JAVA/JEE

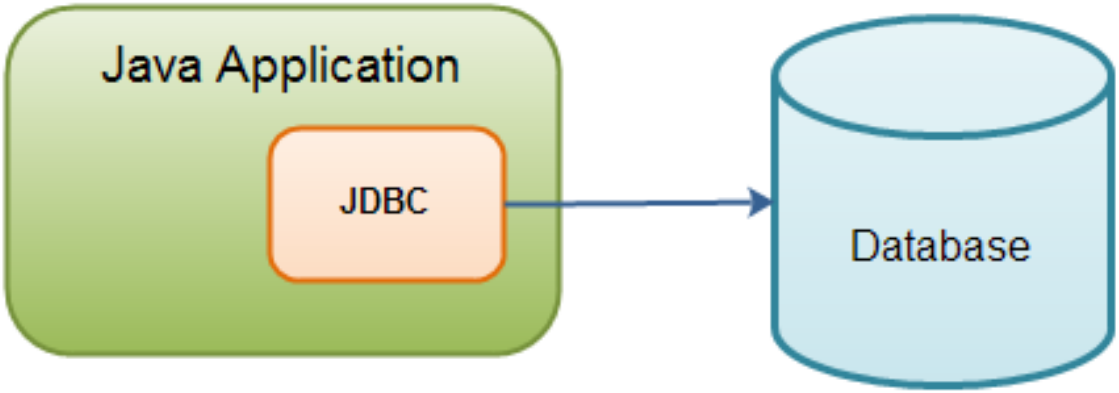
L'accès aux données

L’accès aux données

- Qu’est ce qu’une donnée ?
- Un système peut-il fonctionner sans donnée ?
- Quels sont les différents type de données disponibles pour un serveur web ?
- Comment accède-t-on à une base données ?

JDBC

- **J**ava **D**ata **B**ase **C**onnection
- API qui standardise l'accès à une base de données
- Une implémentation (Driver) existe pour chaque produit du marché (MySQL, PostgreSQL, Oracle…)



JDBC

- Il existe différents type de driver
- JDBC Offre une API simple permettant de requêter une base de données
- Gestion du pooling
- Gestion des transactions

Exemple d'utilisation de JDBC

```
Connection conn = null;
try{
   Class.forName("com.mysql.jdbc.Driver").newInstance();
   System.out.println("JDBC driver successfuly loaded");

   conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "");
   Statement stmt = conn.createStatement();
   ResultSet result = stmt.executeQuery("SELECT * FROM items");

    while(result.next()){
      System.out.print(result.getInt(1) + result.getString(3) );
       System.out.print(result.getInt("id") + result.getString("name") );
   }
}finally{
    if(conn != null){
        conn.close();
    }
}
```

Les POJO

- **P**lain **O**ld Java **O**bject
- Object simple comportant uniquement:
  - Un ou plusieurs constructeur
  - Des attributs privés
  - Des getter et des setter
- Un POJO permet de traduire facilement le modèle de données de l'application en objet JAVA
- Un POJO est transmis à travers toutes les couches de l'application: accès aux données, métier, présentation…

Example

```
public class Facture {

    private String numfact;
    private double montant;
    private Client client;

    public Facture() { } // Constructeur par défaut
    public Facture(String numfact) { this.numfact = numfact; }

    public void setMontant(double montant) { this.montant = montant; }
    public double getMontant( ) { return montant; }

    public Client getClient( ) { return client; }
    public void setClient(Client client) { this.client = client; }

}
```

Le pattern DAO

- Offre une abstraction de la manière dont sont stockés les données
- Pour un même POJO on peut avoir un DAO stockant sur disque, un autre en base, un autre en cache…
- Simplifie le remplacement de la couche d'accès aux données
- Un DAO est souvent constitué des méthodes CRUD (Create Retrive Update Delete) de base ainsi qu'éventuellement de méthodes plus spécifique (ex. recherche par clé étrangère)

Exemple DAO

- Exemple de signature d'un DAO

```
public class TrackDao {

        public TrackDao(Connection connection)
        public boolean create(Item item);

        public boolean update(Item item);

        public boolean delete(Item item);

        public List<Item> find();
}
```

JPA

- **J**ava **P**ersistence **A**PI
- Gestion du mapping objet/relationnel (ORM)
- JPA ne spécifie qu'une interface, il faut ensuite choisir une implémentation:
  - Hibernate
  - TopLink
  - JDO

JPA

- Les dernières version de JPA permettent d'annoter les POJO qu'on appel alors Entité.
  - **@Entity** permet désigner un POJO persisté en base de données
  - **@Id** défini un champ utilisé comme ID
  - **@GeneratedValue** permet de définir un auto incrément
  - Les noms des variables sont automatiquement mappé avec le nom des champs en base de données. Si le mapping n'est pas exact on peut le redéfinir avec l'annotation **@Column**
  - **@JoinColumn** permet de définir le lien entre deux entités par une clé étrangère.
  - Les annotations **@OneToOne**, **@ManyToOne**… permettent de définir la nature de la relation
  - **@JoinTable** permet de définir une relation entre deux entités qui s'effectue à travers une table d'association

Exemple d'annotation avec JPA

```
@Entity
public class Facture {

    @Id
    private String numfact;

    private double montant;
    private Client client;

    public Facture() { } // Constructeur par défaut (obligatoire pour Entity bean)
    public Facture(String numfact) { this.numfact = numfact; }

    public void setMontant(double montant) { this.montant = montant; }
    public double getMontant( ) { return montant; }

    @ManyToOne
    @JoinColumn (name = "RefClient")
    public Client getClient( ) { return client; }
    public void setClient(Client client) { this.client = client; }

}
```

Le fichier persistence.xml

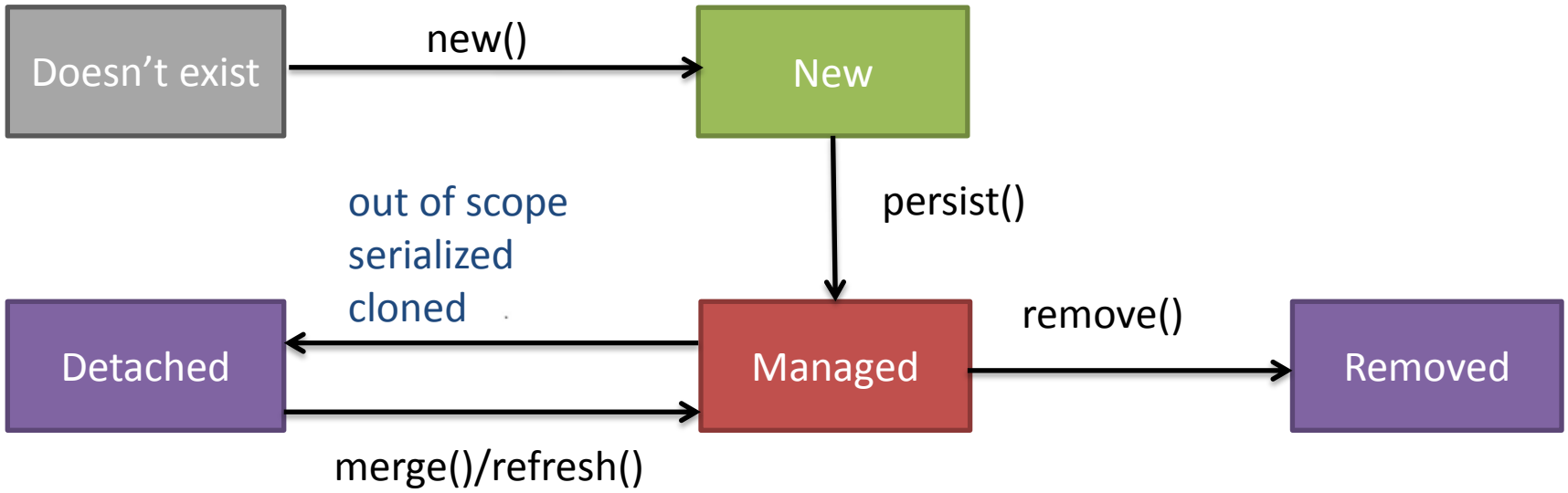
- Doit être défini dans le dossier /META-INF
- Défini un ensemble de persistance unit. La configuration d'une persistance unit comprend notamment les paramètres d'accès à la base de données

Exemple

```
<persistence
    <persistence-unit  name="testPU">
         <provider>org.hibernate.ejb.HibernatePersistence</provider>
             <properties>
                <property name="javax.persistence.jdbc.driver"
                          value="com.mysql.jdbc.Driver" />
                <property name="javax.persistence.jdbc.url"
                          value="jdbc:mysql://localhost:3306/testPU" />
                <property name="javax.persistence.jdbc.user" value="root" />
                <property name="javax.persistence.jdbc.password" value="root" />
                <property name="hibernate.dialect"
                          value="org.hibernate.dialect.MySQLDialect" />
                <property name="hibernate.show_sql" value="true" />
      </properties>
    </persistence-unit>
</persistence>
```

Entity manager

- L'entity manager est une classe qui gère le cyle de vie d'une ou plusieurs entités



Le langage JPQL

- Langage permettant de requêter directement des entités plutôt que des tables
  - Abstrait les spécificités de la base de données
  - Le code Java est plus cohérent
  - Même mots clés qu'en SQL: select, from, where, order by, group by…
- Utilisation de la notation `object.attribute`

```
SELECT a FROM Author a ORDER BY a.firstName,
a.lastName

SELECT DISTINCT a FROM Author a INNER JOIN
a.books b WHERE b.publisher.name = 'XYZ Press'
```

Exemple

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("musciPU");
EntityManger em = emf.createEntityManager();


List<Item> items=
em.createQuery("SELECT i FROM Item i").getResultList();

Query query = el.createQuery("SELECT a FROM Author a  WHERE a.lastName
IS NULL OR LOWER(a.lastName) = LOWER(:lastName)");

query.setParameter("lastName", lastName);

List<Author> authors = query.getResultList();

em.close();
emf.close();
```

Lazy loading

- Par défaut la stratégie utilisée par Hibernate pour charger les relations entre Entités est le lazy loading
- Il est possible de définir manuellement la stratégie à utiliser lors de la définition du POJO
- Le lazy loading permet d'éviter de charger une grosse partie de la base en mémoire mais n'est pas possible dans tous les cas

Les transactions

- Les transactions sont une des solutions possible au problème de la cohérence des données. Elle doit posséder les propriétés suivantes
  - **A**tomicité
  - **C**ohérence
  - **I**solation
  - **D**urabilité

Les transactions

- Une transaction est nécessaire pour toute opération d'**écriture** en base: create, update, delete
- Une transaction doit idéalement être définie au niveau de la couche métier pour pouvoir faire des **rollback** pertinent

Exemple

```
public void create(Item item){
   // Get entity manager
   EntityManagerFactory emf=
Persistence.createEntityManagerFactory(« productsPU");
    EntityManager em = emf.createEntityManager();

    //Get transaction
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();

    em.persist(item);
    transaction.commit();

    //Close entity manager
    em.close();
    emf.close();
}
```