PERSISTANCE DES DONNÉES

PERSISTANCE

mécanisme responsable de la sauvegarde et la restauration de données, afin qu'un programme puisse se terminer sans que ses données ni son état d'exécution soient perdus.

THÉORÈME DE BREWER

- Consistance
- Disponibilité
- Tolérance au partitionnement

Il n'est pas possible d'avoir les trois propriétés

Visual Guide to NoSQL Systems

Availability:
Each client can always read and write.

A

Data Models

- Relational (comparison)
- Key-Value
- Column-Oriented/Tabular
- Document-Oriented

CA

RDBMSs (MySQL, Postgres, etc)

Aster Data
Greenplum
Vertica

AP

Dynamo
Voldemort
Tokyo Cabinet
KAI

Cassandra
SimpleDB
CouchDB
Riak

Pick Two

C

P

Consistency:
All clients always have the same view of the data.

CP

BigTable
Hypertable
Hbase

MongoDB
Terrastore
Scalaris

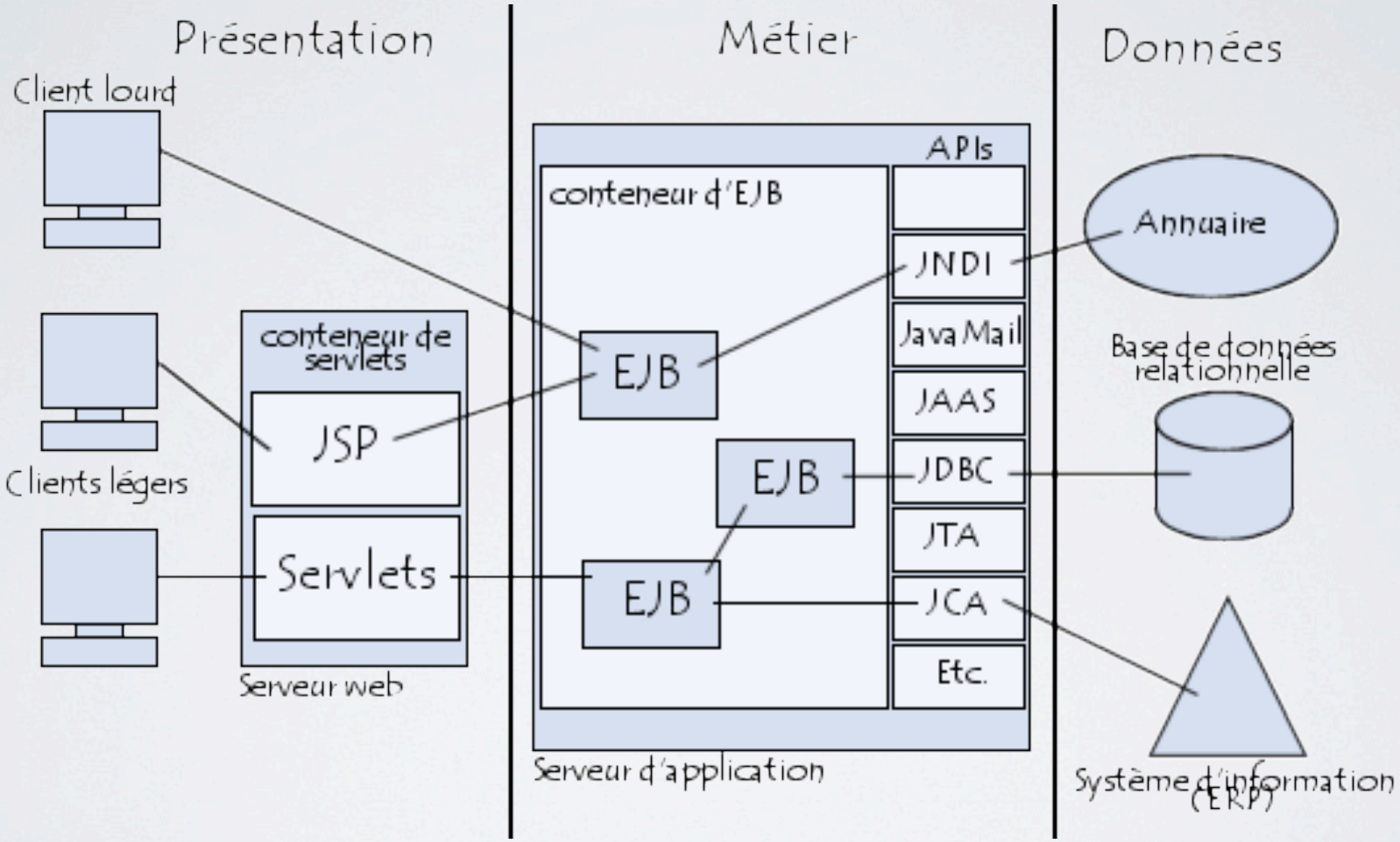
Berkeley DB
MemcacheDB
Redis

Partition Tolerance:
The system works well despite physical network partitions.

PERSISTANCE

- RDBMS
- Cassandra
- MongoDB

RAPPEL D'ARCHITECTURE



PROBLÉMATIQUE

Comment faire le lien entre les tables d'une base de données et des objets Java ?

JDBC

- Interface Java standard d'accès à des bases de données hétérogènes
- Indépendance vis-à-vis du SGBD sous-jacent
- 4 grands types de drivers

TYPE I

Pilotes qui se comportent comme passerelle en permettant l'accès à une base de données grâce à une autre technologie (JDBC-ODBC via ODBC par exemple).

Open DataBase Connectivity est un middleware permettant de manipuler des bases de données mises à disposition par différents SGDB

TYPE II

Pilotes d'API natifs. C'est un mélange de pilotes natifs et de pilotes Java. Les appels JDBC sont convertis en appels natifs pour le serveur de bases de données (Oracle, Sybase, *etc.*) généralement en C ou en C++.

TYPE III

Pilotes convertissant les appels JDBC en un protocole indépendant de la base de données. Un serveur convertit ensuite ceux-ci dans le protocole requis (modèle à 3 couches).

TYPE IV

Pilotes convertissant les appels JDBC directement en un protocole réseau exploité par la base de données. Ces pilotes encapsulent directement l'interface cliente de la base de données et sont fournis par les éditeurs de base de données.

Quel type de driver utilisé ?

TROIS CRITÈRES DE CHOIX

- Portabilité
- Respect des spécifications
- Performances

On ne peut optimiser que 2 choix sur 3 !

ANALYSE RAPIDE

Portabilité	III & IV (Pure Java contrairement aux I & II utilisant des API natives). I & II inutilisable dans une Applet !
A la norme	Type I ne respecte que partiellement la spécification JDBC 3.0. III & IV sont beaucoup plus utilisés ce qui peut impliquer un développement plus suivi.
Performances	De prime abord les drivers I & II qui utilisent des APIs natives. Mais... pas toujours le cas! Le driver Oracle Java aussi voir plus rapide que son homologue natif !

Ou trouver des drivers JDBC ?

DÉPÔT PROPOSÉ PAR SUN

http://developers.sun.com/product/jdbc/drivers

Recherche multi-critères:

- Version de l'API JDBC
- Nom du distributeur
- Certification JEE
- Type du Driver
- SGBD
- Fonctionnalités proposées

Comment charger un driver ?

PREMIÈRE MÉTHODE

En utilisant Class.forName, qui enregistrera le driver auprès du DriverManager:

```
try{
     Class.forName(nomDriver).newInstance();
} catch(ClassNotFoundException e){
     e.printStackTrace();
}
```

DEUXIÈME MÉTHODE

Enregistrement direct du driver dans le DriverManager:

```
Driver monDriver = new com.mysql.jdbc.Driver();
DriverManager.registerDriver(monDriver);
```

TROISIÈME MÉTHODE

Enregistrement du driver comme un argument de la ligne de commande:

```
java -cp .;drivers.jar -Djdbc.drivers=implem.Driver Main
```

Comment se connecter à la base de données ?

CONNECTION AVEC JDBC

Directement:

```
Connection con = DriverManager.getConnection(url);
```

Via une DataSource:

```
Context ic = new InitialContext(env) ;
DataSource ds = ic.lookup(”java :comp/env/jdbc/DS1”);
Connection con = ds.getConnection();
```

Comment exécuter des instructions SQL ?

PRINCIPE JDBC

Une instruction JDBC est un objet de type **statement**:

```
Statement stmt = conn.createStatement();
```

Connexion active pour créer un statement !

Comment exécuter le statement ?

EXÉCUTION

Il faut spécifier l'exécution appropriée...

- executeQuery: pour exécuter un SELECT
- executeUpdate: pour CREATE, INSERT, UPDATE, DELETE

Par exemple:

```
stmt.executeUpdate(
"INSERT INTO AUTHOR VALUES ('TOTO', 'TITI', 1");
```

Et pour le SELECT ?

PRINCIPE POUR LE SELECT

Le résultat d'un SELECT est un... **`ResultSet`**:

```
ResultSet rs = stmt.executeQuery(
"SELECT AUTHOR_FIRSTNAME,AUTHOR_LASTNAME FROM AUTHOR");
```

Comment se déplace t-on dans un ResultSet ?

```
while(rs.next()) {
  String f = rs.getString("AUTHOR_FIRSTNAME");
  String l = rs.getString("AUTHOR_LASTNAME");
  System.out.println(f + " " + s);
}
```

DÉPLACEMENT SUITE...

Atteindre un enregistrement:

```
rs.absolute(Num);
```

Connaître l'enregistrement courant:

```
rs.getRow();
```

Se déplacer de n enregistrements:

```
rs.relative(+/-n);
```

Et la transactionnalité ?

TRANSACTIONALITÉ

Définition:

Une transaction est un jeu de une ou plusieurs instructions qui sont exécutées ensembles de façon unitaire.

Toutes les instructions sont exécutées ou aucune !

Comment être transactionnel avec JDBC ?

JDBC & TRANSACTIONALITÉ

1- Désactivation du mode Auto-commit...

```
conn.setAutoCommit(false);
```

2- ...pour que la transaction soit effective:

```
conn.setAutoCommit(false);
statement1.executeUpdate(...);
statement2.executeUpdate(...);
conn.commit();
conn.setAutoCommit(true);
```

S'il y a un problème lors d'une instruction ?

ROLLBACK

Il faut utiliser la méthode `conn.rollback()`:

```
try {
    conn.setAutoCommit(false);
    //traitements
    conn.commit();
    conn.setAutoCommit(true);
} catch(SQLException e) {
    try{conn.rollback();
} catch(Exception f){
} finally {
    try{conn.close();} catch(Exception e){}
}
```

Peut-on verrouiller (lecture ou écriture) les données durant une transaction ?

ISOLATION DES TRANSACTIONS

Il existe 5 niveaux d'isolation:

- TRANSACTION_NONE: transaction non supportée
- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

ANOMALIES

Lecture impropre (dirty read) : Lorsqu'une transaction lis des données qui sont en train d'être modifiées par votre transaction (non encore validée).

- **Lecture non répétable (non-repeatable read)** : Si une requête ne renvoie pas les mêmes résultats lors d'exécutions successives. C'est le cas si les données que vous lisez sont modifiées par une autre transaction (c'est un peu l'inverse de la lecture impropre).
- **Lecture fantôme (phantom reads)** : Si des exécutions successives d'une même requête renvoient des données en plus ou en moins. Cela peut être le cas si une autre transaction est en train de supprimer ou d'ajouter des données à la table.

RÉCAPITULATIF

	Lecture impropre	**Lecture non répétable**	**Lecture fantôme**
TRANSACTION_READ_UNCOMMITTED	possible	possible	possible
TRANSACTION_READ_COMMITTED	impossible	possible	possible
TRANSACTION_REPEATABLE_READ	impossible	impossible	possible
TRANSACTION_SERIALIZABLE	impossible	impossible	impossible

N’oublions pas quelques petites choses...

APPLICATIONS D'ENTREPRISE

réduction des temps et coûts de développement

portables

disponibles

sécurisée

maintenables

montée en charge

sûres

extensibles

intégrables

adaptables

qualité du code

répondent aux besoins exprimés par les utilisateurs !

Avec JDBC, on fait tout à la main.....

PROBLÉMATIQUE

JDBC permet de se connecter à une base de données et de remonter des informations de celle-ci.

Il est ensuite nécessaire de faire la translation manuellement entre ses données brutes et nos objets de haut niveau.

IDÉE

Ne peut on pas avoir un **moyen automatique** de faire la **translation** entre nos objets Java et les données stockées dans nos bases relationnelles ?

Un Mapping Objet-Relationnel en quelque sorte...

Java Persistance API 1.0

JPA

- Abstraction plus élevée que simplement utiliser JDBC
- Transformation des objets vers la DB et vice-versa
- Pas de ligne de code de type JDBC
- Langage d'interrogation similaire à SQL (JPQL)
- Objets Java annotés
- Gestionnaire d'entités (cycle de vie, état, persistance, ...)

NOTION D'ENTITÉ

- Encapsule les données d'une occurence d'une ou plusieurs tables
- **P**lain **O**ld **J**ava **O**bject (i.e. pas d'héritage !)
- Un POJO est mappé vers une table à l'aide d'annotations
- Un constructeur sans argument
- Ses propriétés sont mappées sur les champs de la table via des annotations
- Respect des règles des Java Beans (set / get / is)

EXEMPLE

```java
@Entity
public class Author {

  @Id
  @GeneratedValue
  private int id;

  private String firstName;
  private String lastName;

  public Author() {
    super();
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;}
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName
  }
}
```

MAPPING ENTITÉ - TABLE

`@Table` permet de lier l'entité à une table de la DB

- Par défaut, le nom de l'entité est utilisé comme nom de table
- L'attribut name spécifie le nom de la table que l'on souhaite

MAPING DES CHAMPS

`@Column` associe un membre de l'entité à une colonne

- L'attribut name spécifie le nom de la colonne
- L'attribut nullable indique si la colonne est nullable
- L'attribut unique indique si la colonne est unique

EXEMPLE

```java
@Entity
@Table(name=«AUTHOR»)
public class Author {

  @Id
  @GeneratedValue
  @Column(name=«AUTHOR_ID»)
  private int id;

  @Column(name=«AUTHOR_FIRSTNAME»)
  private String firstName;
  @Column(name=«AUTHOR_LASTNAME»)
  private String lastName;

  public Author() {
    super();
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;}
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName
  }
}
```

CONTRAINTES

Il faut obligatoirement définir une propriété avec @Id !

- La propriété concernée sera la clé primaire
- La propriété doit être de type primitif ou String
- La clé primaire générée automatiquement **`@GeneratedValue`**

DIFFÉRENTES STRATÉGIES DE GÉNÉRATION

- TABLE: table dédiée stocke les clés des tables
- **SEQUENCE**: séquence de génération fournie par la DB
- IDENTITY: colonne spécifique à la base de donnée
- **AUTO**: L'implantation génère la clé

EXEMPLE

```
@Id
@Column(name=«AUTHOR_ID»)
@SequenceGenerator(name=«seqId»,
                   sequenceName= «AUTHOR_SEQ»)
@GeneratedValue
(strategy=GenerationType.SEQUENCE,
 generator= «seqId»)
```

MAPPING D'ÉNUMÉRATION

- **`@Enumerated`** permet de faire l'association d'une énumération à une colonne de la table

- **`EnumType`** permet de spécifier la forme prise de la sauvegarde en DB

  - Sous forme de chaîne de caractères **`EnumType.STRING`**

  - Sous forme d'entier **`EnumType.ORDINAL`** (défaut)

EXEMPLE

```
@Column(name= «AUTHOR_CIVILITY»)
@Enumerated(EnumType.STRING)
private Civility civility;
```

RELATION ENTRE OBJETS

Différentes cardinalités sont possibles:

- OneToOne
- OneToMany
- ManyToOne
- ManyToMany

ONETOONE

Voir exemple de code

ONETOMANY

voir exemple de code

MANYTOONE

voir exemple de code

HÉRITAGE

voir exemple de code

Comment sont gérées les entités ?

QUELQUES DÉFINITIONS

- Unité de persistance (Persistance Unit)
- Contexte de persistance (Persistance Context)
- Gestionnaire d'entité (Entity Manager)

CONTEXTE DE PERSISTANCE

- Un contexte de persistance est un ensemble d'instances pour lesquelles il existe une relation d'unicité `(id, entité)`.

- Dans un contexte de persistance, les instances et leur cycle de vie sont gérés par un gestionnaire d'entité.

- La portée d'un contexte de persistance peut être une transaction ou un processus de traitement plus large.

UNITÉ DE PERSISTANCE

- Représente l'ensemble des types d'entité qui peuvent être gérées par un gestionnaire d'entité donné.

- Définie l'ensemble des classes en relation avec une application et qui sont stockées dans une DB.

GESTIONNAIRE D'ENTITÉ

- Permet d'accéder à une DB dans un processus de traitement donné.

- Les opérations possibles sont :

  - création d'instances d'entité (insert / update / merge);
  - suppression d'instances d'entité (delete);
  - recherche d'entités en fonction de leur clé (find);
  - réalisation de requêtes (createQuery, createNamedQuery)

PRINCIPES D'ARCHITECTURE

- Portée du contexte de persistance
- Propagation du contexte de persistance
- Etats des entités
- Stratégies de verrous

PORTÉE DU CONTEXTE DE PERSISTANCE

- Un gestionnaire d'entité permet d'inter-agir avec le contexte de persistance.

- Contexte de persistance attachée à une transaction

PORTÉE TRANSACTION

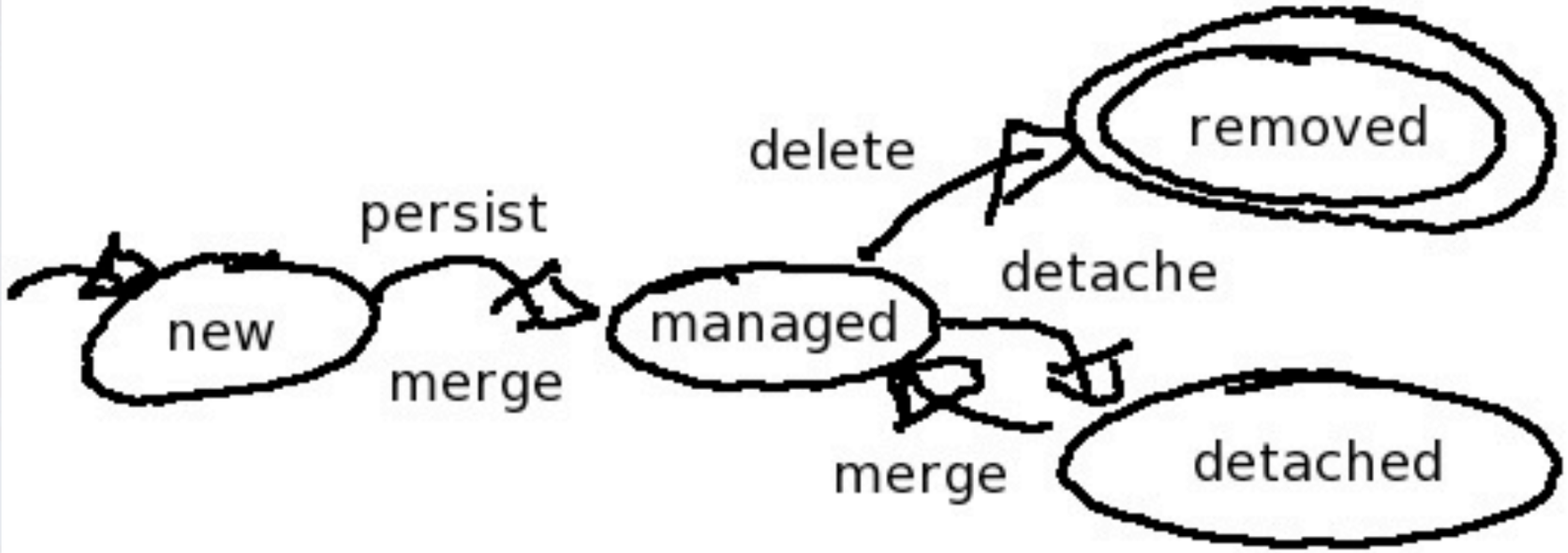
- Utilisable avec Java Transaction API
- Associé avec le cycle de vie
- Quand on invoque un gestionnaire d'entité, on récupère le contexte de persistance :
  - si il n'y a pas de contexte de persistance associé avec la transaction courante on en ouvre un.
  - sinon celui qui est associé avec le contexte de persistance est utilisé.

PROPAGATION DU CONTEXTE DE PERSISTANCE

Un contexte de persistance n'est jamais partagé entre différentes transactions ou entre des gestionnaires d'entité qui ne sont pas issus de la même fabrique de gestionnaire d'entité !

Et les entités dans tout ça ?

ÉTATS DES ENTITÉS



L'ÉTAT NEW

Une entité est à l'état « new » lorsqu'elle vient juste d'être créée et qu'elle n'est pas encore associée à un contexte de persistance. Elle n'a donc pas de représentation relationnel dans la DB et ne possède pas encore d'`id`.

L'ÉTAT MANAGED

Une entité à l'état « managed » est une entité avec un `id` et elle est associée à un contexte de persistance.

L'ÉTAT DETACHED

Une entité à l'état « detached » possède un `id` mais elle n'est plus associée à un contexte de persistance car ce contexte est clos ou l'entité à été détachée manuellement.

L'ÉTAT REMOVED

Une entité à l'état « removed » est une instance avec un `id`, associée à un contexte de persistance et devant être supprimée de la DB.

Comment éviter les les dirty reads and non-repeatable reads sur une entité ?

Lecture impropre (dirty read) : Lorsqu'une transaction lis des données qui sont en train d'être modifiées par votre transaction (non encore validée).
Lecture non répétable (non-repeatable read) : Si une requête ne renvoie pas les mêmes résultats lors d'exécutions successives. C'est le cas si les données que vous lisez sont modifiées par une autre transaction (c'est un peu l'inverse de la lecture impropre).

VERROU

Un verrou peut être posé via :

- la méthode `entityManager.lock();`
- la méthode `entityManager.find();`
- la méthode `entityManager.refresh();`
- la méthode `query.setLockMode();`

DEUX STRATÉGIES...

Les verrou optimistes cherche à verrouiller le plus tard possible en espérant que la donnée ne soit pas modifiée.

Les verrous pessimistes cherche à verrouiller le plus tôt possible et le relâche quand la transaction est commitée.

...PLUSIEURS VARIANTES

- **PESSIMISTIC_READ:** verrou de lecture de la DB à l'appel pour verrouiller. Lecteurs concurrents mais pas d'écrivains.
- **PESSIMISTIC_WRITE:** verrou d'écriture de la DB à l'appel pour verrouiller. Pas de lectures ou écrivains concurrents.
- **OPTIMISTIC_READ:** utilisation d'un numéro de version dans l'entité. Vérification de la version avant le commit.
- **OPTIMISTIC_WRITE:** utilisation d'un numéro de version dans l'entité. Vérification de la version avant le commit. Incrémentation de la version après le commit.

OPTIMISTIC_WRITE EN PRATIQUE

```
public class Author {
@Id
int id;
@Version
int version;

“UPDATE Author ..., version = version+1,
WHERE id = ? AND version = readVersion”
```

CONCLUSION SUR JPA

Pros :

- API standard et interchangeable
- Différents ORM respecte ce standard (TopLink, Hibernate, ...)

Cons :

- Le plus petit dénominateur commun entre ORM !
- Pleins de fonctionnalités manquantes !

Plein de fonctionnalités manquantes

FONCTIONNALITÉS HORS JPA

- Criteria
- Example
- Flush manuel
- Filtrage dynamique de résultats
- Gestion d'un cache de second niveau
- Cascade « DELETE_ORPHAN »
- Attacher et détacher un objet de l'EntityManager

Certains ORM proposent déjà ce type de fonctionnalités et ont inspiré JPA...

HIBERNATE

- Respect du standard JPA
- Standard marché
- Fournit toutes les fonctionnalités décrites ci-avant
- Parfaitement intégré avec Spring
- Forte innovation
- Nombreux outils (par exemple, **`HbmToJava`**)

UN STANDARD DU MARCHÉ

FONCTIONNALITÉS « HIBERNATE »

- Criteria
- Example
- Flush manuel
- Cascade « DELETE_ORPHAN »
- Attacher et détacher un objet de l'EntityManager
- Cache de second niveau

CRITERIA

- Haut niveau
- API de recherche orientée objet
- Intuitive & simple
- Requête dépendant des actions utilisateurs
- Nombreuses fonctionnalités (restrictions, tris, filtres, ...)
- Basée sur le design pattern **Builder**

RESTRICTIONS

- Opérations logiques : **`and`**, **`not`**, **`or`**
- Opérations ensemblistes : **`between`**, **`in`**, **`conjunction`**, **`disjunction`**
- Opérations de comparaisons : **`=`**, **`<=`**, **`>=`**, **`<`**, **`>`**, **`!=`**
- Comparaisons de chaines de caractères : **`like`**, **`ilike`**

EXEMPLE

```
Criteria criteria = sess.createCriteria(Author.class)
    .add(Restrictions.like("lastname", "Toto")));

List results = criteria.list();
```

```
Criteria criteria = sess.createCriteria(Author.class)
    .add(Restrictions.ilike("lastname", "TOTO")));

List results = criteria.list();
```

TRIS & FILTRES

- Trier les résultats de façon ascendante ou descendante
- Limiter le nombre de résultats retournés

Par exemple :

```
Criteria criteria = session.createCriteria(Author.class)
    .add(Restrictions.between("weigth", 55, 80))
    .addOrder(Order.asc("weigth"))
    .addOrder(Order.desc("lastname"));

List results = criteria.list();
```

EXAMPLE

- Recherche en fonction d'un exemple
- On sélectionne les propriétés qui nous intéresse

Par exemple :

```
Book book = new Book(...);
Example example = Example.create(book)
    .excludeProperty("year") //On ne compare pas avec l'année
    .ignoreCase(); //On ne tient pas compte de la casse

Criteria criteria = session.createCriteria(Book.class)
    .add(example);

List results = criteria.list();
```

FLUSH MANUEL

Synchronisation de la DB avec les différents états des entités présentes dans le gestionnaire d'entité.

DELETE_ORPHAN

Dans une relation **`@OneToMany`** ou **`@OneToOne`** des entités liées à une entité supprimée vont rester en base...

```
@OneToMany(cascade = {CascadeType.ALL} )
@Cascade
({org.hibernate.annotations.CascadeType.DEL
ETE_ORPHAN})
```

DELETE ORPHAN

Dans le cas d'une relation **`@ManyToMany`** il faut utiliser une autre annotation Hibernate :

```
@OneToMany( cascade = {CascadeType.ALL} )
@Cascade
({org.hibernate.annotations.CascadeType.SAV
E_UPDATE,
org.hibernate.annotations.CascadeType.DELET
E_ORPHAN})
```

INTÉGRATION AVEC SPRING

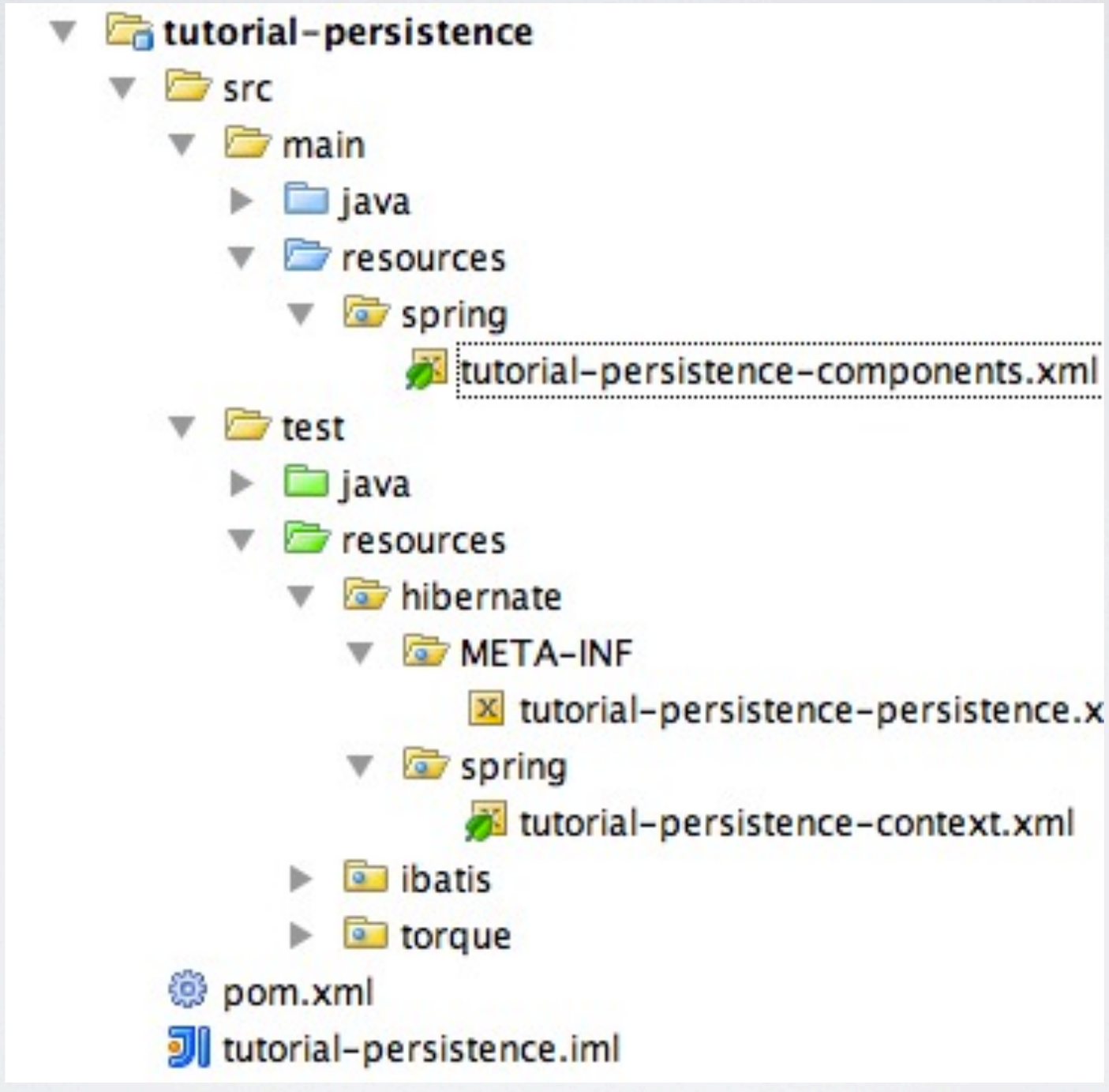
En pratique comment cela se passe t-il ?

FICHIERS DE CONFIGURATION

Définition de trois fichiers :

- persistence-unit
- spring components
- sprint context

ARBORESCENCE DES FICHIERS



PERSISTENCE-UNIT

```
<persistence-unit name="tutorial-persistance" transaction-
type="RESOURCE_LOCAL">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>

 <class>fr.umlv.m2.jee.tutorial.persistence.book.Book</class>

 <properties>
  <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
  <property name="hibernate.show_sql" value="true"/>
  <property name="hibernate.format_sql" value="true"/>
  <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
  <property name="hibernate.batch_size" value="20"/>
 </properties>
</persistence-unit>
```

PROPRIÉTÉS

- **`hibernate.dialect:`** dialecte de la base de données
- **`hibernate.show_sql:`** afficher les requêtes SQL
- **`hibernate.hbm2ddl.auto:`** opération sur le schema de la DB
- **`hibernate.jdbc.batch_size:`** éviter des `OutOfMemory`

http://docs.jboss.org/hibernate/core/3.3/reference/fr/html/session-configuration.html

PRÉCISIONS SUR HBM2DDL

Valeurs possibles de l'option :

- `validate` : validation du schema sans modification de la DB
- `update` : met à jour le schema de la DB
- `create` : création du schema (destruction des données)
- `create-drop` : drop le schema à la fin de la session

PRÉCISIONS SUR BATCH_SIZE

```
// set the JDBC batch size (it is fine somewhere between 20-50)
hibernate.jdbc.batch_size 30

// disable second-lavel cache
hibernate.cache.use_second_level_cache false
```

```
Transaction T = session.beginTransaction();
for (int i = 0; i < 200000; ++i) {
    Record r = new Record(...);
    session.save(record);
    if ((i % 30) == 0) {
        // 30, same as the JDBC batch size
        // flush a batch and release memory
        session.flush();
        session.clear();
    }
}
T.commit();
S.close();
```

SPRING-COMPONENTS

```
<!-- Scans for application components to deploy -->
<context:component-scan
  base-package="fr.umlv.m2.jee.tutorial.persistence" />
```

SPRING-CONTEXT

```
<import resource="tutorial-persistence-components.xml"/>

<tx:annotation-driven />

<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
 <property name="entityManagerFactory"
           ref="entityManagerFactory" />
</bean>

  <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFact
oryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="persistenceXmlLocation"
              value="classpath:hibernate/META-INF/
                     tutorial-persistence-persistence.xml"/>
</bean>
```

SPRING CONTEXT (2)

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
 <property name="driverClassName" value="org.h2.Driver" />
 <property name="url" value="jdbc:h2:mem:tutorial-
persistence;MODE=Oracle" />
</bean>
```

`MODE=Oracle` va dépendre de la base cible

HQL VS. SQL

- Langage de requêtage d'Hibernate
- Hibernate Query Langage

- HQL est au niveau objet, on utilise donc le nom de la classe, les paramètres d'un champ
- En SQL on utilise le nom d'une table et le nom de colonne

HQL EXEMPLE

```
Query query = createQuery("SELECT b FROM AUTHOR b
WHERE b.ssid=:ssid");
```

Il faut éviter l'injection de SQL dans vos applications!

```
query.setParameter("ssid", paramSSID);
```

PAGINATION

```
Query q = session.createFilter( collection, "" );
q.setMaxResults(PAGE_SIZE); // nombre max d'elements
q.setFirstResult(PAGE_SIZE * pageNumber); // on pagine
List page = q.list();
```

NOMBRE D'ÉLÉMENTS EN BASE

Ne jamais faire....

```
Query query = createQuery("SELECT b FROM AUTHOR b
WHERE b.ssid=:ssid");
List<XX> r = query.getResultList();
return r.length();
```

Il faut faire...

```
Query query = createQuery("select count(*) from ....");
```

ORDER BY

Possibilité de tri ascendant ou descendant

```
from Author t order by t.name asc
```

Pattern de persistance

DATA ACCESS OBJECT

- Intervient entre le tiers métiers et le tiers données
- Encapsule l'acccès aux données
- Rend transparent l'aspect technique
- Découple les clients du service technique
- Centralise les accès aux données

DATA ACCESS OBJECT

Pros :

- Abstraction des aspects techniques
- Changement du mode de stockage transparent

Cons :

- Complexité de mise en oeuvre
- Abstraction

DATA ACCESS OBJECT