COURS DE JAVA – Java Spring
El Hadji Gaye - AlizNet

Avec Netbeans.

Auteur	El Hadji Gaye
Pour	Ecole
Date	11/12/2017
Objet	Java Spring

I) **Vocabulaire** 3
II) **Présentation du Framework Spring** 4
III) **Design pattern Injection de dépendance.** 5
IV) **Application 1 : injection de JavaBean avec Spring** 9
V) **Application 2 : Spring avec l'ORM JPA Hibernate** 22
VI) **Application 3 : maven-personnes-jpa-spring** 35

I) Vocabulaire

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.

- **ORM**: Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.

- **JPA** : Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.

- **JPQL** : Le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.

- **HQL** : **Hibernate Query Language** est aussi un langage de requête orienté objet au même titre que JPQL. La principale différence avec le langage JQL est que le en HQL le « **Select** » sur l'objet n'est pas nécessaire. En fin de compte pour le JPQL on aura : **Select person from Personne person** alors que pour le HQL on aura : **from Personne**.

- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.

- **Pattern IoC** : L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les Frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du Framework ou de la couche logicielle sous-jacente. En effet selon un problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances (dependency injection) qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.

- **Pattern AOP** : L'AOP (Aspect Oriented Programming) ou POA (Programmation Orientée Aspect) est un paradigme de programmation ayant pour but de compléter la programmation orientée objet et permettre d'implémenter de façon plus propre les problématiques transverses à l'application. En effet, elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.

II) Présentation du Framework Spring

Le framework Spring a été initialement développé par Rod Johnson et Juergen Holler dans sa version **Spring 1.0** en mars 2004. Spring est un framework libre qui permet de construire et de définir l'infrastructure d'une application java, dont il facilite le développement et les tests. Il fournit de nombreuses fonctionnalités qui peuvent être utilisées de plusieurs manières ce qui laisse le choix au développeur d'utiliser la solution qui répond à ses besoins.Il est l'un des frameworks les plus répandus dans le monde Java, sa popularité a grandie à cause de la complexité des serveurs d'application Java/J2EE.
Le framework a un coeur reposant sur un conteneur de type IoC assure la gestion du cycle de vies des beans et l'injection des dépendances et un autre conteneur dite AOP qui elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.
Spring était un framework applicatif qui permet de faciliter l'intégration avec des projets open source ou API de Java/J2EE. Sa principale interêt est sa grande flexibilité ses fonctionnalités.
Spring est souvent appelé conteneur léger (lightweight container) par opposition aux conteneurs lourds que sont les serveurs d'applications Java/J2EE.

III) Design pattern Injection de dépendance.

Considérons une entreprise avec des employés dont chacun a une adresse et un rôle précis dans l'entreprise. Les entités mis en jeu dans cet example son **Employe**, **Role** et **Adresse** dont les signatures sont :

```
1   public class Role {
2       private int idRole;
3       private String codeRole;
4       private String descriptionRole;
5       public Role() {}
6   }
7   public class Adresse {
8       private int idAdresse;
9       private String nomRue;
10      private String codePostal;
11      private String ville;
12      private String pays;
13      public Adresse() {}
14  }
15  public class Employe {
16      private int idEmploye;
17      private Role role;
18      private Adresse adresse;
19      public Employe() {
20          this.role = new Role();
21          this.adresse = new Adresse();
22      }
23  }
```

En version copiable:

```
public class Role {
   private int idRole;
   private String codeRole;
   private String descriptionRole;
   public Role() {}
}
public class Adresse {
   private int idAdresse;
   private String nomRue;
   private String codePostal;
   private String ville;
   private String pays;
   public Adresse() {}
}
public class Employe {
   private int idEmploye;
   private Role role;
   private Adresse adresse;
   public Employe() {
      this.role = new Role();
      this.adresse = new Adresse();
   }
}
```

Ce code ci-dessus permet de voir la relation de dépendance entre la classe **Employe** et les classes **Role** et **Adresse**. Mais le code comporte quand même quelques soucis :

- Lorsqu'on instancie un objet de type **Employe** alors les objets **Role et Adresse** sont tout temps les même.
- Lorsqu'on change la signature des classes **Role** ou **Adresse** (constructeur par exemple) alors on doit aussi changer le constructeur de la classe **Employe** aussi sous peine d'avoir des erreurs de compilation.

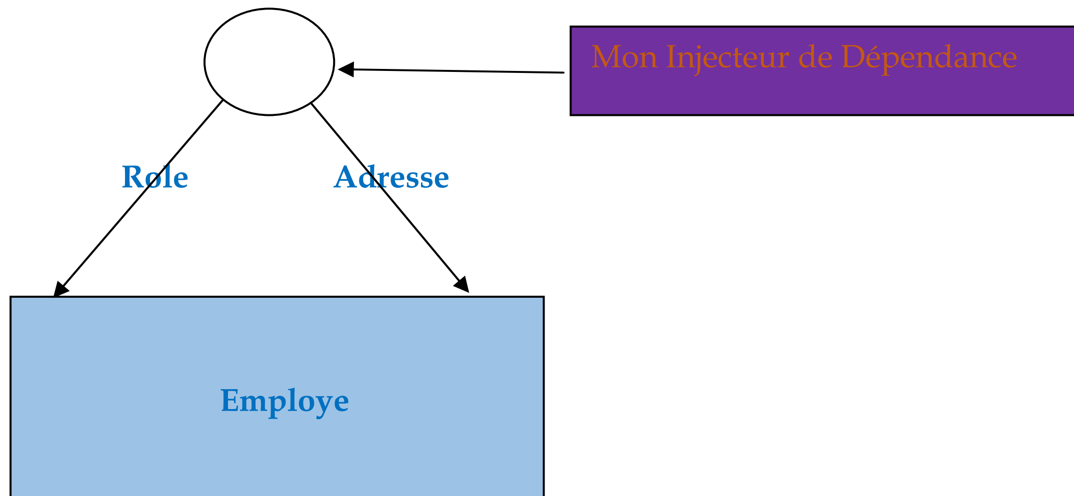
Le premier problème peut être résolu facilement en redéfinissant les constructeurs **Role** et **Adresse**. Ceci va nous obliger à changer les classes **Employe**, **Role** et **Adresse**.

```
1   public class Role {
2       private int idRole;
3       private String codeRole;
4       private String descriptionRole;
5
6       public Role(String codeRole, String descriptionRole) {
7           this.codeRole = codeRole;
8           this.descriptionRole = descriptionRole;
9       }
10  }
11
12  public class Adresse {
13      private int idAdresse;
14      private String nomRue;
15      private String codePostal;
16      private String ville;
17      private String pays;
18
19      public Adresse(String nomRue, String codePostal, String ville, String pays) {
20          this.nomRue = nomRue;
21          this.codePostal = codePostal;
22          this.ville = ville;
23          this.pays = pays;
24      }
25  }
26
27  public class Employe {
28      private int idEmploye;
29      private Role role;
30      private Adresse adresse;
31
32      public Employe(Role role,Adresse adresse) {
33          this.role = role;
34          this.adresse = adresse;
35      }
36  }
37  Role role = new Role("Administrateur","Je suis Admin");
38  Address addresse = new Address("river","5555","New York","Etats Unis");
39  Employe employe = new Employe(role,addresse);
40
```

En version copiable:

```
public class Role {
    private int idRole;
    private String codeRole;
    private String descriptionRole;

    public Role(String codeRole, String descriptionRole) {
        this.codeRole = codeRole;
        this.descriptionRole = descriptionRole;
    }
}

public class Adresse {
    private int idAdresse;
    private String nomRue;
    private String codePostal;
    private String ville;
    private String pays;

    public Adresse(String nomRue, String codePostal, String ville, String pays) {
        this.nomRue = nomRue;
        this.codePostal = codePostal;
        this.ville = ville;
        this.pays = pays;
    }
}

public class Employe {
    private int idEmploye;
    private Role role;
    private Adresse adresse;

    public Employe(Role role,Adresse adresse) {
        this.role = role;
        this.adresse = adresse;
    }
}
Role role = new Role("Administrateur","Je suis Admin");
Address addresse = new Address("river","5555","New York","Etats Unis");
Employe employe = new Employe(role,addresse);
```

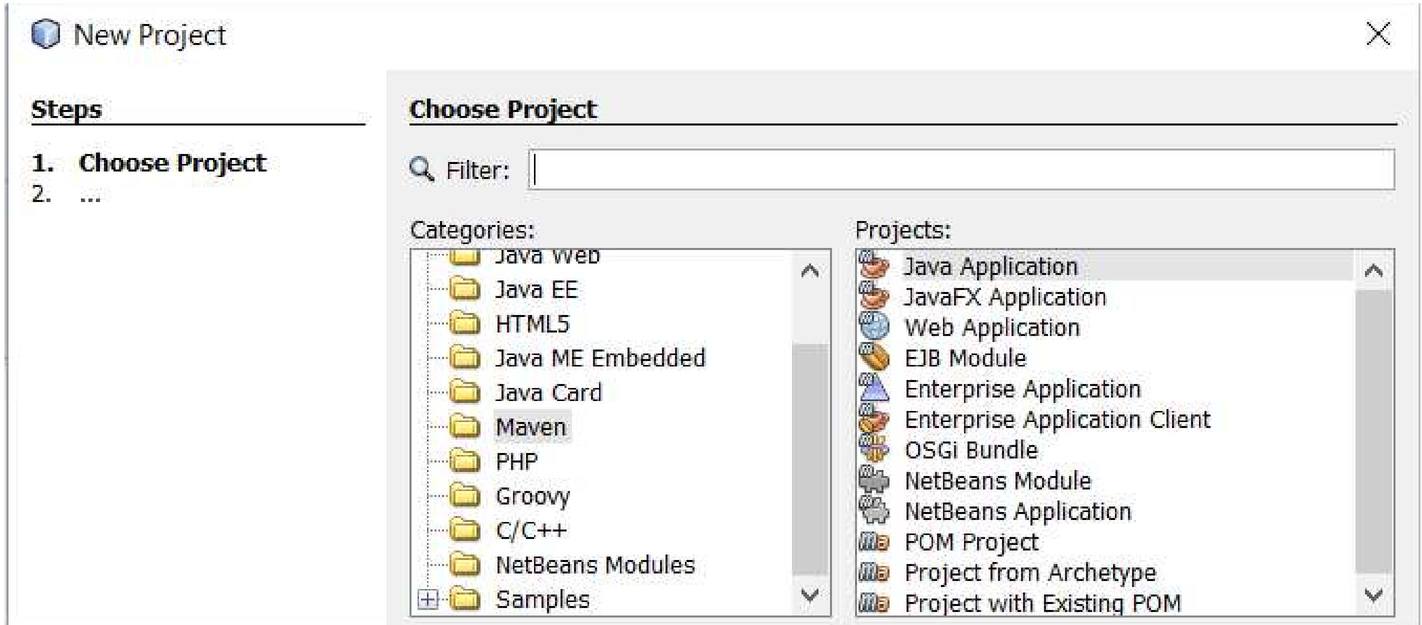
Lignes 37 ,38, 39 : Pour avoir une instance de **Employe** il faut aussi avoir une instance de **Role** et **Adresse** ce qui cause aussi problème.
Ce qui serait trop cool c'est de pouvoir avoir une instance de **Employe** sans se souci des instances qui servent à construire un objet de type **Employe** (ici **Role** et **Adresse**).

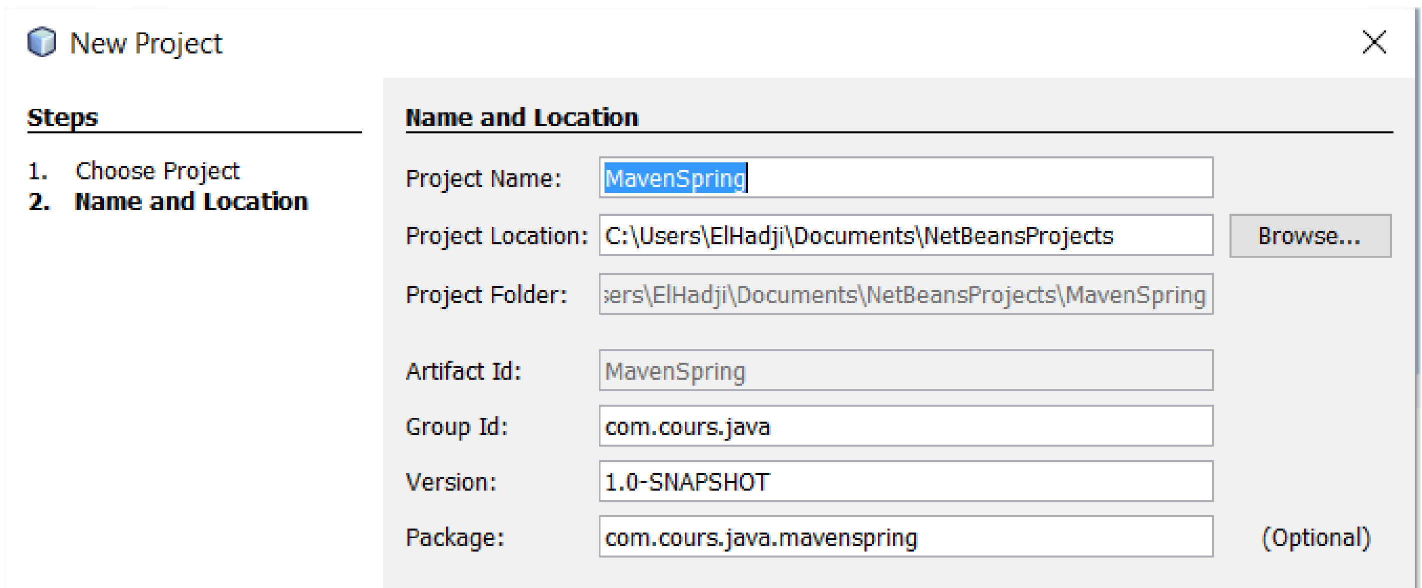


Dans l'exemple ci dessus c'est le framework « **Mon injecteur de dépendance** » qui se charge d'injecter les instances necessaire à la contruction d'un objet de type **Employe** permettant ainsi de renvoyer des instances d'**Employe** sans se soucier de **Role** et **Adresse**.
En POO (programmation orienté objet) c'est ce processus que l'on appelle Injection de dépendance.
Dans les langages tel que **Java** et **.Net** ce processus d'injection est géré par un Framework portant le nom de **Spring**.

IV) Application 1 : injection de JavaBean avec Spring

Considérons l'example de l'application « **MavenSpring** » de type Maven.
« New Project » → « Maven » → « Java Application » → « MavenSpring ».





Le fichier **pom.xml** peut ressembler à :

```
<?xml version="1.0" encoding="UTF-8"?>
<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>monModelVersion</modelVersion>
   <groupId>monGroupId</groupId>
   <artifactId>maVersionArtifactId</artifactId>
   <version>maVersionDeSnap</version>
   <packaging>jar</packaging>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>maVersionDeMaven</maven.compiler.source>
      <maven.compiler.target>maVersionDeMaven</maven.compiler.target>
   </properties>
</project>
```

Dans mon cas :

- **monModelVersion** est égale à **4.0.0**.
- **monGroupId** est égale à **com.cours.java**.
- **maVersionArtifactId** est égale à **MavenSpring**.
- **maVersionDeSnap** est égale à **1.0-SNAPSHOT**.
- **maVersionDeMaven** est égale à **1.7**.

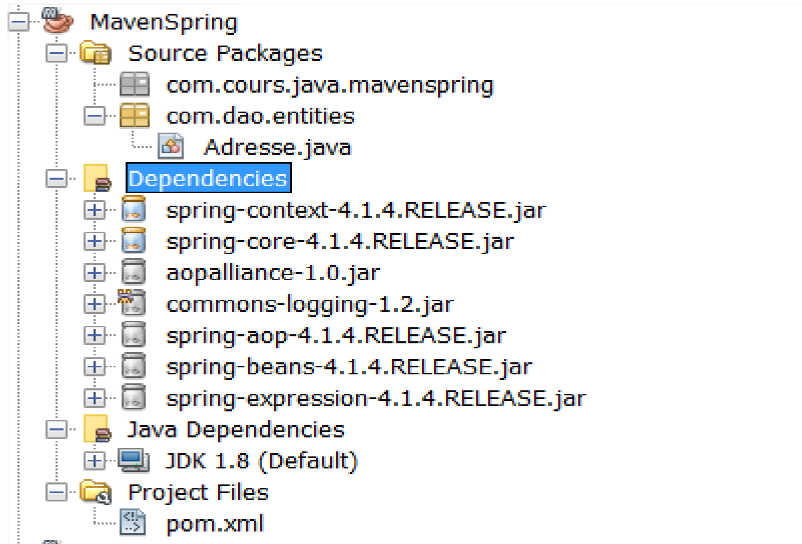
Nous allons maintenant ajouter la liste des Jar à telecharger pour utiliser **Spring**.
Modifier la section « **properties** » de votre fichier **pom.xml** en rajoutant la variable spring.version dans la section « **properties** ».
La section en question devient donc :

```
 <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>maVersionDeMaven</maven.compiler.source>
      <maven.compiler.target>maVersionDeMaven</maven.compiler.target>
      <spring.version>4.1.4.RELEASE</spring.version>
  </properties>
```

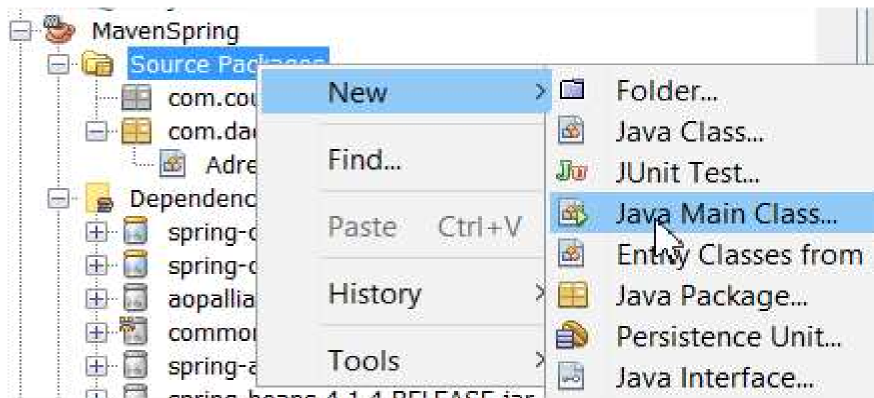
Modifier la section « **dependencies** » du **pom.xml** en mettant :

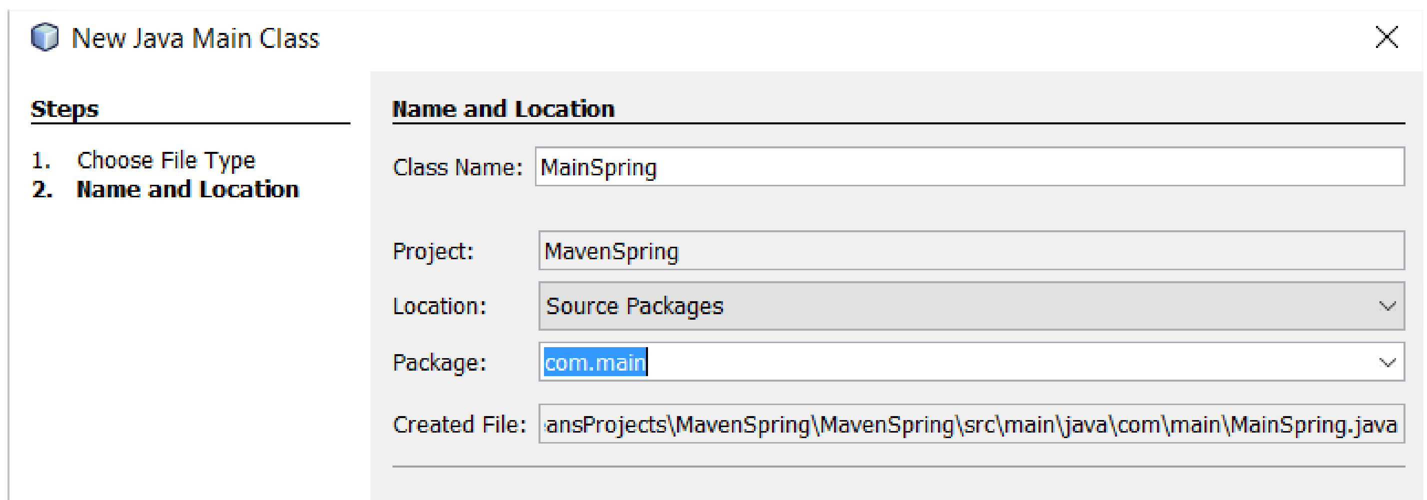
```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency>
</dependencies>
```

Après un **Clean and Build** on retrouve tous les jar de Spring dans la section « **Dependencies** » :

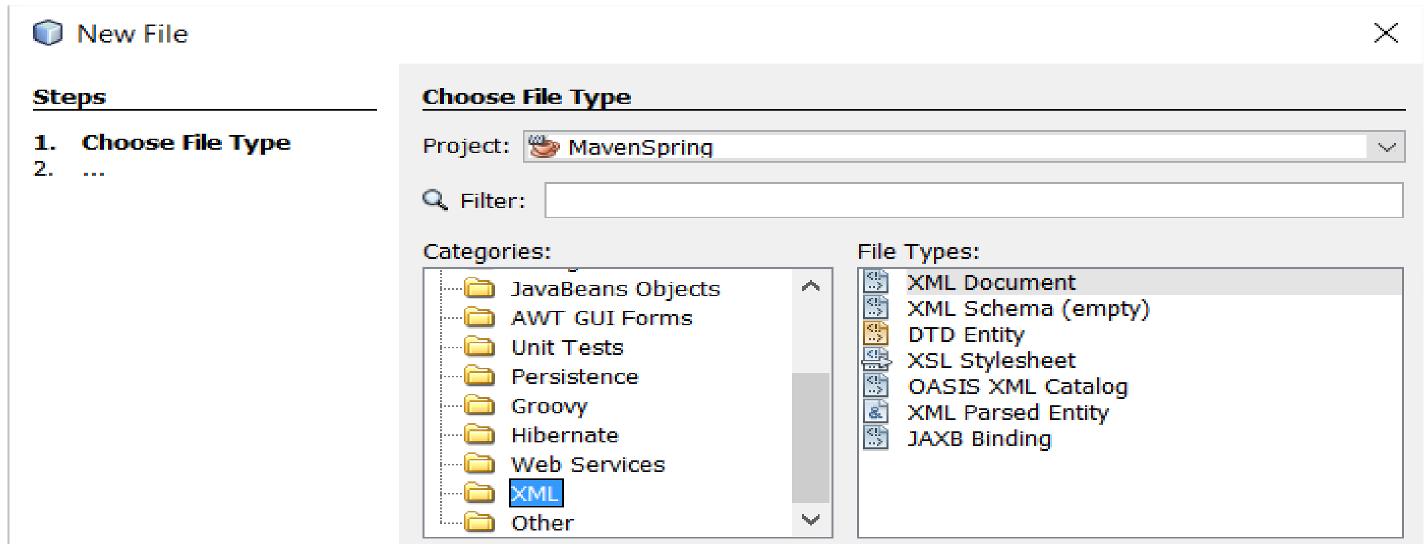


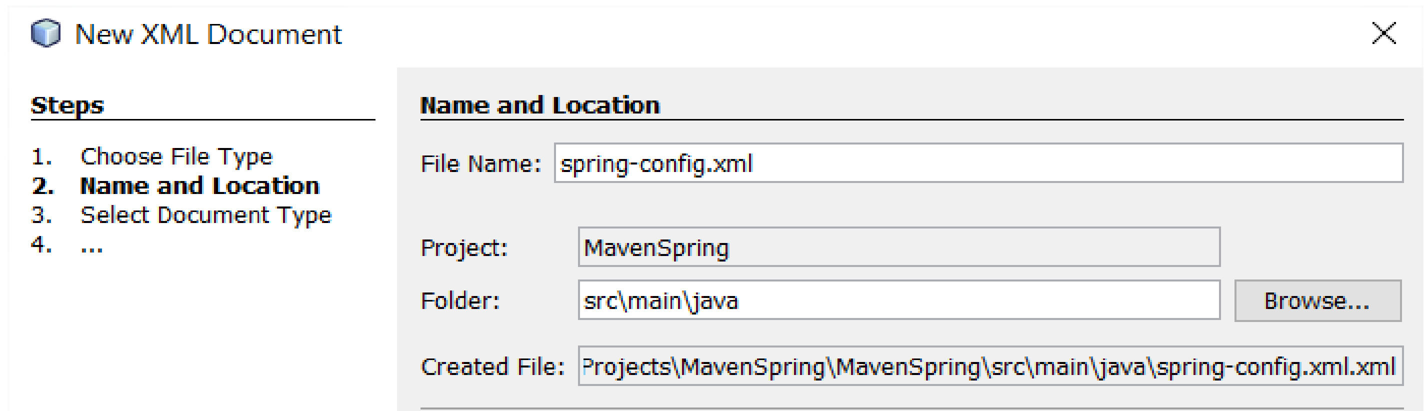
Créer le classe de démarage de type Main « **com.main.MainSpring** » pour tester le projet.
« New » → « Java Main Class » → « **com.main.MainSpring**».



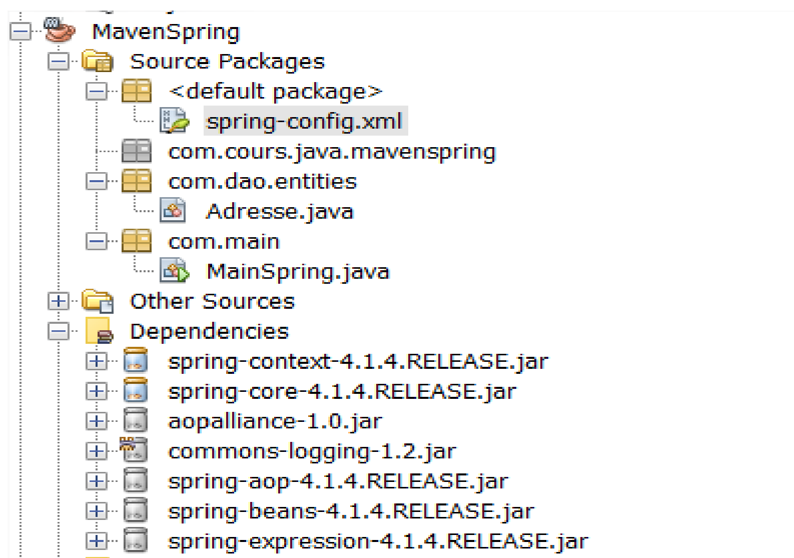


Créer le fichier XML « **spring-config**» :
« Clic droit sur **Source Packages** » → « New » → « Other » → « XML » → « XML Document » → « **spring-config**»





On obtient au final le projet :



Le fichier « **spring-config.xml** » a pour contenu :

```
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-4.0.xsd">
6
7       <bean id="myAdresseBean" class="com.dao.entities.Adresse"/>
8   </beans>
9
10
```

En version copiable :

```
<?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-4.0.xsd">

  <bean id="myAdresseBean" class="com.dao.entities.Adresse"/>
</beans>
```

Ligne 5 : Lien vers le fichier **xsd** de la version **4** du framework Spring. Ce fichier aura pour role de valider le contenu de notre fichier « **spring-config.xml** » et en occurrence la déclaration des JavaBean. Ce fichier **xsd** est obligatoire pour le bon fonctionnement du Framework **Spring**.

Ligne 7 : Declaration d'un bean de nom « myAdresseBean » et de type « com.dao.entities.Adresse ». Le nom associe à ce bean nous permettra dans un programme specifique de recuperer des instances de la classe « com.dao.entities.Adresse ».

La Classe « **com.dao.entities.Adresse** » aura pour contenu :

```
package com.dao.entities;

public class Adresse {

  private int idAdresse = 100;
   private String numeroRue = "11";
   private String nomRue = "rue de Nantes";
   private String codePostal = "53000";
   private String ville = "Laval";
   private String pays = "France";

   public Adresse() {
   }
   public Adresse(String numeroRue, String nomRue, String codePostal, String ville, String pays) {
      this.numeroRue = numeroRue;
      this.nomRue = nomRue;
      this.codePostal = codePostal;
      this.ville = ville;
      this.pays = pays;
   }
   public Adresse(int idAdresse, String numeroRue, String nomRue, String codePostal, String ville, String pays) {
      this.idAdresse = idAdresse;
      this.numeroRue = numeroRue;
      this.nomRue = nomRue;
      this.codePostal = codePostal;
      this.ville = ville;
      this.pays = pays;
   }
   public int getIdAdresse() {
      return this.idAdresse;
   }
   public String getNumeroRue() {
      return this.numeroRue;
   }
   public void setNumeroRue(String numeroRue) {
      this.numeroRue = numeroRue;
   }
   public String getNomRue() {
      return this.nomRue;
   }
   public void setNomRue(String nomRue) {
      this.nomRue = nomRue;
   }

   public String getCodePostal() {
      return this.codePostal;
   }
```

```java
    public void setCodePostal(String codePostal) {
        this.codePostal = codePostal;
    }
    public String getVille() {
        return this.ville;
    }
    public void setVille(String ville) {
        this.ville = ville;
    }
    public String getPays() {
        return this.pays;
    }
    public void setPays(String pays) {
        this.pays = pays;
    }

    @Override
    public String toString() {
        return String.format("[numeroRue=%s, nomRue=%s,codePostal=%s,ville=%s,pays=%s]", numeroRue, nomRu
e, codePostal, ville, pays);
    }
    @Override
    public int hashCode() {
        return this.getIdAdresse();
    }
    @Override
    public boolean equals(Object o) {
        if (o == null) {
            return false;
        }
        if (o instanceof Adresse) {
            return ((Adresse) o).getIdAdresse() == getIdAdresse();
        }
        return false;
    }
}
```

La Classe de démarage « **com.main.MainSpring** » aura pour contenu :

```
1   package com.main;
2
3   import com.dao.entities.Adresse;
4   import org.springframework.context.ApplicationContext;
5   import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7   public class MainSpring {
8       public static String className = MainSpring.class.getName();
9       public static void main(String[] args) {
10          String methodName = "main";
11          ApplicationContext ctx = new ClassPathXmlApplicationContext("file:src/main/java/spring-config.xml");
12          Adresse adresse = (Adresse) ctx.getBean("myAdresseBean");
13          System.out.println(className + " --> " + methodName + ", idAdresse: " + adresse.getIdAdresse() + ", nomRue: " + adresse.getNomRue()
14          + ", ville: " + adresse.getVille() + ", codePostal: " + adresse.getCodePostal() + ", pays: " + adresse.getPays());
15      }
16  }
17
```

En version copiable :

```
package com.main;

import com.dao.entities.Adresse;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainSpring {
   public static String className = MainSpring.class.getName();
   public static void main(String[] args) {
      String methodName = "main";
      ApplicationContext ctx = new ClassPathXmlApplicationContext("file:src/main/java/spring-config.xml");
      Adresse adresse = (Adresse) ctx.getBean("myAdresseBean");
      System.out.println(className + " --
> " + methodName + ", idAdresse: " + adresse.getIdAdresse() + ", nomRue: " + adresse.getNomRue()
      + ", ville: " + adresse.getVille() + ", codePostal: " + adresse.getCodePostal() + ", pays: " + adresse.getPays());
   }
}
```

Ligne 25 : recupération et chargement du fichier « **spring-config.xml** » dans le contexte d'application de Spring.
Ligne 26 : instanciation et recupération du JavaBean « **myAdresseBean** » à partir du contexte d'application de Spring.

On obtient à donc à l'éxécution :

```
com.main.MainSpring --> main, idAdresse: 100, nomRue: rue de Nantes, ville: Laval, codePostal: 53000, pays: France
---------------------------------------------------------------------
```

On remarque bien dans les logs : com.main.MainSpring --> main, idAdresse: 100, nomRue: rue de Nantes, ville: Laval, codePostal: 53000, pays: France
Donc on a bien instancier et recuperer notre JavaBean « com.dao.entities.Adresse ».

Vous pouvez trouver bizarre le fait d'instancier un objet Java sans voir de de new Adresse ( ) mais comme **Spring** ne sais pas par avance quel classe vous allez instancier alors il ne peut utiliser que la réflexivité ou la rétro-inspection.

Voici ci-dessous un exemple de rétro-inspection avec le JavaBean « com.dao.entities.Adresse » :

```
1   public static void reflectAdresseExample() {
2       String methodName = "reflectAdresseExample";
3       try {
4           // Methode 1: Instanciation standard avec le mot clef new.
5           Adresse myStandardAdresse = new Adresse();
6
7           // Methode 2: Instanciation par reflexivite.
8           Adresse myReflectAdresse = null;
9           Class myClass = Class.forName("com.dao.entities.Adresse");
10          Object objectMyAdresse = myClass.newInstance();
11          if (objectMyAdresse instanceof Adresse) {
12              myReflectAdresse = (Adresse) objectMyAdresse;
13          }
14
15          System.out.println(className + " --> " + methodName + ", myStandardAdresse: " + myStandardAdresse);
16          System.out.println(className + " --> " + methodName + ", myReflectAdresse: " + myReflectAdresse);
17      } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
18          System.out.println("Erreur lors de l'execution de la methode , Exception: " + ex);
19      }
20  }
21
```

En version copiable :

```
public static void reflectAdresseExample() {
  String methodName = "reflectAdresseExample";
  try {
    // Methode 1: Instanciation standard avec le mot clef new.
    Adresse myStandardAdresse = new Adresse();

    // Methode 2: Instanciation par reflexivite.
    Adresse myReflectAdresse = null;
    Class myClass = Class.forName("com.dao.entities.Adresse");
    Object objectMyAdresse = myClass.newInstance();
    if (objectMyAdresse instanceof Adresse) {
      myReflectAdresse = (Adresse) objectMyAdresse;
    }

    System.out.println(className + " --> " + methodName + ", myStandardAdresse: " + myStandardAdresse);
    System.out.println(className + " --> " + methodName + ", myReflectAdresse: " + myReflectAdresse);
  } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
    System.out.println("Erreur lors de l'execution de la methode , Exception: " + ex);
  }
}
```

A l'appel de la méthode **reflectAdresseExample** dans le main, on obtient :

```
com.main.MainSpring --> reflectAdresseExample, myStandardAdresse: [numeroRue=11, nomRue=rue de Nantes,codePostal=53000,ville=Laval,pays=France]
com.main.MainSpring --> reflectAdresseExample, myReflectAdresse: [numeroRue=11, nomRue=rue de Nantes,codePostal=53000,ville=Laval,pays=France]
-----------------------------------------------------------------------
```

Les methodes **1** et **2** renvois exactement la même chose. Pour plus de details sur la réflexivité cf. **Cours-Java-Introduction**.
Spring offre aussi la possibilité de modifier les attributs du bean « com.dao.entities.Adresse ».

Le fichier « **spring-config.xml** » peut donc devenir :

```
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-4.0.xsd">
6
7      <bean id="myAdresseBean" class="com.dao.entities.Adresse">
8          <property name="numeroRue" value="22" />
9          <property name="nomRue" value="rue de Nantes Bis" />
10         <property name="codePostal" value="53000 Bis" />
11         <property name="ville" value="Laval Bis" />
12         <property name="pays" value="France Bis" />
13     </bean>
14 </beans>
```

En version copiable :

```
<?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-4.0.xsd">

  <bean id="myAdresseBean" class="com.dao.entities.Adresse">
    <property name="numeroRue" value="22" />
    <property name="nomRue" value="rue de Nantes Bis" />
    <property name="codePostal" value="53000 Bis" />
    <property name="ville" value="Laval Bis" />
    <property name="pays" value="France Bis" />
  </bean>
</beans>
```

Ligne 8-12 : Modification des information contenu dans les attributs du JavaBean « **myAdresseBean**».

On obtient à l'execution du programme :

```
com.main.MainSpring --> main, idAdresse: 100, nomRue: rue de Nantes Bis, ville: Laval Bis, codePostal: 53000 Bis, pays: France Bis
-----------------------------------------------------------------------
```

Le framework Spring offre aussi la possibilité d'injecter un JavaBean dans un autre JavaBean.
Soit la classe « **com.dao.entities.Utilisateur** » dont le contenu est le suivant :

```
package com.dao.entities;

public class Utilisateur {
   private int idUtilisateur = 10;
   private String prenom = "Jack";
   private String nom = "Dupont";
   private Adresse adresse = null;

   public Utilisateur () {
   }
   public Utilisateur (String prenom, String nom, Adresse adresse) {
      this.prenom = prenom;
      this.nom = nom;
      this.adresse = adresse;
   }
   public Utilisateur (int idUtilisateur, String prenom, String nom, Adresse adresse) {
      this.idUtilisateur = idUtilisateur;
      this.prenom = prenom;
      this.nom = nom;
      this.adresse = adresse;
   }
   public int getIdUtilisateur() {
      return this.idUtilisateur;
   }
   public String getPrenom() {
      return this.prenom;
   }
   public void setPrenom(String prenom) {
      this.prenom = prenom;
   }
   public String getNom() {
      return this.nom;
   }
   public void setNom(String nom) {
      this.nom = nom;
   }
   public Adresse getAdresse() {
      return this.adresse;
   }
   public void setAdresse(Adresse adresse) {
      this.adresse = adresse;
   }
   @Override
   public String toString() {
      return String.format("[idUtilisateur=%s, prenom=%s,nom=%s,adresse=%s]", idUtilisateur, prenom, nom, adresse);
   }
```

```
    @Override
    public int hashCode() {
        return this.getIdUtilisateur();
    }
    @Override
    public boolean equals(Object o) {
        if (o == null) {
            return false;
        }
        if (o instanceof Utilisateur) {
            return ((Utilisateur) o).getIdUtilisateur() == getIdUtilisateur();
        }
        return false;
    }
}
```

Le fichier « **spring-config.xml** » devient donc :

```
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-4.0.xsd">
6
7       <bean id="myAdresseBean" class="com.dao.entities.Adresse"/>
8       <bean id="myUtilisateurBean" class="com.dao.entities.Utilisateur">
9           <property name="adresse" ref="myAdresseBean"/>
10      </bean>
11  </beans>
12
```

En version copiable :

```
<?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-4.0.xsd">

  <bean id="myAdresseBean" class="com.dao.entities.Adresse"/>
  <bean id="myUtilisateurBean" class="com.dao.entities.Utilisateur">
    <property name="adresse" ref="myAdresseBean"/>
  </bean>
</beans>
```

Ligne 7 : Déclaration d'un bean de nom « myAdresseBean » et de type « com.dao.entities.Adresse ».
Ligne 8 : Déclaration d'un bean de nom « myUtilisateurBean» et de type « com.dao.entities.Utilisateur».
Ligne 9 : Injection du bean « myAdresseBean » dans le bean « myUtilisateurBean» par le mot cles **ref**.

Les exemples ci-dessus permettre comprendre le fonctionnement de Spring d'une manière générale. Dans un cas de réelle cas d'utilisation (exemple : dans une implémentation **DAO**) nous aurons pour chaque entité une interface **INomDeMonEntiteDao** et sa classe d'implementation **NomDeMonEntiteDao**.
La classe **NomDeMonEntiteDao** se chargera de se connecter à une DataSource (fichier csv,fichier XML, Base de données ect…) pour recuperer les information de **MonEntite** et les renvoyer à **Spring**.

V) Application 2 : Spring avec l'ORM JPA Hibernate

Une des grandes applications du Framework Spring est le fait de gérer les transactions via une source de données.

A quoi sert le Framework Spring combinées avec l'ORM JPA Hibernate ?

1) Construire et injecter un **EntityManager** à travers un fichier de configuration que l'on peut nommer en général par **spring-dao-config.xml**.
2) Gérer les transactions de l'**EntityManager**.
3) Gérer la fermeture de ressources telles que l'**EntityManagerFactory** et l'**EntityManager**.

Considérons le programme suivant :

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MonUniteDePersistance");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();

operationNumero1();
operationNumero2();
operationNumero3();
operationNumero4();
operationNumero5();
operationNumero6();

em.getTransaction().commit();

em.close();
emf.close();
```

Avec le Framework Spring nous aurons :

```
SpringCreeMonEntityManager();
SpringDebuteUneTransaction();

operationNumero1();
operationNumero2();
operationNumero3();
operationNumero4();
operationNumero5();
operationNumero6();

SpringCommitLaTransaction();
SpringFermeLesRessources();
```

Il est donc possible de s'affranchir de la gestion des transactions avec **Spring**.
Nous allons dans un premier temps télécharger les librairies nécessaires pour l'utilisation du Framework **Spring/JPA**.

On ajoute la liste des Jar à telecharger pour utiliser **Spring/JPA**, le « **pom.xml** » de **MavenSpring** devient donc :

Modifier la section « **properties** » de votre fichier **pom.xml** en rajoutant les variables spring.version, hibernate.version et mysql.version.
Le section « **properties** » devient :

```
<properties>
      ........
      <spring.version>4.1.4.RELEASE</spring.version>
      <hibernate.version>4.0.1.Final</hibernate.version>
      <mysql.version>5.1.41</mysql.version>
</properties>
```

Modifier la section « **dependencies** » du **pom.xml** en mettant :

```
  <dependencies>
    <!-- Spring dependencies -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>${spring.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>${spring.version}</version>
    </dependency>
     <!-- Hibernate dependencies -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
     <!-- MySql dependencies -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysql.version}</version>
    </dependency>
  </dependencies>
</project>
```

Créer la classe « **com.dao.entities.Personne**» (généré normalement par l'ORM JPA Hibernate mais dans ce cas précis un simple copier coller de la classe fera l'affaire) dont le contenu est le suivant :

```
package com.dao.entities;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Version;
import javax.xml.bind.annotation.XmlRootElement;


@Entity
@Table(name = "personne")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Personne.findAll", query = "SELECT p FROM Personne p"),
    @NamedQuery(name = "Personne.findByIdPersonne", query = "SELECT p FROM Personne p WHERE p.idPers
onne = :idPersonne"),
    @NamedQuery(name = "Personne.findByPrenom", query = "SELECT p FROM Personne p WHERE p.prenom =
:prenom"),
    @NamedQuery(name = "Personne.findByNom", query = "SELECT p FROM Personne p WHERE p.nom = :nom"
),
    @NamedQuery(name = "Personne.findByPoids", query = "SELECT p FROM Personne p WHERE p.poids = :poi
ds"),
    @NamedQuery(name = "Personne.findByRue", query = "SELECT p FROM Personne p WHERE p.rue = :rue"),
    @NamedQuery(name = "Personne.findByVille", query = "SELECT p FROM Personne p WHERE p.ville = :ville")
,
    @NamedQuery(name = "Personne.findByCodePostal", query = "SELECT p FROM Personne p WHERE p.codeP
ostal = :codePostal"),
    @NamedQuery(name = "Personne.findByVersion", query = "SELECT p FROM Personne p WHERE p.version = :
version"),
    @NamedQuery(name = "Personne.findByPoidsInf", query = "SELECT p FROM Personne p WHERE p.poids < :p
oids"),
    @NamedQuery(name = "Personne.findByPoidsSup", query = "SELECT p FROM Personne p WHERE p.poids > :
poids"),
    @NamedQuery(name = "Personne.findByPoidsInfSup", query = "SELECT p FROM Personne p WHERE p.poids
> :poidsInf AND p.poids < :poidsSup"),
    @NamedQuery(name = "Personne.findByPrenomNom", query = "SELECT p FROM Personne p WHERE p.pren
om LIKE :prenom AND p.nom LIKE :nom")})
```

```java
public class Personne implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "IdPersonne")
    private Integer idPersonne;
    @Column(name = "Prenom")
    private String prenom;
    @Column(name = "Nom")
    private String nom;
    @Column(name = "Poids")
    private Double poids;
    @Column(name = "Rue")
    private String rue;
    @Column(name = "Ville")
    private String ville;
    @Column(name = "CodePostal")
    private String codePostal;
    @Column(name = "Version")
    @Version
    private Integer version;

    public Personne() {
    }

    public Personne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public Personne(String prenom, String nom, double poids, String rue, String ville, String codePostal) {
        this.prenom = prenom;
        this.nom = nom;
        this.poids = poids;
        this.rue = rue;
        this.ville = ville;
        this.codePostal = codePostal;
    }

    public Integer getIdPersonne() {
        return idPersonne;
    }

    public void setIdPersonne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }
```

```java
public String getPrenom() {
    return prenom;
}

public void setPrenom(String prenom) {
    this.prenom = prenom;
}

public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}

public Double getPoids() {
    return poids;
}

public void setPoids(Double poids) {
    this.poids = poids;
}

public String getRue() {
    return rue;
}

public void setRue(String rue) {
    this.rue = rue;
}

public String getVille() {
    return ville;
}

public void setVille(String ville) {
    this.ville = ville;
}

public String getCodePostal() {
    return codePostal;
}

public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
}
```

```java
    public Integer getVersion() {
        return version;
    }

    public void setVersion(Integer version) {
        this.version = version;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idPersonne != null ? idPersonne.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Personne)) {
            return false;
        }
        Personne other = (Personne) object;
        if ((this.idPersonne == null && other.idPersonne != null) || (this.idPersonne != null && !this.idPersonne.equ
als(other.idPersonne))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return String.format("[idPersonne=%s, prenom=%s, nom=%s, poids=%s, rue=%s, ville=%s, codePostal=%s]",
idPersonne, prenom, nom, poids, rue, ville, codePostal);
    }

}
```

Créer l'interface « **com.dao.service.IPersonneDao**» et sa classe d'implémentation associé « **com.dao.service.PersonneDao**».
L'interface « **com.dao.service.IPersonneDao**» aura pour contenu :

```
package com.dao.service;

import com.dao.entities.Personne;
import java.util.List;


public interface IPersonneDao {
    // créer une nouvelle Personne
    Personne create(Personne person);
    // modifier une Personne existante et la retourner
    Personne update(Personne person);
    // supprimer une Personne existante
    boolean delete(Personne person);
    // chercher une Personne particulière
    Personne find(Integer id);
    // obtenir tous les objets Personne
    List<Personne> findAll();
}
```

La classe d'implémentation « **com.dao.service.PersonneDao**» aura pour contenu :

```
package com.dao.service;

import com.dao.entities.Personne;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.transaction.annotation.Transactional;

@Transactional
public class PersonneDao implements IPersonneDao {

    @PersistenceContext
    private EntityManager em;

    @Override
    public Personne create(Personne person) {
        String methodName = "create";
        try {
            em.persist(person);
        } catch (Exception e) {
            System.out.println("Erreur lors de l'execution de la methode : " + methodName + " , Exception : " + e);
        }
        return person;
    }
```

```java
    @Override
    public Personne update(Personne person) {
        String methodName = "update";
        try {
            person = em.merge(person);
        } catch (Exception e) {
            System.out.println("Erreur lors de l'execution de la methode : " + methodName + " , Exception : " + e);
        }
        return person;
    }
    @Override
    public boolean delete(Personne person) {
        String methodName = "delete";
        boolean isOk = true;
        try {
            em.remove(em.merge(person));
        } catch (Exception e) {
            isOk = false;
            System.out.println("Erreur lors de l'execution de la methode : " + methodName + " , Exception : " + e);
        }
        return isOk;
    }

    @Override
    public Personne find(Integer id) {
        String methodName = "find";
        Personne person = null;
        try {
            person = (Personne) em.find(Personne.class, id);
        } catch (Exception e) {
            System.out.println("Erreur lors de l'execution de la methode : " + methodName + " , Exception : " + e);
        }
        return person;
    }

    @Override
    public List<Personne> findAll() {
        String methodName = "findAll";
        List<Personne> persons = null;
        try {
            persons = em.createNamedQuery("Personne.findAll").getResultList();
             // Equivalent à
            //personnes = em.createQuery("select person from Personne person").getResultList();
        } catch (Exception e) {
            System.out.println("Erreur lors de l'execution de la methode : " + methodName + " , Exception : " + e);
        }
        return persons;
    }
}
```

On retrouve dans cette classe l'annotation **@Transactional** ce qui veut dire que toutes methodes de la classe PersonneDao sont transactionnelles.
L'annotation **@ PersistenceContext** au-dessus de l'attribut « **javax.persistence.EntityManager em** » veut dire qu'un objet [EntityManager] est injecté dans le champ **em** grâce à cette annotation JPA. La classe DAO **PersonneDao** est instanciée une unique fois. C'est un singleton utilisé par tous les threads utilisant la couche JPA. Ainsi donc l'EntityManager **em** est commun à tous les threads. Sping gére aussi cette aspect de Multithreading qui est quand meme un des problèmes majeur en programmation Java.

Créer le fichier de configuration de spring « **spring-config-dao.xml** » qui aura pour contenu :

```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-2.0.xsd
    <!-- couches applicatives -->
    <bean id="personneDao" class="com.dao.service.PersonneDao" />
    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <!-- <property name="showSql" value="true" />-->
                <property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
                <property name="generateDdl" value="true" />
            </bean>
        </property>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
        </property>
    </bean>
    <!-- Simple implementation of the standard JDBC DataSource interface,
    configuring the plain old JDBC DriverManager via bean properties -->
    <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/base_personnes_jpa" />
        <property name="username" value="application" />
        <property name="password" value="passw0rd" />
    </bean>
    <!-- le gestionnaire de transactions -->
    <tx:annotation-driven transaction-manager="txManager" />
    <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
    <!-- traduction des exceptions -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
    <!-- persistence -->
    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
</beans>
```

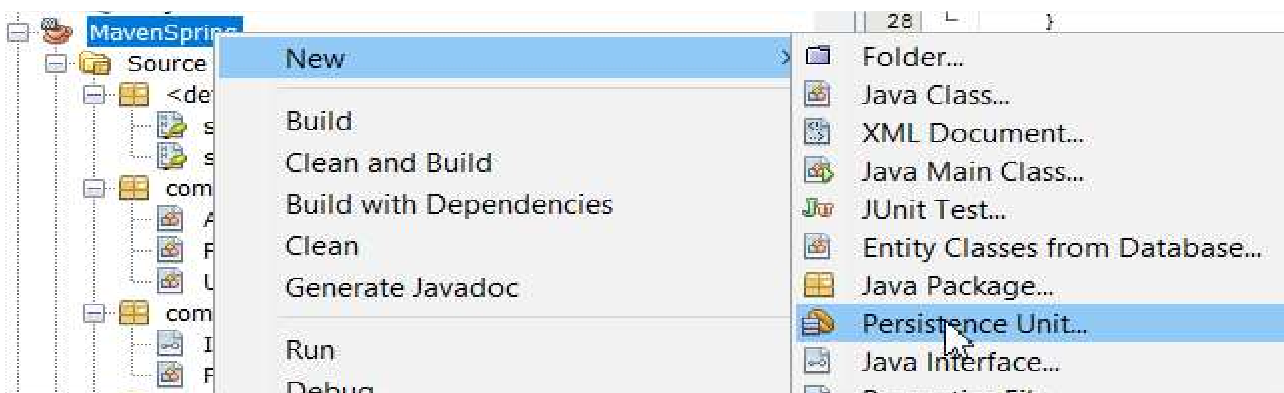
En version copiable :

```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-2.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
  <!-- couches applicatives -->
  <bean id="personneDao" class="com.dao.service.PersonneDao" />
  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <!-- <property name="showSql" value="true" />-->
        <property name="databasePlatform" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
        <property name="generateDdl" value="true" />
      </bean>
    </property>
    <property name="loadTimeWeaver">
      <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
    </property>
  </bean>
```

```
    <!-- Simple implementation of the standard JDBC DataSource interface,
    configuring the plain old JDBC DriverManager via bean properties -->
    <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/base_personnes_jpa" />
      <property name="username" value="application" />
      <property name="password" value="passw0rd" />
    </bean>
    <!-- le gestionnaire de transactions -->
    <tx:annotation-driven transaction-manager="txManager" />
    <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
    <!-- traduction des exceptions -->
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
    <!-- persistence -->
    <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
</beans>
```

Ligne 6 : Création d'un JavaBean de nom « personneDao » et de type « **com.dao.service.PersonneDao** ».
Ligne 7 : Création de « entityManagerFactory » de la librairie **JPA** ici Hibernate.
Ligne 22-27 : Configuration de la source de données qui correspond ici à une base de données MYSQL.
Ligne 29-32 : Configuration des transactions qui sont maintenant géré par **Spring**. On voit bien l'injection du bean « entityManagerFactory » précédamment crée.

Créer un fichier « **persistence.xml** » : « New » → « Persistence Unit » → « **PersonnesPU** ».



New Persistence Unit

Steps

1. Choose File Type
2. **Provider and Database**

Provider and Database

Persistence Unit Name: PersonnesPU

Specify the persistence provider and database for entity classes.

Persistence Library: Hibernate (JPA 2.1)

Database Connection:

Table Generation Strategy: (•) Create ( ) Drop and Create ( ) None

Le contenu du fichier « **persistence.xml** » peut être le suivant :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persi
3      <persistence-unit name="PersonnesPU" transaction-type="RESOURCE_LOCAL">
4          <provider>org.hibernate.ejb.HibernatePersistence</provider>
5          <class>com.dao.entities.Personne</class>
6          <properties>
7              <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/base_personnes_jpa"/>
8              <property name="javax.persistence.jdbc.user" value="root"/>
9              <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
10             <property name="javax.persistence.jdbc.password" value=""/>
11         </properties>
12     </persistence-unit>
13 </persistence>
14
```

En version copiable :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="PersonnesPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.dao.entities.Personne</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/base_personnes_jpa"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

Ligne 2 : la balise racine du fichier XML est **<persistence>**.
Ligne 3 : **<persistence-unit>** sert à définir une unité de persistance. Il peut y avoir plusieurs unités de persistance. Chacune d'elles a un nom (attribut name) et un type de transactions (attribut transaction-type). L'application aura accès à l'unité de persistance via le nom de celle-ci, ici JPA. Le type de transaction **RESOURCE_LOCAL** indique que l'application gère elle-même les transactions

avec le SGBD. Ce sera le cas ici. Lorsque l'application s'exécute dans un conteneur EJB3, elle peut utiliser le service de transactions de celui-ci. Dans ce cas, on mettra transaction-type=**JTA** (Java Transaction API).
JTA est la valeur par défaut lorsque l'attribut transaction-type est absent.

Ligne 4 : la balise **<provider>** sert à définir une classe implémentant l'interface [javax.persistence.spi.PersistenceProvider], interface qui permet à l'application d'initialiser la couche de persistance. Parce qu'on utilise une implémentation JPA / Hibernate, la classe utilisée ici est une classe d'Hibernate.

Ligne 5 : la balise **<properties>** introduit des propriétés propres au provider particulier choisi. Ainsi selon qu'on a choisi **Hibernate, Toplink, EclipseLink, Kodo**, ... on aura des propriétés différentes. Celles qui suivent sont propres à Hibernate.
Ligne 6 : l'url de la base de données utilisée
Ligne 7 : l'utilisateur de la connexion au SGBD.
Ligne 8 : le mot de passe de la connexion au SGBD.
Ligne 9 : la classe du pilote JDBC du SGBD, ici MySQL.

Nous pouvons à present tester notre nouvelle implementation à travers la methode « testSpringHibernate » de la classe « **com.main.MainSpring** » :

```
public static void testSpringHibernate() {
      String methodName = "testSpringHibernate";
      boolean isDeleteOk;
      ApplicationContext ctx = new ClassPathXmlApplicationContext("file:src/main/java/spring-config-dao.xml");
      IPersonneDao personneDao = (IPersonneDao) ctx.getBean("personneDao");
      Personne personneCRUD = new Personne("Barro", "Barro", 75, "rue des passeurs", "Laval", "53000");
      personneCRUD = personneDao.create(personneCRUD);
      personneCRUD.setPrenom("Barro Bis");
      personneCRUD.setNom("Barro Bis");
      personneCRUD = personneDao.update(personneCRUD);
      isDeleteOk = personneDao.delete(personneCRUD);
      System.out.println(className + " --> " + methodName + " , personnes : " + personneDao.findAll() + " , isDeleteOk : " + isDeleteOk);
}
```

VI) Application 3 : maven-personnes-jpa-spring

Nous avions précédemment créé le projet **LibraryPersonnesJPA** dans lequel nous avions la classe **ManagerHelper** qui gérait l'**EntityManager** (instanciation ressources BDD, transactions, fermeture ressources BDD), vous allez éliminer les appels de cette classe et nous gèrerons alors l'**EntityManager** avec le Framework Spring.