Formation JPA
04/03/2024

Votre formateur

- François Caremoli

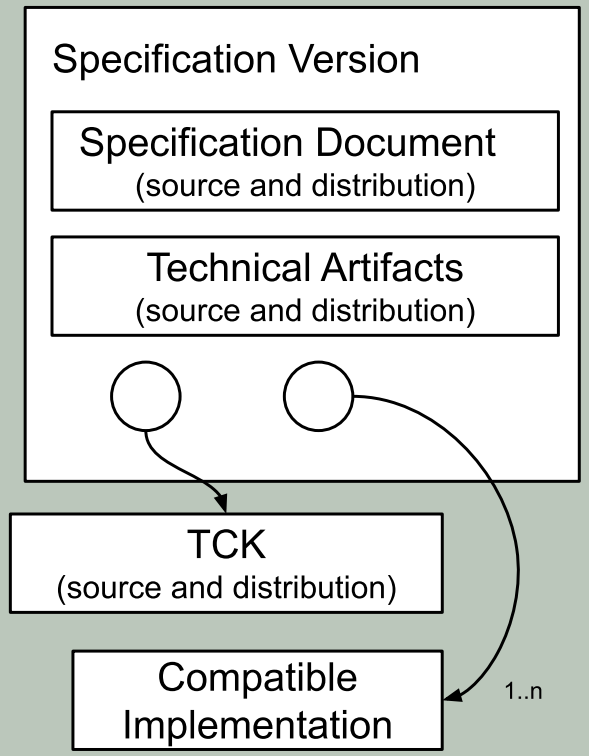
- Développeur pendant 7 ans sur des applications Java/JavaEE.
- Profil full stack (du SQL ↔ HTML ou HTTP)
- Lead Developper pendant 8 ans
- Architecte SI pendant 2 ans.
- Formateur en parallèle.

Introduction
Téléchargement et installation des composants

-
- Télécharger et décompresser :
- Un Eclipse à jour, version développeur, contenant Git, Maven et une JDK avec une version >= 17
- Un client Git à jour. Vérifier que vous pouvez cloner le repo de la formation.
-
- Un IDE . La formation se fera avec Eclipse.

JPA dans JEE

JPA dans JEE
Spécifications et implémentations



JavaEE ou Jakarta EE est un ensemble de spécifications, ou API. Une spécification se compose au moins d'une documentation, d'un jar contenant des interfaces, et d'un TCK.

Une bibliothèque ou un framework qui implémente la spécification doit réussir les tests du TCK. Elle se compose d'un ou plusieurs jars qui contiennent (entre autres) des classes qui implémentent les interfaces.

Les spécifications évoluent avec le temps, les implémentations aussi.

JPA dans JEE
Principales API

JAX-RS

Webservices REST

Faces

Framework MVC

Servlet/JSP

Interfaces HTTP/HTML

Context & Dendendency Injection

Injection de dépendance

EJB

Enterprise Java Beans

JPA

Mapping Objet/Relationnel

JTA

Transactions

Validations

Validation de Beans

Environ une quarantaine : https://jakarta.ee/specifications/

JPA dans JEE
Serveurs compatibles

Eclipse Glassfish

L'implémentation de référence

Apache TomEE

Comme Tomcat, version J2EE

IBM Websphere/ Liberty

Oracle Weblogic

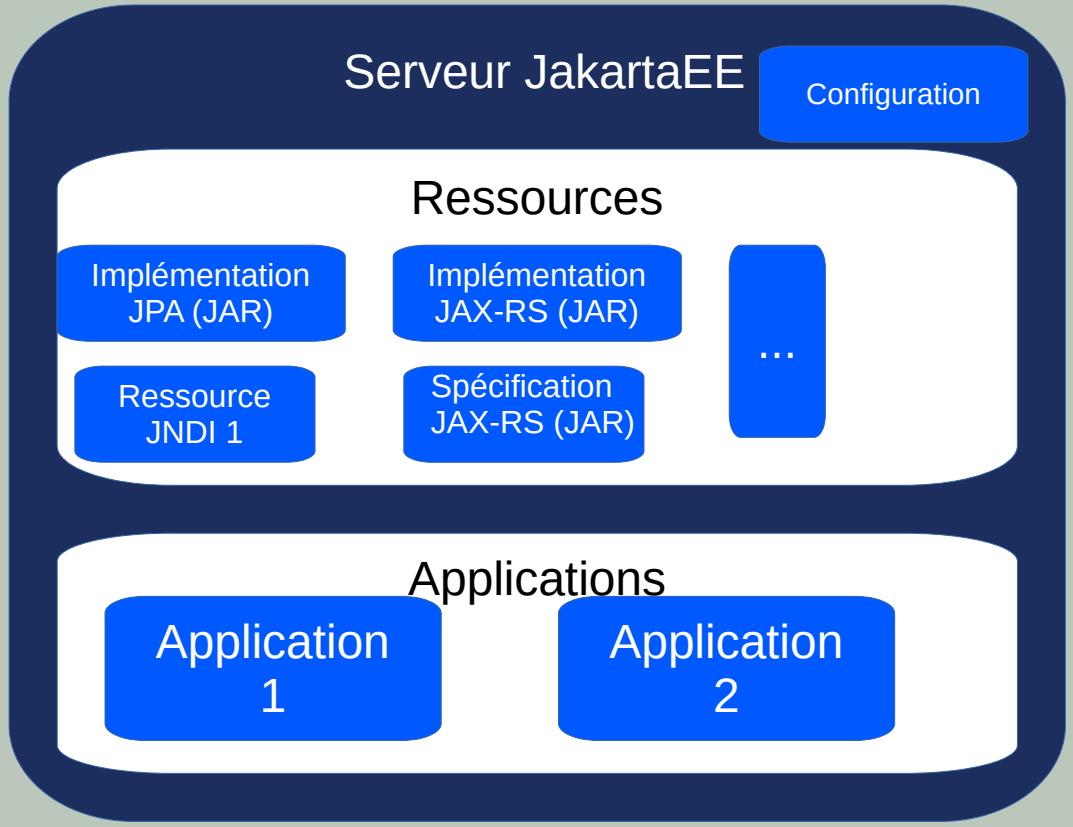
Paraya

Widlfly

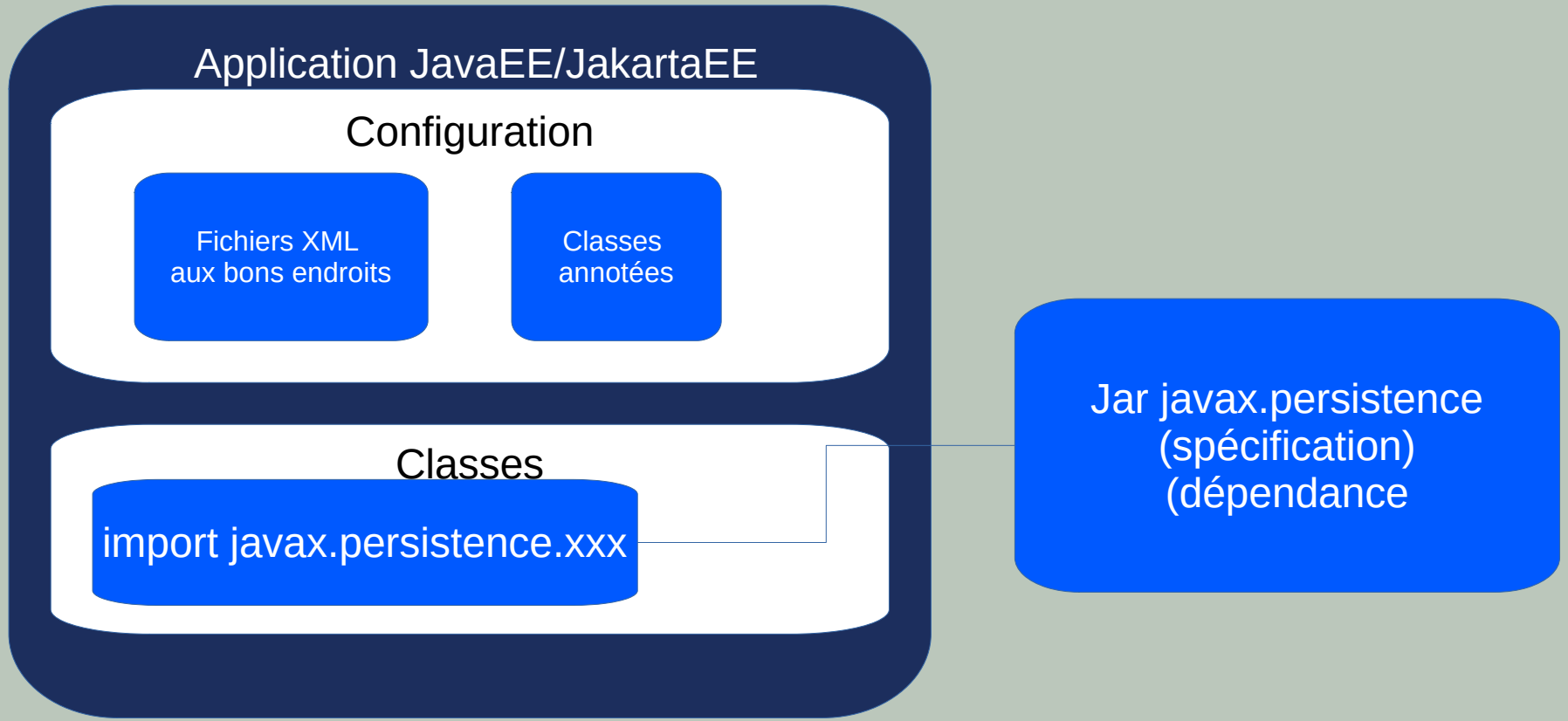
Nouveau JBOSS

Et d'autres : https://jakarta.ee/compatibility

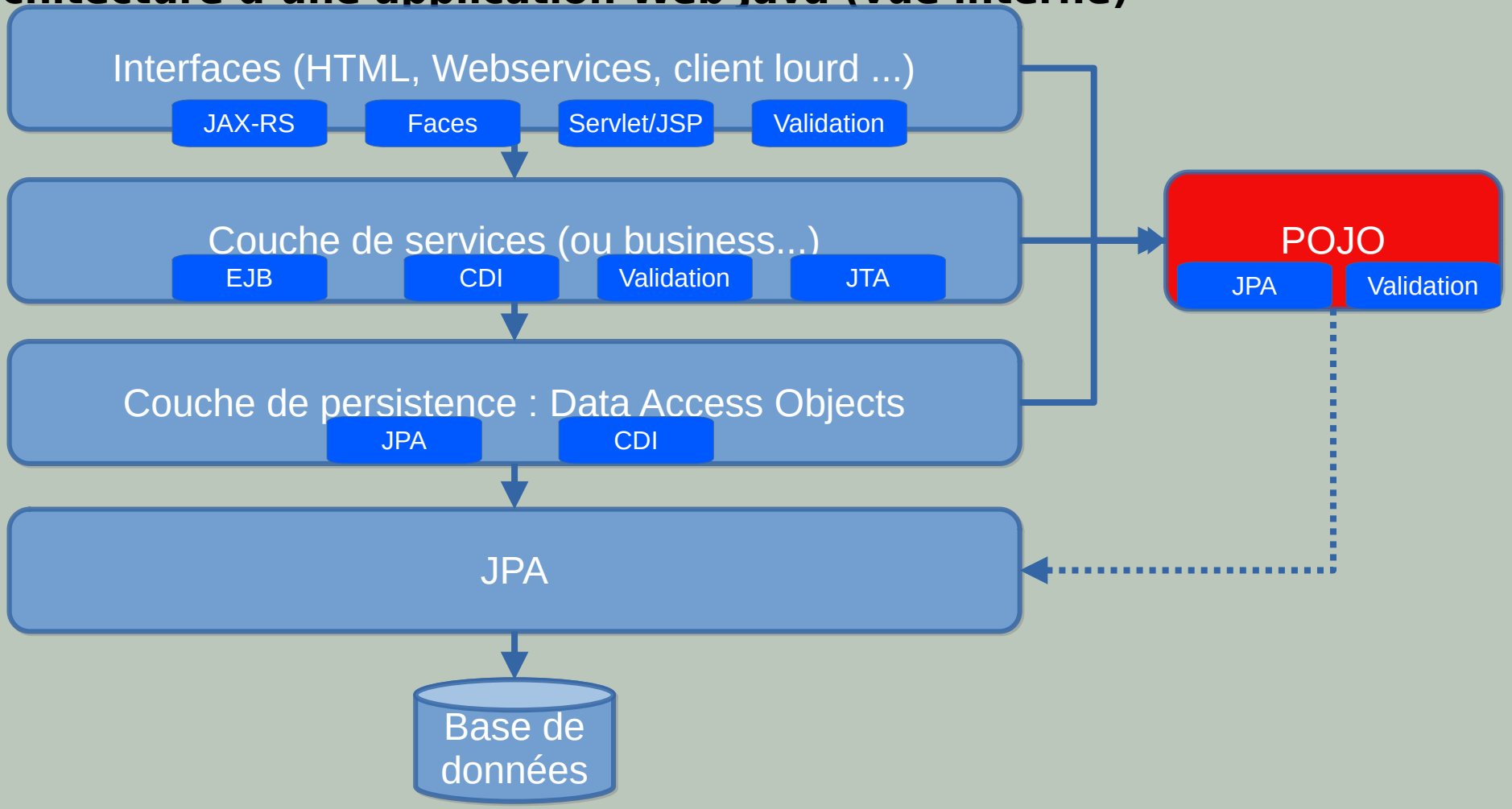
JPA dans JEE
Contenu d'un serveur JavaEE/JakartaEE



JPA dans JEE
Architecture d'une application Web Java (vue externe)



JPA dans JEE
Architecture d'une application Web Java (vue interne)



JPA dans JEE
Passé et futur de JEE

Historique :

- Java EE 1 1996
- Java EE 8 2017

Depuis, transfert de Java EE à une fondation OpenSource (Eclipse), MAIS :

- Javax est une marque déposée, qui doit disparaître
- La montée de version de la JDK n’est pas simple.

Trois versions suivent donc Java EE8 :

- Jakarta EE8 : équivalent techniquement et fonctionnellement à JavaEE8, mis à part le nom ← version utilisée pour la formation
- Jakarta EE9 : équivalent fonctionnellement à JavaEE9, techniquement les noms de package J2EE sont passés de javax.* à jakarta.*
- Jakarta EE9.1 : supporte la JDK11
- Jakarta EE10 : ajoute des fonctionnalités. ← version à utiliser dans le futur

Les concurrents à un serveur JEE

- Le framework Spring est bâti sur un CDI, il s'est étoffé de nombreuses fonctionnalités et concurrence JEE.
- Les implémentations JEE étant assez peu dépendantes, on peut choisir certaines de ces implémentations pour un projet, et les assembler "à la main" dans un serveur non JEE (c'est le cas de bon nombre d'applications qui tournent sous Tomcat).
- JEE s'est simplifié au fur du temps, ce qui en fait aujourd'hui une alternative viable à ces concurrents. De plus, il s'est rapproché de ses concurrents, développer une application JEE ou Spring n'est pas beaucoup éloigné au niveau architecture.
- Concernant JPA, la majorité des utilisations de JPA sur les projets récents se fait en utilisant Spring Boot et Spring Data JPA, ce qui réduit le code redondant créé pour utiliser JPA.
- Attention, ceci cache aussi une partie de la complexité de JPA, qu'il faut néanmoins connaître.

JPA dans JEE
Ce qu'il faut retenir

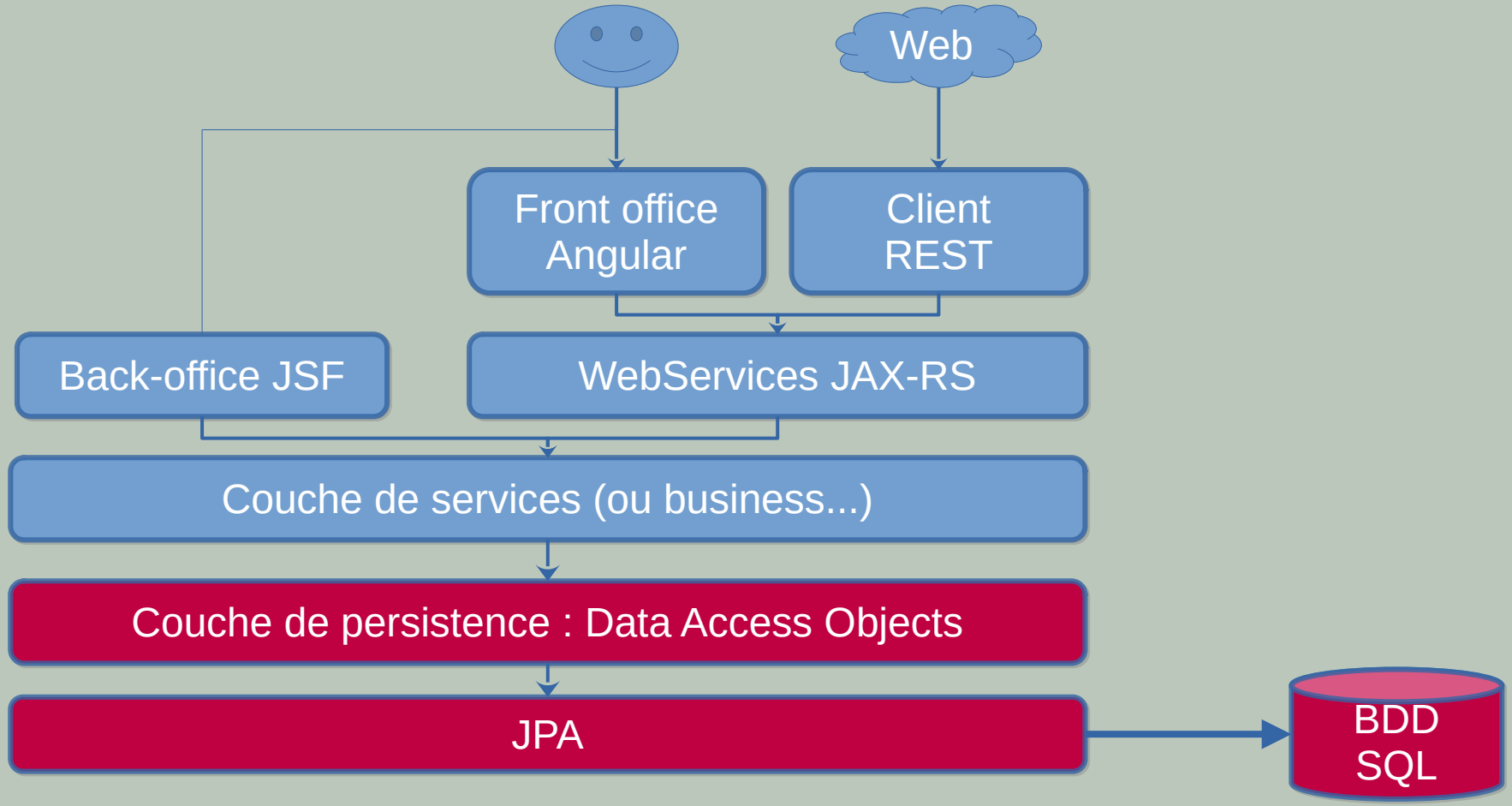
- JEE offre de nombreuses spécifications (pour persister les données, afficher des pages Web dynamiques, créer des webservices REST ...).
- Les serveurs JEE fournissent des implémentations à ces spécifications. Ils peuvent être libres, gratuits, payants ...
- JPA est une spécification JEE, de nombreuses implémentations de cette spécification existent, dont Hibernate.
- Une application devant persister des données en base va utiliser JPA et donc une de ses implémentations.

JPA dans JEE
Les conseils du formateur

- Découpez votre code applicatif en différentes couches. Ceci est une extension du principe : “une classe = une responsabilité”.
- Utilisez un serveur JEE, ou un concurrent (type Spring), pour automatiser le maximum de choses, et éviter de ré-écrire du code.
- Les implémentations de JPA sont en théorie interchangeables. En pratique, des différences existent, ce qui rend le changement coûteux. Il faut, au mieux, retester tout le code de persistance. Au pire, il faut recoder une partie de la couche de persistance.

ORM et JPA

ORM et JPA
Place de la couche de persistance dans une application



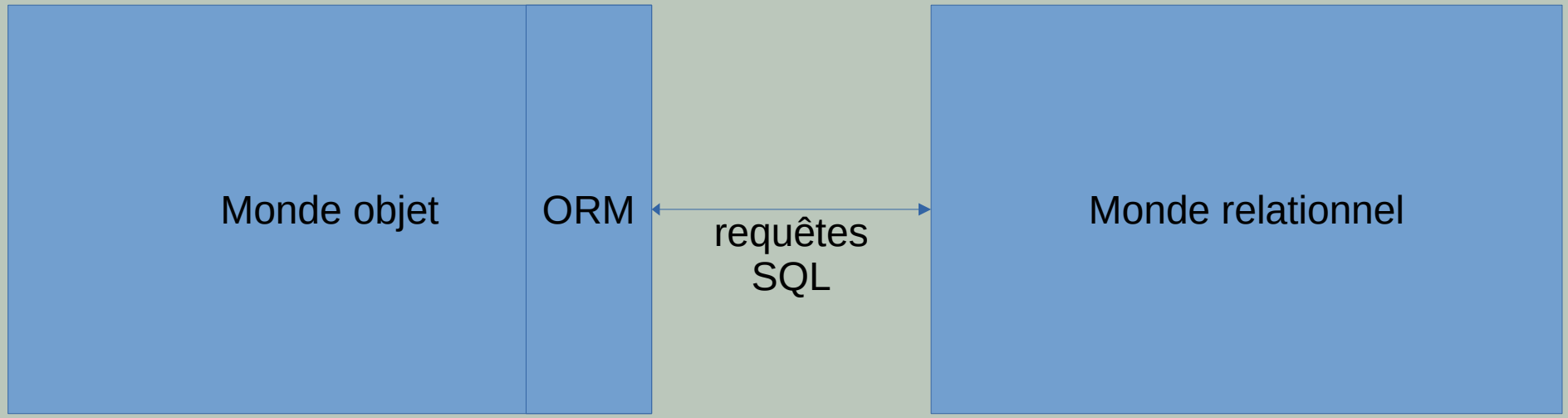
Persistance et ORM

- ORM signifie Object Relational Mapping (ou Mapper). Un ORM gère donc la correspondance entre le monde objet et le monde relationnel.
- Cet ORM va servir à diminuer la quantité de code requis pour faire correspondre des classes et des objets avec des tables et des colonnes (entre autres) dans une base de données relationnelle.
- En simplifiant, un ORM automatise de nombreuses tâches pour :
  - sauvegarder une classe dans une table,
  - en sauvegardant chaque attribut dans une colonne,
  - et chaque instance dans une ligne.
- Les ORMs ne sont pas obligatoires pour qu'un programme Java (ou autre) corresponde avec une base SQL, mais ils sont tellement utiles qu'ils sont incontournables depuis 20 ans.

Persistance sans ORM

- Il est tout à fait possible de gérer la persistance sans ORM. C’est ainsi que les développeurs procédaient avant 2005 (en Java).
- Dans ce cas, il faut :
  - Créer une requête SQL ad hoc pour chaque classe, pour les commandes INSERT, UPDATE, DELETE, afin de les persister en base.
  - Créer des requêtes SQL pour chaque classe ou grappe de classes, afin de les récupérer en base.
  - Créer et maintenir du code pour transformer la réponse SQL en instances de ces classes. Le code fait correspondre à chaque colonne un attribut.
- Ceci est coûteux, contient du code redondant. C’est acceptable pour une poignée de classes, mais pas pour un projet contenant des dizaines (voire plus) de classes à persister, ou de tables.

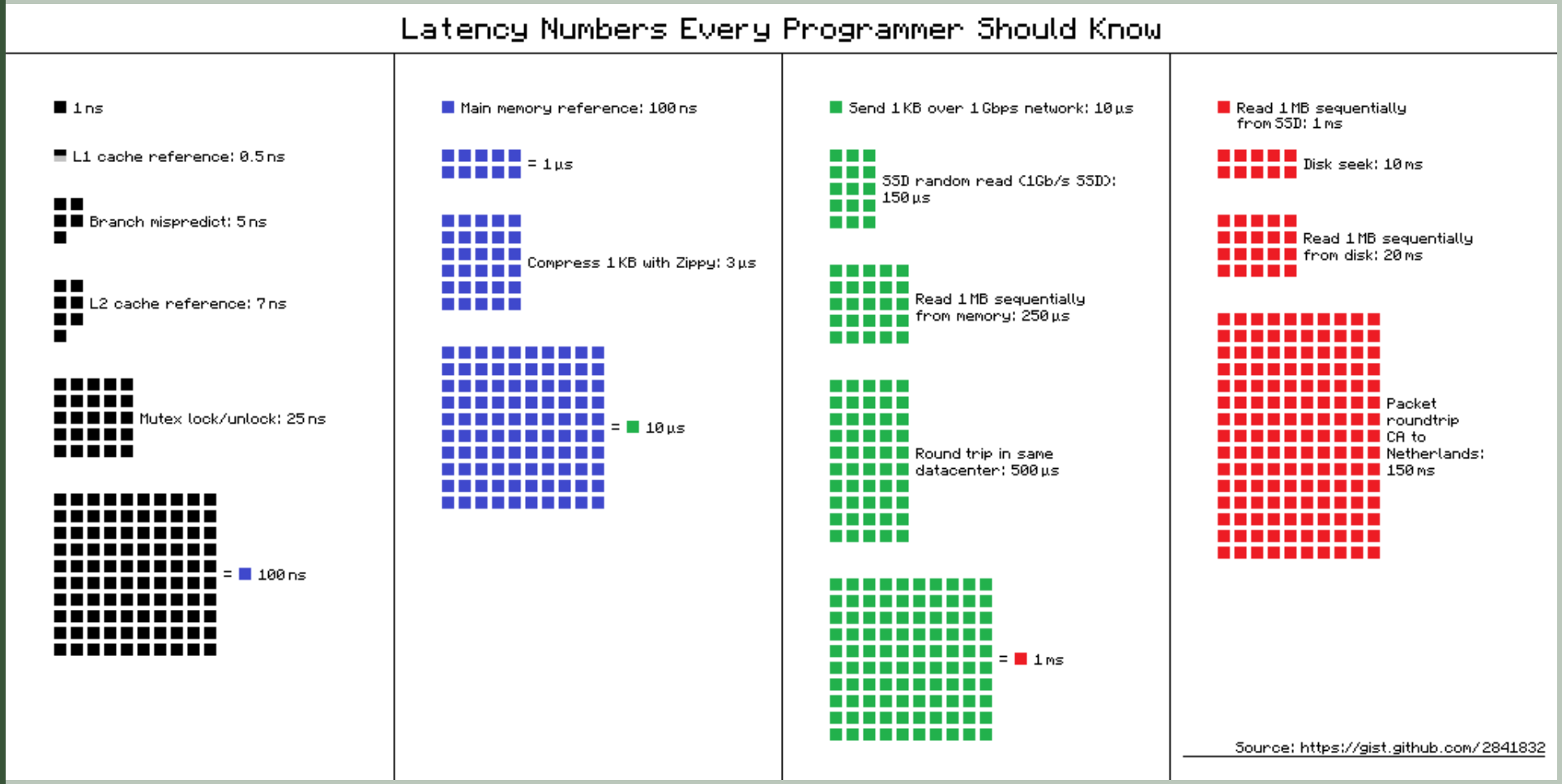
ORM et JPA
ORM côté objet : schéma



ORM et JPA
Contexte et objectif d'un ORM

- Un ORM a pour but de faire correspondre le monde objet et le monde relationnel.
- Il vise donc à combler le fossé entre ces deux mondes. Ce fossé est dû à :
  - Le cycle de vie des relations en SQL, qui se base sur des fichiers. Le temps de ce cycle est long.
  - Le cycle de vie des objets en Java, qui sont en mémoire. Le temps de ce cycle est bien plus court.
  - L'identité des relations en SQL, qui se base sur une clé primaire.
  - L'identité des objets en Java, basé sur une référence à un objet.
  - Les associations en SQL, qui se basent sur des clés étrangères (en vrai des contraintes d'intégrité réferentielle).
  - Les associations en Java, qui se basent sur des références entre objets.
  - La performance des échanges entre ces monde.

ORM et JPA
Coût des échanges entre les deux mondes



Historique :

- **Java DataBase Connectivity**
- **Hibernate (2003)**
- **JPA, JPA 2, JPA 3 …**
- **… et ses implémentations (Hibernate, Toplink, OpenJPA …)**
- **Spécification JavaEE puis JakartaEE (intégrables aux autres technologies JavaEE)**

https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html

ORM et JPA
Choix faits par JPA

- Les choix de persistence sont faits du « côté objet » de l'ORM
- Pas d'injection de code du framework dans la classe (ou très peu :))
- Utilisation de POJO (Plain Old Java Object) :
- Les objets persistés peuvent être utilisés pour toute autre chose que de la persistence

Votre classe persistée doit :

- Être un POJO
- Être non final, et aucune de ses méthodes ne l'est
- Contenir un constructeur sans argument (avec au moins la visibilité package, mais en général public)
- Être annotée de façon à ce que JPA puisse savoir comment la persister
- Avoir un attribut qui servira d'identifiant à JPA, et de clé primaire à la base de données

ORM et JPA
POJO

Un POJO (Plain Old Java Object) est un objet qui contient un constructeur sans argument. Il contient aussi des attributs, chacun accessible via un getter et un setter. ClassA ci-dessous est un POJO :

```java
public class ClassA {

	private boolean ready = true;

	private String firstName = "Jean";

	private String lastName = "Dupont";

	private Long age = 31;

	public boolean isReady() {
		return ready;
	}

	public void setReady(boolean ready) {
		this.ready = ready;
	}


...
```

- Persistence : classe qui représente l’API JPA, permet de générer des EntityManagerFactory.
- EntityManagerFactory : permet de générer des entityManager.
- EntityManager : L’objet qui permet de persister les classes. Si on utilise JTA, sera injectable facilement dans les beans CDI ou EJB.
- Entity : une classe Java persistable

- Toute couche de persistence sera composée de DataAccessObjects, classe ayant la responsabilité de persister une Entity
- Une Entity est une classe ayant “du sens” dans le monde objet, et pouvant être persistée dans le monde relationnel, grâce à ses annotations



ORM et JPA
Ce qu'il faut retenir

- Les mondes objet et relationnel ont des similitudes ...
- ... mais aussi des différences.
- L'utilisation de JDBC permet de lier le monde Java et le monde relationnel avec du code Java ...
- ... mais ceci amène à créer du code redondant et difficile à maintenir.
- Un ORM réduit ce code et adresse les différences entre les deux mondes.
- JPA est un ORM pour Java, utilisable côté objet, et se basant sur JDBC.

ORM et JPA
Les conseils du formateur

- Il est toujours possible de ne pas utiliser un ORM ...
- ... en pratique, cela n’a d’intérêt que pour les cas marginaux.
- Utilisez un ORM pour persister des objets en base de données.

Modèle de persistance

Modèle de persistence
persistence.xml

- Point de configuration principal de JPA
- Se trouve dans /META-INF du classpath

```xml
<persistence ...>

    <persistence-unit name="com.bigcorp.persistence.jpa">   <-- Nom unique de l'unité de persistence
        <description>... </description>

        <class>org.hibernate.tutorial.em.Event</class>

        <properties>
            <!-- Configuration de connexion à la base de donnnées -->
            <property name="jakarta.persistence.jdbc.url" value="jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1" />
            <property name="jakarta.persistence.jdbc.user" value="sa" />
            <property name="jakarta.persistence.jdbc.password" value="" />

            <!-- Création du schéma au démarrage de la persistance -->
            <property name="jakarta.persistence.schema-generation.database.action" value="create" />

            <!-- Affichage des requêtes SQL dans la console (bien en dev/débug, moins en production) -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.highlight_sql" value="true" />
        </properties>

    </persistence-unit>

</persistence>
```

Modèle de persistence
persistence.xml

- Contient une (ou plusieurs) configurations de persistence unit.
- Une persistence unit fournit un ORM avec une base de données.
- Définit la connexion à la base de données.
- Définit aussi la configuration globale de l’unité de persistance.

Modèle de persistence
DataSource

- Littéralement ‘source de données’ .
- Une datasource est généralement configurée dans un serveur d’applications. C’est ce serveur qui est en charge de créer la datasource et de la donner aux applications du serveur.
- Cette datasource doit être configurée.
- De plus, le serveur doit contenir le ou les JARs contenant les classes nécessaires pour créer la datasource : ex les classes des clients JDBC pour une connexion à une base de données.

Modèle de persistence
Rendre une classe persistante : une entité

- Une entity (Entity), pour JPA est une classe qui peut être persistée : elle peut être sauvegardée en base de données, et récupérée de celle-ci.

- Il existe deux manières de rendre une classe persistante :
- Par un fichier de configuration xml (historique, obsolète)
- Par annotations (actuel)

Modèle de persistence
Une entité

- Responsabilité : persiste les données (peut les valider et peut contenir des fonctionnalités)

```
@Entity ← est une entité, une classe persistante de JPA. Sera gérée par l'EntityManager
@Table(name = "EXAMPLE")
public class Example {

      @Id ← est le champ qui identifie les entités pour la base de données (la clé
                  primaire), et pour l'EntityManager.
      @GeneratedValue(strategy=GenerationType.TABLE)← le générateur utilisé par le contexte de
                                                      persistence
      private Long id;

      private String name; ← cet attribut n'est pas annoté, il sera persisté avec les
                           réglages « par défaut » de l'EntityManager

      (getters et setters)

}
```

Modèle de persistence
Entité et mapping

- Une entité est très généralement “mappée” à une table. Le nom de l’entité (le nom de la classe) est le même que celui de la table.
- L’identité d’une entité côté Java est son “adresse en mémoire”. Elle est mappée à l’identité côté base de données, la clé primaire de la ligne dans la table. Ce mapping se fait via l’annotation @Id.
- Les attributs sont par défaut tous persistés : un attribut côté Java est mappé sur une colonne côté base de données. Le nom de l’attribut est celui de la colonne. Ceci vaut aussi pour l’attribut stockant la clé primaire.
- Ces caractéristiques sont celles du comportement par défaut du mapping. Des annotations permettent de configurer finement ce mapping. Par exemple, l’annotation @Column permet de spécifier une colonne dont le nom est différent de celui de l’attribut.

Modèle de persistence
CRUD

- CRUD signifie Create Read Update Delete : créer, lire, mettre à jour, supprimer.
- Ce sont les quatre opérations nécessaires pour gérer la persistance d’un objet.
- Tout bon framework d’ORM propose des méthodes de CRUD, et le développeur peut utiliser ces méthodes du framework pour gérer ses objets persistents : les entités.
- Généralement, ces méthodes de gestion d’entité sont regroupées dans une classe : un Data Access Object ou DAO.

Modèle de persistence
L'EntityManager

Est créé par un EntityManagerFactory.

Dans un contexte JEE, cette création est faite par le serveur d'application, et l'EntityManager est injecté (généralement dans un DAO).

Est LE point d'entrée de l'API : (i.e la classe qu'un développeur utilise pour JPA).

Propose des méthodes de persistance.

Propose des méthodes pour créer des transactions.

Peut se fermer.

Contient un cache de données, de façon à optimiser les échanges avec la base.

Modèle de persistence
entityManager.find(...)

- Récupère une entité par son identifiant. La place dans le contexte de persistance.
- Le contexte de persistance peut être considéré comme un cache (il est appelé cache de premier niveau).
- L'entityManager doit être considéré en pratique comme :
  - Un objet "léger" : on peut en créer et en supprimer à volonte (!= EntityManagerFactory et Persistence).
  - Créé par le serveur selon le contexte.
  - Dans une application Web, un par requête HTTP.

Modèle de persistence
entityManager : exemple d'utilisation

```
	public static void main(String[] args) {
		// AutoCloseable, pourrait être mis dans un try with resources
		EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory( "com.bigcorp.persistence");
		Booking booking1 = new Booking();
		booking1.setName("Réservation Dupont");

		saveBooking(entityManagerFactory, booking1);

		// Récupère l'entité par son identifiant
		EntityManager entityManager = entityManagerFactory.createEntityManager();
		Booking booking = entityManager.find(Booking.class, booking1.getId());
		if (booking == null) {
			System.out.println("Aucun booking trouvé en base.");
		} else {
			System.out.println("Une réservation a été trouvée, avec le nom : " + booking.getName());
		}
		entityManagerFactory.close();
	}
```

Modèle de persistence
Exercice : première entité

- Créer (ou réutiliser) un POJO correspondant à une entité.
- L’annoter avec @Entity et annoter son identifiant avec @Id et @GeneratedValue (comme sur Booking).
- Dupliquer le code de PersistAndFindBookingMain pour l’adapter à cette entité.
- Lancer le main et analyser les dernières requêtes SQL émises.

- Bonus : sauvegarder deux POJOs.

Modèle de persistence
Un DAO (exemple)

- Responsabilité : gère la persistance (donc appelle JPA et utilise l'entityManager).
- En "vrai", un DAO sera bâti sur un framework (serveur JEE, Spring) qui injectera l'entityManager.
- Les transactions seront gérées à un niveau plus haut (niveau service). Les transactions sont plus du domaine fonctionnel.

```
public class BookingDao {
	/**
	 * Récupère un Booking par son id.
	 * Peut renvoyer null si aucun booking ne correspond
	 * @param id
	 * @return
	 */
	public Booking findById(Long id) {
		EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
		return em.find(Booking.class, id);
	}
	/**
	 * Insère ou met à jour Booking
	 * @param booking
	 */
	public void saveOrUpdate(Booking booking) {
		EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
		EntityTransaction tx = em.getTransaction();
		tx.begin();
		em.merge(booking);
		tx.commit();
	}
}
```

Modèle de persistence
Un Service utilisant le DAO

Responsabilités : gère les transactions, et contient du code fonctionnel.

L’API JPA
Une classe de test du DAO

Responsabilités : teste le DAO
Cette classe nécessite une assez lourde configuration pour initialiser le contexte de persistance et la DataSource

Modèle de persistence
Une classe de test du DAO (exemple dans un environnement JEE, il y a un équivalent dans un environnement Spring)

```
@RunWithApplicationComposer
public class PojoJpaDaoTest {

      @EJB
      private PojoJpaDao pojoJpaDao;

      @org.apache.openejb.testing.Module
      public EjbJar beans() {
            //Configure les EJB
      }

    @org.apache.openejb.testing.Module
    public PersistenceUnit persistence() {
        // configuration du contexte de persistence de test
    }

    @Configuration
    public Properties config() throws Exception {
        //Configuration de la Datasource
    }

      @Test
      public void testSave() throws Exception {
         //test
      }
```

Modèle de persistence
Une classe de test du DAO (exemple sans JEE)

```java
public class BookingDaoTest {

	private BookingDao bookingDao = new BookingDao();

	@Test
	public void testSaveAndFind() {
		Booking booking = new Booking();
		String bookingName = "Réservation Dugenou";
		booking.setName(bookingName);
		booking = bookingDao.save(booking);


		Booking savedBooking = bookingDao.findById(booking.getId());
		Assert.assertNotNull(savedBooking);
		Assert.assertEquals(bookingName, savedBooking.getName());
	}

}
```

Modèle de persistence
Exercice : créer le DAO associé à l’entité précédente

- Créer une classe DAO offrant les services de récupération par identifiant.
- En s’inspirant de la classe de test BookingDaoTest, tester avec JUnit.
- La base de données étant vide, s’assurer que l’appel à la méthode de récupération se fasse sans erreur (ne pas vérifier si la valeur de retour est non nulle).

Modèle de persistence
A retenir

- Tout le mapping objet relationnel d’une classe est dans un POJO, et généralement sous la forme d’annotations JPA.
- Le POJO doit obéir à un contrat avec JPA, mais ce contrat n’est pas trop contraignant. Ceci permet d’utiliser l’entité (l’objet persisté) dans toute partie d’une application Java (ou J2EE).
- Il vaut mieux utiliser les wrappers de types primitifs, plutôt que les types primitifs eux-mêmes, et gérer la non nullité avec des annotations.
- Organiser le code dans une entité, un DAO, une classe de service et au moins une classe de test est un standard de facto.

Modèle de persistence
Conseils du formateur

- En amont, au niveau conception, faîtes en sorte que le « monde objet » et le « monde relationnel » soient proches (en général, l’architecture qui les définit obéit aux mêmes principes).
- Ne pas oublier que l’EntityManager est un cache.
- La machinerie nécessaire aux tests est plus lourde à créer et à lancer, mais elle est quand même absolument nécessaire.

Manipulation des entités

Manipulation des entités
Tester et consulter sa base de données

- Grâce au DROP-CREATE au chargement de JPA, c'est possible. (cf. paramètre schema-generation.database.action de persistence.xml)

- Normalement, les tests jUnit se font avec une base en mémoire, directement instanciée à la connexion JDBC. Ceci offre de nombreux avantages (pas de problème d'hystérésis entre chaque build, pas besoin de s'assurer qu'une base réponde …) et est la norme pour les tests jUnit.

- Pour notre formation, nous allons procéder différemment, et utiliser H2.

- Attention : H2, tel que configuré, ne supporte qu'une connexion à la fois à chaque base de données : il faut donc se déconnecter de la base pour laisser les tests l'utiliser.

Manipulation des entités
Exercice : se connecter à la BDD de test

- Télécharger H2 dans sa dernière version
- Lancer H2.w.bat
- Se connecter à la base de données dont les coordonnées sont dans persistence.xml de src/test/resources/META-INF
- Vérifier l’état du schéma.

Modèle de persistence
entityManager.persist(...)

- Persiste (sauvegarde) une entité.
- Ne fonctionne que si l’entité n’a pas d’identifiant (l’entité n’a jamais été persistée).
- Modifie l’entité pour mettre à jour son identifiant.
- Cette opération peut nécessiter l’envoi d’un SELECT en base.
- Cette opération devrait envoyer un INSERT en base.
- Cette opération va nécessiter l’emploi de transactions pour fonctionner (détaillées plus tard).

Modèle de persistence
entityManager.merge(...)

- Persiste ou fusionne (sauvegarde) une entité.
- Fonctionne si l’entité a des identifiants ou non.
- Peut modifier l’entité pour mettre à jour son identifiant.
- Cette opération peut nécessiter l’envoi d’un SELECT en base.
- Cette opération devrait envoyer un INSERT ou un UPDATE en base.
- Ces opérations vont nécessiter l’emploi de transactions pour fonctionner (détaillées plus tard).
- Ne modifie pas l’entité, mais renvoie l’entité mise à jour.

Modèle de persistence
entityManager : exemple de persistence d'objets

```
//Création d'un entityManager
EntityManager entityManager = entityManagerFactory.createEntityManager();

//Gestion de la transaction
EntityTransaction tx = entityManager.getTransaction();
tx.begin();

//Demande de persistence de booking
entityManager.persist(booking);

//Validation de la transaction
tx.commit();

entityManager.close();
```

Manipulation des entités
JPA et les transactions

- Pour qu'une transaction soit validée et effectivement persistée, une commande SQL commit doit être exécutée.
- Pour qu'une transaction soit annulée, une commande SQL rollback doit être exécutée.
- JPA permet de gérer ces transactions :
  - Via JTA, ce qui permet d'automatiser la gestion des commits et rollbacks, et de gérer des transactions distribuées sur plusieurs bases de données (si le SGBDR le supporte).
  - Directement. Dans ce cas, l'entityManager propose des méthodes pour manipuler ces transactions.

Manipulation des entités JPA et les transactions

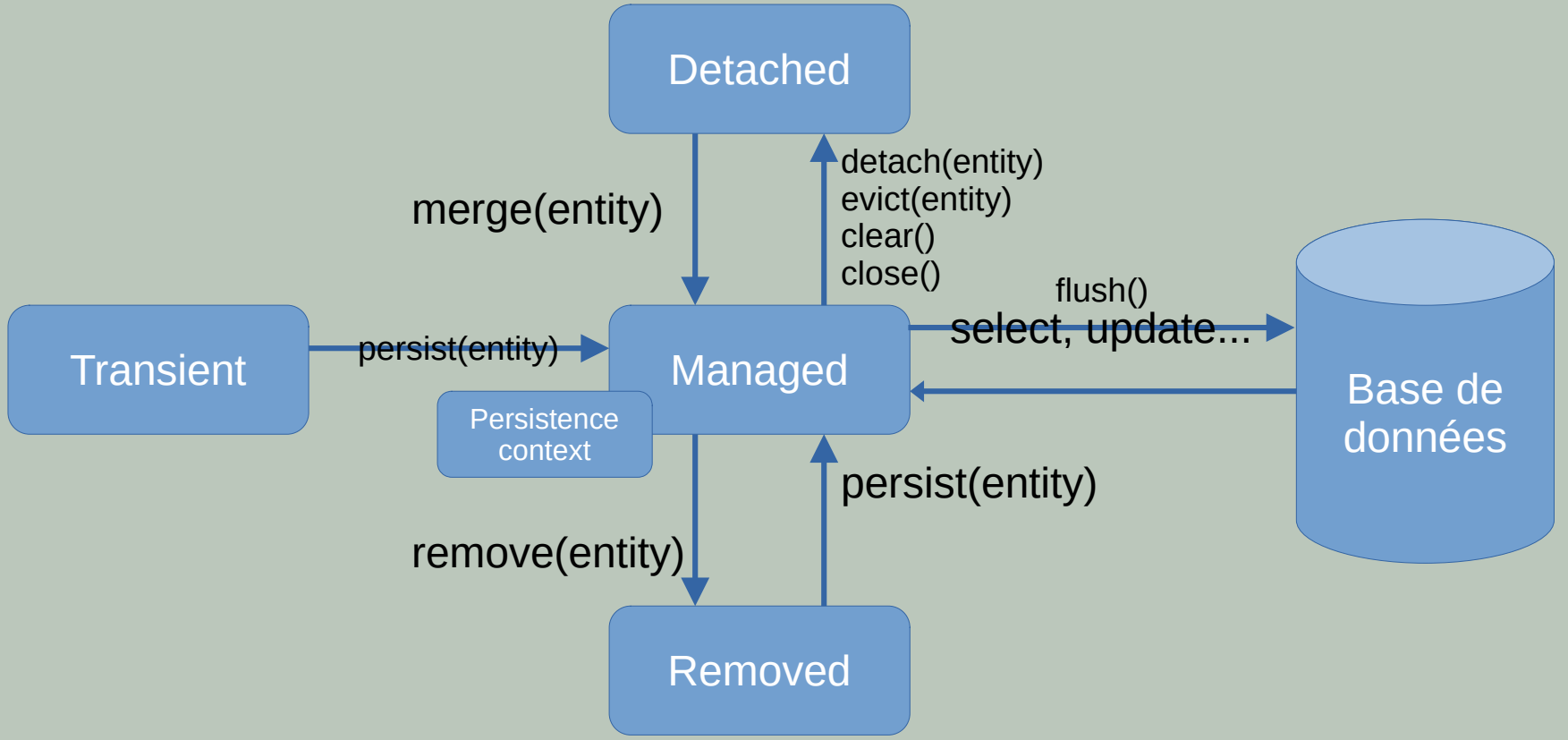
- EntityManager permet de créer une instance de EntityTransaction avec getTransaction()
- Il est possible d'appeler begin(), commit() ou rollback() sur cette instance pour commencer, valider ou annuler la transaction.
- Ainsi, la cohérence de la base de données peut être conservée via un code correct.

```
EntityTransaction transaction = entityManager.getTransaction();
try {
	transaction.begin();
	//méthodes de mise à jour de la base
	transaction.commit();
	}
catch (Exception e) {
	if (transaction.isActive()) {
		transaction.rollback();
	}
	throw e;
}
```

Manipulation des entités
Exercice : Ajouter la méthode save au DAO

- Cette méthode utilise les transactions pour persister correctement une entité, et merge pour traiter les cas d’INSERT et d’UPDATE.
- Tester avec JUnit un l’utilisation de save(...) et findById(...)

Manipulation des entités
Cycle de vie des entités



Manipulation des entités
Cycle de vie des entités : états

- Etats des entités :
- Transient : nouvelle entité, non reliée à un EntityManager, et sans identifiant.
- Managed : entité liée à un EntityManager. JPA détectera tout changement sur l’entité et déclenchera des envois de requêtes SQL par via flush() pour synchroniser l’état des entités avec la base de données.
- Detached : entitée non liée à un EntityManager, mais possédant un identifiant.
- Removed : entité marquée comme étant à supprimer : une requête SQL DELETE sera émise via flush() (peut posséder un id).

Manipulation des entités
Exercice : Créer une nouvelle entité, le DAO et la classe de tests associés.

- Le DAO propose des méthodes pour récupérer une entité par son identifiant et sauvegarder une entité.
- La classe de tests teste ces deux méthodes.

Manipulation des entités
Les fonctionnalités communes aux DAOs : le CRUD

Tous les DAOs offrent de base les mêmes fonctionnalités :

- Create (INSERT)
- Read (SELECT)
- Update (UPDATE)
- Delete (DELETE)

- Les nôtres ne vont pas déroger à cette règle, nous pourrions même mutualiser ce qui peut l’être dans une classe abstraite.
- Ce pattern est utilisable dans vos projets. Il est aussi possible d’utiliser des bibliothèques (comme Spring-Data-JPA) qui vous fournissent ce genre de DAO générique.

Modèle de persistence
entityManager.delete

- Supprime une entité.
- Ne fonctionne que si l’entité est managed, c’est à dire présente dans l’entityManager.
- Cette opération enverra un DELETE en base.
- Cette opération nécessite une transaction pour fonctionner.

Manipulation des entités

Exercice : Ajouter une méthode delete dans le nouveau DAO .

- Ajouter la méthode et la tester.

Manipulation des entités
Les attributs simples

- Ils sont automatiquement persistés.
- Les types primitifs ou leurs wrappers sont gérés par JPA, mais il vaut mieux toujours utiliser les wrappers.
- Les enums sont persistés aussi avec une annotation.
- Si la classe d’un attribut est Embeddable, ou est Serializable, JPA peut les persister.
- Sinon, une erreur sera lancée au démarrage.
- La place de la première annotation (@Id) est déterminante :
- Si elle est sur un attribut, JPA mettra à jour les attributs directement.
- Si elle est sur un getter, JPA passera par les accesseurs.

Manipulation des entités
Autres annotations

- @Table permet de spécifier un nom de table différent de celui de la classe
- @Column permet de spécifier un nom de colonne différent de celui de l'attribut
- @GeneratedValue permet de spécifier une stratégy de génération de valeur. Celle-ci peut être :
  - IDENTITY : une colonne auto incrémentée
  - SEQUENCE : une séquence (il faudra préciser la séquence à utiliser via @SequenceGenerator)
  - TABLE : une table dans laquelle JPA gère les identifiants
  - UUID : un UUID
- .... et d'autres : cf. spec-jee

Manipulation des entités
Les Enums

- On peut persister des enums, en annotant l'attribut comme suit :
  - @Enumerated(EnumType.STRING) (← le choix du formateur)
  - @Enumerated(EnumType.ORDINAL)
- Le type STRING sauvegarde en base la chaîne de caractères de la valeur énumérée.
- Le type ORDINAL sauvegarde en base de données la position de la valeur énumérée dans la liste des valeurs de l'enum. La position la plus petite vaut 0.

Manipulation des entités
Exercice : Ajouter des attributs à l'entité

- Ajouter au moins
  - un attribut Boolean (ex : active)
  - un attribut String (ex : phone, name)
  - un attribute Integer (ex : score)
- Vérifier qu'ils sont persistés dans la classe de test, et dans la base de données.

- Bonus :
  - ajouter un attribut de type enum
  - changer le nom de la colonne d'un attribut avec @Column

Manipulation des entités
Ce qu'il faut retenir

- Les DAOs proposent très généralement des services de CRUD.
- Via les méthodes persist(...), merge(...), remove(...), find(...), l'entityManager permet de coder ces services de CRUD.
- L'entityManager gère ainsi ses entités via un cycle de vie, avec des états (TRANSIENT, MANAGED, DETACHED, REMOVED).
- Ceci lui permet d'optimiser les allers-retours avec la base de données et de gérer correctement les entités.

Manipulation des entités
Les conseils du formateur

- Les DAOs contiennent en général du code dupliqué, et donc mutualisable ...
- ... ce qui fait que de nombreuses bibliothèques contiennent ce code et évitent de le dupliquer : ex : Spring-Data-JPA.
- Le cycle de vie des entités JPA n’est pas à connaître par coeur, mais il faut savoir qu’il existe. Ce cycle peut dépanner en cas de problème avec JPA (qui ne manqueront pas d’arriver).

JPQL

JPQL
Requêtes SQL natives

- Il est tout à fait possible d’exécuter des requêtes SQL natives via JPA.
- EntityManager propose une méthode createNativeQuery(String sqlQuery) pour ce faire.
- Il est possible d’éxécuter des requêtes d’UPDATE , de DELETE ou d’INSERT ainsi.
- Il est aussi possible d’exécuter des requêtes SELECT. Dans cas, une liste de Object[] sera renvoyée, correspondant aux lignes renvoyées par la requête SELECT.
- Ceci est possible, mais n’est pas très orienté objets ….

JPQL
Requêtes SQL natives (exemple)

```java
/**
 * Renvoie le nom d'un booking par son id.
 * Utilise une requête SQL native
 * @param id
 * @return
 */
public String getNameWithNativeSQL(Long id) {
	EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
	Query sqlQuery = em.createNativeQuery("SELECT NAME FROM BOOKING WHERE ID = :id");
	sqlQuery.setParameter("id", id);
	List<?> result = sqlQuery.getResultList();
	if(result.isEmpty()) {
		return null;
	}
	//else
	return Objects.toString(result.get(0));
}
```

JPQL
JPQL

- ... le JPQL (Java Persistence Query Language) permet d’écrire une sorte de SQL orienté objets.
- La méthode createQuery de entityManager crée une requête JPQL, qui permet au développeur de requêter la base de données, avec des notions de classes, attributs, relations, plutôt que tables, colonnes, clés étrangères.
- Ce sont des instances de classes que renvoie la Query.
- JPA s’occupe du mapping objet -> relationnel lors de la création de la requête, et du mapping relationnel -> objet lors du traitement de la réponse.

https://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html

JPQL
JPQL

- JPQL peut gérer le type de retour d'une requête en précisant la classe que l'on veut obtenir.
- Il suffit de mettre cette classe en paramètre de createQuery : cela crée une requête typée, ou TypedQuery.
- Il est possible (et recommandé) d'utiliser des alias dans les requêtes JPQL, tout comme en SQL.
-

```
	/**
	 * Renvoie tous les Bookings qui ont le nom "name"
	 * @param name
	 * @return
	 */
	public List<Booking> findByName(String name) {
		EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
		TypedQuery<Booking> query = em.createQuery("select b from Booking b where b.name = :name",
Booking.class);
		query.setParameter("name", name);
		return query.getResultList();
	}
```

https://docs.oracle.com/cd/E11035_01/kodo41/full/html/ejb3_langref.html

JPQL
JPQL : requêtes paramétrées.

JPQL permet de paramétrer des requêtes.

L'exemple précédent montre une requête pour laquelle name est un paramètre.

Ceci est infiniment préférable à la concaténation de chaînes de caractères :

- le code est plus maintenable,
- il est plus facile à lire,
- il évite les injections SQL.

- Les paramètres sont :
  - nommés, dans ce cas, le nom est une chaîne de caractères précédée par ':'
  - positionnés, dans ce cas le paramètre est un entier précédé par '?' .
- setParameter permet de définir la valeur d'un paramètre (avec son nom, par exemple).

JPQL : autres possibilités

JPQL permet :

- Les jointures (vues plus tard, avec les associations)
- l’utilisation de distinct, max, min, group by, order by
- des comparaisons avec = , <= , > ...
- l’utilisation de upper, lower et des caractères joker ? et * pour les chaînes de caractères,
- l’utilisation de delete et update,
- ...

JPQL
Exercice : Ajouter une méthode de recherche par le nom

- Dans le DAO, ajouter une méthode de recherche par nom.
- Tester.

- Bonus : ajouter une méthode de recherche par nom insensible à la casse.
- Bonus++ : ajouter une méthode de recherche par nom avec uniquement une partie du nom comme critère de recherche.

JPQL
Criteria

- JPQL est très pratique, mais oblige à manipuler des chaînes de caractère. Si une requête JPQL contient 1 ou 5 critères de requêtes, en fonction de ce que l’utilisateur choisit, cela force le développeur à manipuler une chaîne de caractères.
- Si ceci est gênant, l’API Criteria peut aider à construire des requêtes dynamiquement, en utilisant des méthodes Java.
- Il faut utiliser getCriteriaBuilder() de entityManager.

JPQL
Criteria : exemple

```
public List<Booking> findWithCriteria(String name) {
		if (name == null) {
			name = "";
		}

		EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
		// Crée un criteriaBuilder pour récupérer des Bookings
		CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
		CriteriaQuery<Booking> criteriaQuery = criteriaBuilder.createQuery(Booking.class);
		Root<Booking> root = criteriaQuery.from(Booking.class);

		// Crée deux prédicats à partir du CriteriaBuilder
		Predicate greaterThanId = criteriaBuilder.gt(root.get("id"), 1000);
		Predicate nameLike = criteriaBuilder.like(root.get("name"), name + "%");

		// Ajoute ces critères, séparés par un or, à la requête.
		criteriaQuery.select(root).where(criteriaBuilder.or(greaterThanId, nameLike));

		// Crée la query associée et renvoie le résultat
		TypedQuery<Booking> query = em.createQuery(criteriaQuery);
		List<Booking> results = query.getResultList();
		return results;
	}
```

Exercice : Utiliser Criteria pour générer une requête avec un AND

- Par exemple, pour chercher toutes les entités dont le nom est égal à une chaîne de caractères, et dont une valeur numérique est inférieure ou égal à une valeur passée en paramètre.

JPQL
A retenir

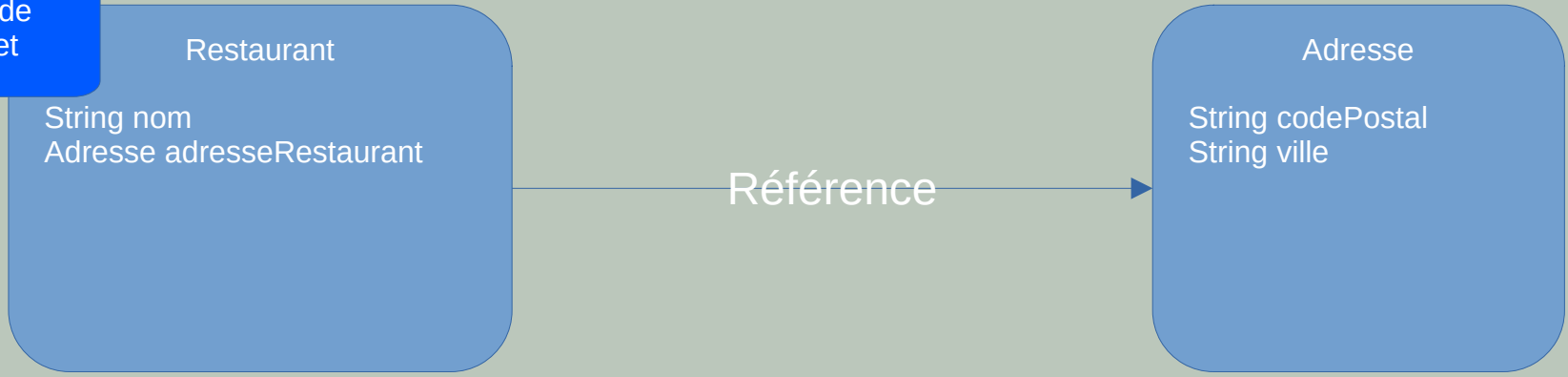
- JPQL est proche du SQL, mais orienté objet.
- Il peut servir à requêter, mais aussi à mettre à jour, insérer ou supprimer des données, voire verrouiller des lignes en base de données.
- On peut aussi utiliser l’API Criteria pour construire des requêtes via des appels à une API. Ceci peut être plus pratique dans le cas où les restrictions (ce qui est après le WHERE) sont dynamiques. Un exemple est un formulaire avec de nombreux champs de recherche optionnels.
- Les requêtes JPQL peuvent être déportées dans des annotations, afin de les regrouper dans une classe.

Relations en JPA

Relations en JPA
Embedded et Embeddable

Monde relationnel

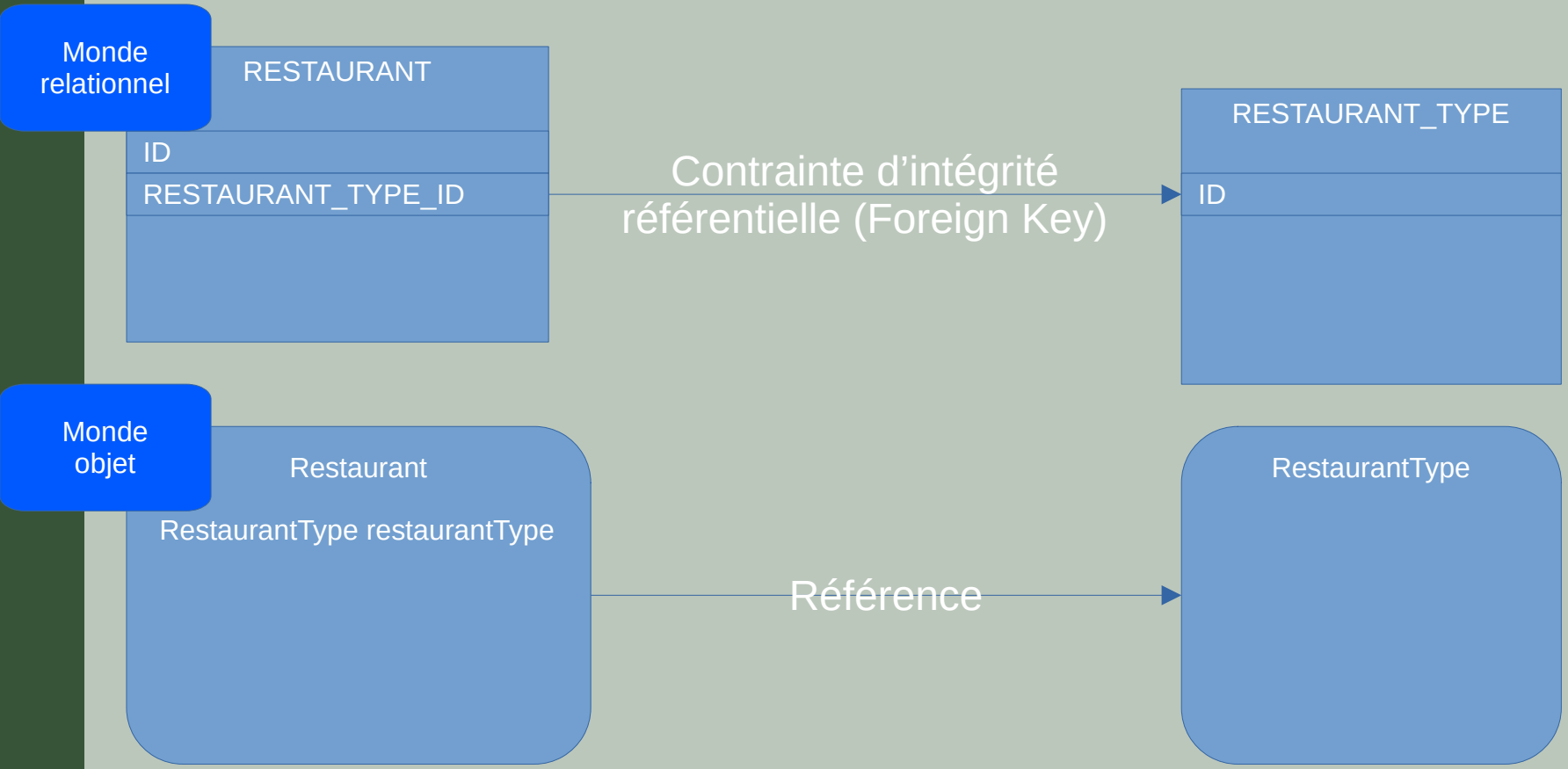
Monde objet



Relations en JPA
Embedded et Embeddable

- Il est possible d’associer plusieurs classes avec une table.
- La manière la plus simple est d’annoter une classe avec @Embeddable. Les attributs de cette classe seront persistés dans des colonnes. Mais ces colonnes ne seront pas celles de la classe annotée avec @Embeddable.
- Si une autre classe définit un attribut, dont le type est celui de la classe @Embeddable. JPA essaiera de persister les attributs de la classe @Embeddable dans la table de la permière classe. L’attribut doit être annoté avec @Embedded.

Relations en JPA
Many-To-One



Relations en JPA
Many-To-One

- La configuration de notre ORM se faisant côté objet, une “bonne” association ManyToOne se fait comme suit :

```java
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "RESTAURANT_TYPE_ID")
private RestaurantType restaurantType;
```

- A part @ManyToOne, tout est optionnel, mais :
- Par défaut Fetch = EAGER, et ce n’est pas recommandé (ce sera vu plus tard)
- Par défaut, les noms des colonnes de jointure ne sont pas excellentes (de même pour les noms des clés étrangères, ou foreign keys).

  Ceci crée une relation Lazy ET unidirectionnelle.

Relations en JPA
Many-To-One : persistence de la relation

```
//Persistence de bookingTable
BookingTable bookingTable = new BookingTable();
bookingTable.setName("Notre meilleure table");
bookingTable = new BookingTableDao().save(bookingTable);

//Rattachement de l'objet bookingTable à booking via setTable(...)
Booking booking = new Booking();
String bookingName = "Réservation sur une table";
booking.setName(bookingName);
booking.setTable(bookingTable);
```

Relations en JPA
Lazy, Eager

- Des solutions compliquées pour des problèmes compliqués. “Fetch” signifie récupérer l’objet derrière une association. Dans l’exemple précédent, cela signifie : récupérer des informations sur le RestaurantType quand on récupère des informations sur le restaurant.

- Les “fetch” sont de deux types par défaut en JPA :
  - Lazy : l’entité derrière l’association n’est pas chargée quand l’entité que l’on requête l’est.
  - Eager : l’entité derrière l’association est chargée quand l’entité que l’on requête l’est.

- Avec un fetch “eager”, toute récupération d’une entité se fera par une requête SQL avec un FROM pour la table de l’entité + autant de jointures (LEFT OUTER JOIN) qu’il y aura de relations eager TRANSITIVES (ce qui peut faire beaucoup).
  A la fin de la requête et de la transformation en instances, le graphe d’instances sera correct.

Relations en JPA
Lazy, Eager

- Avec un fetch “lazy”, toute récupération d’une entité se fera par une requête SQL avec un FROM. L’instance sera remplie avec les colonnes “basiques” et des Proxys pour chacune des entités “lazy”. Ces proxys seront instanciés à la demande :
  - Par exemple, lors d’un appel à restaurant.getRestaurantType().getName() SEULEMENT SI l’entityManager est encore actif lors de l’appel au get. Dans ce cas, un SELECT est envoyé en base.
  - Si l’entityManager est fermé, le proxy retourne null, ou une LazyInitializationException (dans tous les cas, le résultat est mauvais).

Relations en JPA
Lazy, Eager

- Si on laisse l'entityManager envoyer des SELECT avec un fetch LAZY non contrôlé, ceci peut gréver les performances de l'application. (problème du N+1 SELECT).

- Solution recommandée pour garder la maîtrise du modèle :
  - Par défaut, tous les fetch sont à lazy.
  - Si une entité doit TOUJOURS être chargée lorsqu'une autre entité l'est, on peut mettre un fetch à EAGER (attention aux EAGER en cascade).
  - Si l'on doit récupérer une grappe d'objets (cas très fréquent), autant forcer le fetch à la demande, par exemple dans une requête JPQL.

Relations en JPA
Fetch d'une relation Lazy en JPQL

- Il est tout à fait possible de configurer le fetch d'une relation en lazy, et de construire un graphe d'objets à façon, en JPQL.
- Pour cela, il faut construire la jointure, comme en SQL, mais du point de vue objet,
- Et utiliser le mot clé fetch.

```
	public Booking findWithBookingTable(Long id) {
		EntityManager em = PersistenceFactory.INSTANCE.getEntityManager();
		TypedQuery<Booking> query = em.createQuery("select b from Booking b
			left join fetch b.table
			where b.id = :id",
			Booking.class);
		query.setParameter("id", id);
		List<Booking> results = query.getResultList();
		if(results.isEmpty()) {
			return null;
		}
		return results.get(0);
	}
```

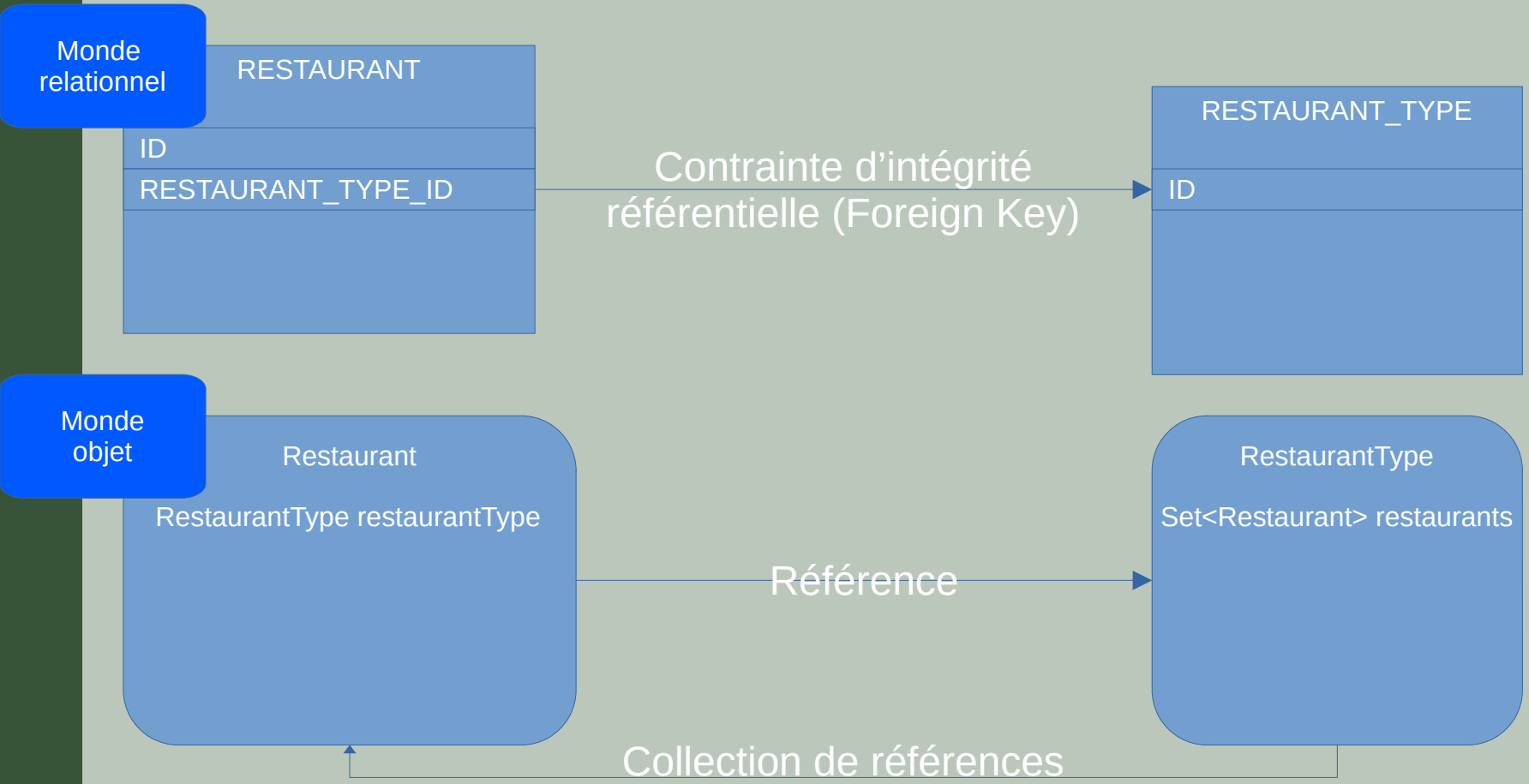
Relations en JPA
Exercice : Ajouter une relation ManyToOne

- Créer une deuxième entité, lui donner quelques attributs.
- Rattacher la première entité à la seconde via un ManyToOne, configuré avec Fetch=EAGER (laisser par défaut le fetch).
- Tester la sauvegarde de l’association et la récupération d’entités. Jeter notamment un oeil à la dernière requête : elle doit contenir un left outer join.

Relations en JPA
Exercice : Passer la relation en LAZY

- Passer la relation précédente en LAZY.
- Le test précédent va passer en erreur, puisque l'on essaie de récupérer un Client alors que l'entityManager ne peut plus lancer un nouveau SELECT pour récupérer les données (il est fermé).
- Ajouter une méthode dans le DAO pour récupérer le graphe d'objets correct, et faire refonctionner le test.

Relations en JPA
Relation bidirectionnelle



Relations en JPA
Relation bidirectionnelle

```java
@OneToMany(mappedBy = "restaurantType", fetch = FetchType.LAZY)
private Set<Restaurant> restaurants = new HashSet<>();
```

- Seul le nom de l'attribut dans la classe RESTAURANT suffit à JPA pour connaître la relation ManyToOne. Ce nom est dans l'attribut mappedBy. Ainsi JPA connaît la clé primaire et la table sur laquelle faire la jointure.
- Ici, le fetch est lazy par défaut, tant mieux.

- Comme précédemment, appeler restaurantType.getRestaurants.get(0).getName() , hors du contexte de persistance va renvoyer une LazyInitializationException ou une NullPointerException.

Relations en JPA
Direction des relations bidirectionnelles

Une relation bidirectionnelle a bien une direction, comme dans la base de données. Cette direction n'a pas d'influence lors de la récupération d'entités, mais elle en a lors de la sauvegarde.

Le code ci-dessous persiste la relation :

```
restaurant.setRestaurantType(restaurantType);
restaurant = this.restaurantService.save(restaurant);
```

Le code ci-dessous ne le fait pas :

```
restaurantType.getRestaurants().add(restaurant);
restaurantType = this.restaurantTypeService.save(restaurantType);
```

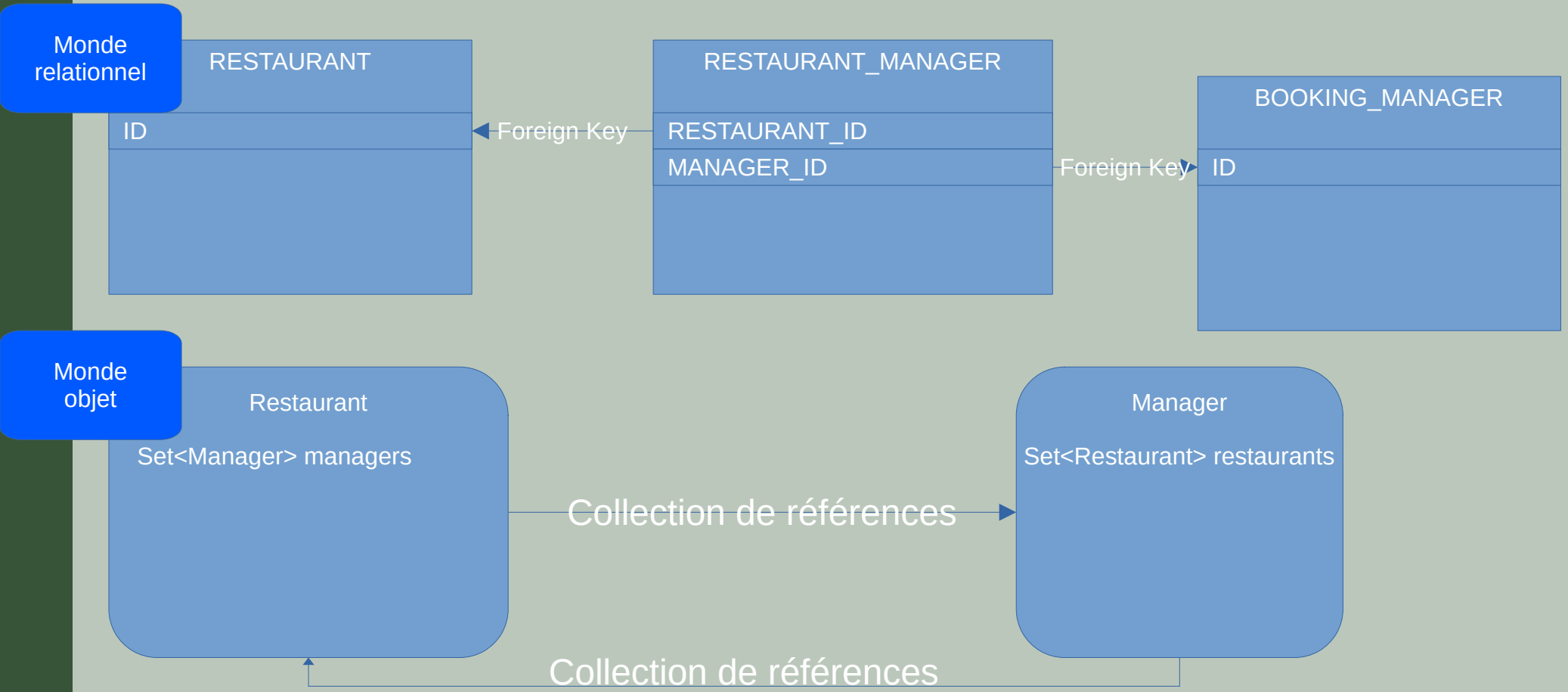
Relations en JPA
Direction des relations bidirectionnelles

- La relation persistée sera la relation ManyToOne.
- Est-ce grave ? Pas vraiment, de toutes façons, pour que le graphe d’objets en mémoire soit cohérent, il est conseillé d’associer les objets correctement. JPA le fera lors de la récupération des entités. Par contre, il faut connaître la règle de persistence pour éviter les mauvaises surprises.

Relations en JPA
Exercice : Rendre la relation précédente bidirectionnelle

- Tester la récupération d’une entité côté “One” du ManyToOne, avec les entités côté “Many”.
- Il faudra aussi utiliser JPQL.
- Regarder la requête SQL émise.

Relations en JPA
Relation ManyToMany



Relations en JPA
Relation ManyToMany

Cette fois-ci, il faut indiquer à JPA la structure de la table d'association. De plus, la relation étant symétrique, il faut expliciter quelle direction de l'association va porter la persistance.

Pour ce faire, on aura côté Restaurant :

```
@ManyToMany
@JoinTable(
name="RESTAURANT_MANAGER",
joinColumns = @JoinColumn(name="RESTAURANT_ID"),
inverseJoinColumns = @JoinColumn(name="MANAGER_ID"))
private Set<Manager> managers = new HashSet<>();
```

Et côté Manager :

```
@ManyToMany(mappedBy = "managers")
private Set<Restaurant> managers = new HashSet<>();
```

Relations en JPA
Relation ManyToMany

- On peut créer des méthodes permettant de rendre la relation bidirectionnelle via l'appel à une méthode :

```
public void associateWith(Manager manager) {
	if(manager == null) {
	return;
	}
	this.managers.add(manager);
	manager.getRestaurants().add(this);
}
```

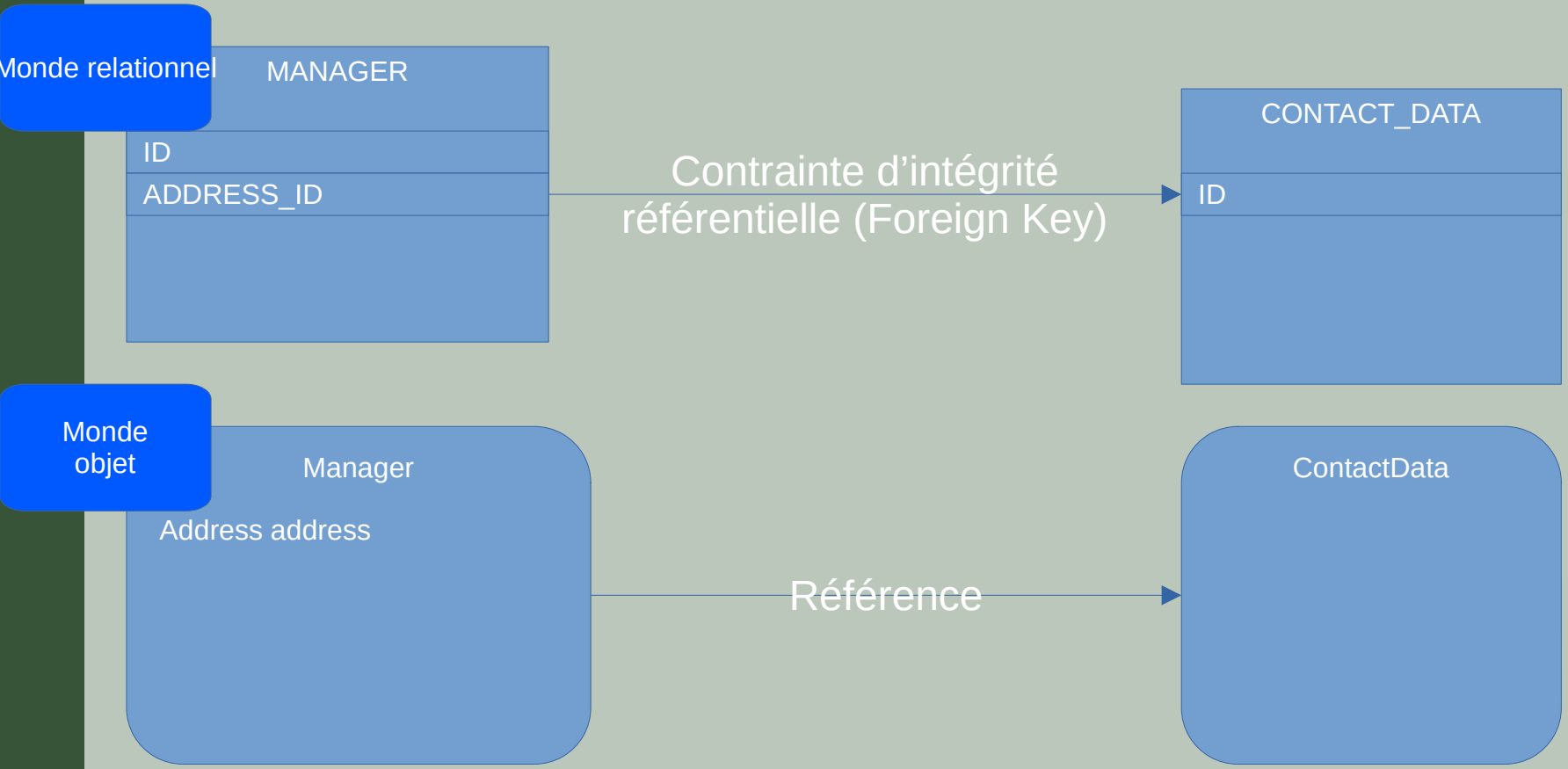
Relations en JPA
Exercice : Ajouter une relation ManyToMany, bidirectionnelle

- Créer une nouvelle entité avec un id et un nom.
- Associer les deux entités via une relation ManyToMany.

- Créer le DAO de la nouvelle entité et tester la sauvegarde de la relation. Voir en base l'état de la table d'association.

- Bonus : avec une requête JPQL, récupérer un graphe d'objets liés par cette relation.

Relations en JPA
Relation OneToOne bidirectionnelle



Relations en JPA
Relation OneToOne

La relation OneToOne est une relation ManyToOne pour la base de données. Néanmoins, elle peut être modélisée via JPA. Il existe trois manières de la modéliser dans le monde relationnel (partage de clé primaire, table intermédiaire et clé étrangère), nous allons nous baser sur la manière la plus simple : grâce à une clé étrangère.

Pour ce faire, on aura côté Manager :

```
@OneToOne(fetch = FetchType.EAGER)
@JoinColumn(
      name="CONTACT_DATA_ID", unique=true, nullable=false, updatable=false)
private  ContactData contactData;
```

Et côté Address … rien , cette relation est unidirectionnelle pour une fois. Notez l'utilisation de Eager : ici, récupérer un manager sans son adresse n'a pas de sens.

Relations en JPA
Propagation par cascade

La relation OneToOne précédente lie très fortement nos instances de Manager et d'adresse. Fonctionnellement, on peut penser qu'à chaque fois que l'on va modifier notre Manager, on va modifier notre adresse. On peut appliquer ceci à l'ORM avec une cascade sur la relation.

Toute opération de l'entityManager (persist, merge, refresh, remove …) peut déclencher une cascade. On peut définir une, ou des cascades d'opérations sur une relation :

```
@OneToOne(fetch = FetchType.EAGER, cascade = {CascadeType.DETACH,CascadeType.MERGE})
@OneToOne(fetch = FetchType.EAGER, cascade = {CascadeType.ALL})
```

On peut aussi automatiquement supprimer les enfants orphelins (sur une OneToOne ou OneToMany). La suppression de l'entité supprimera l'entité derrière la relation:

```
@OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, orphanRemoval = true)
```

Relations en JPA
Propagation par cascade

- Une cascade va propager l’opération de l’autre côté de l’association.
- Par exemple, si on demande de faire un merge sur une entité, qui a une association avec une autre entité, qui contient une cascade merge, JPA appliquera un merge sur la première entité, et sur la deuxième.

Relations en JPA
Exercice : Sur une des relations, Cascader la mise à jour et la suppression.

- Vérifier que cela fonctionne avec la mise à jour et la suppression.

- Bonus : utiliser orphanRemoval sur une relation ManyToOne pour supprimer automatiquement des orphelins (utiliser CascadeType.ALL pour activer orphanRemoval)

Relations en JPA
A retenir

- La grande partie de la différence entre le monde objet et le monde relationnel vient des relations entre entités.
- Il existe de nombreux moyens de modéliser ces relations, il faut connaître au moins ManyToOne, OneToMany et ManyToMany.
- Il faut aussi configurer prudemment les politiques de fetch par défaut, ou forcées par les requêtes (par exemple dans JPQL).
- Finalement, les cascades offrent des options intéressantes de persistance.

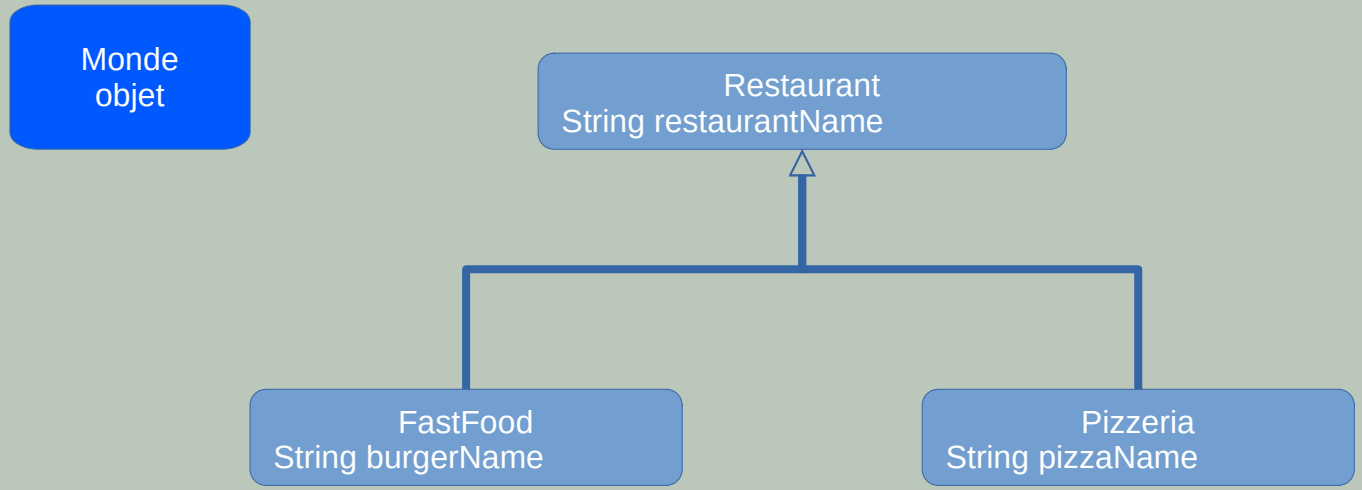
Relations en JPA
Conseils du formateur (diapo subjective)

- Par défaut, pas de Cascade, et des Fetch = Lazy.
- Dans le cas où les cycles de vie de deux entités liées par une relation sont très liés, mettre une Cascade et/ou un Fetch EAGER, mais faire attention aux transitivités.
- Privilégier une récupération explicite des graphes d’objet dans les services, si possible en documentant ce qui est retourné.
- Ce faisant, les problèmes qui vont survenir vont être des LazyInitialisationException, assez facile à débugger en phase de conception.
- Dans le cas contraire, des problèmes de performance vont survenir plus tard dans la vie du projet, et les refactorings qui vont en découler vont amener d’autres problèmes (dont les LazyInitialisationExceptions ….)

- Les SGBDRs offrent aussi des fonctionnalités de cascade (ON DELETE, ON UPDATE ….), il vaut mieux choisir entre les cascades côté objet ou côté SQL.

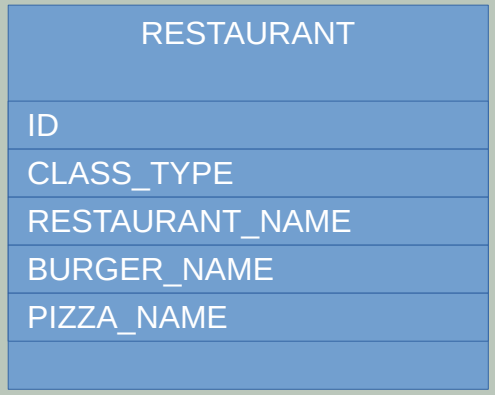
Héritage en JPA

Héritage en JPA
Hiérarchie de classes



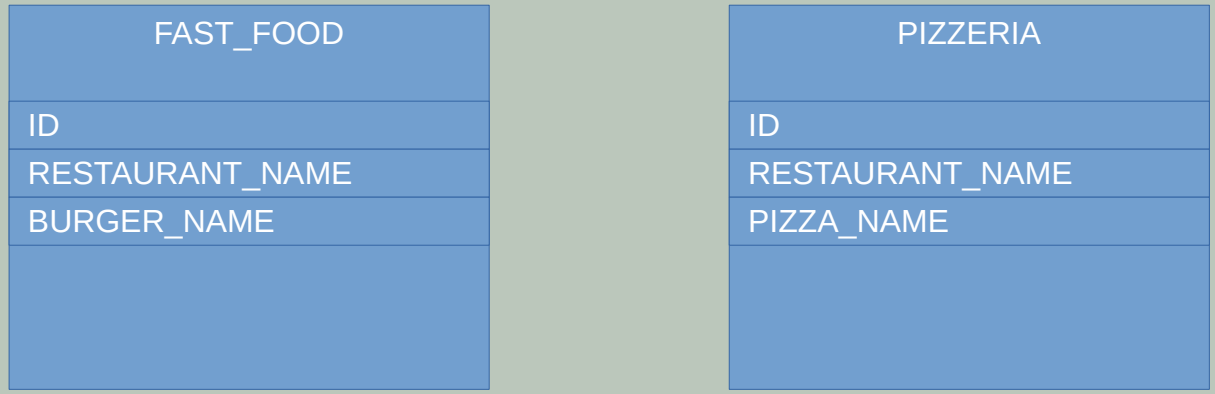
Héritage en JPA
Stratégie une table par hiérarchie de classes

Monde relationnel



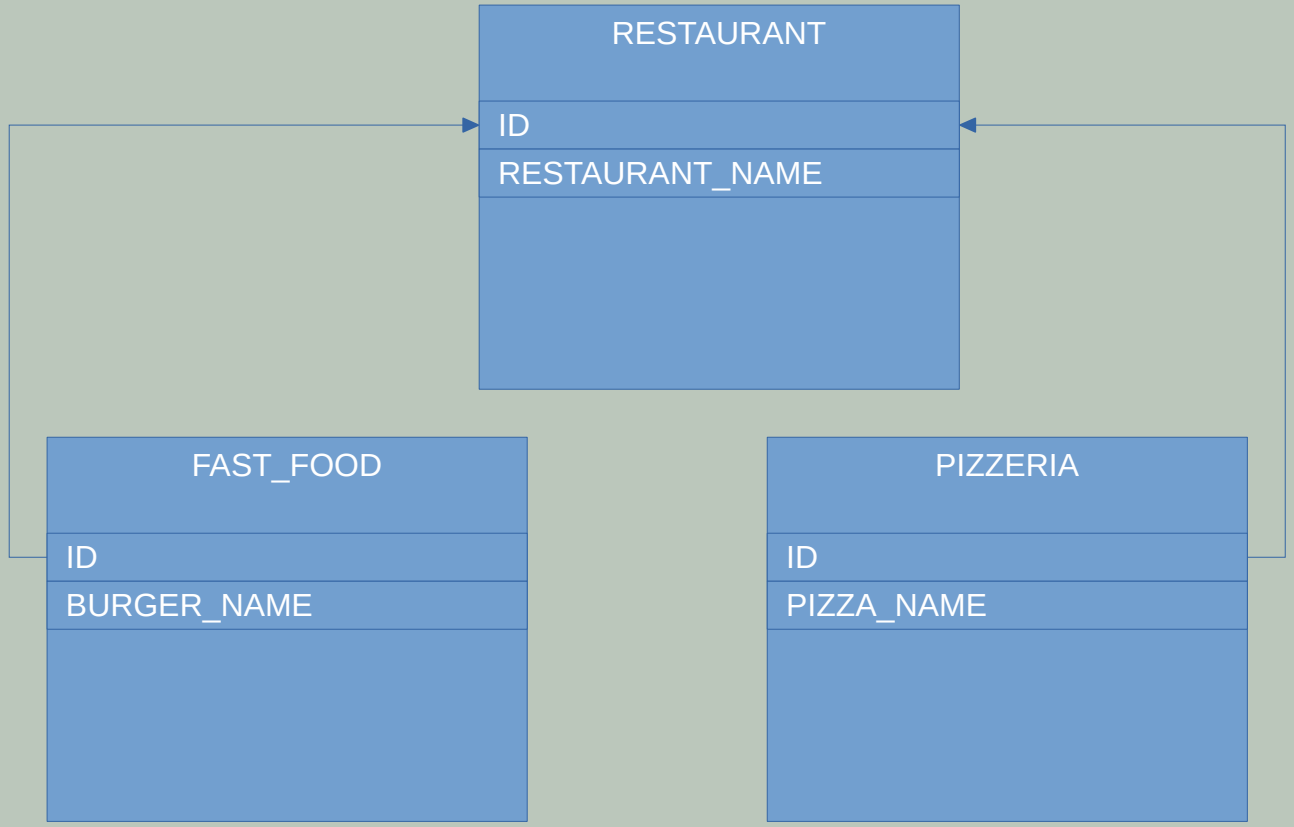
Héritage en JPA
Stratégie une table par classe concrète

Monde relationnel



Héritage en JPA
Stratégie une table par sous-classe

Monde relationnel



Héritage en JPA
Que choisir ? (diapo très subjective)

- En bref : table par classe concrète par défaut, ou table par classe.

- Pourquoi (attention, arguments subjectifs) :
- La stratégie une table par hiérarchie est efficace, mais mélange vraiment tout.
- La stratégie une table par classe semble bien architecturellement, mais est inefficace à l'usage (une jointure par classe concrète minimum).
- Dans un modèle, si l'on a une hiérarchie de classes, toutes les classes parentes devraient être abstraites.
- Architecturellement, il vaut mieux (en général) utiliser l'héritage pour des problèmes techniques, et moins pour des problèmes fonctionnels : dans ce cas, préférer la composition.

Héritage en JPA
Que choisir ? (slide très subjectif)

- Par contre, la stratégie une table par classe concrète ne permet pas les requêtes polymorphiques. Si l’on a vraiment besoin de ce genre de requêtes, utiliser table par classe concrète.

Héritage en JPA
Héritage : table par classe concrète en pratique

```java
@MappedSuperclass
public abstract class AbstractRestaurant {



@Entity
@AttributeOverride(name ="name", column = @Column(name="nome"))
public class Pizzeria extends AbstractRestaurant {
```

Héritage en JPA
Exercice : Ajouter une classe abstraite, et deux implémentations de cette classe

- Rendre la hiérarchie de classes persistable par JPA (en utilisant la stratégie une table par classe concrète), et créer un DAO par classe concrète pour persister ces entités.

Héritage en JPA
Ce qu'il faut retenir

- JPA propose plusieurs stratégies de mapping d'une hiérarchie de classes vers une ou plusieurs tables.
- Il faut connaître ces stratégies :
  - une table par hiérarchie de classes,
  - une table par classe concrète,
  - une table par classe,

  Afin d'utiliser la meilleure, selon le contexte technique et fonctionnel.

JPA avancé

JPA avancé
A propos des verrous

- Les verrous peuvent se faire de plusieurs manières :
  - côté base de données, de façon “pessimiste”, sur une ligne, une colonne, une table ... (dépend du système de gestion de base de données).
  - côté code, de façon optimiste.
  - Un verrou optimiste se gère ainsi :
    - Chaque entité verrouillable contient une colonne spéciale (verrou) avec un incrément.
    - Toute opération de modification sur une entité augmente la valeur du verrou ...
    - Mais s’assure auparavant que la valeur du verrou n’a pas changé.
  - Les requêtes émises sont du type : UPDATE MA_TABLE SET .... , VERROU = 12 WHERE ID=1 AND VERROU = 11;
  - Elles échoueront si jamais un autre processus a augmenté la valeur de VERROU.

JPA avancé
A propos des verrous

Optimiste

@Version

Sur un attribut (Integer par exemple), indique une colonne de version, qui est utilisé pour le verrouillage optimiste.

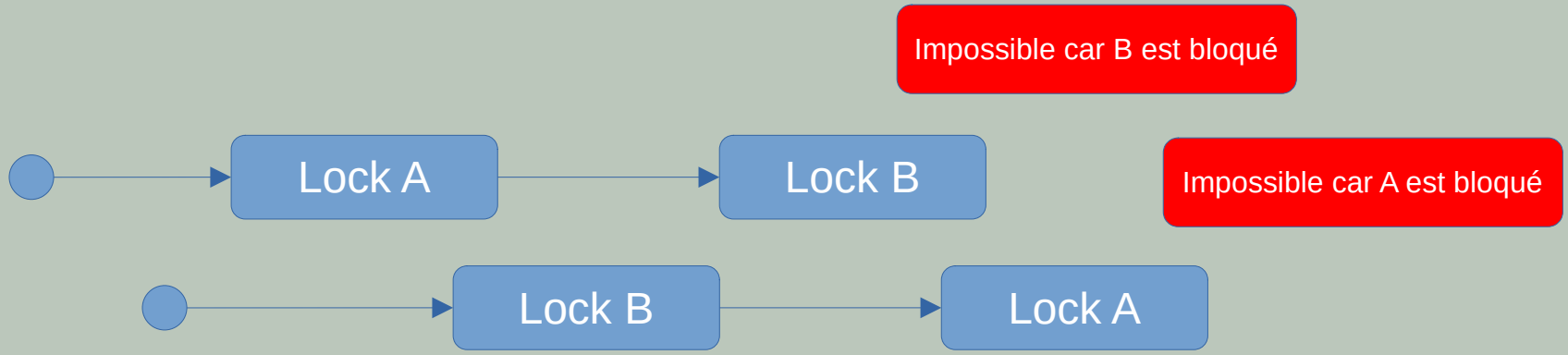
Pessimiste

```
em.createQuery("select i from Item i where i.category.id = :catId")
.setLockMode(LockModeType.PESSIMISTIC_READ)
.setHint("javax.persistence.lock.timeout", 5000)
.setParameter("catId", categoryId)
.getResultList();
```

Peut lancer une exception

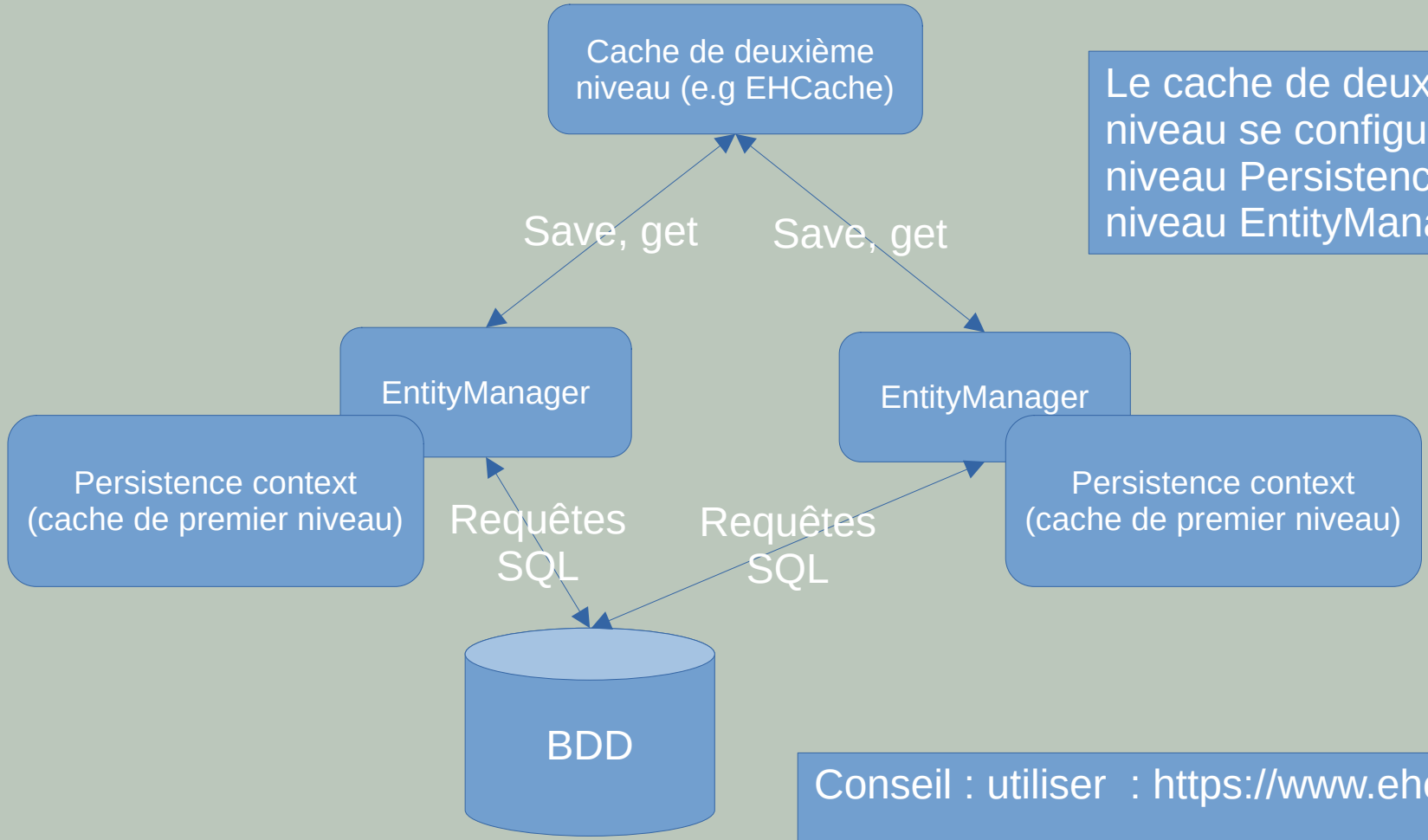
Lock

JPA avancé
A propos des deadlocks



Verrouiller une table, ou mieux une ligne, est parfois nécessaire, par exemple dans le cas d'applications de réservations. Il faut faire attention à réduire la taille de ce qui est verrouillé, et parfois construire son modèle relationnel en conséqunce.
Pour éviter certains deadlocks, une technique est de s'assurer que les algorithmes créant les locks les font dans le même ordre.

JPA avancé
Le cache de deuxième niveau



Le cache de deuxième niveau se configure, au niveau Persistence, non au niveau EntityManager

Conseil : utiliser : https://www.ehcache.org/

JPA avancé
Implémentations de JPA

- JPA (Java Persistence API) a été renommé en Jakarta Persistence en 2019. La version 3.0 est sortie 2020. Quelques fournisseurs de JPA 3 :
  - DataNucleus (version >=6.0)
  - EclipseLink (version >=3.0)
  - Hibernate (version >=5.5)
- JPA 3.1 a été spécifiée en 2022, avec Jakarta EE 10. JPA 3.1 nécessite une version de Java >= 11. Cette version améliore la gestion des UUIDs ajoute de nouvelles fonctions JPQL (sur les mathématiques, les jours, les heures). Quelques fournisseurs de JPA 3.1 :
  - DataNucleus (version >= 6.0)
  - EclipseLink (version >= 4.0)
  - Hibernate (version >= 6.0)

https://fr.wikipedia.org/wiki/Jakarta_Persistence

JPA avancé
Ce qu’il faut retenir

- Les verrous optimistes et pessimistes sont gérés par JPA, et peuvent être hautement configurables.
- L’entityManager est un cache de premier niveau. On peut y adjoindre un cache de second niveau.

Les conseils du formateur

- Il ne faut verrouiller que ce qui doit l’être. C’est important pour des raisons de performance ...
- ... et aussi pour éviter les deadlocks.
- EHCache est un cache de second niveau sur-utilisé en production, vous pouvez vous en servir sans problème.

Fin
Conclusion

- Questions ? Réponses !

Merci pour votre attention !