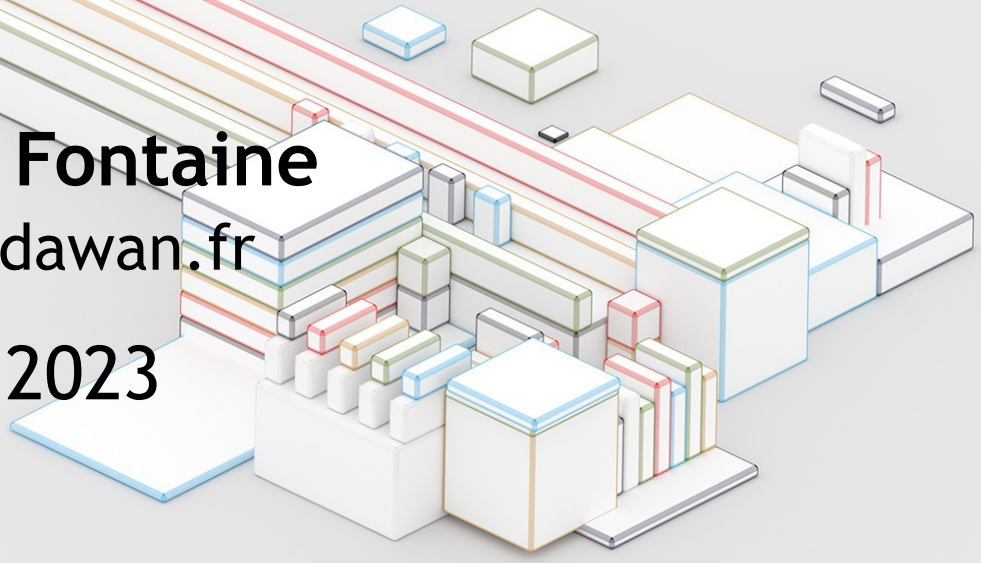
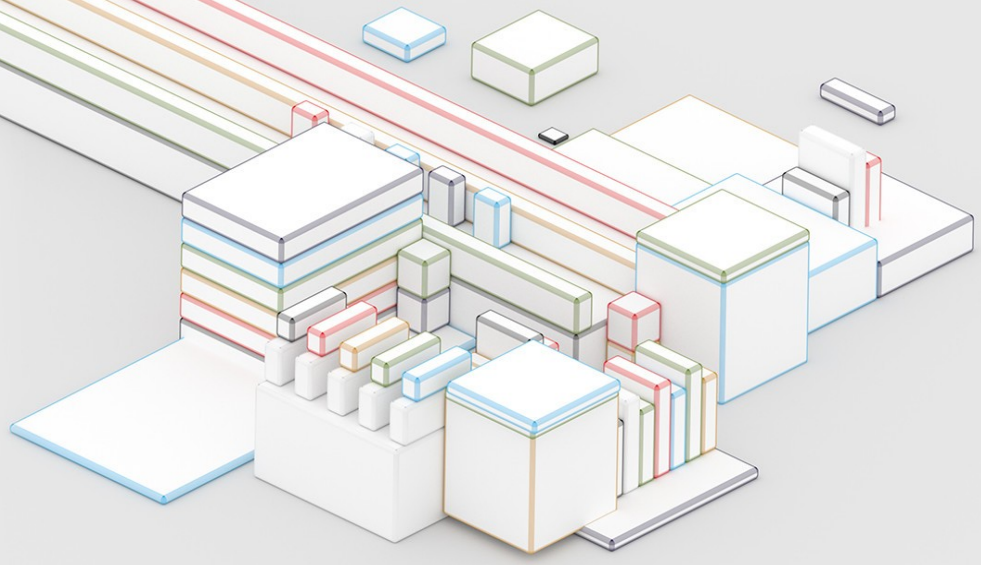
Spring MVC

Christophe Fontaine
cfontaine@dawan.fr

02/01/2023



Vue



Types de vues

Spring permet d'utiliser différents types de vues :

- JSP/JSTL
- Thymeleaf
- FreeMarker
- Groovy Markup
- Tiles
- XML, CSV, JSON

Liaison vues-controleur

- Utilisation d'un modèle contenant des attributs

```
@Controller
public class WelcomeController {
    @RequestMapping("/home")
    public String home() {
        return "home";
    }

    @GetMapping("/greeting")
    public String greetingByGet() {
        return "greetingGet";
    }

    @PostMapping("/greeting")
    public String greetingByPost(Model model) {
        model.addAttribute("titre", "Test de vue");
        model.addAttribute("description", "pour tester le passage
                                           d'infos du controleur");
        return "default";
    }
}
```

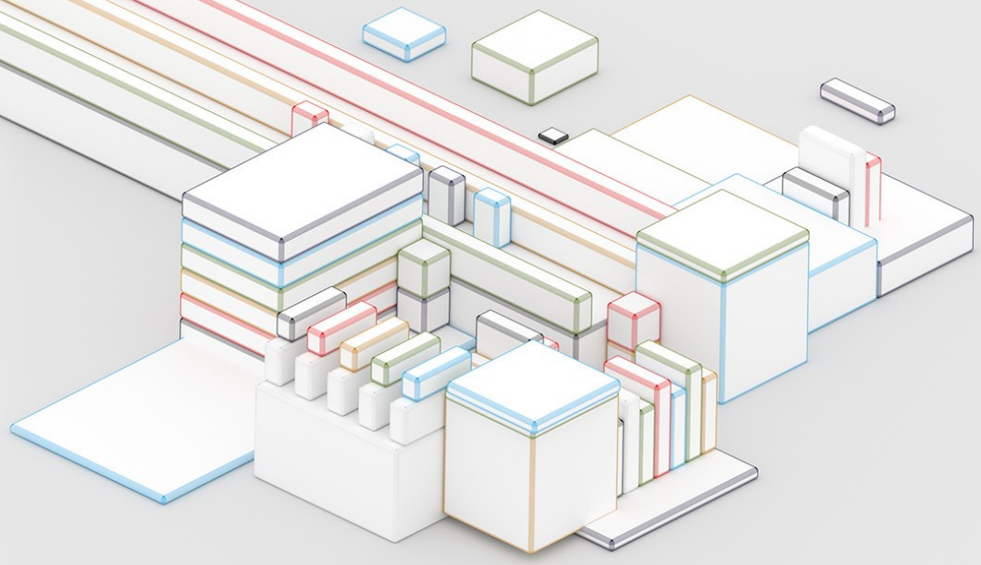
Définition d'une vue JSP

- Utilisation d'un ViewResolver pour déterminer le type et l'emplacement du fichier

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
                                                    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>${titre}</title>
    </head>
    <body>
        <h2>${titre}</h2>
        <p>${description}</p>
    </body>
</html>
```

Dawan
Training is our way

Templating



Templating

Complément de JSP

- SiteMesh

  http://wiki.sitemesh.org/wiki/display/sitemesh3/Getting+Started+with+SiteMesh+3

Alternatives à JSP

- Thymeleaf

  https://www.thymeleaf.org/

- FreeMarker

  https://freemarker.apache.org/

- Velocity

  https://velocity.apache.org/

FreeMarker

- Apache FreeMarker est un moteur de template qui génére des sorties texte (pages HTML, e-mails, fichiers de configuration, code source …)
- **Dépendances maven**

```
<dependency>
    <groupId>org.freemarker</groupId>
    <artifactId>freemarker</artifactId>
    <version>2.3.31</version>
</dependency>
```

- **Extension des vues** → .ftl, .flth

FreeMarker

- **Configuration du view resolver**

```
<beans:bean id="freemarkerConfig"
    class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <beans:property name="templateLoaderPath"
        value="/WEB-INF/views/ftl" />
</beans:bean>

<beans:bean id="viewResolverFreemarker"
    class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <beans:property name="order" value="1" />
    <beans:property name="cache" value="true" />
    <beans:property name="prefix" value="" />
    <beans:property name="suffix" value=".ftl" />
</beans:bean>
```

order permet d'indiquer l'ordre d'appel des view resolvers (commence à 1)

le view resolver pour les pages JSP doit toujours être le dernier

FreeMarker

- **Data-model**

  Le data-model peut être visualiser comme un arbre

```
(root)
  +- animals
  |    |
  |    +- elephant
  |    |    |
  |    |    +- size = "large"
  |    |    |
  |    |    +- price = 5000
  |    +- python
  |         |
  |         +- size = "medium"
  |         |
  |         +- price = 4999
  +- message = "It is a test"
```

  Il peut contenir :

  - **hash** → objet
  - **scalar** → variable (String, nombre, date-time, date, time, boolean)
  - **sequence** → tableau, list (type indexé)

FreeMarker

- **Contenu d'une page**
  - **Interpolation : ${…}**

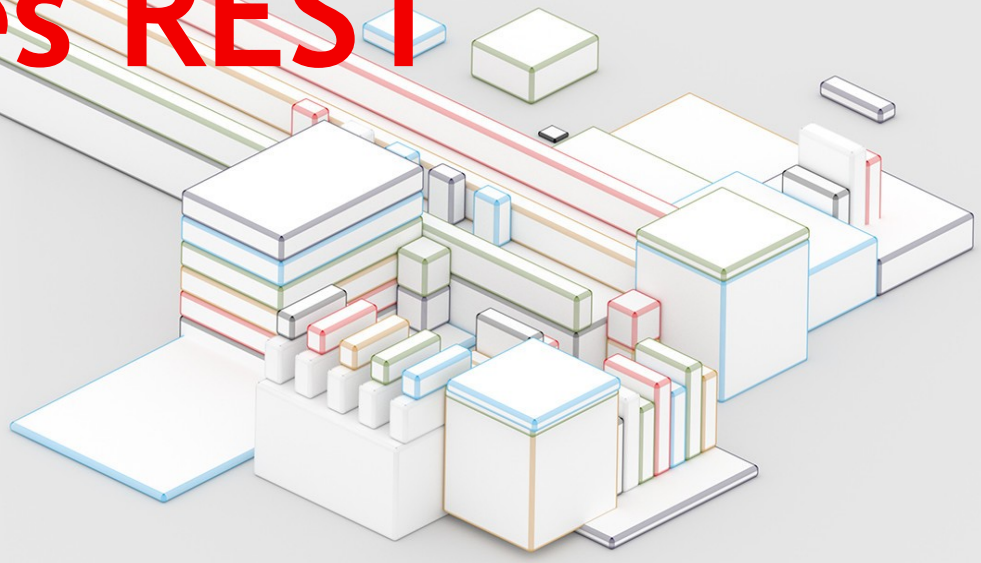
    FreeMarker le remplacera par la valeur réelle de l'expression à l'intérieur des accolades
  - **FTL tags : #…**

    tag freemarker qui seront interprété par freemarker et ne feront pas partie de la sortie, le nom des tags commence par **#**

    avec les tags FTL, on fait référence à des directives
  - Commentaire : **<#-- -->**

Dawan
Training is our way

Implémenter des web services REST



Architecture REST

Qu'est ce que c'est REST ?

- Representational State Tranfer
- Repose sur l'architecture originelle du Web
- Utilise uniquement HTTP et un ensemble restreint d'actions :

  GET, **POST**, **PUT**, **DELETE**

- Utilise une URI (Uniform Resource Identifier) comme moyen d'interroger le service
- Des types MIME pour indiquer la nature des informations retournées par le service

Architecture REST

Selon son créateur, **Roy Fielding,** une architecture REST doit respecter :

- **Client-serveur** : on sépare le producteur du consommateur pour garantir un faible couplage
- **Sans état** : une requête doit contenir l'ensemble des informations nécessaires au serveur pour traiter la demande
- **Mise en cache** : la réponse serveur peut contenir des informations comme la date de création ou de validité de la réponse
- **Système par couches** : on accède à des ressources individuelles, une fonctionnalité complexe entraîne un ensemble d'appel au serveur

Contrôleur REST

- Annoter le contrôleur avec **@RestController**
  - Évite de spécifier **@ResponseBody** sur chacune des méthodes du contrôleur
  - Format de la réponse : données au format JSON, XML possible
- Dépendance : jackson-databind
- Conversion automatique des données du bean au format JSON

Contrôleur REST

- **Lecture**
  - @GetMapping
- **Création**
  - @PostMapping
  - Données dans @RequestBody
- **Mise à jour**
  - @PutMapping
  - Données dans @RequestBody
  - Id en PathVariable
- **Suppression**
  - @DeleteMapping
  - Id en PathVariable

Utilisation de XML

- Annoter le bean avec **JAXB**

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.PROPERTY)
public class User implements Serializable{
    private int id;
    private String firstName;
    private String lastName;

    @XmlAttribute
    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;
    }
}
```

- Indiquer au contrôleur que la méthode produit du XML

Exemple de contrôleur REST

```java
@RestController
@RequestMapping("trainings") //@Path
public class WSTrainingController {
    @Autowired
    private TrainingDao trainingDao;
    @GetMapping(value = "/json", produces = "application/json")
    public List<Training> listAll() {
        return trainingDao.findAllBasicInfosTrainings();
    }
    @GetMapping(value = "/xml", produces = "application/xml")
    public List<Training> listAllXml() {
        return trainingDao.findAllBasicInfosTrainings();
    }
    @GetMapping(value = "/{id}", produces = "application/json")
    public Training find(@PathVariable int id) {
        return trainingDao.findBasicInfosTrainingById(id);
    }
    @PostMapping(value = "/insert", consumes = "application/json")
    public void find(@RequestBody Training training) {
        trainingDao.persist(training);
    }
}
```

Codes retour HTTP

- Annoter la méthode du contrôleur avec @ResponseStatus
  - Classe **HttpStatus**

Dawan
Training is our way

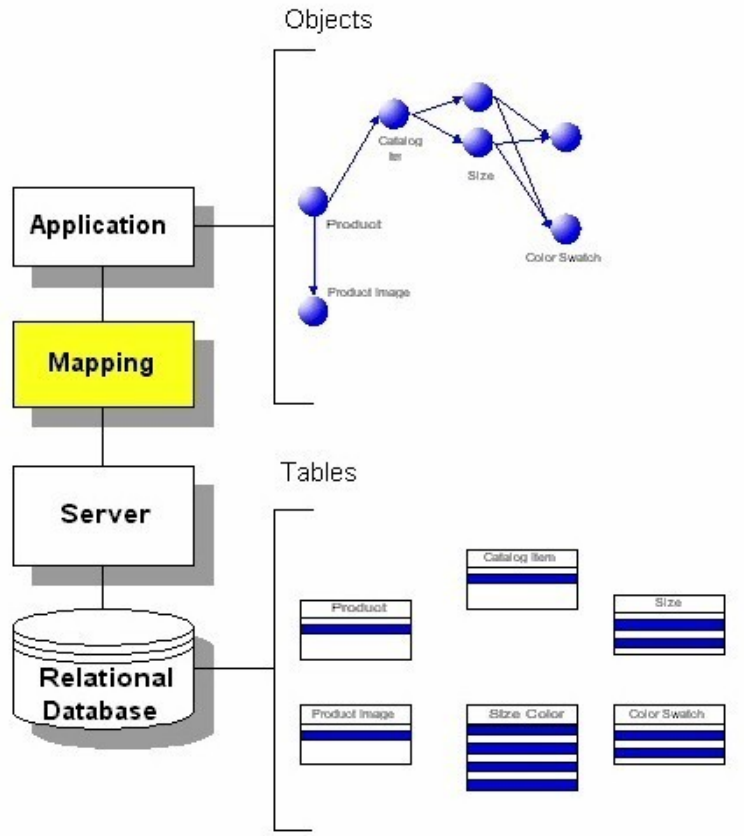
Réaliser un mapping des données avec Spring ORM

Accès aux Bdds en Java

- JDBC (Java DataBase Connectivity)
- Inconvénients :
  - Nécessite l'écriture de nombreuses lignes de codes répétitives
  - La liaison entre les objets et les tables est un travail de bas niveau
- Exemple de code + pattern DAO

Impedance Mismatch

- Désigne la difficulté de stocker des graphes d'objets interconnectés e mémoire sous forme de tables dans des système de base de données relationnelles (RDBMS)
- Granularité
- Héritage
- Égalité
- Association
- Navigation



Mapping relationnel-objet

Concept permettant de connecter un modèle objet à un modèle relationnel

Couche qui va interagir entre l'application et la base de données

- **Pourquoi utiliser ce concept?**
  - Pas besoin de connaître l'ensemble des tables et des champs de la base de données
  - Faire abstraction de toute la partie SQL d'une application

Mapping relationnel-objet

- **Avantages** :
  - Gain de temps au niveau du développement d'une application
  - Abstraction de toute la partie SQL
  - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients :**
  - L'optimisation des frameworks/outils proposés
  - La difficulté à maîtriser les frameworks/outils

Critères de choix d'un ORM

- La facilité du mapping des tables avec les classes, des champs avec les attributs
- Les fonctionnalités de bases des modèles relationnel et objet
- Les performances et optimisations proposées : gestion du cache, chargement différé
- Les fonctionnalités avancées : gestion des sessions, des transactions
- Intégration IDE : outils graphiques
- La maturité

JPA

- Une API (Java Persistence API)
- Des implémentations

- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

Concepts vs Classes

Concept	JDBC	Hibernate	JPA
Ressource	Connection	Session	EntityManager
Fabrique de ressources	DataSource	Sessionfactory	EntityManagerFactory
Exception	SQLException	HibernateException	PersistenceException

Configuration de Spring Hibernate

- Modification du contexte Spring pour la configuration :
  - dataSource (locale ou distante avec JNDI)
  - sessionFactory
  - transactionManager
  - hibernateTemplate

Supports de Spring

Classes de support	JDBC	Hibernate	JPA
Classe template	JdbcTemplate	HibernateTemplate	JPATemplate
Classe de support de DAO	JdbcDaoSupport	HibernateDaoSupport	JPADaoSupport
Gestionnaire de transactions	DataSource TransactionManager	HibernateTransaction Manager	JPATransaction Manager

Objets Hibernate

- **SessionFactory**
  Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de Session et un client de ConnectionProvider
  Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau processus ou au niveau cluster

- **Session**
  Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance
  Encapsule une connexion JDBC, une Factory (fabrique) des objets Transaction. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant

Objets Hibernate

- **Objets et Collections persistants**
  Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets métier ; la seule particularité est qu'ils sont associés avec une (et une seule) Session

  Dès que la Session est fermée, ils seront détachés et libres d'être utilisés par n'importe quelle couche de l'application (i.e. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données)

- **Objets et collections transitoires (Transient)**
  Instances de classes persistantes qui ne sont actuellement pas associées à une Session. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elles n'ont pu être instanciées par une Session fermée

Objets Hibernate

- **Transaction : (Optionnel)**
  Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique.
  Abstrait l'application des transactions sous-jacentes
  Une Session peut fournir plusieurs Transactions dans certains cas

- **TransactionFactory : (Optionnel)**
  Une fabrique d'instances de Transaction. Non exposé à l'application, mais peut être étendu/implémenté par le développeur

HibernateTemplate

- Gère toutes les interactions avec la base de données :
- **Recherche** : find, getReference
- **Sauvegarde** : persist, merge
- **Suppression** : remove
- **Requêtage** : langage JPQL (HQL), createQuery,
- **Transaction** : begin, commit, rollback
  (support des transactions par annotations)

Annotations JPA

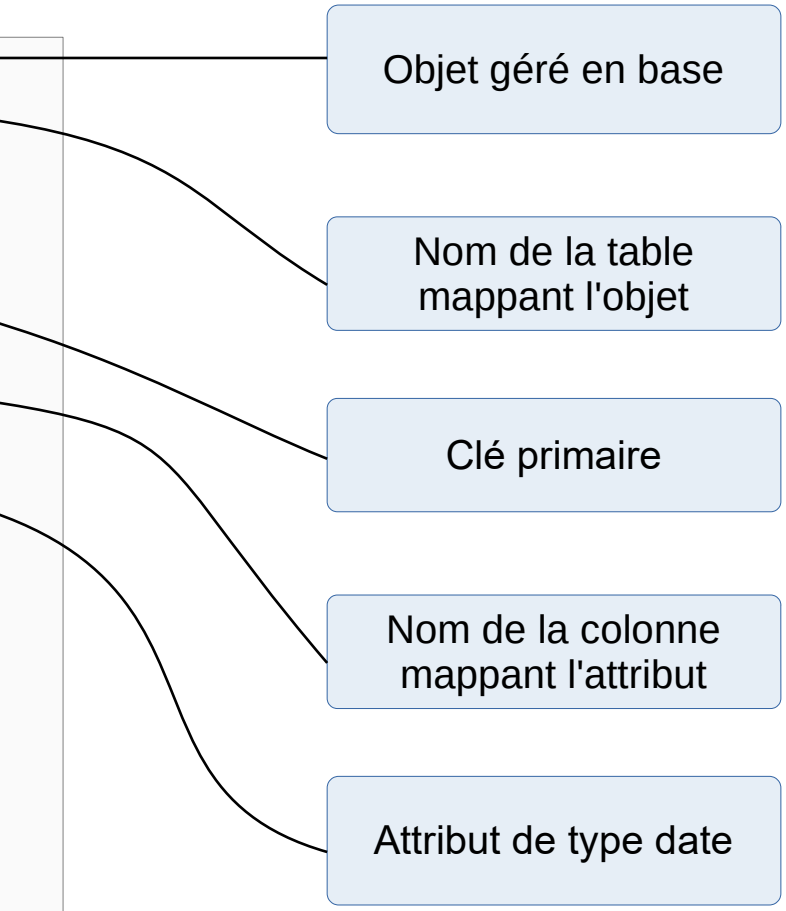
- JPA est l'API de spécification de la couche de persistance au sein d'une application client lourd / léger. (les spécifications EJB 3 ne traitent pas directement de la couche de persistance)

- JPA 2 est la partie de la spécification EJB 3.1 qui concerne la persistance des composants :

  http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/

Mapping relationnel

```java
@Entity
@Table(name = "person")
public class Person implements Serializable {

    @Id()
    private long id;

    @Column(name = "first_name")
    private String firstName;

    @Temporal(TemporalType.DATE)
    @Column(name = "birth_day")
    private Date birthDay;public Person() {
        super();
    }

    // Getters / Setters

    …
```



Gestion de la concurrence

- La gestion de la concurrence est essentielle dans le cas de longues transactions.

  Hibernate possède plusieurs modèles de concurrence :

- **None** : la transaction concurrentielle est déléguée au SGBD. Elle peut échouer
- **Optimistic (Versioned)** : si nous détectons un changement dans l'entité, nous ne pouvons pas la mettre à jour
  @Version(Numeric, Timestamp, DB Timestamp) :
  on utilise une colonne explicite Version (meilleure stratégie)
- **Pessimistic** : utilisation des LockMode spécifiques à chaque SGBD

  https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html

Gestion de la concurrence Versioned

- L'élément @Version indique que la table contient des enregistrements versionnés. La propriété est incrémentée automatiquement par Hibernate

  Automatiquement, la requête générée inclura un test sur ce champ :

- UPDATE Player SET version = @p0, PlayerName = @p1

  WHERE PlayerId = @p2

  AND version = @p3;

Gestion de la concurrence Pessimistic

- Nous pouvons exécuter une commande séparée pour la base de données pour obtenir un verrou sur la ligne représentant l'entité :

```
player = session.get(Player.class,1);
session.Lock(player, LockMode.Upgrade);

player.PlayerName = "other";
tx.Commit();

SELECT PlayerId FROM Player with (updlock, rowlock) WHERE
PlayerId = @p0;

UPDATE Player SET PlayerName = @p1
WHERE PlayerId = @p2 AND PlayerName = @p3;
```

- **Inconvénient** : l'attente pour l'obtention du verrou (pour la modification si la ligne est verrouillée)
  Une exception est déclenchée après le TimeOut parce que nous ne pouvions pas obtenir le verrou

Relations entre Entity Beans

- **One-To-One**
  @OneToOne
  @PrimaryKeyJoinColumn
  @JoinColumn
- **Many-To-One**
  @ManyToOne
  @JoinColumn
- **One-To-Many**
  @OneToMany
  (pas de @JoinColumn)
- **Many-To-Many**
  @ManyToMany
  @JoinTable

Traitement en cascade

- Les annotations @OneToOne, @OneToMany, @ManyToOne et @ManyToMany possèdent l'attribut cascade
- Une opération appliquée à une entité est propagée aux relations de celle-ci par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également
- 4 Types :
  - **PERSIST**
  - **MERGE**
  - **REMOVE**
  - **REFRESH**
- **CascadeType.ALL** : cumule les 4

Héritage

- Tout est dans la même table
- Une seule table
- Une colonne, appelée “Discriminator” définit le type de la classe enregistrée.
- De nombreuses colonnes inutilisées

```
@Entity(name="COMPTE")
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
      discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{…}

@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte implements Serializable {…}
```

Héritage : SINGLE_TABLE

- Tout est dans la même table
- Une seule table
- Une colonne, appelée “Discriminator” définit le type de la classe enregistrée.
- De nombreuses colonnes inutilisées

```
@Entity(name="COMPTE")
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
      discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{...}

@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte implements Serializable {...}
```

Héritage : TABLE_PER_CLASS

- Chaque Entity Bean fils a sa propre table
- Lourd à gérer pour le polymorphisme

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable
{…}
La clé @Id ne peut pas être @GeneratedValue (Identity) (le mettre en TABLE)


@Entity
public class CompteEpargne extends Compte implements Serializable {…}
```

Héritage : JOINED

- Chaque Entity Bean a sa propre table
- Beaucoup de jointures

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable
{…}

@Entity
public class CompteEpargne extends Compte implements
Serializable {…}
```

Héritage : récapitulatif

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Avantages	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
Inconvénients	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse des performances

Requêtes

- JPA-QL (HQL)

  https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/queryhql.html

- Criteria

  https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/querycriteria.html

- SQL

  https://docs.jboss.org/hibernate/orm/3.6/reference/fr-FR/html/querysql.html

- Requêtes nommées

Gérer les exceptions

- Récupérer les DataAccessException

  La cause de l'exception est l'erreur SQL

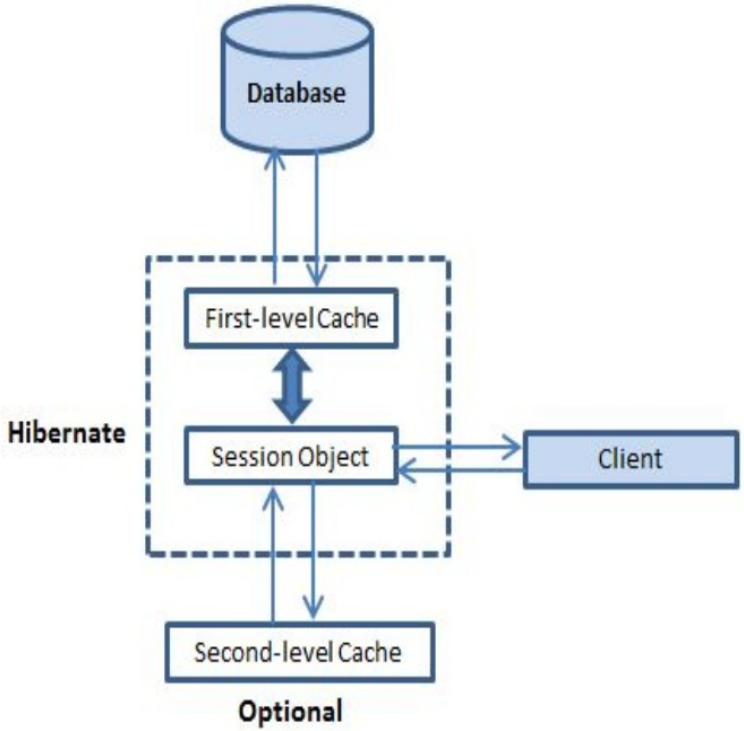
```
try {
	dao.delete(delete1);
} catch (DataAccessException e) {
	System.out.println(e.getMessage());
	System.out.println(e.getCause());
	SQLException sqlex = (SQLException) e.getCause();
	System.out.println(sqlex.getErrorCode());
	System.out.println(sqlex.getSQLState());
}
```

Cache de niveau 1

Hibernate possède 2 niveaux de cache

- **Cache de niveau 1** :

  - Chaque fois que vous passez un objet à **Save**(), **Update**() ou **SaveOrUpdate**() ou que vous récupérez un objet avec **Load()**, **Get**(), **List**(), ou **Enumerable**(), cet objet est ajouté au cache interne de la session (Isession)



  - Quand **Flush**() est ensuite appelée, l'état de cet objet va être synchronisé avec la base de données. Si vous ne voulez pas que cette synchronisation se produit ou si vous traitez un grand nombre d'objets et la nécessité de gérer efficacement la mémoire :

  - La méthode **evict**() peut être utilisée pour supprimer l'objet et ses collections dépendantes du cache de premier niveau

  - La méthode clear() permet de vider complètement le cache de la session

Cache de niveau 1 (suppression)

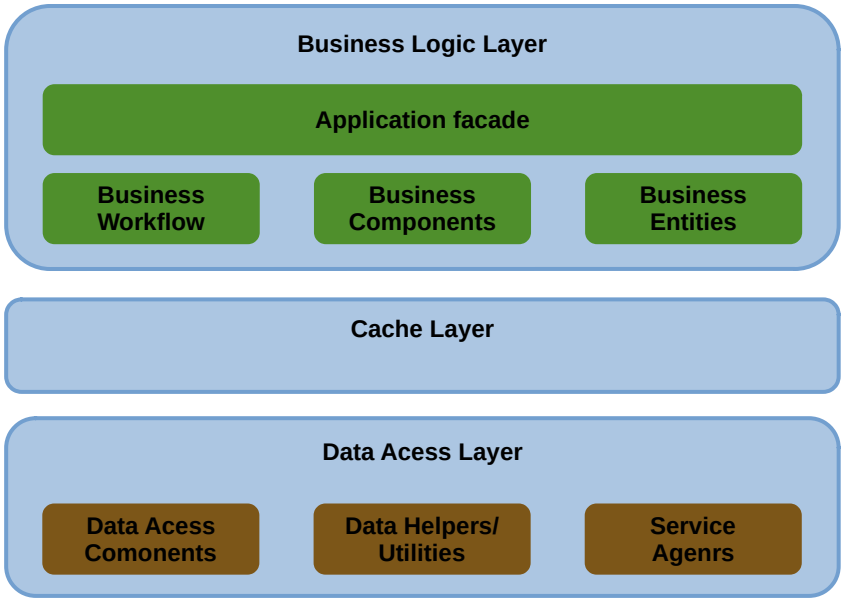
```
List<Book> books = session.createQuery("from Book").list();

//un grand résultat
for (Book b : books) {
      DoSomethingWithABook(b);
      Session.evict(b);
}
```

Cache de niveau 2 Introduction

- La mise en cache des entités est une technique très importante pour améliorer les performances de l'application
  Généralement, on introduit une couche de mise en cache dans une architecture multi-couches **avant la couche d'accès aux données**

  Parfois, on utilise les composants web côté présentation pour mettre en cache les entités :

  - Session
  - Application (servletContext)

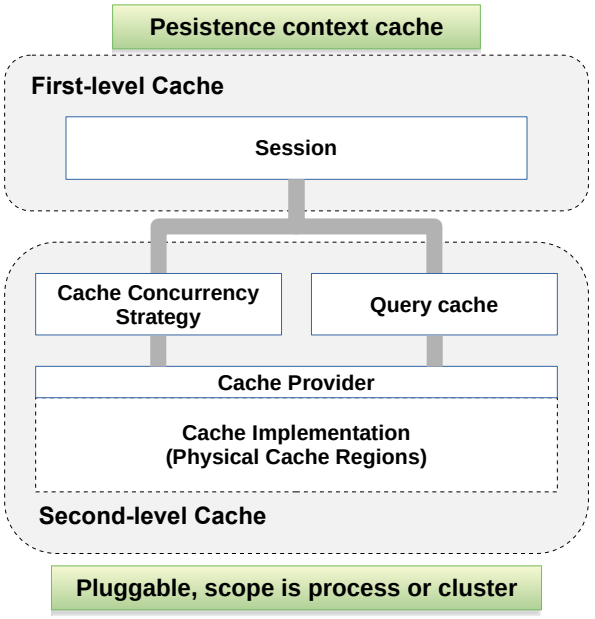


Hibernate fournit un cache de niveau 2 (couche accès aux données) qui permet de s'abstraire de l'utilisation des composants de la couche présentation ou métier

Cache de niveau 2 Configuration

```
<property name="cache.use_second_level_cache">true</property>
<property name="cache.provider_class">...</property>
<property name="cache.use_query_cache">true</property>
<property name="prepare_sql">true</property>
```

- De multiples cache providers sont disponibles :
  https://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/performance-cache.html
- Exemple Cache :
  http://www.baeldung.com/hibernate-second-level-cache



Cache de niveau 2
Caching des entités

- Pour spécifier la mise en cache d'une entité ou d'une collection, on ajoutela'annotation @Cache dans le mapping de la classe ou de la collection ou une conf. Xml :

- L'attribut **usage spécifie** la stratégie de gestion de la concurrence

  - **read only :** cache en lecture seule, pas de modification d'instances
    persistantes (manière la plus simple et la plus performante)
  - **read/write :** si l'application doit mettre a jour des données On doit s'assurer que la transaction est terminée et que la session est fermee (strict).
  - **nonstrict read/write :** si l'application doit occasionnellement mettre a jour des donnees (multiples transactions simultanees)

Cache de niveau 2
Suppression d'entités

- **Suppression d'un book spécifique** :
  sessionFactory.evict(typeof(Book), bookId);
- **Suppression de toutes les instances Book** :
  sessionFactory.evict(typeof(Book));
- **Suppression d'une collection Chapters d'un objet Book :**
  sessionFactory.evictCollection("Book.Chapters", bookId);
- **Suppression de toutes les collections Chapters :**
  sessionFactory.evictCollection("Book.Chapters");

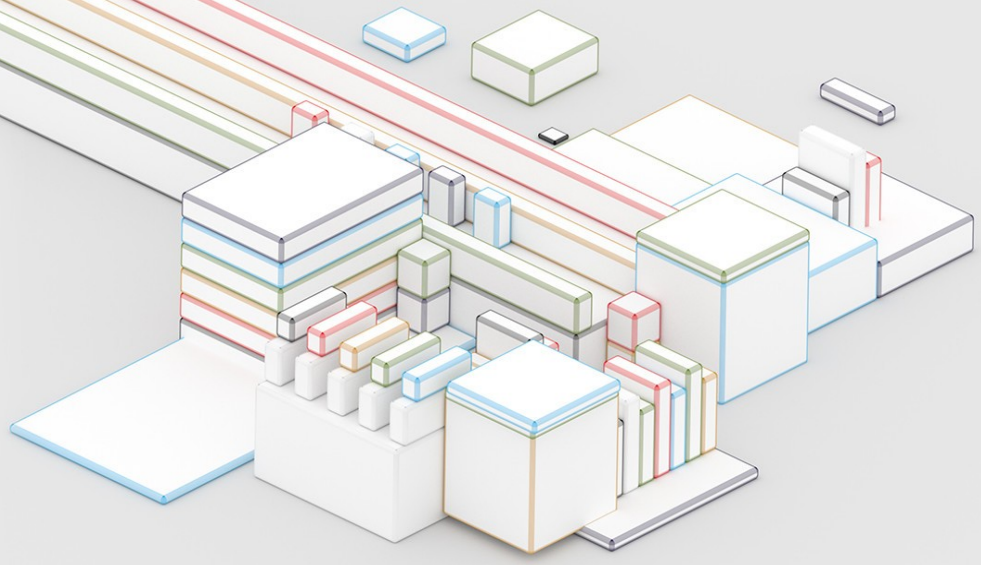
Cache de niveau 2
Caching de requêtes

- Les résultats d'une requête peuvent être mis en cache
  Utile uniquement pour les requêtes exécutées fréquemment avec les mêmes paramètres.
- Configuration :
  `cache.use_query_cache = true`
- Utilisation : `Query.setCacheable(true)`

```
List<Book> booksList2 = session.createQuery("FROM Book b")
.setCacheable(true).setCacheRegion("ShortTermCacheRegion")
.list();
// ……
```

- Forcer le rafraîchissement :
  `sessionFactory.evictQueries(regionName)`

Annexe



Namespaces de Spring

- **Beans** : paramétrage des objets
- **Utils** : faciliter le paramétrage des beans en XML, collections ...
- **Jee** : accéder à des objets JEE (ressources JNDI, stateless bean)
- **Lang** : utilisation d'objets écrits en langage dynamique type Groovy
- **Jms** : utilisation de l'API de messaging Java
- **Tx** : gérer des transactions
- **Aop** : programmation par aspect de Spring
- **Context** : contexte de l'application (annotations, …)
- **Jdbc** : configuration de datasources
- **Cache** : gestion de cache