**Cours 29 janvier 2018**

**. WEB Services & JPA .**

Professeur : Sébastien Péricard

Services distribuées, interopérabilité, standardisation.  
Standard basé sur le XML.  
L’interface est fournie à l’application client, dans un format standard le wsdl.

Dans un service web, le type (interface exposée) est définit par un port.

WSDL :  
- objets manipulés par le service, sous forme de  
- méthode : input et output.  
- type (interface) : Port  
- location : protocole : //IP :Port/Nom 🡪 http et rpc.

La communication d’un webservice doit être standard.

Dans une entreprise, base de données UDDI qui sont une sorte de DNS pour l’entreprise, et stockent les wsdl et wsdl2 *(pour pouvoir les localiser et leur faire appel, si j’ai bien compris)*.

Sérialisation des objets d’un côté, désérialisation de l’autre ; et pour la réponse idem : sérialisation pour l’émetteur, désérialisation pour le récepteur.

Message : un entête et un corps.

Langage SOAP : [clic](https://openclassrooms.com/courses/les-services-web)  
Le message SOAP permet les messages, permet l’interopérabilité entre les applications.

Sérialisation : [clic](https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/serialization/)

**Services REST :**  
Contrairement à un service SOAP, pas d’API : on ne sait pas ce qu’il y a derrière (pas de descripteur).

Il est basé sur le http.  
Il dit « si je reçois telle requête <http://IP:port/App/traitement1>, alors je vais effectuer telle méthode » : association URL / méthode, à un URL donné est associée une méthode.

Ici, on fournit juste une doc associant les URL aux méthodes associées.

Méthodes pour contacter un service REST, pour effectuer le type de traitement que l’on veut effectuer :  
GET  
POST  
PUT  
DELETE  
OPTIONS  
…

Les applications mobiles utilisent les services REST  
Le traitement des requêtes est fait par REST : présentation vue + controller 🡪 service REST pour le traitement.

REST appelle son modèle, avec couche business et data jusqu’à la bdd.  
Le service REST intercepte toutes les requêtes, il fait office de Controller.

Dans le modèle applicatif (côté traitement), on ne sera plus que dans du service SOAP : jamais entre le business et le data.

ECMA : standardise et fait évoluer le javascript et le actionscript.

Node JS : asynchrone : utilisation des Callback

Pour faire de l’asynchrone côté serveur, a été développé en Node JS libUC, qui fournit un eventloop permettant d’empiler des traitements.

LibUC envoie à l’eventloop tout ce qui touche aux réseaux, IO :file, etc,  
Le callback arrive à la fin des événements.

Google engine V8 (moteur d’interprétation javascript)

Chromium (navigateur) ne peut pas manipuler l’OS directement, il passe donc par un serveur qui lui, peut. Il passe par Node JS, avec lequel il discute en http avec JSon.

Services web 🡪 Java EE 7

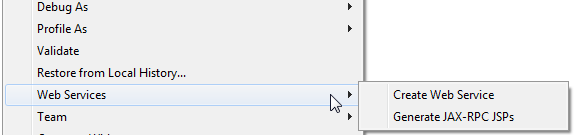
Axis intercepte les demandes d’un consommateur de service, c’est donc une servlet. Utilisée avant JSE 1.6.  
CXF permet à la lecture des annotations du standard, d’implémenter.

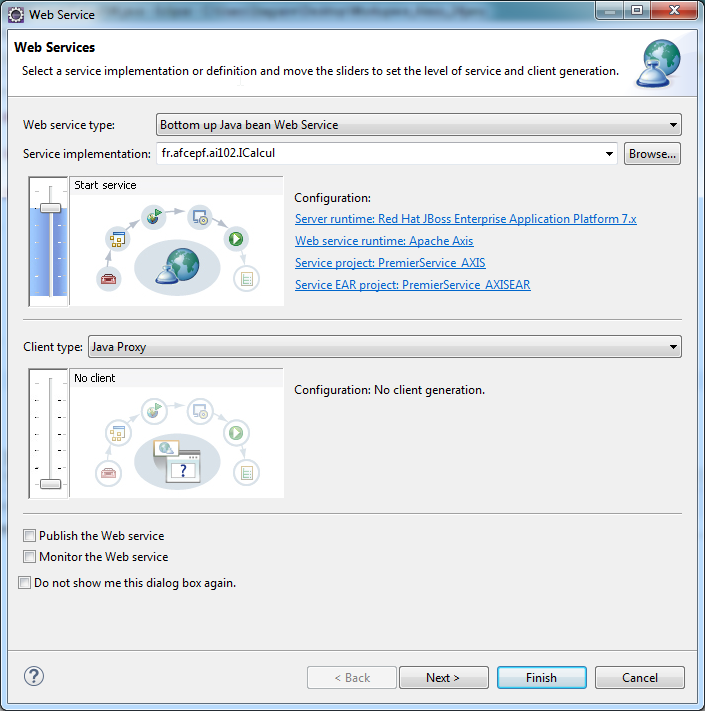
Nous feront un service AXIS et son client, un jax-ws et son client, et .Net et son client,  
Puis on mixera.

Un attribut est comme une annotation, définissant un comportement son à la compilation, soit à l’exécution du programme.

Axis est intégré à Eclipse : il peut générer la servlet.

On peut générer le webservice après clic droit sur l’implémentation :



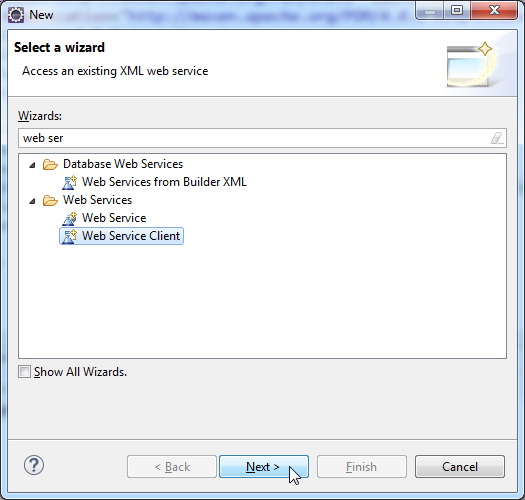


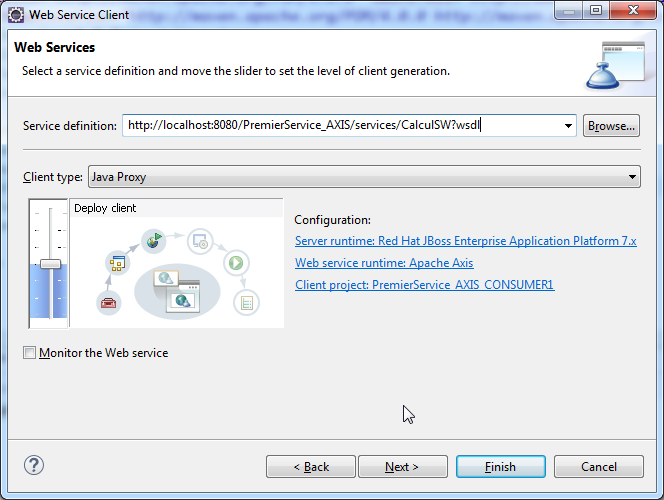
On utilise pour tester les webservices SOAPUI (java) ou Postman.

SOAPUI permet de faire beaucoup plus de choses que Postman, dont génération de tests unitaires.

Le proxy est généré à la lecture du wsdl.

Le wsdl définit ce que l’on voit de l’extérieur de l’application.  
On utilise donc le chemin donné dans le wsdl pour accéder à la ressource recherchée.

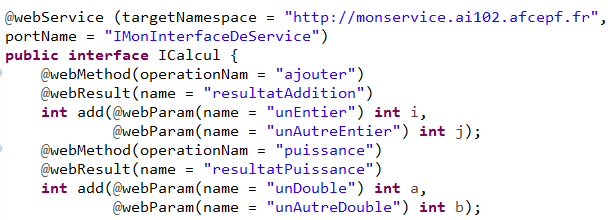




On peut utiliser PowerShell (console)



Génération avec le JDK du proxy



« Méthodes exposées »

**. JAVA PERSISTENCE API Oracle e**

Hibernate est une communauté (ORM),  
Reddhat a fait le choix d’utiliser Hibernate.

JPA fournit des **annotations et des interfaces**Implémentations grace à Reddhat 🡪 Hibernate, et EclipseLink.

Un ORM fournit un moyen de faire des associations entre tables et classes, l’ORM travaille sur la bdd que l’on peut « oublier » : il simplifie les traitements à la base ; il fournit aussi des systèmes très simplifiés pour les CRUD et propose deux types de query, le SQL natif et le HQL (Hibernate Query Language qui permet de requêter sur les propriétés de l’objet).

On a besoin pour que l’ORM fonctionne de :  
- pour le mapping : @nnotations ou fichier hbm.xml, ou orm.xml

Dans les fichiers persistence.cfg.xml (ou persistence.xml son homologue) on a besoin de paramétrer le SessionFactory (en hibernate) ou PersistenceUnit (JPA) : on y paramètre   
- le DATASOURCE (url, login, pwd, driver)  
- les classes  
- les properties : dialect, hbm2ddl.auto (import.sql) create, validate, update

En Hibernate, on a une Session : session.insert(obj) insère en base lors du commit, et récupère sa primary key PK ; hibernate ne va pas rechercher en base mais directement la PK, s’il vient d’insérer (optimisation).   
session.update(obj)  
session.delete(obj)  
sessions.get/load(type, PK)

Dans un ORM, une classe a obligatoirement une PK.

Dans le projet Maven, on crée les propriétés et dépendances :

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<oriject.build.sourceEncoding>UTF-8</oriject.build.sourceEncoding>

<hibernate.version>4.3.8.Final</hibernate.version>

<jpa.version>1.0.0.Final</jpa.version>

<log4j.version>1.2.17</log4j.version>

<mysql-connector-java.version>5.1.27</mysql-connector-java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.1-api</artifactId>

<version>${jpa.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>${hibernate.version}</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql-connector-java.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

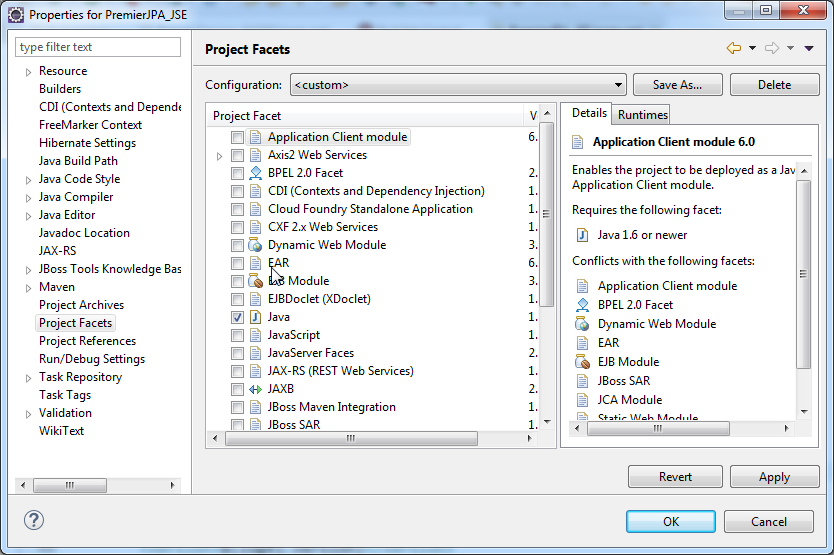
<version>${log4j.version}</version>

<scope>compile</scope>

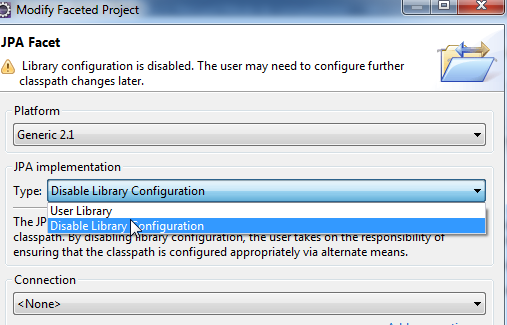
</dependency>

</dependencies>

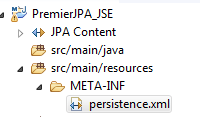
Propriétés du projet (ALTR ENTER) :



Ensuite on disable la library :



On bouge le persistence avec son META-INF dans Resources

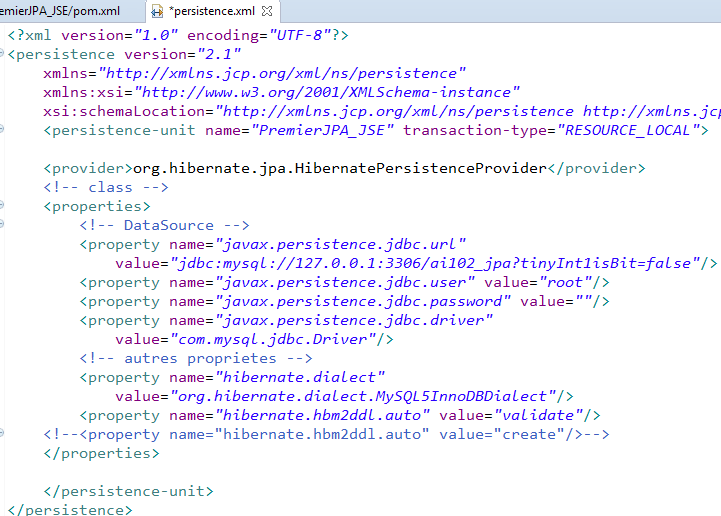


Rajout de cette ligne dans le fichier persistence :

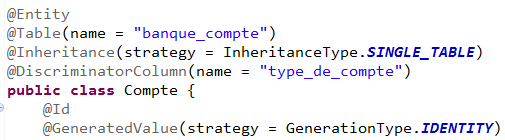
transaction-type=*"RESOURCE\_LOCAL"*

tinyInt est un bit ! Dans la connexion, pour générer de bons booleans, on doit mettre tinyInt1isBit=false.

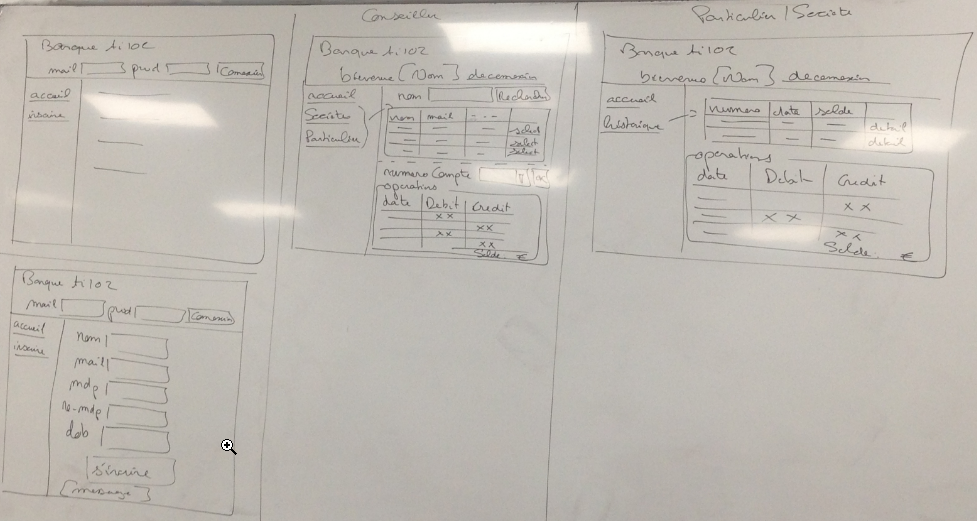
On a donc le fichier persistence :



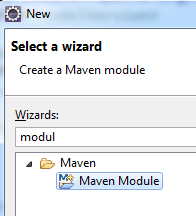
Représenter de l’héritage dans une bdd qui est relationnelle :  
- single table : mais on ne peut pas forcer à non null (justement) donc pas top  
-joined table :   
- TABLE\_PER\_CLASS



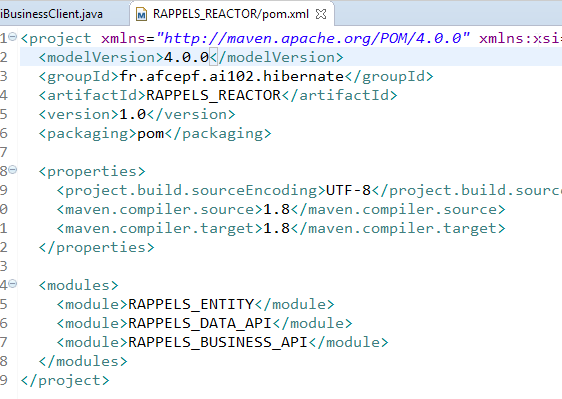
TP – Continuation – Le projets modules



Création du maven project en packaging **pom** cette fois !  
Dedans, CTRL N et ajout des MODULES qui sont en fait les petits sous projets :

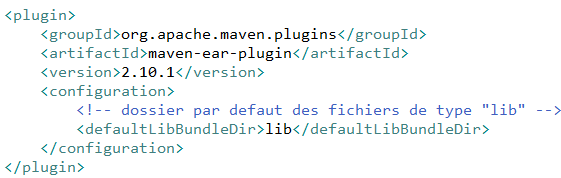


On a donc dans le projet principal ce smodules dans le pom :



Dans ce même pom, on peut rajouter un pluginManagement, sorte d’héritage pour les sous projets maven, qui « héritent » de la version définie ici



Ajout aussi de celui là :  


On se fait un petit EAR, package ear :  
On rajoute dedans les dépendances vers les aures projets, avec   
<type>jar</type>