

Conception d'applications internet

JPA

Olivier Cailloux

LAMSADE, Université Paris-Dauphine

Version du 10 mars 2016

Java Persistence API : introduction

- Standard pour gérer la persistance
- Pour Java SE et Java EE
- Modèle Objet / Relationnel (ORM)
- JPA définit les concepts et interfaces, fournisseur JPA les implémente (exemple : Hibernate)
- Pour cette présentation : ORM ; Accès concurrents ; JPA
- Appuié sur JDBC

Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ?

Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ? Java Community Process

Standards JCP

- Implication de « la communauté » pour standards Java
- JCP ? Java Community Process
- Définit les JSR : standards utilisés en Java SE ou Java EE
- Spécifications tiennent compte de nombreux avis d'horizons divers
- JSR 338 : JPA 2.1 ; JSR 345 : EJB 3.2 ; JSR 342 : Java EE 7 ; JSR 346 : CDI...
- Tensions entre standard ouvert et contrôle ! (2010, Apache [quitte](#) le comité JCP ; Doug Lea [également](#), en faveur de OpenJDK...)

Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ?

Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ? Éviter de charger tout le graphe !

Faiblesses du modèle relationnel pur

Problème : « Mismatch » Objet / Relationnel

- Modèle de données (DM) : objets
- Références directionnelles
- Modèle Entité / Relationnel : sans comportement
- Pas de notion de navigation
- Chargement automatique ? Éviter de charger tout le graphe !
- Problème classique : $n+1$ select
- Cohérence à maintenir entre BD et objets : types, colonnes, contraintes not null ou autres...
- Répétition lors écriture des requêtes de base
- Difficultés particulières : héritage et autres concepts objet

Avantages d'une solution ORM

ORM ?

Avantages d'une solution ORM

ORM ? Object / Relational Mapping

Avantages d'une solution ORM

ORM ? Object / Relational Mapping

- Détection des modifications avec accès BD optimisés
- Réduction code répétitif
- Meilleure portabilité
- Permet modèle objet fin
- Facilite refactoring et développement agile

De : [diapos](#) Maude Manouvrier, p. 22

Entité

- Lien entre DM et BD : *entités* et annotations sur entités
- DM contient des classes *entités*
- Entité : instance représente pas toujours une ligne d'une table
- Marquer classe `@Entity` (voir aussi `@Table`)
- Marquer champs transients (ou méthodes `get*`) `@Transient` (et persistants `@Column` si désiré)
- Marquer un champ persistant id `@Id` (et `@GeneratedValue`)
- Id représente la clé primaire
- Id initialisé par le fournisseur de persistance (pas de `setId public`) (sauf si clé naturelle, généralement déconseillé)
- Pour permettre concurrence optimiste : marquer un champ persistant version `@Version` (écriture : slmt fournisseur)

Unité de persistance

- Ensemble d'entités contenu dans une « persistance unit »
- Unité liée à un pilote JDBC et des propriétés de connexion par configuration
- Unité associée à `EntityManagerFactory`
- `EntityManager` gère entités dans unité persistance

Contexte de persistance

- EntityManager lié à un *contexte de persistance*
- Contexte : associe une ligne DB à (max.) une instance d'entité

Cycle typique :

- `e = new MyEntity(); e.setTruc(...);` // e est « new »
- `entityManager.persist(e);` // e est « managed »
- fermeture du entityManager : e est « detached »

États d'une instance d'entité

new sans identité persistante, pas dans le contexte

managed avec identité persistante, dans le contexte

detached avec identité persistante, pas dans le contexte

removed avec identité persistante, dans le contexte de persistance, marquée pour effacement

EM et transactions

- Les changements du contexte de persistance via l'entity manager sont synchronisés avec la BD au commit (ou flush)
- Les changements à la BD sont annulés par un rollback
- Injection dans EJB : `@PersistenceContext` EntityManager
- Persistence context (par défaut) a le scope de la transaction en cours

Exemple de changement :

- `e=entityManager.find(...);` // e est « managed »
- `e.setTruc(...);` // changement marqué
- fermeture du entityManager : synchronisation

Configuration unité de persistance

- Persistence unit définie dans META-INF/`persistence.xml` dans un jar comme bibliothèque du `.ear` (aussi possible dans EJB ou `.war`)
- Génération du schéma par JPA : dans `persistence.xml`, propriété `javax.persistence.schema-generation.database.action`, valeur `drop-and-create`
- Fournisseur trouvé dans classpath

Unité de persistance en Java SE

- `Persistence.createEntityManagerFactory("helloworld");`

Unité de persistance en Java EE

- En général, préciser `jta-data-source` : un nom JNDI
- Conteneur Java EE définit `java:comp/DefaultDataSource` : nom utilisé par défaut

Transactions et Java EE

- Pilotes JDBC supportent généralement JTA (Java Transaction API)
- Permet entre autres la gestion des transactions par le conteneur
- Transactions gérées par le conteneur par défaut pour EJB (ou utiliser `@TransactionManagement`)
- Chaque méthode participe alors par défaut à une transaction
- Si pas de transaction en cours lors de l'appel : le conteneur démarre puis termine automatiquement une transaction (commit si ok, rollback si exception sauf exception application)
- Si transaction en cours, la méthode y participe
- Ou annoter la méthode `@TransactionAttribute`

Exercices

- Créer une entité pour un type de votre projet.
- Faire en sorte que la table correspondante soit créée automatiquement lors du déploiement de l'entité.
- Permettre CR.D : Create, Retrieve, Delete *aussi simple que possible*, via un ou plusieurs servlets. (N'utilisez pas de paramètres complexes, ce n'est pas le but de cet exercice.)
- Programmer une méthode qui transforme un attribut d'un objet.
- Permettre l'application de cette méthode via un SLSB appelé par un servlet. Votre servlet ne doit pas nécessairement accepter de paramètres. Quid de l'atomicité de la transaction ?

Définition

- Entity type : une classe représentant une table
- Value type : une classe représentant une *partie* d'une table
- Entité a un cycle de vie propre
- Value type attachée à une entité
- Cycle de vie dépend de l'entité parente

Sortes de value types

- Value type de base : champ int par exemple
 - Représente une colonne avec un type simple Java
 - Utilise les conversions JDBC
 - Exemple : Date, int...
- Value type collection : pour liens multiples entre instances
- Value type embarqué
 - Représente *plusieurs* colonne avec un type Java non élémentaire

Justification

Value type embarqué

- Dans BD, une seule table User
 - User a une adresse
 - Dans modèle objet, classe User et classe Adresse
-
- Modèle objet : *granularité* plus fine peut être désirable
 - Pourquoi cette différence ?

Justification

Value type embarqué

- Dans BD, une seule table User
 - User a une adresse
 - Dans modèle objet, classe User et classe Adresse
-
- Modèle objet : *granularité* plus fine peut être désirable
 - Pourquoi cette différence ?
 - Classes ont des responsabilités
 - Réutiliser une classe, ne pas la dupliquer
 - Schéma BD stable dans le temps

Utilisation

- Annoter le value type `@Embeddable`
- Annoter son utilisation `@Embedded`
- On peut aussi utiliser un value type dans un value type

Exemple d'utilisation

```
@Embeddable
public class Zip {
    private String postalCode;
    private String city;
}

@Embeddable
public class Address {
    private String street;
    @Embedded
    private ZipCode zipCode;
}

@Entity
public class User {
    private String name;
    @Embedded
    private Address address;
}
```

Utilisation via EntityManager

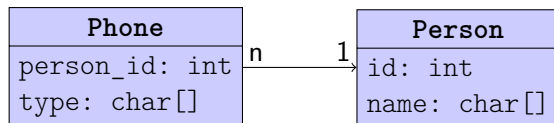
- Créer une instance user
- Créer une instance address
- `user.setAddress(address)`
- `em.persist(user)`
- L'EM persiste le user *et* son adresse
- Cycle de vie address lié à cycle user
- De même, effacement d'un user efface son adresse

Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ?

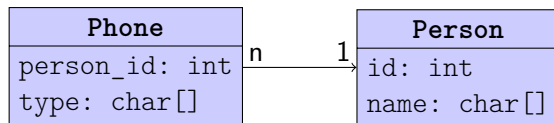
Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



Association one-to-many

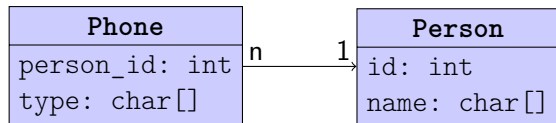
- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



Représentation avec ORM ?

Association one-to-many

- Une personne a plusieurs numéros de téléphone
- BD ? clé étrangère Phone référence Person



Représentation avec ORM ?

- Classe Phone référence Person
- Classe Person a une collection de Phones
- Les deux

Mapping proche BD

Classe Phone référence Person

- Entités normales Phone et Person
- Dans Phone, inclure champ Person
- L'annoter `@ManyToOne` voir aussi `@JoinColumn`
- Retenir : "...ToOne" plutôt que "Many" !
- Crée schema présenté précédemment

`@Entity`

```
public class Phone {  
    @Id ...  
  
    @ManyToOne  
    private Person person;  
}
```


Usage mapping ManyToOne

- Par défaut : nécessaire persister *Phone* et *Person*
- Nécessaire effacer *Phone* en plus de *Person*

Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ?

Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle

Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle
- Comment trouver les n° de téléphone d'une personne ?

Limites du mapping ManyToOne

- Solution précédente simple
- Mais... ? Peu naturelle
- Comment trouver les n° de téléphone d'une personne ?
- On peut toujours le faire par requête SQL !
- Mais on peut souhaiter pouvoir le faire via navigation de pointeurs

Mapping OneToMany

- Mapping inverse : ref `@OneToMany` unidirectionnelle
-



Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ?)

Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ? pour indépendance à ORM)

Liens entre entités

Exemple : un item a plusieurs bid associés tiré de Java Persistence with Hibernate

- Chaque bout d'une association : to one ou to many
- To one (Item forItem dans Bid) : annoter `@ManyToOne`, préciser éventuellement `optional=false` et `fetch`
- Si bidirectionnelle : votre code doit maintenir la synchronisation, malgré annotations (Pourquoi ? pour indépendance à ORM)
- Pour autre bout d'un lien bidirectionnel (ne compte pas dans DDL) : `@OneToMany` et préciser `mappedBy`
- Liens one to one similaire, utiliser `@OneToOne`











Concurrence

- Entité avec `@Version` : protection optimiste par défaut
- Après update, effacement, merge : vérification automatique du n° de version en cache
- Vérification effectuée au moment même ou au flush
- Si versions ne correspondent pas :
`OptimisticLockException`
- Protection pessimiste : utiliser
`EntityManager.lock(entity, lockModeType)`

Différentes égalités

Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ?

Différentes égalités

Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ? Tant que non persistantes, ne fonctionne pas ; change lors sauvegarde

Différentes égalités

Trois types d'égalités

- Égalité en mémoire : `a == b`
- Égalité objet : `a.equals(b)`
- Égalité DB : `a.getId().equals(b.getId())`

Quelles propriétés prendre en compte dans `hashCode` et `equals` ?

- Table de hachage : objet ne peut changer de hash / d'égalité
- Dans une session il faut éviter deux objets not `equals()` et concernant la même ligne de la table
- Id pour égalité : pourquoi pas ? Tant que non persistantes, ne fonctionne pas ; change lors sauvegarde

⇒ Ensemble d'attributs déterminants pt de vue utilisateur !
(username...)

Patterns

- DM *sans* dépendance persistance (tests ; simplification dépendances)
- Entités transversales (couche web et business)
- Qqs classes service business
- Qqs classes persistance visibles du business
- Ou pattern DAO...
- Explorer : SFSB et `@PersistenceContext(text(PersistenceContextType.EXTENDED))`

Usages plus avancés et divers

- Entité peut être non chargée entièrement (`getReference`)
- Accès éventuellement impossible après fermeture du contexte
- Contexte peut persister au-delà de la transaction ; peut être non synchronisé (`@PersistenceContext`)
- Transitivité de la persistance, cf. cascade sur `@OneToMany` (par exemple)
- Type « Value » plutôt que Entity pour des objets de cycle de vie dépendants d'autres

Références

- Java Persistence with Hibernate : [1^{re}](#) édition, [2^e](#) édition
- Hibernate 5.1 [User Guide](#)
- [JSR 338](#) (JPA 2.1) ([direct](#))
- [JSR 907](#) (JTA) (moins utile pour développeur Java EE)
- ENORM: An Essential Notation for Object-Relational Mapping, ACM SIGMOD Record 43(2), June 2014, pp 23–28 ([doi](#), [pdf](#) article)

Pattern DAO

Extraction des aspects propres à la persistance

- DAO ?

Pattern DAO

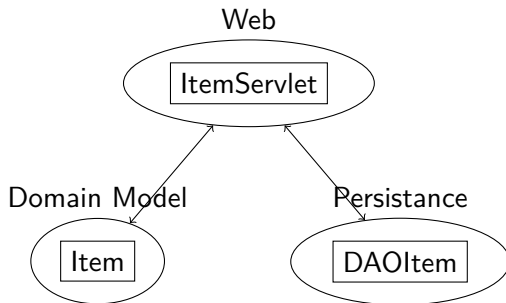
Extraction des aspects propres à la persistance

- DAO ? Data Access Object

Pattern DAO

Extraction des aspects propres à la persistance

- DAO ? Data Access Object
- « Modèle » découpé en aspects Persistance et Domain Model
- Domain Model : opérations logiques, connaissance métier
- Persistance : seule autorisée à communiquer avec la BD



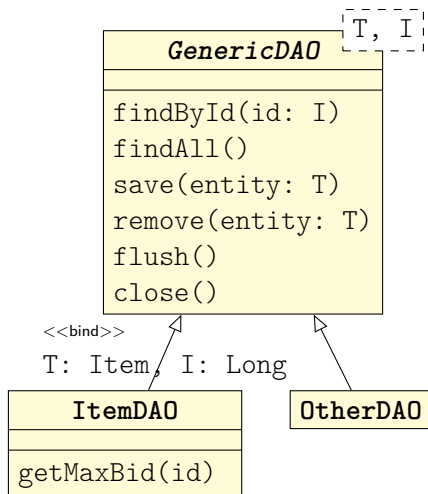
DAO : mise en œuvre

Classe parent

- abstraite
- générique :
 - T = type entité
 - I = type ID
- contient les méthodes *CRUD* : Create / Read / Update / Delete

Sous-classes

- contiennent les méthodes spécialisées



Inspiré par : Java persistence with hibernate

Autres remarques architecturales et références

- Alternative au DAO : Active Record
- Avec JPA, DAO peut être superflu

Références

- Java Persistence with Hibernate
- Patterns of Enterprise Application Architecture

Licence

Cette présentation, et le code LaTeX associé, sont sous [licence MIT](#). Vous êtes libres de réutiliser des éléments de cette présentation, sous réserve de citer l'auteur.

Le travail réutilisé est à attribuer à [Olivier Cailloux](#), Université Paris-Dauphine.

(Ceci ne couvre pas les images incluses dans ce document, puisque je n'en suis généralement pas l'auteur.)