tra le altre cose di tutte le operazioni di **creazione**, **inizializzazione** ed **invocazione** dei metodi degli oggetti.
- IoC invece inverte il control flow facendo in modo che non sia più lo sviluppatore a doversi preoccupare di questi aspetti, ma il framework, che reagendo a qualche “stimolo” se ne occuperà per suo conto. Questo principio è anche conosciuto come Hollywood Principle (“Non chiamarci, ti chiameremo noi”).



Spring ioc +DI: esempio

Class **Employee**{
Address address;
Employee(){
address=new Address();
} }

dependency between the Employee
and Address (tight coupling)



```
class Employee{  
Address address;  
Employee(Address address){  
this.address=address;  
} }
```

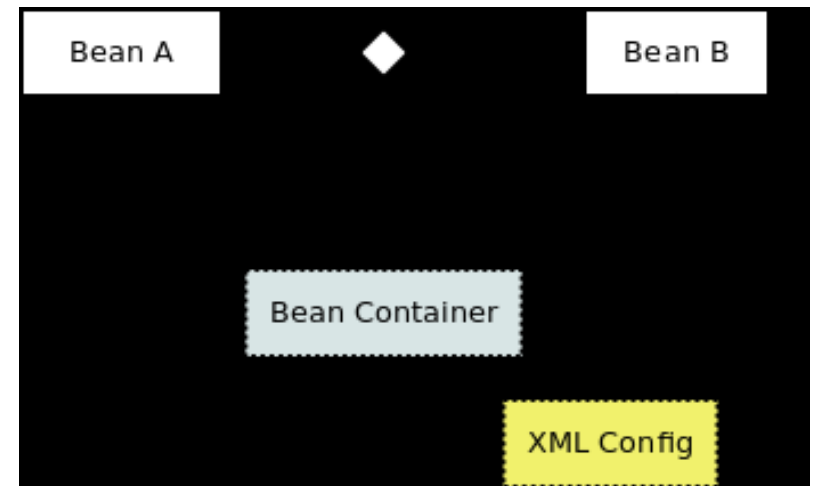
In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Advantage:

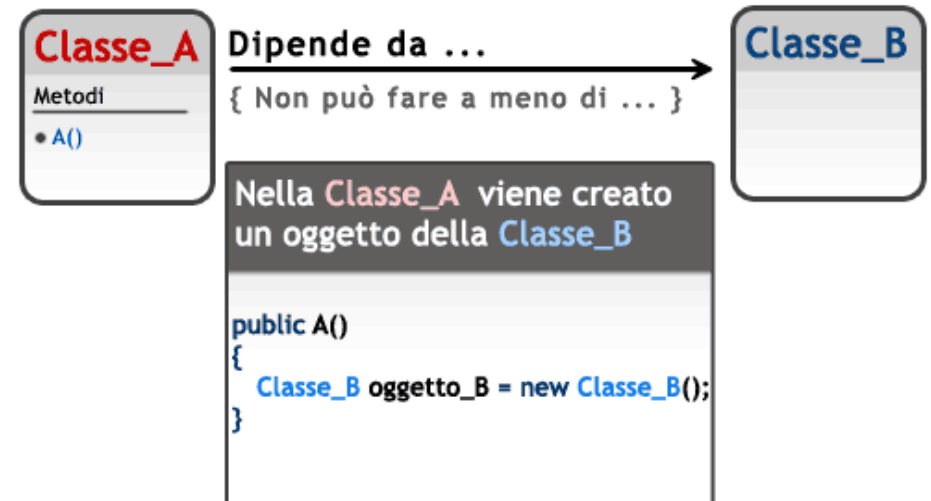
- makes the code loosely coupled so easy to maintain
- makes the code easy to test

Dependency Injection

- **Dependency Injection (DI)** per riferirsi ad una specifica implementazione dello **IoC** rivolta ad invertire il processo di risoluzione delle dipendenze, facendo in modo che queste vengano iniettate dall'esterno. Banalmente, nel caso della programmazione Object Oriented, una classe A si dice dipendente dalla classe B se ne usa in qualche punto i servizi offerti.



IoC e DI



- Perché questo tipo di collaborazione abbia luogo la classe A ha diverse alternative:
 - istanziare e inizializzare (attraverso costruttore o metodi setter) la classe B,
 - ottenere un'istanza di B attraverso una factory oppure effettuare un lookup attraverso un servizio di naming (es JNDI).
 - Ognuno di questi casi implica che nel codice di A sia "sculpita" la logica di risoluzione della dipendenza verso B.
- Per chiarire introduciamo un esempio. Si vuole creare un semplice generatore di report in grado di generare output in formato testuale.

IoC e Dependency Inj

```
public class TxtReport {  
    public void generate(String data) {  
        System.out.println("genera txt report");  
    }  
}  
  
public class ReportGenerator {  
    TxtReport report = null;  
    public Report generate(String data) {  
        report = new TxtReport();    // risoluzione della dipendenza  
        report.generate(data);  
        return report;  
    }  
}
```

La classe ReportGenerator ha una dipendenza verso TxtReport e questa è risolta nel corpo del metodo generate().

Questo modo di operare, oltre a vincolare la creazione della dipendenza nel codice limitandone il riuso (cosa succede se in futuro vogliamo generare report in formato HTML?), tende a generare un forte accoppiamento tra le classi.
(**Coupling**)

Il problema risiede nel fatto che senza l'ausilio di un apposito sistema la risoluzione delle dipendenze è ad esclusivo appannaggio delle classi stesse, che dovranno preoccuparsi di creare gli oggetti da cui dipendono o di ottenerne delle reference attraverso operazioni di lookup.

IoC e Dependency Injection

- L'idea alla base della Dependency Injection è quella di avere un componente esterno (assembler) che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l'utilizzo dell'injection. In particolare esistono tre forme di injection:
 - **Constructor** Injection, dove la dipendenza viene iniettata tramite l'argomento del costruttore
 - **Setter** Injection, dove la dipendenza viene iniettata attraverso un metodo "set"
 - **Interface** Injection che si basa sul mapping tra interfaccia e relativa implementazione (non utilizzato in Spring)
- L'iniezione di dipendenza può essere realizzata in molteplici modi, tra cui il più semplice consiste nell'utilizzo di una **factory**. Riprendiamo ora l'esempio del generatore di report e cerchiamo di capire come realizzare una semplice iniezione della dipendenza.

IoC e Dependency Injection

Creiamo un'interfaccia Report che definisce le operazioni di base che un report deve avere e facciamola implementare alla classe TxtReport:

```
public interface Report {                                // Interfaccia Report
    public void generate(String data);
    public void saveToFile();
}

public class TxtReport implements Report {              // Classe TxtReport
    String path;
    public TxtReport(String path) { this.path = path; }
    public void generate(String data) {
        System.out.println("genera txt report");
    }
    public void saveToFile() {
        System.out.println("File salvato!");
    }
}
```

IoC e Dependency Injection

Modifichiamo il ReportGenerator in modo che sia in grado di utilizzare oggetti di tipo Report e aggiungiamo il relativo metodo setter:

```
public class ReportGenerator {  
    Report report;  
    public Report generate(String data) {  
        // report = new TxtReport();  
        report.generate(data);  
        return report;  
    }  
    public void setReport (Report report) {  
        this.report = report;  
    }  
}
```

IoC e Dependency Injection

Creiamo infine una classe factory che avrà il compito di istanziare oggetti di tipo ReportGenerator e di risolverne le dipendenze:

```
public class ReportGeneratorFactory {  
    public static ReportGenerator createTxtReportGenerator() {  
        ReportGenerator rg = new ReportGenerator();  
        rg.setReport(new TxtReport());    // risoluzione della dipendenza  
        return rg;  
    }  
}
```

IoC e Dependency Injection

Il client della nostra applicazione si presenterà in questo modo:

```
public class ReportClient {  
    public static void main(String[] args) {  
        String data = null;           // reperimento dati  
        ReportGenerator gen = ReportGeneratorFactory.createTxtReportGenerator()  
        gen.generate(data).saveToFile();  
    }  
}
```

La soluzione proposta ci ha permesso di disaccoppiare le classi ReportGenerator e TxtGenerator attraverso l'utilizzo della classe ReportGeneratorFactory. In particolare la factory, dopo aver creato l'oggetto ReportGenerator, vi ha iniettato attraverso il relativo metodo setter (**setter injection**) un oggetto di tipo Report.

Esempio00.zip

IoC e Dependency Injection

Pur essendo efficace, questo tipo “manuale” di DI continua ancora a non risolvere il problema di avere scolpito nel codice la creazione del report. Il problema è infatti stato semplicemente trasferito nella **classe factory**, che dovrà essere ogni volta modificata se si vorrà cambiare il tipo di report da utilizzare.

Un modo migliore per poter lavorare con la Dependency Injection è quello di utilizzare come assembler uno **IoC Container** in grado di compiere operazioni di injection.

Per definizione un container è un componente esterno che si prende carico di una serie di compiti esonerando così lo sviluppatore dal preoccuparsene. Uno IoC Container non è altro che un container specializzato nella dependency injection, che basandosi su apposite configurazioni definite dall'utente (generalmente attraverso l'utilizzo di un file xml) è in grado di compiere opportune operazioni di injection.

IoC Container

Considerato il cuore di Spring, lo **IoC Container** fornisce un contesto altamente configurabile per la creazione e risoluzione delle dipendenze di componenti che qui vengono chiamati **bean** (da non confondere con i JavaBean).

in Spring un bean non deve aderire a nessun tipo di contratto e può essere rappresentato da una qualunque classe Java.

Tecnicamente parlando, lo IoC Container è realizzato da due interfacce:

- **BeanFactory**, che definisce le funzionalità di base per la gestione dei bean
- **ApplicationContext**, che estende queste funzionalità basilari aggiungendone altre tipicamente enterprise come ad esempio la gestione degli eventi, l'internazionalizzazione e l'integrazione con AOP

IoC Container

Di seguito analizzeremo le funzionalità di base fornite dallo IoC Container fornite da BeanFactory, tenendo presente che tutto ciò che verrà detto è valido anche per ApplicationContext.

L'interfaccia BeanFactory rappresenta la forma più semplice di IoC Container in Spring e ha il compito di:

- creare i bean necessari all'applicazione
- inizializzare le loro dipendenze attraverso l'utilizzo dell'injection
- gestirne l'intero ciclo di vita

Per svolgere questi compiti, il container si appoggia a configurazioni impostate dall'utente che, riflettendo lo scenario applicativo, specificano i bean che dovranno essere gestiti dal container, le dipendenze che intercorrono tra questi oltre alle varie configurazioni specifiche.

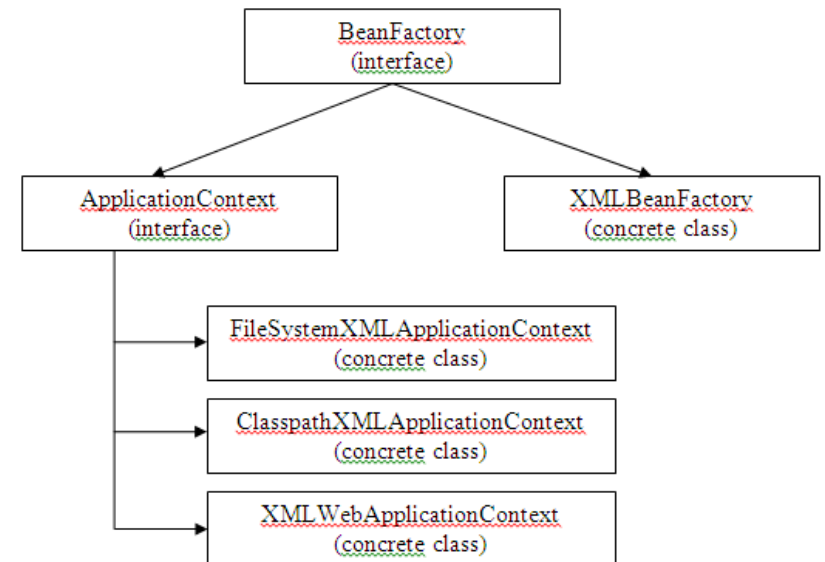
XmlBeanFactory

In Spring esistono diverse implementazioni di BeanFactory, la più comune delle quali è senza dubbio la **XmlBeanFactory** che permette di utilizzare uno o più file XML per descrivere la configurazione da utilizzare. I file di configurazione della XmlBeanFactory hanno la seguente forma:

```
<?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="..." class="...">
    <!-- eventuali risoluzioni di dipendenze e proprietà -->
  </bean>

  <bean id="..." class="...">
    <!-- eventuali dipendenze e proprietà -->
  </bean>

  <!-- altri bean applicativi -->
```



XmlBeanFactory

Dopo l'intestazione del file XML, racchiuse tra i tag <beans>, troviamo le definizioni dei bean e, per ognuno di questi, le eventuali proprietà che ne descriveranno la struttura e il comportamento:

- **l'Id** associato al bean per essere richiamato tramite invocazioni al container
- **Il nome della classe** che implementa il bean in caso questa venga istanziata direttamente o della relativa factory che si occuperà della sua creazione
- Risoluzione di dipendenze attraverso metodi setter o costruttori appositi
- Proprietà comportamentali che definiscono come il bean deve essere trattato dal container (scope, ciclo di vita, etc.)

XmlBeanFactory

IoC Container di Spring. I nostri **bean** sono rappresentati dalle classi **TxtReport** e **ReportGenerator** visti in precedenza:

```
public class TxtReport implements Report {  
    String path;  
    public TxtReport(String path) {  
        this.path = path;  
    }  
    public void generate(String data) {  
        System.out.println("genera txt report");  
    }  
    public void saveToFile() {  
        System.out.println("File salvato");  
    }  
}
```

```
public class ReportGenerator {  
    Report report;  
    public Report generate(String data) {  
        // report = new TxtReport();  
        report.generate(data);  
        return report;  
    }  
    public void setReport (Report report) {  
        this.report = report;  
    }  
}
```

XmlBeanFactory

- Creiamo un file bean.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-3.0.xsd">
```

```
<!-- Indica che il bean 'report' è implementato da TxtReport -->
```

```
<bean id="report" class="it.spring.report.TxtReport">
```

```
<!-- imposta il parametro per il costruttore-->
```

```
<constructor-arg value="/report" />
```

```
</bean>
```

```
<!-- associa il bean 'reportGenerator' al nostro ReportGenerator -->
```

```
<bean id="reportGenerator" class="it.spring.report.ReportGenerator">
```

```
<!-- indica al setter 'report' del reportGenerator di riferirsi ad oggetti -->
```

```
<!-- istanziati con il bean 'report' (quindi, in questo caso, TxtReport)-->
```

```
<property name="report" ref="report" />
```

```
</bean>
```

```
</beans>
```

XmlBeanFactory

Analizzando il file di configurazione possiamo notare che è stato chiesto al container di gestire due classi:

- `it.spring.report.ReportGenerator` (con id `reportGenerator`)
- `it.spring.report.TxtReport` (con id `report`).

sono state anche inserite le direttive per indicare al container come risolvere le dipendenze di queste classi:

- Al bean `report` è stata iniettata, tramite una injection di tipo constructor, una stringa indicante il percorso in cui salvare il report
- A `ReportGenerator` è stato iniettato il bean `report`, con una injection di tipo setter.
- Infine istanziamo il container e vediamo come interagire con esso per utilizzare i bean configurati:

```
public class ReportClient {  
    public static void main(String[] args) {  
        String data = null;           // ... reperimento dati  
        BeanFactory ctx = new XmlBeanFactory(new ClassPathResource("beans.xml"));  
        ReportGenerator gen = (ReportGenerator) ctx.getBean("reportGenerator");  
        gen.generate(data);  
    }  
}
```

XmlBeanFactory

Una volta istanziato il container è possibile utilizzarlo alla stregua di una factory (da qui il nome BeanFactory) reperendo i bean attraverso il metodo `getBean (nomebean)`.

l'esempio completo (progetto Eclipse) [Esempio_01.zip](#)

Come si può ben capire questo modo di operare ci offre enormi vantaggi in termini di riuso dei bean. Se in seguito fosse necessario di un nuovo report che produca file in formato HTML basterà semplicemente scrivere una classe che aderisca all'interfaccia `Report` e modificare il bean corrispondente nel file di configurazione.

Accesso ai dati

- Spring supporta:
 - JDBC
 - Java Persistence API (JPA)
 - Java Data Objects (JDO)
 - Hibernate
 - Common Client Interface (CCI)
 - iBATIS SQL Maps
 - Oracle TopLink

Accesso ai dati

- Creiamo il nostro modello da persistere: è composto da una semplice classe Book contenente le proprietà necessarie per rappresentare un libro e dai relativi metodi accessori.

```
public class Book {  
    String isbn;  
    String author;  
    String title;  
    public String getIsbn() { return isbn; }  
    public void setIsbn(String isbn) { this.isbn = isbn; }  
    public String getAuthor() { return author; }  
    public void setAuthor(String author) { this.author = author; }  
    public String getTitle() { return title; }  
    public void setTitle(String title) { this.title = title; }  
}
```


Accesso ai dati

La persistenza del modello sarà effettuata nella tabella books del database library; di seguito i dettagli di connessione e creazione della tabella.

```
CONNECT 'jdbc:derby://localhost:1527/library;create=true';
```

```
CREATE TABLE BOOKS ( ISBN VARCHAR(13) NOT NULL,  
    AUTHOR VARCHAR(20),  
    TITLE VARCHAR(20),  
    PRIMARY KEY (ISBN));
```

Accesso ai dati

Una possibile scelta è Apache Derby, un database open source estremamente leggero e di facile configurazione.

Per avviare il server basterà semplicemente eseguire il file **startServerNetwork.bat** presente nella sottodirectory bin del path dove è stato installato Apache.

A questo punto sarà possibile aprire la console(Apache Derby Console) applicativa attraverso il comando ij, da dove lanciare i comandi mostrati in figura

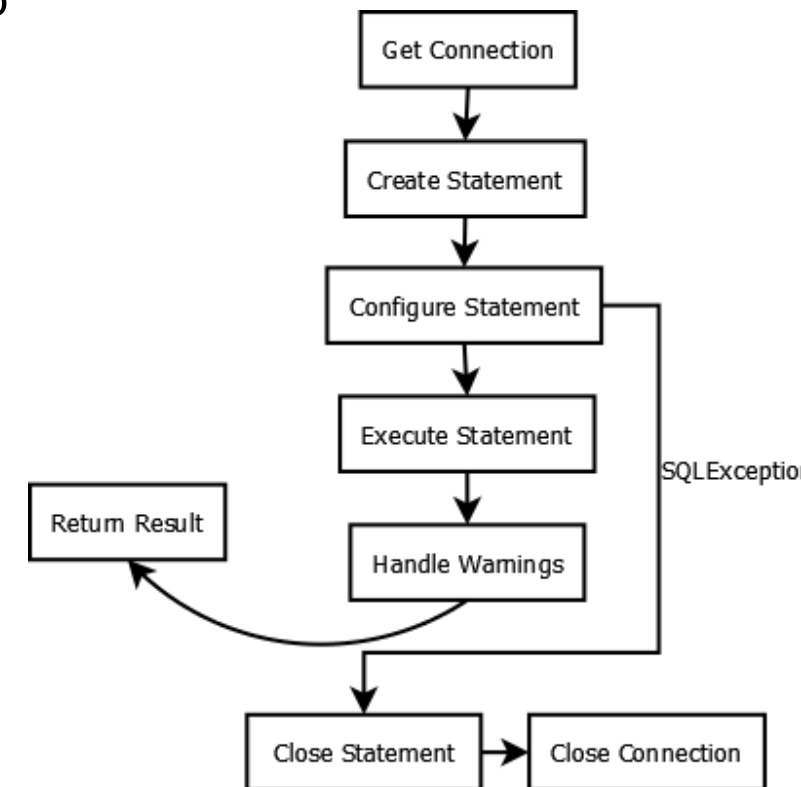
```
C:\>ij
Versione ij 10.5
ij> CONNECT 'jdbc:derby://localhost:1527/library;create=true';
ij> CREATE TABLE BOOKS ( ISBN VARCHAR(13) NOT NULL, AUTHOR VARCHAR(20), TITLE VARCHAR(20), PRIMARY KEY (ISBN));
0 righe inserite/aggiornate/eliminate
ij>
```

Spring JDBC

La forma primaria di accesso ai dati in Java è senza dubbio **JDBC**, un'API creata per rendere platform-independent l'interazione con i RDBMS. Il suo utilizzo offre notevoli vantaggi in termini di performances e facilità di utilizzo, richiedendo per contro allo sviluppatore di farsi carico dell'intera gestione dei processi di accesso ai dati, ovvero:

- Richiesta della connessione al **datasource**
- Creazione del **PreparedStatement** e valorizzazione dei parametri
- Esecuzione del PreparedStatement
- Estrazione dei risultati dal **ResultSet** (in caso di interrogazioni)
- Gestione delle **eccezioni**
- Chiusura della **connessione**

Se per applicazioni di piccole dimensioni queste “complicazioni” possono essere accettabili, sicuramente provocano dei grossi disagi in ambito enterprise, dove nel corso degli anni sono sorti appositi framework che, astruendo JDBC, tendono a semplificare questi aspetti. In questo ambito si colloca Spring JDBC.



Spring Jdbc

- Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.
- The problems of JDBC API are as follows:
 - We need to write a **lot of code** before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
 - We need to perform **exception handling** code on the database logic.
 - We need to handle **transaction**.
 - Repetition of all these codes from one to another database logic is a time consuming task.
 - Advantage of Spring JdbcTemplate
 - Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

Spring Jdbc Approaches

- Spring framework provides following approaches for JDBC database access:
 - **JdbcTemplate**
 - NamedParameterJdbcTemplate
 - SimpleJdbcTemplate
 - SimpleJdbcInsert and SimpleJdbcCall

Spring JdbcTemplate class

- It is the central class in the Spring JDBC support classes. It takes care of **creation and release of resources** such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.
- It handles the exception and provides the informative exception messages by the help of exception classes defined in the *org.springframework.dao* package.
- We can perform all the database operations by the help of **JdbcTemplate** class such as **insertion, updation, deletion** and **retrieval** of the data from the database.
- Let's see the **methods** of spring JdbcTemplate class.

No.	Method	Description
-----	--------	-------------

- | | | |
|----|---|--|
| 1) | public int update (String query) | is used to insert, update and delete records. |
| 2) | public int update(String query, Object... args)
PreparedStatement using given arguments. | is used to insert, update and delete records using |
| 3) | public void execute (String query) | is used to execute DDL query. |
| 4) | public T execute(String sql, PreparedStatementCallback action)
PreparedStatement callback. | executes the query by using |
| 5) | public T query(String sql, ResultSetExtractor rse) | is used to fetch records using ResultSetExtractor. |
| 6) | public List query(String sql, RowMapper rse) | is used to fetch records using RowMapper. |

Spring JdbcTemplate example

We are assuming that you have created the following table inside the Oracle10g database.

```
create table employee(  
  id number(10),  
  name varchar2(100),  
  salary number(10)  
);
```

Employee.java contains 3 properties with constructors and setter and getters.

```
public class Employee {  
  private int id;  
  private String name;  
  private float salary;  
    //no-arg and parameterized constructors  
    //getters and setters  
}
```

Spring JdbcTemplate example

```
import org.springframework.jdbc.core.JdbcTemplate;

public class EmployeeDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

        this.jdbcTemplate = jdbcTemplate;

    }

    public int saveEmployee(Employee e){

        String query="insert into employee values(

            '"+e.getId()+"', '"+e.getName()+"', '"+e.getSalary()+"'";

        return jdbcTemplate.update(query);

    }

    public int updateEmployee(Employee e){

        String query="update employee set

            name='"+e.getName()+"', salary='"+e.getSalary()+"' where id='"+e.getId()+"' ";

        return jdbcTemplate.update(query);

    }

    public int deleteEmployee(Employee e){

        String query="delete from employee where id='"+e.getId()+"' ";

        return jdbcTemplate.update(query);

    } }
```


Spring JdbcTemplate example

ApplicationContext.xml

The DriverManagerDataSource is used to contain the information about the database such as driver class name, connection URL, username and password.

There are a property named datasource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.

Here, we are using the JdbcTemplate object in the EmployeeDao class, so we are pass

Spring JdbcTemplate example

```
<?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:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
    <property name="username" value="system" />
    <property name="password" value="oracle" />

  </bean>

  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

    <property name="dataSource" ref="ds"></property>

  </bean>

  <bean id="edao" class="com.javatpoint.EmployeeDao">

    <property name="jdbcTemplate" ref="jdbcTemplate"></property>

  </bean>

</beans>
```

Spring JdbcTemplate example

Test.java

This class gets the bean from the applicationContext.xml file and calls the saveEmployee() method. You can also call updateEmployee() and deleteEmployee() method by uncommenting the code as well.

Spring JdbcTemplate example

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

    public static void main(String[] args) {

        ApplicationContext ctx=new ClassPathXmlApplicationContext("applicationContext.xml");

        EmployeeDao dao=(EmployeeDao)ctx.getBean("edao");

        int status=dao.saveEmployee(new Employee(102,"Amit",35000));

        System.out.println(status);

        /*int status=dao.updateEmployee(new Employee(102,"Sonoo",15000));

        System.out.println(status);

        */

        /*Employee e=new Employee();

        e.setId(102);

        int status=dao.deleteEmployee(e);

        System.out.println(status);*/

    }

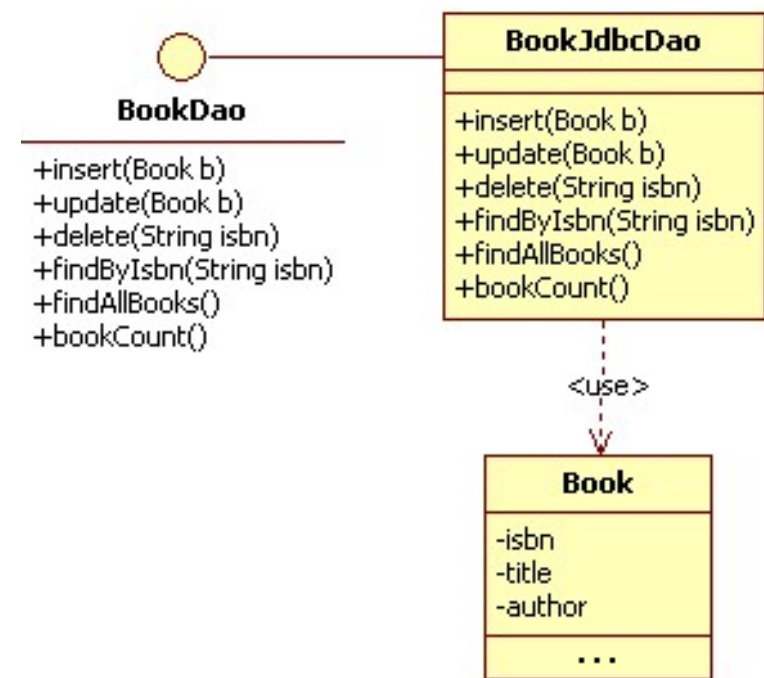
}
```

Test.java

Accesso ai dati

Una delle peculiarità principali di Spring è quella di favorire la scrittura di codice modulare favorendone il riuso. Per questa ragione Spring incoraggia l'utilizzo del **DAO** (Data Access Object), un pattern architetturale che ha come scopo quello di separare le logiche di business da quelle di accesso ai dati.

L'idea alla base di questo pattern è quello di descrivere le operazioni necessarie per la persistenza del modello in un'interfaccia e di implementare la logica specifica di accesso ai dati in apposite classi.



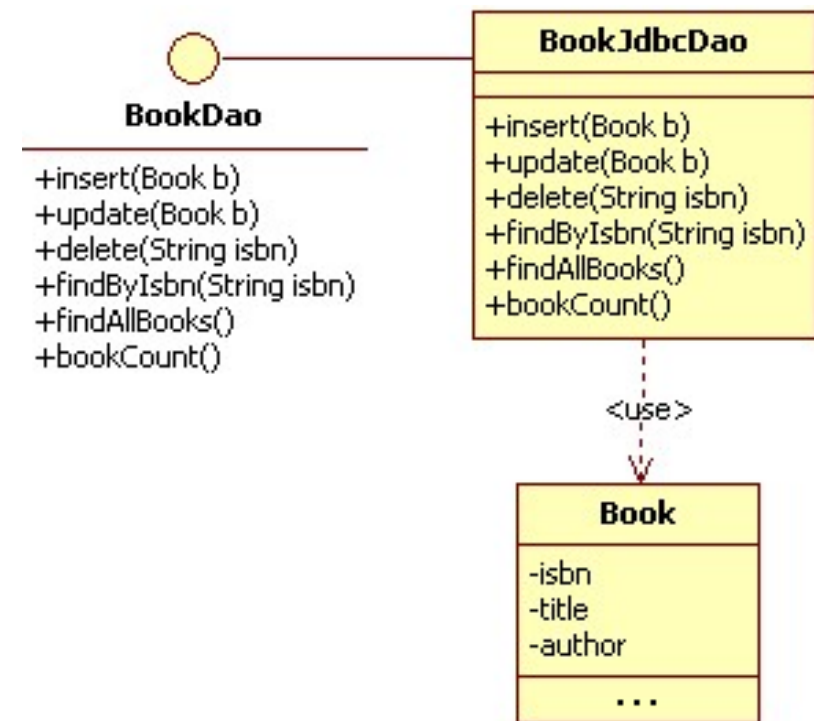
Accesso ai dati

La logica di business necessaria per la persistenza del nostro modello è descritta nella classe **BookDao**, mentre nella classe **BookJdbcDao** troviamo un'implementazione specifica di questa interfaccia rivolta a gestire le logiche di accesso ai dati mediante tecnologia JDBC.

In questo scenario è possibile notare come grazie all'utilizzo del pattern DAO sia possibile con poco sforzo fornire ulteriori implementazioni della classe BookDao per introdurre nuove logiche di accesso ai dati (ad esempio per l'utilizzo di Hibernate un ipotetico BookHibernateDao).

L'interfaccia DAO che utilizzeremo sarà così fatta:

```
public interface BookDao {  
    public void insert(Book book);  
    public void update(Book book);  
    public void delete(String isbn);  
    public Book findByISBN(String isbn);  
    public List<Book> findAllBooks();  
    public int bookCount();  
}
```



Spring JDBC Template

Spring offre diverse possibilità per la persistenza dei dati mediante JDBC, la principale delle quali è l'utilizzo della **classe JDBCTemplate**.

Questa classe implementa l'intero processo di accesso ai dati attraverso **template methods**, rendendo possibile la personalizzazione di tutte le fasi di tale processo mediante l'override dei metodi specifici.

Vediamo ora attraverso degli snippet di codice come eseguire operazioni sul Database mediante JDBCTemplate.

Iniziamo con la creazione dell'implementazione specifica per JDBC del **BookDao** visto prima. Come è possibile notare dal codice, questo possiede un riferimento alla classe JDBCTemplate, inizializzata attraverso il metodo setDataSource() che provvede a fornire il DataSource necessario.

```
public class BookJdbcDao implements BookDao {  
    private JdbcTemplate jdbcTemplate;  
  
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {  
        this.jdbcTemplate = jdbcTemplate;  
    }  
    // ...  
}
```

Spring JDBC Template: Update

Attraverso il metodo **update** è possibile eseguire tutte quelle operazioni che comportano una modifica dei dati nel database.

```
public class BookJdbcDao implements BookDao {  
    // ...  
    public void insert(Book book)           //Inserimento  
    {  
        jdbcTemplate.update("insert into books (isbn, autore, titolo) values (?, ?, ?)",  
            new Object[] { book.getIsbn(), book.getAutore(), book.getTitolo() });  
    }  
  
    public void update(Book book)           //Modifica  
    {  
        jdbcTemplate.update("update books set autore = ?, titolo = ? where isbn = ?",  
            new Object[] { book.getIsbn(), book.getAutore(), book.getTitolo() });  
    }  
  
    public void delete(String isbn)         // Eliminazione  
    {  
        jdbcTemplate.update("delete from books where isbn = ?",  
            new Object[] { isbn });  
    }  
}
```


Spring JDBC Template: Ricerca

Passando alle interrogazioni, questo è il caso di ricerca di un intero.

```
public class BookJdbcDao implements BookDao {  
    // ...  
    // Query di un intero  
    public int bookCount() {  
        int rowCount = jdbcTemplate.queryForInt("select count(1) from books");  
        return rowCount;  
    }  
}
```

Passando alle interrogazioni di oggetti oltre alla query e i suoi parametri è necessario specificare come deve avvenire il mapping tra i risultati presenti nel `ResultSet` e le proprietà dell'oggetto. Per fare questo si utilizza l'interfaccia **RowMapper**, nel nostro codice implementata dalla classe `BookRowMapper`.

Spring JDBC Template: Ricerca

```
public class BookJdbcDao implements BookDao {  
    ...  
    public Book findByISBN(String isbn) { //Query di un singolo oggetto  
        Book book = (Book) jdbcTemplate.queryForObject("select * from books where isbn = ? ", new Object[] { isbn }, new BookRowMapper());  
        return book;  
    }  
    public List<Book> findAllBooks() { // Query di una lista  
        List<Book> books = (List<Book>) jdbcTemplate.query("select * from books", new BookRowMapper());  
        return books;  
    }  
}  
  
public class BookRowMapper implements RowMapper { // Classe RowMapper  
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Book book= new Book();  
        book.setIsbn(rs.getString("isbn"));  
        book.setAutore(rs.getString("autore"));  
        book.setTitolo(rs.getString("titolo"));  
        return book;  
    } }  
}
```

la classe BookRowMapper contiene le regole di mapping tra una riga del ResultSet e classe Book ed è utilizzata dal jdbcTemplate per ritornare i valori della query.

Jdbc Template Injection

Grazie all'utilizzo dell'**Inversion of Control** è possibile iniettare nella nostra classe un oggetto di tipo **JdbcTemplate** già inizializzato con un datasource. Esaminiamo il **file di configurazione**:

```
<?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-2.0.xsd">

    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url" value="jdbc:derby://localhost:1527/library;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookDao" class="it.spring.book.BookJdbcDao">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>

</beans>
```

Jdbc Template Injection

La classe **JdbcDaoSupport** permette di agevolare l'utilizzo di JdbcTemplate implementando direttamente i metodi setDataSource() e setJdbcTemplate().

Facendo estendere questa classe ai propri DAO sarà sufficiente iniettare un datasource per poter utilizzare un JdbcTemplate attraverso il metodo **getJdbcTemplate()** (ereditato anch'esso da **JdbcDaoSupport**).

```
public class BookJdbcDao extends JdbcDaoSupport implements BookDao {  
  
    ...  
  
    //Inserimento  
    public void insert(Book book)  
    {  
        getJdbcTemplate().update("insert into book (isbn, autore, titolo) values (?, ?, ?)",  
            new Object[] { book.getIsbn(), book.getAuthor(), book.getTitle() });  
    }  
}
```

Simple Jdbc Template

La classe **SimpleJdbcTemplate** aggiunge alcune funzionalità offerte da Java 1.5 alla classe base `JdbcTemplate`, semplificando ulteriormente le procedure di accesso ai dati. Grazie al supporto dei parametri a lunghezza variabile è possibile passare direttamente i valori ai metodi di `update` senza dover utilizzare un array di oggetti.

```
public class BookJdbcDaoSupport extends SimpleJdbcDaoSupport implements BookDao {  
  
    public void insert(Book book) {  
        getSimpleJdbcTemplate().update("insert into book (isbn, autore, titolo) values (?, ?, ?)", book.getIsbn(), book.getAutore(),  
book.getTitolo());  
        ...  
    }  
}
```

Con l'utilizzo dei generics e dell'autoboxing anche le query vengono notevolmente semplificate.

Simple Jdbc Template

In particolare Spring fornisce la classe **ParameterizedBeanPropertyRowMapper**, un'implementazione dell'interfaccia `RowMapper` che si occupa di effettuare le operazioni di mapping viste in precedenza in maniera trasparente allo sviluppatore.

```
public Book findByISBN(String isbn) {                //Query di un singolo oggetto
    Book book = getSimpleJdbcTemplate().queryForObject("select * from books where isbn =?",
        ParameterizedBeanPropertyRowMapper.newInstance(Book.class), isbn);
    return book;
}
```

```
public List<book> findAllBooks() {                // Query di una lista di oggetti
    List<book> books = (List<book>) getJdbcTemplate().query( "select * from
books",ParameterizedBeanPropertyRowMapper.newInstance(Book.class));
    return books;
}
```

```
</book></book></book>
```

Esempio completo file esempio_02.zip

Parametri nominali con Jdbc Template

Sempre all'insegna della semplicità, Spring aggiunge alla programmazione JDBC il supporto ai parametri nominali, consentendo l'utilizzo di label come segnalibri al posto del classico punto interrogativo (?). Questo da un lato favorisce la leggibilità delle query e dall'altro elimina possibili complicazioni dovute alla posizionalità dei parametri.

In Spring questa funzionalità viene offerta sia attraverso l'utilizzo diretto della classe `NamedParameterJdbcTemplate`, che grazie al supporto del `SimpleJdbcTemplate` visto in precedenza. Quest'ultima è la soluzione adottata per il prossimo esempio.

Il binding tra i parametri identificati dalle label e i relativi valori avviene tramite la classe `SqlParameterSource`, la cui implementazione base è la **`MapSqlParameterSource`**.

```
public class BookSimpleJdbcDaoSupportNamedValue extends SimpleJdbcDaoSupport {  
  
    public void update(Book book) {  
        MapSqlParameterSource parameters = new MapSqlParameterSource();  
        parameters.addValue("isbn", book.getIsbn());  
        parameters.addValue("author", book.getAuthor());  
        parameters.addValue("title", book.getTitle());  
        getSimpleJdbcTemplate().update("update books set author = :author, title = :title where isbn = :isbn",  
                                       parameters);  
    }  
    // ...  
}
```

Parametri nominali con Jdbc Template

SqlParameterSource offre anche un'altra implementazione in grado di effettuare un mapping automatico dei parametri, la classe **BeanPropertySqlParameterSource**.

```
public class BookSimpleJdbcDaoSupportNamedValue extends SimpleJdbcDaoSupport {

    public void insert(Book book) {
        // Binding automatico dei parametri
        SqlParameterSource parameters = new BeanPropertySqlParameterSource(book);
        //Inserimento
        getSimpleJdbcTemplate().update("insert into books (isbn, author, title) values (:isbn, :author, :title)",
            parameters);
    }
    ...
}
```

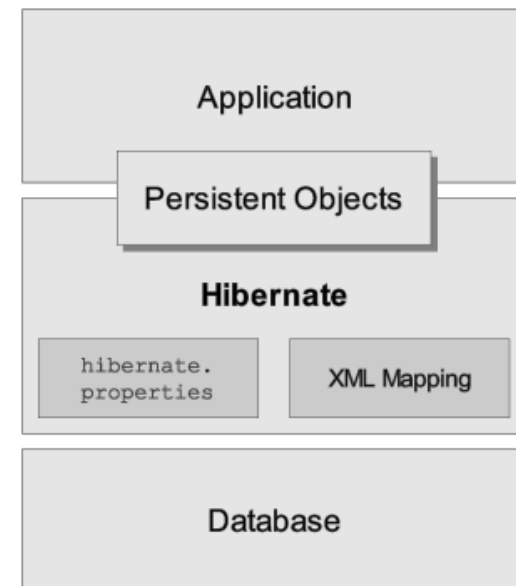

Spring e Hibernate

Un ORM, acronimo di Object Relation Mapping, è una tecnologia che nasce per semplificare la persistenza di oggetti in un database relazionale, generando automaticamente il codice SQL necessario.

Nel mondo J2EE esiste una moltitudine di ORM e, dal canto suo, Spring offre loro un grande supporto che include tra gli altri Hibernate, Oracle TopLink, iBATIS SQL Maps, JDO e JPA.

Hibernate è uno strato di middleware che consente allo sviluppatore di automatizzare le procedure per le operazioni cosiddette CRUD (Create, Read, Update, Delete) dei database

Lo scopo è indicare al middleware in maniera dichiarativa (con dei descrittori testuali, xml) l'associazione tra la classe del javabean e la tabella in cui risiedono i dati. L'infrastruttura si occuperà di recuperare dinamicamente le informazioni associate (leggendo i descrittori) e creando automaticamente le query necessarie .



Hibernate mapping

Perché un ORM sia in grado di generare codice SQL è necessario specificare il mapping tra gli oggetti da persistere e le tabelle nel database.

Mapping con XML

In accordo con le specifiche Hibernate, uno dei metodi per effettuare questa operazione è utilizzare un file XML che, per convenzione, ha nome:
<classe>.hbm.xml

Di seguito è illustrato il file book.hbm.xml responsabile del mapping tra la classe Book vista in precedenza e la relativa tabella nella base dati.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="it.html.spring.book">
  <class name="Book" table="BOOK">
    <id name="isbn" type="string" column="ISBN"></id>
    <property name="titolo" type="string" column="TITOLO"></property>
    <property name="autore" type="string" column="AUTORE"></property>
  </class>
</hibernate-mapping>
```

N.B. La gestione del mapping per mezzo dei file XML è relativamente semplice, ma può risultare difficile da gestire, soprattutto quando il numero delle classi da persistere è alto.

Hibernate mapping Annotation

Un'alternativa è data dall'adozione delle JPA annotations introdotte da Sun a partire da Java 5 per permettere di specificare il mapping all'interno delle stesse classi da persistere. Le JPA annotations possono essere utilizzate da tutti gli ORM JPA-compliant come Hibernate.

Di seguito è mostrata la classe Book con aggiunte le annotations necessarie per la sua persistenza.

@Entity

@Table(name = "BOOK")

public class **Book** {

 @Id

 @Column(name = "ISBN")

 String isbn;

 @Column(name = "AUTORE")

 String autore;

 @Column(name = "TITOLO")

 String titolo;

 public String getIsbn() { return isbn; }

 public void setIsbn(String isbn) { this.isbn = isbn; }

 public String getAutore() { return autore; }

 public void setAutore(String autore) { this.autore = autore; }

 public String getTitolo() { return titolo; }

 public void setTitolo(String titolo) { this.titolo = titolo; }

 public String toString() {

 return isbn + " - " + autore + " - " + titolo;

 }

}

Dal codice risultante si nota subito come l'utilizzo di queste annotations renda le operazioni di mapping più semplici e manutenibili rispetto ai rispettivi file XML. Per questa ragione, da ora in poi, prenderemo in considerazione questa alternativa.

HibernateTemplate

Proseguiamo ora creando un'implementazione specifica per Hibernate del nostro BookDao. Come per JDBC, Spring ci viene in aiuto fornendo **HibernateTemplate**: uno specifico template che si fa carico delle operazioni di gestione, tra le quali:

- Richiesta della connessione alla factory
- Apertura della transazione
- Gestione delle eccezioni
- Commit/Rollback della transazione
- Chiusura della connessione

Unitamente a HibernateTemplate Spring mette a disposizione la classe di supporto **HibernateDaoSupport** per facilitarne l'injection all'interno dei DAO. Estendendo questa classe sarà sufficiente richiamare il metodo getHibernateTemplate() per poter avere un hibernate template da utilizzare.

```
public class BookHibernateDaoSupport extends HibernateDaoSupport
implements BookDao {
```

```
    @Transactional
```

```
    public int bookCount() {
        return findAllBooks().size();
    }
```

```
    @Transactional
```

```
    public void delete(String isbn) {
        Book book = (Book) getHibernateTemplate().get(Book.class, isbn);
        getHibernateTemplate().delete(book);
    }
```

```
    @Transactional(readOnly = true)
```

```
    public List<book> findAllBooks() {
        return getHibernateTemplate().find("from Book");
    }
```

HibernateTemplate

```
    @Transactional(readOnly = true)
    public Book findByISBN(String isbn) {
        return (Book) getHibernateTemplate().get(Book.class, isbn);
    }
```

```
    @Transactional
```

```
    public void insert(Book book) {
        getHibernateTemplate().saveOrUpdate(book);
    }
```

```
    public void update(Book book) {
        getHibernateTemplate().saveOrUpdate(book);
    }
}
</book>
```

HibernateTemplate

Quello che si vuole ottenere è che le operazioni effettuate dai metodi del DAO siano transazionali, cioè abbiano un senso compiuto nella loro interezza. Questo comportamento è reso possibile dall'utilizzo dell'apposita annotation **@Transactional** con la quale è possibile richiedere al framework che tutte le chiamate ai metodi coinvolti siano trattati nella stessa transazione.

In realtà, i componenti che entrano in gioco per permettere a **HibernateTemplate** di svolgere il proprio lavoro sono molteplici, anche se non direttamente visibili, perchè mascherati dall'utilizzo dell'Inversion of Control. È proprio qui infatti che l'utilizzo di IoC esprime un ruolo fondamentale, semplificando notevolmente le operazioni di persistenza.

Passiamo quindi a specificare come configurare questi componenti.

Datasource

Session factory

Datasource

Il primo bean da configurare è quello responsabile della connessione verso la sorgente dati. Come per gli esempi relativi a JDBC utilizzeremo il **DriverManagerDataSource** un semplice datasource in grado di aprire una connessione ogni volta che questa viene richiesta. Questo modo di operare non è certamente dei migliori quanto a gestione delle risorse; per questo motivo, in ambienti più complessi, si suggerisce l'utilizzo di datasource più efficienti come il `SingleConnectionDataSource`.

```
<?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="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">

        <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url" value="jdbc:derby://localhost:1527/books;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>
    <!-- ... -->
</beans>
```

SessionFactory

Per far sì che Hibernate svolga il proprio lavoro di persistenza sono necessarie alcune configurazioni che indicano al framework quali parametri utilizzare per la creazione del SessionFactory, l'oggetto responsabile dell'apertura delle sessioni verso il database. Tali parametri includono:

- Datasource da utilizzare per la connessione con il database
- Dialecto SQL utilizzato da Hibernate per l'ottimizzazione delle query
- Lista degli eventuali file di mapping
- Altre informazioni accessorie

Nell'utilizzo classico di hibernate queste configurazioni possono essere espresse sotto forma di file xml (utilizzando il file hibernate.cfg.xml) oppure tramite annotations. A queste modalità Spring aggiunge una terza alternativa sfruttando il meccanismo **dell'Inversion of Control** per la creazione di SessionFactory e la relativa injection dei parametri necessari.

Di seguito sono mostrate due possibili configurazioni per il bean sessionFactory da impiegare nell'applicazione presa in esempio. In particolare la prima in caso di utilizzo delle **annotations JPA** per il mapping delle classi da persistere, la seconda in caso di adozione di file **XML**.

SessionFactory

```
<!-- beans ... -->
```

```
<!-- Session Factory da utilizzare per mapping attraverso JPA Annotations -->
```

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
```

```
  <property name="dataSource" ref="dataSource" />
```

```
  <property name="annotatedClasses">
```

```
    <list>
```

```
      <value>it.html.spring.book.Book</value>
```

```
    </list>
```

```
  </property>
```

```
  <property name="hibernateProperties">
```

```
    <props>
```

```
      <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
```

```
      <prop key="hibernate.show_sql">true</prop>
```

```
      <prop key="hibernate.hbm2ddl.auto">update</prop>
```

```
    </props>
```

```
  </property>
```

```
</bean>
```

SessionFactory

```
<!-- Session Factory da utilizzare per mapping attraverso file xml-->
<!-- bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>book.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean -->

<!-- ... -->

</beans>
```

Progetto completo Eclipse: Esempio_03.zip

Transaction Manger

Responsabile delle transazioni è il `TransactionManager`, un componente in grado di creare e gestire le transazioni attraverso un `datasource` e al cui utilizzo si affida `HibernateTemplate`. Spring fornisce diversi transaction manager, uno per ogni tecnologia ORM supportata. Nel caso preso in esempio quello da utilizzare è il **`HibernateTransactionManager`** in grado di operare tramite una `session factory`. Occorre anche specificare, attraverso il tag **`tx:annotation-driven`** che le direttive di transazione vengono fornite attraverso l'utilizzo di annotations.

```
<?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-3.0.xsd">

    <!-- ... -->
    <tx:annotation-driven />

    <bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- ... -->

</beans>
```

Hibernate Template

Passiamo ora alla configurazione del template hibernate discusso in precedenza

```
<!-- ... -->
```

```
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">  
  <property name="sessionFactory" ref="sessionFactory" />  
</bean>
```

```
<!-- ... -->
```

```
</beans>
```

Book Dao

A questo punto l'ultimo bean rimasto da configurare è semplicemente quello rappresentante l'implementazione specifica del Book Dao.

```
<!-- ... -->
```

```
<bean id="bookDao" class="it.spring.book.BookHibernateDaoSupport">  
  <property name="hibernateTemplate" ref="hibernateTemplate" />  
</bean>
```

```
<!-- ... -->
```

```
</beans>
```

In caso di utilizzo della classe `HibernateDaoSupport` è possibile iniettare nel `BookDao` direttamente un `TransactionManager` rendendo di fatto inutile configurare `HibernateTemplate`. Sarà poi la stessa classe di supporto responsabile di istanziare un oggetto di tipo `HibernateTemplate`.

Applicazioni di esempio

<https://spring.io/guides/gs/accessing-data-mysql/>

<https://spring.io/guides/gs/relational-data-access/>

<https://spring.io/guides/gs/spring-boot/>

<https://spring.io/blog/2015/08/19/migrating-a-spring-web-mvc-application-from-jsp-to-angularjs>