Compléments JPA pour les serveurs d'application

Université Française d'Egypte
Version 0.8 – 13/10/12
Richard Grin

Plan du support

- Introduction
- Gestionnaire d'entités géré par le container
- Gestionnaire d'entités géré par l'application
- Types de transaction
- Conception
- Entité détachée
- Définir une source de données

Objectif du support

- Ce support complète le support sur JPA 2 avec Java SE
- Il indique les différences lorsque JPA est utilisé avec un serveur d'application, dans un environnement Java EE 6

- Dans la suite de ce support de cours, GE signifiera « gestionnaire d'entités » et CP « contexte de persistance »

Introduction

Gestionnaire d'entités dans Java SE

- Rappel : un seul type de GE :
  - créé par l'application (**`EntityManagerFactory.emf.createEntityManager()`**)
  - attaché à un même CP durant toute son existence (un seul type de CP)
  - gère les transactions (interface **`EntityTransaction`**) qui sont le plus souvent du type « locales », non JTA

Plus complexe avec Java EE

- Avec Java EE la taxinomie des gestionnaires d'entités JPA (interface **`EntityManager`**), des contextes de persistance et des transactions est plus complexe que dans Java SE

Types de GE dans un serveur d'application

- EM géré par le container (cas le plus fréquent)
  - CP délimité par les transactions (TRANSACTION)
  - CP peut survivre à une fin de transaction (EXTENDED) ; nécessite un bean session avec état (*stateful*)
- EM géré par l'application (c'est la fabrique qui est injectée) ; CP pas délimité par les transactions

Exemple – géré par le container

```
@Stateless
public class DeptFacade {
  @PersistenceContext(unitName = "employes")
  private EntityManager em;

  public void create(String nom, String lieu) {
    Dept dept = new Dept(nom, lieu);
    em.persist(dept);
    dept.setLieu("Paris");
  }
  ...
}
```

EM injecté par le container

Transaction démarrée au début de la méthode et fermée à la fin

CP ne dure que le temps de la transaction

Exemple – géré par l'application

```
@Stateless
public class DeptFacade {
  @PersistenceUnit(unitName = "employes")
  private EntityManagerFactory emf;

  public void create(String nom, String lieu) {
    EntityManager em = emf.createEntityManager();
    Dept dept = new Dept(nom, lieu);
    em.persist(dept);
    dept.setLieu("Paris");
    em.close();
  }
  ...
}
```

Fabrique d'EM injectée par le container

Choix du type d'EM

- Bien plus simple de le faire gérer par le container (propagation des transactions, pas besoin de le fermer)
- Un EM géré par l'application ne doit être utilisé que lorsque c'est nécessaire ; par exemple, avec un servlet car une fabrique d'EM est *thread-safe*, ce que n'est pas un EM
- Éviter de mélanger les 2 types de gestion

2 types de transactions

- Transactions JTA ; transactions qui peuvent englober plusieurs sources de données ; le cas de loin le plus fréquent avec Java EE
- Transactions JDBC (comme avec Java SE) ; ne peuvent concerner qu'une seule source de données ; « *resource local* » dans la spécification JPA ; très rarement utilisé avec Java EE

2 types de gestion des transactions

- Les transactions peuvent être gérées par le container ; typiquement des annotations indiquent comment les méthodes doivent se comporter pour ce qui concernent les transactions
- Les transactions peuvent être gérées par l'application (par les beans de l'application)

2 types de contexte de persistance

- Avec Java EE, le contexte de persistance peut être limité à une seule transaction (le cas le plus fréquent)
- Il peut aussi, comme pour Java SE, être « étendu », c'est-à-dire qu'il peut survivre à la fin de la transaction et donc être utilisé par plusieurs transactions ; en ce cas, l'association entre un CP et plusieurs GE est plus complexe qu'avec Java SE où il y a une bijection entre les CP et les GE

Le cas le plus fréquent

- Les transactions sont gérées par le container
- Un bean session sans état injecte dans son code un GE dont le CP est limité à une transaction

Gestionnaire d'entités géré par le container

GE fourni par injection

- Le GE est alors fourni à l'application par le container par injection de ressource (annotation ou fichier XML) ou par un lookup JNDI ; les exemples de ce cours utiliseront essentiellement les annotations
- Les cycles de vie du GE et du CP associé sont gérés par le container : le container crée et supprime le GE et le CP

Injection du GE par annotation

- Le GE est injecté par le serveur d'applications dans un EJB :

```
@PersistenceContext(unitName="Xxxxx")
EntityManager em;
```

- La durée de vie du CP associé à un tel GE est la transaction : il est fermé par le container quand une transaction se termine

Exemple

```
@Stateless
public class DeptFacade {
  @PersistenceContext(unitName = "employes")
  private EntityManager em;

  public void create(String nom, String lieu) {
    Dept dept = new Dept(nom, lieu);
    em.persist(dept);
    dept.setLieu("Paris");
  }
  ...
}
```

EM injecté par le container

Durée de vie du CP d'un GE géré par le container

- Un GE géré par le container peut avoir un CP d'un des 2 types (**`PersistanceContextType`**) suivants :
  - d'un durée de vie limitée à une transaction ; un commit ou rollback ferme le CP (**`TRANSACTION`** ; c'est le type par défaut)
  - peut survivre à une fin de transaction (**`EXTENDED`**) ; nécessite un bean session avec état (*stateful*)

CP étendu

- Annotation pour injecter un GE dont le CP peut durer pendant plusieurs transactions :

```
@PersistenceContext(unitName="Xxxxx",
   type=PersistenceContextType.EXTENDED)
EntityManager em;
```

- Ce type de GE ne peut être injecté que dans un bean session avec état (*stateful*) ; il n'est pas disponible dans un bean session sans état (*stateless*)
- Le CP est fermé à la suppression du bean session avec état (ou à l'exécution de la méthode **`@Remove`** du bean session)

Exemple

```
@Stateful
public class CaddyImpl implements Caddy {
  @PersistenceContext(type=EXTENDED)
  EntityManager em;
  ...
}
```


Utilité des CP étendus

- Ils permettent d'établir des conversations longues entre l'utilisateur et la BD, sans avoir à gérer des entités détachées (le contexte reste ouvert entre les transactions)

Opérations d'un GE et type de CP

- Dans le cas d'un contexte de persistance de portée limitée à une transaction,
  - **`persist`**, **`merge`**, **`remove`** et **`refresh`** doivent être invoquées dans le contexte d'une transaction
  - **`find`** et **`getReference`** (comme les résultats de l'exécution d'un **`Query`**) peuvent être invoqués en dehors du contexte d'une transaction mais dans ce cas les entités renvoyées sont détachées

CP et transaction JTA

- Les transactions utilisées pour les GE gérés par le container sont toutes de type JTA
- Le CP d'un GE géré par le container est attaché à la transaction JTA et il peut ainsi être utilisé par plusieurs GEs utilisé dans la même transaction

Propagation de contexte

- Quand un GE est injecté, le container regarde si un CP est attaché à la transaction
- Si c'est le cas, ce CP est utilisé par le GE, sinon le container en crée un nouveau et l'attache à la transaction

Utilité de la propagation de contexte

- Ce mécanisme évite au développeur de passer un même GE en paramètre de plusieurs méthodes appelées dans la même transaction pour que le même contexte soit utilisé pendant toute la transaction

Gestionnaire d'entités géré par l'application

GE géré par l'application

- L'application peut gérer elle-même la création d'un GE en utilisant une fabrique de GE (méthode **`createEntityManager()`**)
- La fabrique de GE, pas le GE, s'obtient par injection de ressource (fournie par le serveur d'applications)

```
@PersistenceUnit(unitName="Xxxxx")
EntityManagerFactory emf;
```

ou par la méthode **`Persistence.createEntityManagerFactory`**, ou par lookup JNDI

Exemple

Fabrique d'EM injectée par le container

```
@Stateless
public class DeptFacade {
  @PersistenceUnit(unitName = "employes")
  private EntityManagerFactory emf;

  public void create(String nom, String lieu) {
    EntityManager em = emf.createEntityManager();
    Dept dept = new Dept(nom, lieu);
    em.persist(dept);
    dept.setLieu("Paris");
    em.close();
  }
  ...
}
```

- Il est plus intéressant d'utiliser un GE géré par l'application dans un bean session avec état
- Le GE peut ainsi être utilisé par plusieurs requêtes

Exemple (1/2)

```
@Stateful
public class DeptFacade {
  @PersistenceUnit(unitName = "employes")
  private EntityManagerFactory emf;
  private EntityManager em;
  private Dept dept;

  public void init(int idDept) {
    em = emf.createEntityManager();
    dept = em.find(Dept.class, idDept);
}
```


Exemple (2/2)

```
  public void addEmploye(int empId) {
    em.joinTransaction();
    Employe emp = em.find(Employe.class, empId);
    dept.getEmployes().add(emp);
    emp.setDept(dept);
  }
  ...
  @Remove
  public void fin() {
    em.close();
  }
}
```


Transactions

- Comme avec Java SE, le CP n'est alors pas limité à une seule transaction : le GE garde le même CP pendant toute son existence
- Une requête (**`find`** ou **`query`**) peut être lancée en dehors de toute transaction, sans que les entités retrouvées soient détachées (alors que c'est le cas pour les CP limités à une transaction)

Synchronisation avec une transaction

- Si le GE est obtenu dans une transaction JTA, il est automatiquement synchronisé avec la transaction : un commit génèrera automatiquement un flush dans la base
- Sinon (GE créé avant le début de la transaction ou dans une transaction précédente qui est finie) le GE doit se synchroniser avec une transaction par la méthode **`joinTransaction()`** de **`EntityManager`**

Pas recommandé

- Il n'est pas recommandé d'utiliser les GE gérés par l'application
- Il est plus difficile de les gérer soi-même que de laisser le container le faire ; par exemple il ne faut pas oublier de les fermer et il peut être nécessaire de synchroniser avec une transaction
- Un tel contexte de persistance ne se propage pas ; pour le partager, il faut transmettre le GE (attention, un GE n'est pas « *thread-safe* »)

Types de transactions

Transactions disponibles dans Java EE

- Les serveurs d'applications utilisent par défaut les transactions JTA, mais les transactions locales à une ressource (celles qui utilisent l'API de JDBC pour les transactions) sont aussi disponibles
- Les transactions peuvent être gérées par le container du serveur d'applications (obligatoirement des transactions JTA) ou par l'application (transactions JTA ou non)

Démarcation des transactions dans Java EE

- Transactions JTA gérées par le serveur d'applications : les transactions sont démarrées (ou non) au début d'une méthode et terminées (ou non) à la fin d'une méthode, suivant les indications données par le développeur par annotation ou dans un fichier XML
- Transactions gérées par l'application : l'application décide de démarrer et de terminer une transaction n'importe où dans le code

Exemple(s)

- Des exemples sont donnés dans le cours sur les transactions des EJB

Transactions gérées par l'application dans Java EE

- Le code utilise l'interface **`javax.transaction.UserTransaction`** qui possède des méthodes pour démarrer, terminer ou marquer pour invalidation une transaction JTA (ne pas confondre avec l'interface **`EntityTransaction`** étudiée dans le cours sur JPA dans Java SE, qui concerne les transactions JDBC, non JTA)
- Une instance de **`UserTransaction`** peut être injectée par une annotation **`@Resource`**

Interface **`UserTransaction`** (1)

- **`void begin()`** : crée une transaction et l'associe avec le thread courant
- **`void commit()`** : valide la transaction associée au thread courant
- **`void rollback()`** : rollback de la transaction associée au thread courant

Interface **`UserTransaction`** (2)

- **`void setRollbackOnly()`** : après cet appel, la transaction ne pourra être qu'invalidée
- **`int getStatus()`** : donne le statut de la transaction sous la forme d'une constante de la classe **`javax.transaction.Status`**
- **`void setTransactionTimeout(int millisecondes)`** : nombre de millisecondes après lesquelles la transaction est annulée ; la valeur 0 remet la valeur par défaut (qui dépend de l'implémentation)

Transactions JDBC

- C'est rarement utile mais l'application peut aussi utiliser des transactions non JTA (simples transactions JDBC) avec l'interface **`EntityTransaction`**

2 types de transaction

- Le type des transactions d'un GE est indiqué dans la configuration de sa fabrique
- Par défaut le GE est JTA :

```
<persistence-unit name="Employes">
<jta-data-source>clients</jta-data-source>
```

- Un GE non JTA :

```
<persistence-unit name="Employes"
  transaction-type=RESOURCE_LOCAL>
<non-jta-data-source>jdbc/MaDB</jta-data-
source>
...
```


Conception

- Quelques considérations pour la conception d'une application Java EE

Environnement par défaut dans un serveur d'applications

- Contexte de persistance géré par le container (injecté par annotation ou par *lookup* JNDI)
- Contexte de persistance limité à une seule transaction
- Transaction JTA dont les démarcations sont fixées déclarativement (par annotation ou dans un fichier XML)

Un principe

- Pour que tout se passe bien, tous les appels de méthodes des GE durant une même transaction doivent utiliser le même contexte de persistance
- Sinon, des modifications effectuées sur des entités durant la transaction risquent d'être perdues par la suite dans la transaction
- JPA facilite l'application de ce principe (voir transparent suivant)

Modèle de conception dans Java EE

- La configuration par défaut correspond au modèle de conception « une transaction par requête de l'utilisateur » :
  - la transaction démarre au début du traitement d'une requête
  - et elle se termine à la fin du traitement
- Pour faciliter l'utilisation de ce modèle, le contexte de persistance peut se propager aux méthodes appelées durant toute la durée d'une transaction (détaillé plus loin)

Autre modèle de conception

- Avec un *bean* session avec état (*stateful*) il est possible d'utiliser un même contexte de persistance étendue pour plusieurs transactions JTA
- Le modèle de conception correspond à des traitements « conversationnels » qui peuvent couvrir plusieurs requêtes de l'utilisateur

Propagation de contexte

- Pour faciliter la tâche du développeur, les transactions JTA permettent la propagation de contexte pour les GE gérés par le container
- Cette propagation de contexte peut occasionner des collisions de contexte dans certaines situations dans lesquelles des beans sessions avec état sont impliqués
- L'héritage de contexte limite les possibilités de conflit

Définition

- Un contexte de persistance peut être propagé durant une transaction JTA s'il est utilisé par des GEs gérés par le container
- Ce contexte est alors utilisé à tour de rôle durant la transaction par les GEs
- Cette propagation s'appelle une propagation de contexte
- Quand un GE reçoit un message, il vérifie s'il existe un contexte propagé ; si c'est le cas, il utilise celui-ci, sinon il en crée un nouveau

Avantage de la propagation

- Les modifications non encore enregistrées dans la base (elles le seront seulement au commit de la transaction), mais enregistrées dans le contexte de persistance, sont disponibles durant toute la transaction, quel que soit l'emplacement du code placé dans la transaction

Conflit de contextes

- La propagation de contexte ne fonctionne pas lors de l'appel d'une méthode d'un *bean* session avec état (*stateful*) depuis un autre *bean*
- En effet, une *bean* session avec état ne peut travailler qu'avec un seul contexte et ne peut accepter de travailler avec un autre contexte propagé
- Une exception est levée
- Voir la spécification JPA pour plus de détails

Héritage de contexte

- Une circonstance qui peut éviter la collision de contexte : un *bean* avec état créé par un autre *bean* avec état hérite du contexte de ce *bean* si les 2 beans utilisent un contexte de persistance étendu
- Ainsi des appels de méthodes pourront se faire entre les 2 *beans* sans collision de contexte

Synchronisation

- Il est possible de synchroniser plusieurs contextes de persistance gérés par l'application avec une même transaction JTA
- Mais un seul contexte géré par le container peut être synchronisé avec une transaction JTA

Conversation longue (1)

- L'utilisation d'un contexte étendu qui n'est pas limité à une seule transaction, permet d'implémenter des conversations longues entre un client et un bean session avec état
- Il n'est pas indispensable, et souvent néfaste, qu'une seule transaction longue couvre toute la conversation

Conversation longue (2)

- On peut commencer par une transaction, valider la transaction, passer des données au client qui les modifie et ensuite reprendre une autre transaction pour terminer la conversation
- Dès que la 2ème transaction démarre, toutes les modifications faites entre les 2 transactions sont prises en compte par cette 2ème transaction

Entité détachée

Cas d'utilisation

- Une application multi-tiers
- Le serveur d'application récupère des données dans la BD
- Ces données sont passées à la couche client, montrées à l'utilisateur qui peut les modifier
- Les modifications sont repassées au serveur et enregistrées dans la BD
- Les entités détachées facilitent l'implémentation d'un tel cas d'utilisation

EJB 2.0

- Dans la norme EJB 2.0, les données de la BD étaient enregistrées dans un objet EJB entité
- Le problème était qu'un objet entité ne pouvait exister que dans l'environnement du serveur d'application
- Les données étaient donc souvent passées au client sous forme de DTO (pour éviter de nombreux appels distants aux EJB entités, qui sont coûteux)

EJB 3.0

- Plus ce problème dans la norme EJB 3.0
- Les entités peuvent être détachées de leur contexte de persistance d'origine et rattachées par la suite

EJB 3.0

- Les entités sont détachées quand elles sont passées à la couche cliente
- La couche cliente peut modifier les entités détachées
- Ces entités peuvent ensuite être rattachées à un GE et les modifications effectuées dans la couche cliente peuvent être alors enregistrées dans la base de données lors d'un flush

Rattachement

- La méthode **`merge`** de **`EntityManager`** permet d'obtenir une entité gérée à partir d'une entité détachée
- Sa signature :
  `<T> T merge(T entité)`
- Attention, l'entité passée en paramètre n'est pas rattachée ; c'est l'entité renvoyée par la méthode **`merge`** qui est rattachée ; cette entité a le même état et la même clé primaire que l'entité passée en paramètre

- Si l'entité passée en paramètre de merge est déjà gérée par le GE, elle est renvoyée par la méthode merge

Exemples de détachement

- Un servlet peut décider d'enregistrer des entités dans une session d'utilisateur ; cette session peut être sérialisée par un conteneur de servlet (et les entités sont alors détachées)
- Une application peut éviter l'utilisation de DTO en transférant les données entre les clients et le serveur dans des objets détachés, évitant ainsi des recopies inutiles

État d'une entité détachée

- Lorsqu'une entité est détachée, la synchronisation avec la BD n'est pas garantie
- De plus, l'état d'une entité détachée peut ne pas être entièrement disponible

État d'une entité

- Pour des raisons de performances, l'état d'une entité gérée par un GE peut ne avoir été complètement récupéré dans la BD (récupération retardé ; *lazy*)
- Le reste de l'état ne sera récupéré, avec l'aide du GE, que lorsque l'entité en aura vraiment besoin
- Si l'entité est détachée alors qu'une partie de son état n'a pas encore été récupérée, la partie manquante de l'entité détachée ne sera pas disponible

Attribut disponible

- Un attribut persistant d'une entité est immédiatement disponible dans les 2 cas suivants :
  - l'attribut a déjà été utilisé
  - l'attribut n'a pas été marqué par **`fetch=LAZY`**
    (par défaut, les valeurs des attributs sont chargées en mémoire)

Association d'une entité détachée

- Si une association d'un objet détaché a le mode de récupération LAZY, il est possible que l'association ne puisse être récupérée (dépend des circonstances et des fournisseurs de JPA)
- En ce cas, une fois détachée et loin du GE, la norme n'assure pas que l'entité puisse récupérer l'état manquant

Association « lazy » disponible

- Si une association d'un objet détaché a le mode de récupération LAZY, il est possible de naviguer à travers cette association si
  - l'application a déjà effectué cette navigation alors que l'entité était attachée
  - ou si l'association a été chargée par un **`join fetch`** lors d'une requête

Rendre accessible l'état d'une entité détachée

- 2 solutions pour accéder à tout l'état d'une entité détachée :
  - L'application récupère tout l'état de l'entité gérée avant le détachement
  - L'application rattache l'entité par la méthode **`merge`** pour récupérer l'état manquant

Récupérer une association avant le détachement

- Si le but est une entité (OneToOne ou ManyToOne), il suffit de lire une des propriétés de l'entité
- Si le but est une collection (OneToMany ou ManyToMany), l'appel de la méthode **`size()`** de la collection suffit le plus souvent

Entité détachée et concurrence

- Avant que les modifications sur l'entité détachée ne soient enregistrées dans la BD, l'entité doit être rattachée à un contexte de persistance (celui d'origine ou un autre)
- A ce moment, ou au moment du commit qui enregistre vraiment les modifications, le contexte de persistance vérifie qu'il n'y a pas de conflit de concurrence

Conflit de concurrence

- Si les données de la BD associée à un objet détaché ont été modifiées depuis le détachement de l'objet, **`merge`** lance une exception, ou le commit échouera
- Tout se passe donc comme si un conflit de concurrence avait été détecté (avec une stratégie optimiste)

Détachement automatique

- Quelques situations provoquent un détachement automatique des entités gérées :
  - Après un commit si le contexte de persistance est limité à la transaction (avec un serveur d'application)
  - Après un rollback
  - Après un clear du GE
  - Après la fermeture du GE
  - Après le passage par valeur de l'entité avec une méthode « remote »

Associations

- X entité détachée ; association entre X et Y
- L'état de Y peut être « disponible » dans ces cas :
  - Y a été retrouvée par **`find`**
  - Y a été retrouvée par une requête avec une clause **`join fetch`**
  - une propriété de Y non clé primaire a déjà été accédée par l'application
  - Y est navigable depuis une entité attachée avec l'association marquée **`fetch=EAGER`**

Définir une source de données

Source de données

- Lorsque JPA est utilisé avec un serveur d'applications, une source de données fournie par le serveur d'applications est le plus souvent utilisée pour obtenir les connexions (voir **`DataSource`** dans le cours sur JDBC)

persistence.xml

- Le fichier **`persistence.xml`** contiendra une référence à cette source de données plutôt que l'URL de la base de données
- A la différence de Java SE, il est inutile de donner la liste des classes gérées dans le fichier persistence.xml s'il n'y a qu'une seule unité de persistance (car le serveur d'application les découvrira automatiquement au moment du déploiement de l'application)

Exemple de persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ...>
  <persistence-unit name="candidaturesPU"
                    transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa
           .PersistenceProvider</provider>
    <jta-data-source>
      jdbc/candidatures</jta-data-source>
    <properties>
      ...
    </properties>
  </persistence-unit>
</persistence>
```


Définir une source de données

- Il est possible de définir la source de données directement dans le serveur d'application ; la procédure n'est pas standardisée mais c'est la seule façon possible si on veut partager cette définition entre plusieurs applications
- On verra plus loin que depuis Java EE 6 il est aussi possible de définir une source de donnée d'une façon standard dans l'application elle-même avec l'annotation **`@DataSourceDefinition`**

Source de données dans le serveur

- La procédure dépend du serveur d'application
- On va voir ici un exemple de la déclaration d'une source de données MySQL dans un serveur d'application GlassFish
- 3 étapes :
  1. Ajouter le driver MySQL dans GlassFish
  2. Créer un pool de connexion
  3. Créer une ressource JDBC

Ajouter le driver MySQL

- Ajouter le jar du driver JDBC de MySQL, par exemple, mysql-connector-java-5.1.12-bin.jar dans le répertoire glassfish\lib placé sous le répertoire d'installation de Glassfish, par exemple C:\glassfishv3\glassfish\lib

Définition du pool de connexion (1/2)

- Il faut faire afficher la console d'administration de Glassfish
- Ressources / JDBC / Pools de connexions JDBC
- Cliquer sur Nouveau
- Nom du pool : candidatures (ou un autre)
- Type de ressource : javax.sql.DataSource
- Fournisseur de pilote de base de données : MySQL
- Cliquer sur Suivant

Définition du pool de connexion (2/2)

- Laisser les valeurs proposées
- En bas de la page, saisir les bonnes valeurs pour les propriétés données : databaseName, serverName, user, password (enlever les autres propriétés)
- Pour vérifier, faire un ping (bouton dans l'onglet « Général » pour le pool de connexions)

Définir une ressource JDBC

- Console d'administration de Glassfish
- Ressources / JDBC / Ressources JDBC
- Cliquer sur Nouveau
- Nom JNDI : jdbc/candidatures
- Donner le nom du pool de connexion
- Cliquer sur OK pour enregistrer

Définition avec une annotation

- Java EE 6 permet aussi (en plus de la définition habituelle dans le serveur d'application) de définir une source de données avec l'annotation **`@DataSourceDefinition`** du paquetage **`javax.annotation.sql`**, ou dans le fichier web.xml
- La source de données sera définie, et son nom sera enregistré dans le registre JNDI, au moment du déploiement de l'application

`@DataSourceDefinition`

- Cette annotation peut être mise sur une classe servlet, EJB ou une classe de la partie cliente
- La portée de la définition dépend du nom JNDI donné (attribut name de l'annotation) ; par exemple java:module:… indique que la portée sera le module de la classe qui contient l'annotation

@DataSourceDefinition

- Les attributs de l'annotation permettent de définir les propriétés principales de la source de données (voir javadoc de l'annotation)
- Il est possible de donner d'autres propriétés particulières aux différents types de source de données avec l'attribut **properties**

Exemple

```
@DataSourceDefinition(
  name = "java:app/env/jdbc/dataSource1",
  className = "org...ClientXADataSource",
  portNumber=6689, serverName="test.unice.fr",
  user = "APP", password = "APP",
  databaseName = "maDb",
  properties = {"create=true"})
@Stateless
public class … {
  ...
  @Resource(lookup="java:app/env/jdbc/dataSource1")
  private dataSource ds;
```


Définir une source de données dans un fichier de configuration

- Il est aussi possible de définir une source de données dans le fichier **web.xml** (et aussi **application.xml**, **ejb-jar.xml** et **application-client.xml**)

Définition dans web.xml

- Le fichier web.xml peut contenir une définition de la source de données, qui (re)définit la définition donnée par l'annotation, ce qui permet de donner une définition sans recompilation
- Il suffit de donner le même nom mais avec une définition différente
- On peut ainsi utiliser le SGBD « Java DB » pour les tests et le SGBD MySQL pour la production

Exemple

```
<data-source>
  <description>MySQL</description>
  <name>java:global/MonApp/maDS</name>
  <class-name>
    com.mysql.jdbc.jdbc2.optional.MysqlDataSource
  </class-name>
  <server-name>machin.unice.fr</server-name>
  <port-number>3306</port-number>
  <database-name>testDB</database-name>
  <user>toto</user> <password>secret</password>
  <property>
    <name>prop1</name> <value>val1</value>
  </property>
  ...
</data-source>
```


Bibliographie

Sites Web

- Spécification officielle de JPA 2.0 : http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html (cliquer sur le 1er bouton « *To view the Specification for evaluation* »
- Eclipselink : http://www.eclipse.org/eclipselink/
- Hibernate : http://www.hibernate.org/

Livres

- Pro JPA 2
  de Mike Keith et Merrick Schincariol
  Edition Apress
  (en anglais)