ISIMA

Java EE : Persistance

JPA

Persistance

- Les enjeux économiques
- La persistance est un élément important du logiciel
- Le temps passé à rechercher des objets ou à les stocker doit être le plus bas possible
  - Les développeurs doivent se préoccuper de créer des fonctions (business) plutôt que de les présenter ou les enregistrer
- C'est avec cet état d'esprit que de nombreux outils ont été créés !

ISIMA

Java™

ORACLE

Quels solutions JEE ?

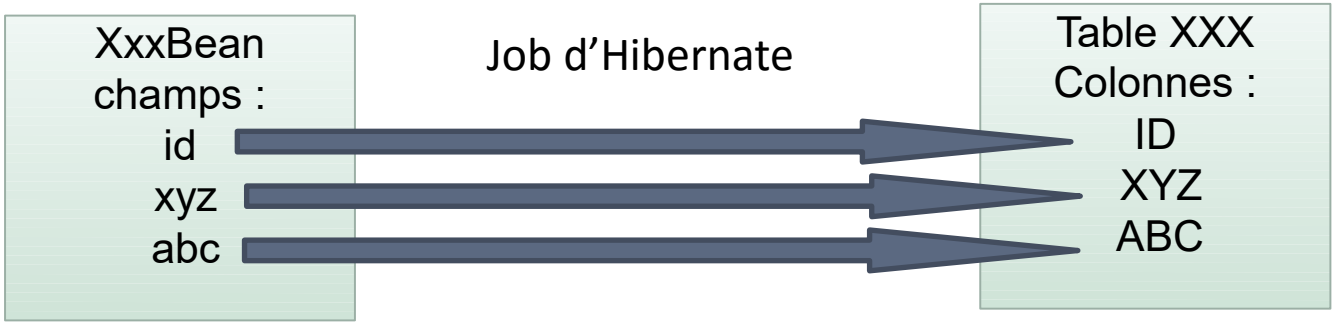
- Les EJB
  - *Enterprise Java Bean*
  - Invention d'IBM en 1997
    - *Intégré par SUN dans JEE en V1.0 98*
    - *V3.2 2013*
  - Augmenter la productivité ! Se concentrer sur le business !
  - Le but est de gérer la couche métier, et par extension la couche persistence

EJB

- Que s'est il passé ?
- Premières versions complexes
  - Adoptées par les entreprises mais impopulaires auprès des développeurs
  - Tomcat a fait le choix de ne pas l'intégrer à sa solution
- Des solutions plus indépendantes se sont popularisées
  - SPRING pour le métier et les interfaces
  - Hibernate pour la persistence
  - Spring ou JAX RS pour les services REST

Hibernate

- Des frameworks populaires et plus simples
- Hibernate est démarré au début des années 2000
  - Très simple d'utilisation
  - Compatible avec de nombreuses bases
  - Mapping objet-relationnel (*Object-Relationnal Mapping* ORM)



Hibernate

- Par rapport aux ResultSet de JDBC, c'est une vraie simplification
- De plus, hibernate manage la connexion à la base de donnée plus simplement et plus intelligemment que le Connection de JDBC
- Cette nouvelle approche permet d'être plus productif

Alternatives

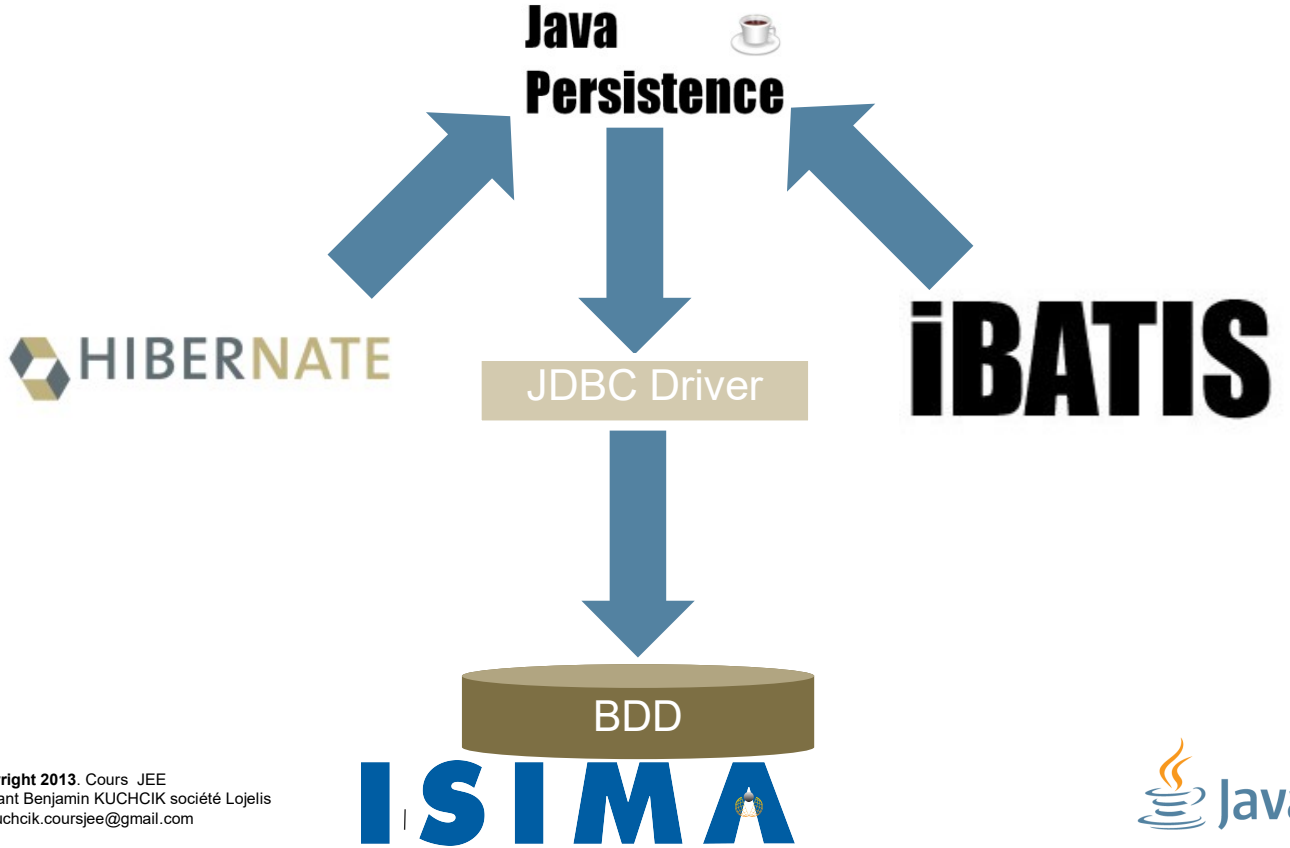
- Il existe des oppositions à ces techniques arguant de moindres performances des ORM par rapport à JDBC
  - La surcouche génère forcément des performances moindres qu'une requête parfaitement optimisée
    - Mais attention les ORM integre des systèmes de cache très puissant
- Java 8 grâce à ses lambda offre une alternative intéressante entre JDBC et les ORM par jOOQ
  - http://www.jooq.org/

JEE normalise

- A partir d'EJB
- A partir des concepts d'Hibernate nait JPA (2006)
  - Orienté base de données relationnelles (MySql, Oracle, SQLServer)
  - Nombreuses compétences dans les entreprises
- Une norme plus générique antérieure existe : JDO (2002)
  - Couvre tout type de persistence
    - Bases orientées objets
    - NOSql

JPA

- Comment cela fonctionne ?



Résumons

- Pour JPA
- Nous avons besoin
  - de la norme JPA (un jar)
  - d'une implémentation de JPA (hibernate)
  - d'une base de données
    - Prenons SQLite
    - Bien intégrée à java
    - Sauvegarde tout dans un fichier
    - Très (très) simple à mettre en place

Créons un CRUD avec JPA

- Create, read, update and delete
- Ce sont les fonctions basiques de stockage
- Il s'agit simplement d'enregistrer et de retrouver les informations en base de données
- Le modèle est très simple
  - Nous avons des articles attachés à une catégorie
- Notre CRUD utilise
  - TwitterBootstrap pour la présentation
  - JPA2 pour le stockage

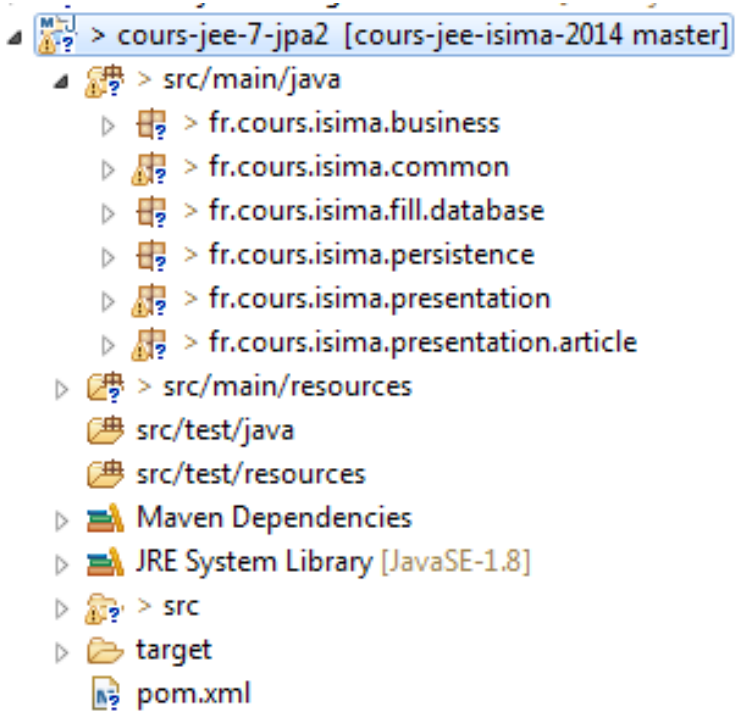
Les fonctionnalités de notre exemple

- cours-jee-7-jpa2
- Il s'agit d'articles classés par catégorie
  - Un article n'a qu'une catégorie
- L'application permet de lister, créer, modifier et supprimer des articles rien de plus
- Le modèle de données est très simple
  - Une table Article avec un Foreign Key sur une Categorie
  - Une table Catégorie

Conception

- Architecture 3-tiers
- L'application respecte une architecture 3-tiers
  - Data Access
  - Business
  - Présentation
- Pour le cours nous allons nous concentrer uniquement sur la couche persistence
- Toutes les autres couches sont détaillées précisément dans le document sur l'exemple

Listing des packages



- Le listing de packages est détaillé dans la documentation supplémentaire
- Pour notre démonstration nous nous concentrerons sur le package « fr.cours.isima.persistence »

ISIMA

Java™ ORACLE

ArticleBean

- Commençons par observer notre objet ArticleBean
- C'est le pendant de notre table « Article »
- Notre ORM va donc lier notre table « Article » avec notre objet « ArticleBean »
- Grâce à JPA cette opération est très simple nous allons utiliser des annotations

L’entête de notre bean

```
@Entity(name = "Article")
@Table(name = "ARTICLE")
public class ArticleBean {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @NotNull(message = "Reference obligatoire")
    @Column(name = "reference", unique = true)
    @Size(min = 3, max = 10)
    private String reference;

    @Column(name = "description")
    @Size(max = 250, message = "La description doit faire moins de 250 caracteres")
    private String description;

    @JoinColumn(name = "category", nullable = false, referencedColumnName = "id")
    @OneToOne(targetEntity = CategorieBean.class)
    @NotNull(message = "Catégorie obligatoire")
    private CategorieBean category;
```

Détaillons le code

```
@Entity(name = "Article")
@Table(name = "ARTICLE")
public class ArticleBean {
```

`@Entity` déclare le bean comme un bean JPA. Le nom est celui utilisé dans les requêtes JPQL

`@Table` pour préciser le nom de la table associée.

`@UniqueConstraint` pour préciser que la colonne référence doit être unique (tout en étant pas la primary key)

Détaillons le code

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
```

- `@Id` déclare le champ comme étant la clé technique primaire
- `@GeneratedValue` : permet de sélectionner la stratégie de génération des identifiants auto signifiant qu'on laisse le driver décider

Détaillons le code

```
@NotNull(message = "Reference obligatoire")
@Column(name = "reference", unique = true)
@Size(min = 3, max = 10)
private String reference;
```

- `@NotNull` cette annotation n'est pas une annotation de mapping mais bien de contrôle des données
  - Ils se basent sur « les beans validators »
  - Les contrôles sont effectués au moment de la sauvegarde
- `@Column` lie la colonne de la table Article au champ référence
- `@Size` est aussi une annotation « bean validator »

Détaillons le code

```
@NotNull(message = "Reference obligatoire")
@Column(name = "reference", unique = true)
@Size(min = 3, max = 10)
private String reference;
```

- `@NotNull` cette annotation n'est pas une annotation de mapping mais bien de contrôle des données
  - Ils se basent sur « les beans validators »
  - Les contrôles sont effectués au moment de la sauvegarde
- `@Column` lie la colonne de la table Article au champ référence
- `@Size` est aussi une annotation « bean validator »

Détaillons le code

```
    @Column(name = "description")
    @Size(max = 250, message = "La description doit faire
moins de 250 caracteres")
    private String description;
```

- Analogue au code précédent à l'exception de la taille du message autorisée

Détaillons le code

```
@JoinColumn(name = "category", nullable = false, referencedColumnName = "id")
@OneToOne(targetEntity = CategorieBean.class)
@NotNull(message = "Catégorie obligatoire")
private CategorieBean category;
```

- Il est possible de déclarer les jointures au niveau même d'un bean
- Ici notre relation est le `@OneToOne` ce qui signifie que nous n'avons qu'une référence d'une entité vers une autre
- `@JoinColumn` signifie que l'identifiant de la catégorie (id) est stocké dans la colonne catégorie

Le modèle de données

- Avec le OneToOne nos deux tables ressemblent à ça

ARTICLE

ID	REFERENCE	LIBELLE	ID_CATEGORIE
1	ART-01	….	1

CATEGORIE

ID	LIBELLE
1	…

Les autres types de relation

- `@OneToMany` : signifie que notre colonne est liée à plusieurs objets. Par exemple plusieurs catégories. Dans notre cas la relation change puisque c'est la catégorie qui stockerait la relation :

ARTICLE

ID	REFERENCE	LIBELLE
1	ART-01	….

CATEGORIE

ID	LIBELLE	ID_ARTICLE
1	…	1

Voyez-vous pourquoi cette représentation est inadaptée à nos fonctionnalités ?

Les autres types de relation

- `@ManyToOne` : c'est la référence opposée à notre exemple précédent. Les deux configurations deviendraient respectivement :

```
public class CategorieBean {

    @ManyToOne
    @JoinColumn(nullable = false, name = "id_article")
    private ArticleBean article;

public class ArticleBean {

    @OneToMany(mappedBy = "article")
    @Size(min = 1)
    private List<CategorieBean> categories;
```

Les autres types de relation

- Toutefois nous ne voulons pas que la catégorie ne soit liée qu'à un seul Article !
- La relation OneToMany/ManyToOne signifie que les deux données sont étroitement liées
- Par exemple une personne peut avoir n numéro de téléphone.
- Dans ce cas là cette relation est parfaite !

La dernière

- Si nous voulions modéliser le fait que plusieurs articles puissent être liés à plusieurs catégories, la solution est le ManyToMany
- Typiquement ce cas surviendrait si nous souhaitions attacher à un article plusieurs Catégories. Toutefois ces catégories ne pourraient pas être liées à un article mais bien à plusieurs

Modèle de données

- Pour modéliser ce cas, il faut une table de jointure

ARTICLE

ID	REFERENCE	LIBELLE
1	ART-01	….

ARTICLE_CATEGORIE

ID_ARTICLE	ID_CATEGORIE
1	3

CATEGORIE

ID	LIBELLE
3	…

Les autres types de relation

- Voici dans le CategorieBean
- Pour l'ajouter dans ArticleBean il suffit d'inverser le « joinColumns »avec le « inverseJoinColunms »
- A noter le mode lazy pour éviter de ramener des enregistrements nombreux pour rien

```
public class CategorieBean {
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "ARTICLE_CATEGORIE", joinColumns = { @JoinColumn(name
= "ID_CATEGORIE", referencedColumnName = "ID") }, inverseJoinColumns = {
            @JoinColumn(name = "ID_ARTICLE", referencedColumnName =
"ID") })
    private List<ArticleBean> articles;
```

Et comment utilise-t-on nos beans ?

- Comme nous l'avons vu précédemment les annotations sont juste des méta-données, des informations
- Si aucun acteur ne les lit elles ne servent à rien !
- C'est pourquoi nous avons au niveau de notre persistence des objets permettant de lire et de sauvegarder nos données
- Ce sont les *DAO – Data Acces Object.* Parfois aussi appelé « *Repository* ».

ArticleDao

- Dans notre application, nous avons un premier écran qui liste l'intégralité des articles
- En toute logique nous avons donc un ArticleDao

```
public interface ArticleDao extends Dao<ArticleBean> {

    /**
     *
     * @return la liste de tous les articles
     */
    List<ArticleBean> findAllArticles();
…
}
```

ArticleDao

- Ciel mais c'est une interface !
- Pourquoi utiliser une interface plutôt qu'une classe concrète ??
- Il y a plusieurs raisons à cela :
  - D'une part nous découplons l'intention (nous voulons retrouver les articles) du moyen (retrouver les articles en bases de données).
  - Interface == contrat, Implementation == moyen
  - La testabilité est considérablement accrue (avec des outils comme JUnit & Mockito)

ArticleDao hérite de Dao

```
public interface Dao<T> {

    /**
     * Trouve l'objet par son id
     *
     * @param id
     * @return ne retourne jamais null, doit renvoyer une exception à la place
     */
    T findById(long id);

    /**
     *
     * @return la classe de bean utilisée pour les tests
     */
    Class<T> getBeanClass();

    /**
     * Sauvegarde le bean
     *
     * @param bean
     */
    void save(T bean);

    /**
     * Supprime toutes les donnes de la table. Attention de ne pas l'exposer en
     * tant que service !
     */
    void deleteAll();

}
```

Dao a quoi sert cette interface ?

- Tout simplement elle définit un ensemble d'opération minimales à définir quand on souhaite créer un Dao
- Le T est un type générique. Il devra contenir la classe du bean correspondant au Dao
  - Il s'agira toujours d'entités au sens Jpa c'est-à-dire des classes déclarées avec le @Entity
  - Il faut noter que ces classes sont porteuses de données mais ne doivent pas contenir de traitement
  - Les contrôles autres que ceux liés à la BeanValidation doivent se trouver dans les couches métiers

Ouvrons JpaArticleDao

- La sauvegarde des articles à la sauce JPA
- Revue du code de la méthode findAllArticles

```
public class JpaArticleDao implements ArticleDao {

    private final EntityManagerExecutor entityManagerExecutor = new EntityManagerExecutor();

    @Override
    public List<ArticleBean> findAllArticles() {
        return entityManagerExecutor.execute(em -> em.createQuery("select a from Article a",
ArticleBean.class).getResultList());
     }
}
```


ISIMA

Java™

ORACLE

A propos de findAllArticles

- Que fait-elle ?

  Elle exécute une requête JPQL ”`select a from Article a`”
- Pour pouvoir faire cela, il y a toute une gestion des connexions à réaliser
- Tout est géré au niveau du EntityManagerExecutor
- Si vous souhaitez en connaitre plus je vous invite à regarder la documentation de cette classe

ISIMA Java ORACLE

Regardons d'un peu plus près le corps

```
return entityManagerExecutor.execute(em -> em.createQuery("select a
from Article a", ArticleBean.class).getResultList());
```

- Le code est très, très simple. Nous écrivons notre requête, demandons le résultat sous forme de list getResultList
- Vous avez beaucoup d'autres exemples dans les classes JpaArticleDao et JpaCategorieDao

Observons la méthode save()

```
@Override
public void save(ArticleBean articleBean) {
    if (articleBean.getId() > 0) {
        entityManagerExecutor.update(articleBean);
    } else {
        try {
            entityManagerExecutor.insert(articleBean);
        } catch (final RuntimeException e) {
            // On repasse l'id à zero sinon on ne pourra pas enregistrer la
            // valeur

            articleBean.setId(0);
            throw e;
        }
    }
```

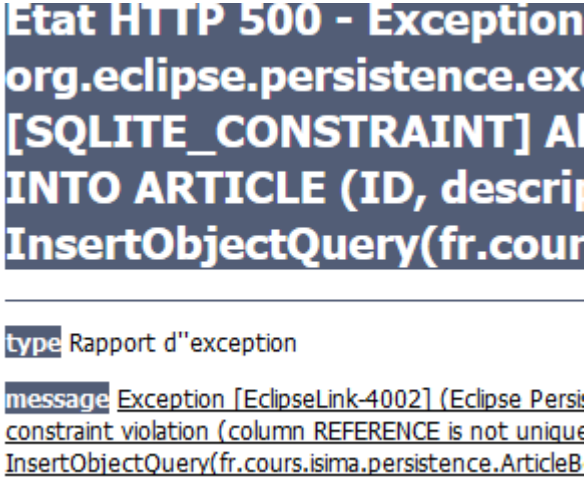
- Remarquez simplement le catch pour remettre l'id à 0
- Quand la validation échoue (champ mal rempli) l'id est positionné à une valeur qui n'existe pas et qui peut tromper nos traitements ! Le remettre à 0 résout le problème

Configurer et mettre en place

- JPA permet de mettre en place facilement une persistence
- Je vous invite à lire la documentation supplémentaire en annexe du cours, notamment sur la partie « persistence »
- Celle-ci détaillera la configuration du fichier « persistence.xml » nécessaire pour lancer votre application

Limites

- La contrainte Unique est mal gérée nativement dans l'implémentation EclipseLink de JPA
- Par exemple si vous n'utilisez pas l' EntityManagerExecutor cela ne fonctionnera pas
- Si vous ajoutez une contrainte d'unicité sur une colonne, n'hésitez pas à copier la gestion qui se trouve dans SaveArticleServlet



ISIMA

Java™ ORACLE