

JPA + RMI + Spring

Une application qui utilise la technologie des EJB offre essentiellement deux services : la persistance et l'accès à distance. Cependant, les EJB requièrent un serveur d'application lourd tel que JBoss et les applications EJB sont difficilement débogables. C'est pourquoi des frameworks comme Spring ont été créés.

Cet article montre comment on peut utiliser le framework Spring couplé avec Java Persistence Api (pour assurer la persistance), ainsi que RMI (pour assurer l'accès à distance). On obtient ainsi les mêmes services essentiels offerts par les EJB.

Matériel requis

- Spring 3.0
- Java 1.6
- Eclipse avec le plug-in WTP (ici c'est Eclipse Galileo qui a été utilisé en version JEE developer)

Les fichiers sources de l'application côté serveur sont à l'adresse :

perso.efrei.fr/~charroux/JPA+RMI+Spring/sourcesServeur

Vous pouvez récupérer toutes les librairies utiles à l'application côté serveur à l'adresse :

perso.efrei.fr/~charroux/JPA+RMI+Spring/libServeur

Décompressez le fichier compressé des librairies dans un répertoire sur votre machine.

Les fichiers sources de l'application côté client sont à l'adresse :

perso.efrei.fr/~charroux/JPA+RMI+Spring/sourcesClient

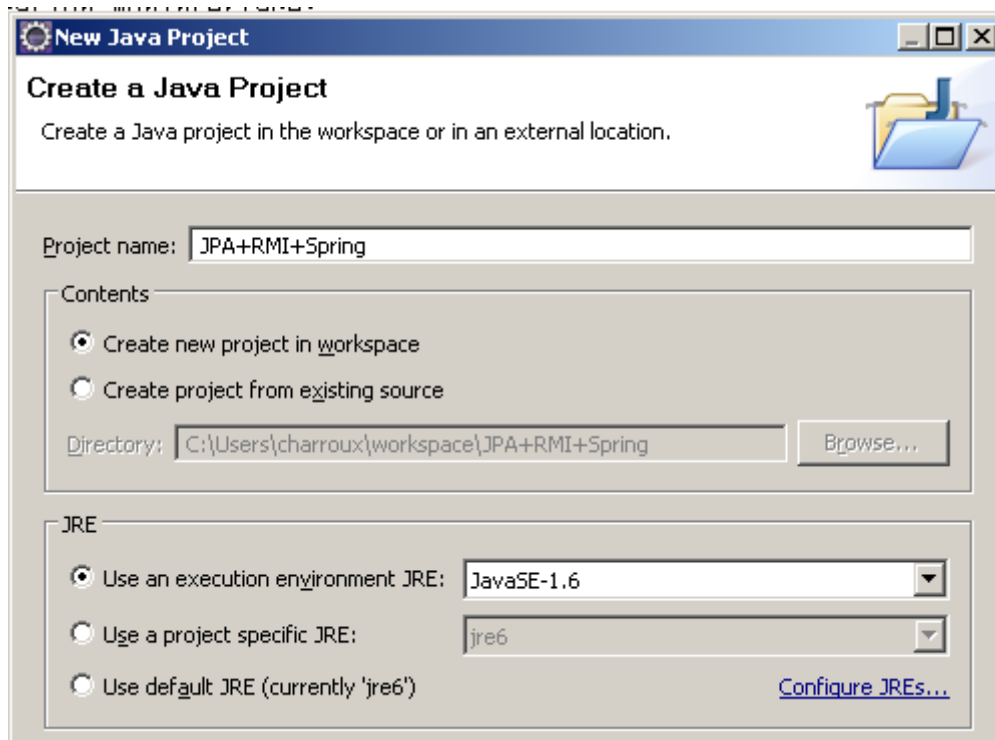
Vous pouvez récupérer toutes les librairies utiles à l'application côté client à l'adresse :

perso.efrei.fr/~charroux/JPA+RMI+Spring/libClient

Décompressez le fichier compressé des librairies dans un répertoire sur votre machine.

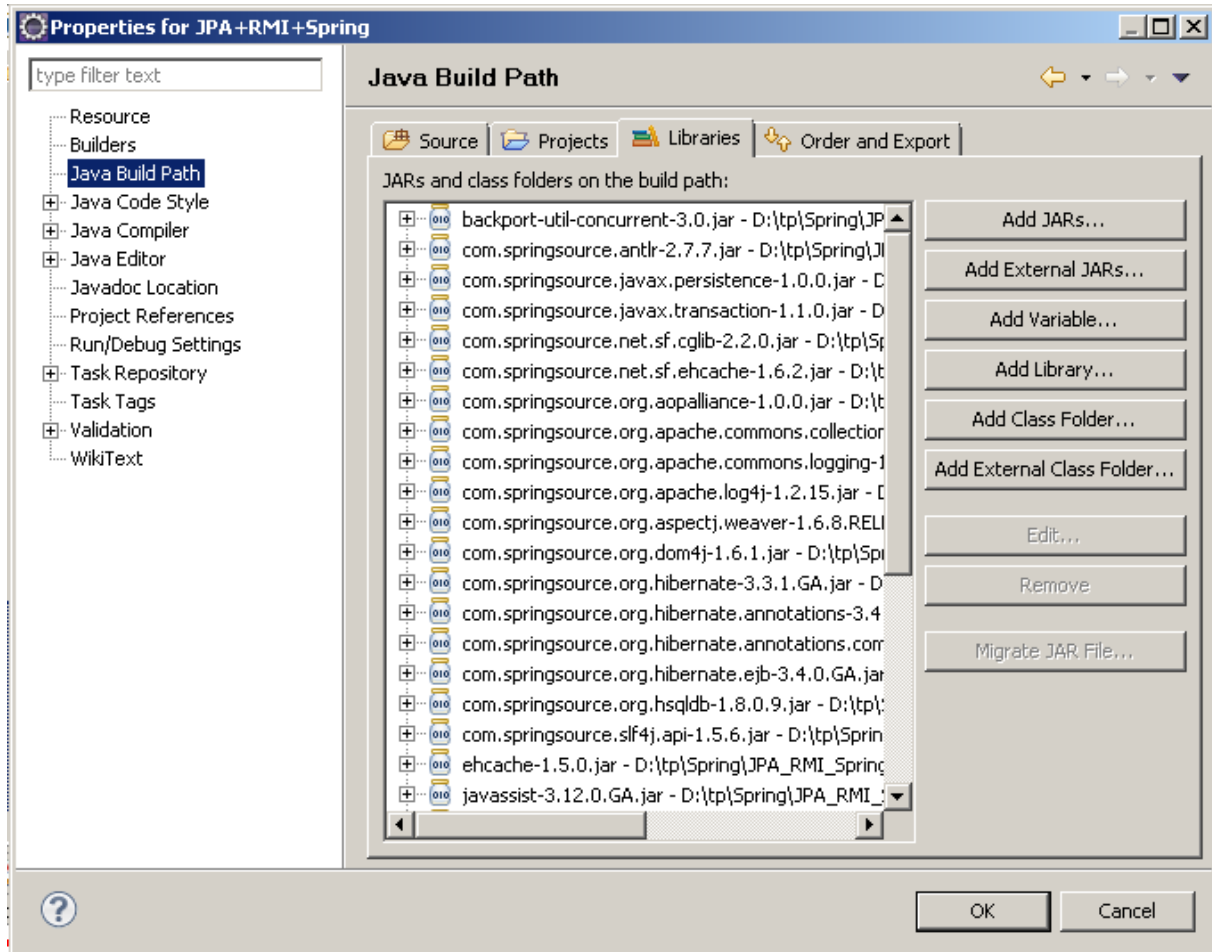
Création d'un projet côté serveur

Création d'un projet Java côté serveur :



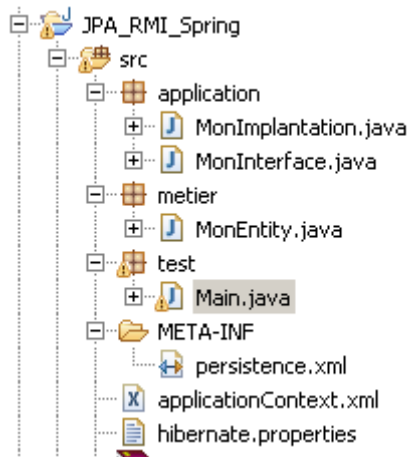
Remarque : ici, et contrairement aux EJB, il n'y a pas de dépendance à une API spécifique dans le code de l'application, car le code est écrit en Java standard.

Ajout des libraires de Spring via : clic droit sur le projet -> properties -> Java Build Path -> Libraries -> Add External JARs où les librairies sont celles que vous avez récupéré au début de cet article.



Ajout des fichiers sources

Décompressez les fichiers sources côté serveur récupérés au début de cet article dans le répertoire src de votre projet. Mettez à jour Eclipse via un F5 sur le projet. Celui-ci doit à présent se présenter comme suit :



Où le package application contient le service qui va être exposé à distance via RMI. Ce service est défini par une interface :

```
public interface MonInterface {  
  
    public void createEntity();  
  
}
```

C'est la méthode createEntity qui va être invoquée à distance.

Le package metier contient une classe persistante :

```
@Entity  
public class MonEntity {  
  
    private long id;  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    public long getId() {  
        return id;  
    }  
  
    public void setId(long id) {  
        this.id = id;  
    }  
  
}
```

C'est cette classe qui va être synchronisée avec une table dans une base de données via JPA.

Le package test contient le programme principal qui démarre le service RMI. Le package services contient quant à lui une classe qui permet de récupérer un accès au point d'entrée de l'API JPA : l'EntityManager.

Le répertoire META-INF contient le fichier persistence.xml qui déclare principalement les classes persistantes :

```
<class>metier.MonEntity</class>
```

Ainsi que la configuration de l'accès à la base de données.

Le fichier de définition des composants pour Spring est le fichier applicationContext.xml. Il contient essentiellement la définition du composant principal de l'application (le code qui va être exposé à distance via RMI) :

```
<bean id="monImplantation" class="application.MonImplantation">
    <constructor-arg ref="entityManagerFactory"/>
</bean>
```

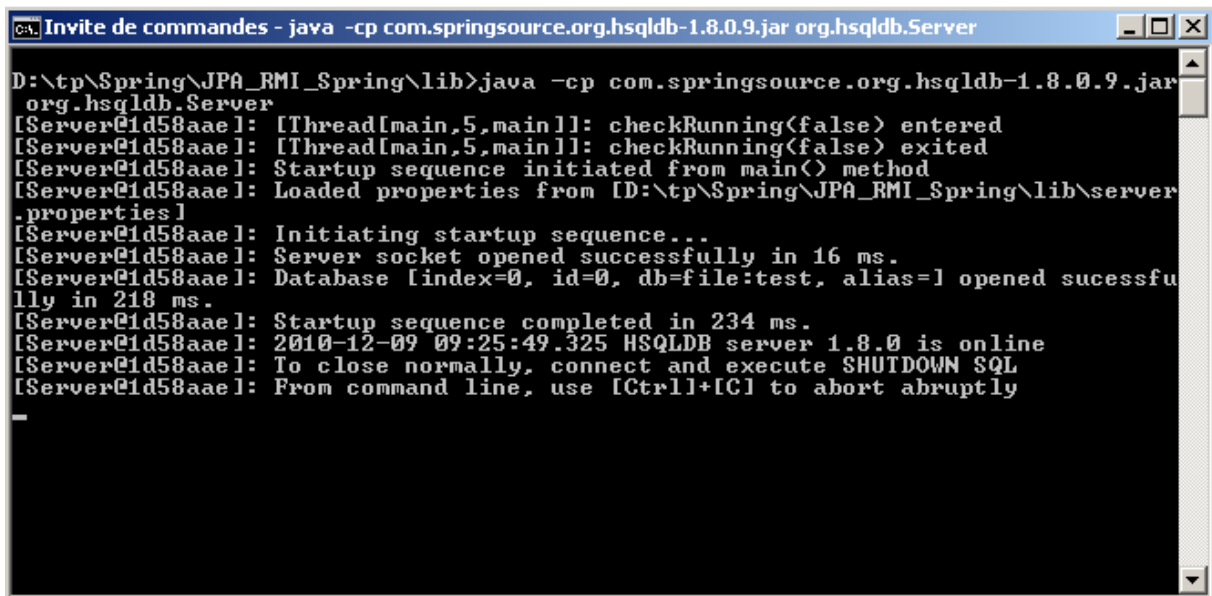
Il contient aussi un utilitaire offert par Spring pour exporter facilement un composant via RMI :

```
<bean id="rmiService"
class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="MonImplantation"/>
    <property name="service" ref="monImplantation"/>
    <property name="serviceInterface" value="application.MonInterface"/>
    <!-- defaults to 1099 -->
    <property name="registryPort" value="1199"/>
</bean>
```

On y voit que c'est le port 1199 qui va être utilisé pour le service d'annuaire des objets installés sur le serveur.

Lancement de l'application

Ici, c'est la base de données HSQLDB qui est utilisée. Elle se lance de la façon suivante à partir du répertoire où sont stockées les librairies :



```

C:\> Invite de commandes - java -cp com.springsource.org.hsqldb-1.8.0.9.jar org.hsqldb.Server
D:\tp\Spring\JPA_RMI_Spring\lib> java -cp com.springsource.org.hsqldb-1.8.0.9.jar
org.hsqldb.Server
[Server@1d58aae]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@1d58aae]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@1d58aae]: Startup sequence initiated from main() method
[Server@1d58aae]: Loaded properties from [D:\tp\Spring\JPA_RMI_Spring\lib\server
.properties]
[Server@1d58aae]: Initiating startup sequence...
[Server@1d58aae]: Server socket opened successfully in 16 ms.
[Server@1d58aae]: Database [index=0, id=0, db=file:test, alias=] opened successfu
lly in 218 ms.
[Server@1d58aae]: Startup sequence completed in 234 ms.
[Server@1d58aae]: 2010-12-09 09:25:49.325 HSQLDB server 1.8.0 is online
[Server@1d58aae]: To close normally, connect and execute SHUTDOWN SQL
[Server@1d58aae]: From command line, use [Ctrl]+[C] to abort abruptly

```

Un utilitaire pour inspecter la base peut aussi être démarré :

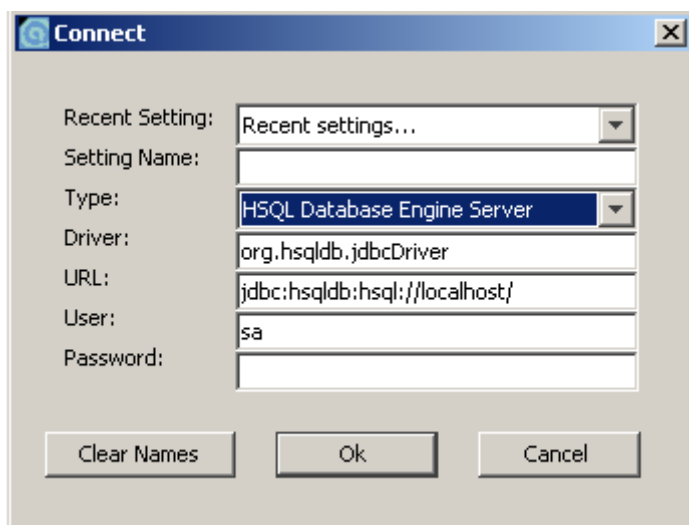


```

C:\> Invite de commandes - java -cp com.springsource.org.hsqldb-1.8.0.9.jar org.hsqldb.util.DatabaseMan...
D:\tp\Spring\JPA_RMI_Spring\lib> java -cp com.springsource.org.hsqldb-1.8.0.9.jar
org.hsqldb.util.DatabaseManagerSwing
Failed to load preferences. Proceeding with defaults:

```

Attention ! Pour accéder à la base, il faut s'y connecter via le serveur :



Connect

Recent Setting: Recent settings...

Setting Name:

Type: HSQL Database Engine Server

Driver: org.hsqldb.jdbcDriver

URL: jdbc:hsqldb:hsqldb://localhost/

User: sa

Password:

Clear Names Ok Cancel

Le démarrage de l'application se fait en lançant le programme de la classe test.Main :

```
package test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.remoting.rmi.RmiServiceExporter;

public class Main {

    public static void main(String[] args) {

        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        RmiServiceExporter rmiServiceExporter =
(RmiServiceExporter) context.getBean("rmiService");

    }

}
```

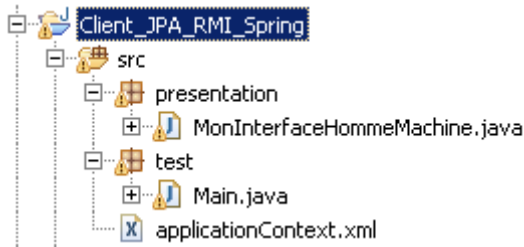
Où on demande à Spring le composant qui exporte le service via RMI.

Un serveur RMI est à présent démarré. L'annuaire des services RMI est accessible via le port 1199.

Création d'un projet côté client

Là encore un simple projet Java suffit avec les librairies ainsi que les fichiers sources spécifiques au client.

Une fois le projet créé, il ressemble à cela sous Eclipse :



La classe `MonInterfaceHommeMachine` contient l'interface utilisateur codé avec Java Swing :

```
public class MonInterfaceHommeMachine extends JDialog implements
ActionListener{

    MonInterface monInterface;

    public MonInterface getMonInterface() {
        return monInterface;
    }

    public void setMonInterface(MonInterface monInterface) {
        this.monInterface = monInterface;
    }

    // ...
}
```

On y voit le référence vers l'interface contenant la méthode à appeler à distance. Cette référence est initialisée par Spring à partir des informations données dans le fichier `applicationContext.xml` :

```
<bean id="monInterfaceHommeMachine"
class="presentation.MonInterfaceHommeMachine">
    <property name="monInterface" ref="monInterface"/>
</bean>

<bean id="monInterface"
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl"
value="rmi://localhost:1199/MonImplantation"/>
    <property name="serviceInterface" value="application.MonInterface"/>
</bean>
```

On y voit comment utiliser l'utilitaire offert par Spring qui permet d'utiliser RMI.

Le client doit accéder à l'interface du service distant pour invoquer les méthodes :

benoit.charroux@efrei.fr

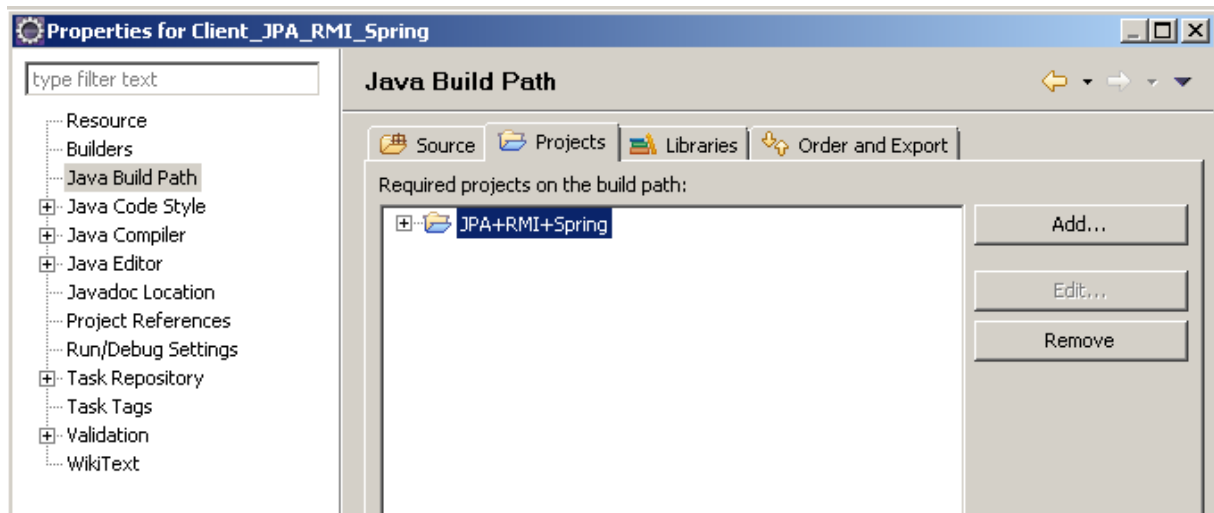
```
package application;

public interface MonInterface {

    public void createEntity();

}
```

Or, cette interface a été définie côté serveur. Il faudrait transmettre au projet client un fichier jar contenant cette interface. Mais comme ici les projets client et serveur sont développés avec Eclipse, il suffit de donner au client un accès aux programmes du serveur.



Le lancement du programme se fait en exécutant le programme principal de la classe Main.

Conclusion

Tentons de comparer à présent les EJB avec l'utilisation de Spring.

Les EJB sont simples à programmer mais le code dépend de l'API EJB. Le code utilisé par Spring est tout aussi simple et en plus il s'écrit en Java standard. Un point pour Spring donc.

La configuration de la persistance est simple avec les EJB couplés à un serveur d'application comme JBoss : un seul fichier persistence.xml est nécessaire. Cette simplicité est à tempérer car, pour mettre un serveur d'application comme JBoss en production et ne pas se contenter d'un serveur de développement, il y a une réelle complexité de configuration. Spring quant à lui n'intervient pas pour la persistance puisque c'est JPA qui est utilisé ici. La configuration de JPA est, du moins dans une configuration de développement, un peu plus complexe qu'avec les EJB : le fichier persistence.xml est plus complexe. Disons donc, 1 point pour les EJB tant qu'on reste dans une configuration de développement.

Pour l'accès à distance, il n'y a pas photo à l'arrivée : les EJB intègrent cela car il suffit d'utiliser des annotations telles que Remote pour que du code d'accès à distance soit généré automatiquement. Spring quant à lui propose des classes utilitaires pour simplifier l'utilisation de RMI. L'accès à distance n'est donc pas pris en charge à partir de simples annotations. Ou

alors il fait utiliser JAX WS mais c'est une autre histoire car il s'agit alors d'utiliser des Web Services. Un point donc pour EJB, du moins en comparaison de Spring + RMI.

Cependant, l'inconvénient majeur des EJB et qu'ils requièrent un gros serveur d'application et qu'ils sont très difficilement débogables vus leur haut niveau d'abstraction. Pour ce dernier point donc, un avantage majeur à Spring.