Cours - Appli Web - S8

> Écrit par Clément Lizé

1 - Introduction

1.1 - Principe et désignation

- Appli web = système client (navigateur) ↔ serveur (serveur web, ex: Apache)
- URI = URN (très peu utilisé) ou URL (couramment utilisé)

1.2 - Langage HTML

(Hyper Text Markup Language)

- Format : `<balise attributs> contenu </balise>`

  ▶ Exemple : `<a href="http://enseeiht.fr">Cliquez ici</a>`

  ➡️ Contenu = texte ou autres balises → imbrication
- Exemple de fichier html :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page title</title>
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>This is a paragraph.</p>
    <p>This is another paragraph.</p>
  </body>
</html>
```

- Principales balises

```
<img src=""/> <!-- Image, src=PATH -->
<div></div> <!-- Bloc, sert à grouper les balises -->
<a href=""></a> <!-- Lien cliquablen href=URL -->
<p></p> <!-- Paragraphe -->
<h1></h1>, <h2></h2> <!-- Titres -->
<ol><ul></ul></ol> <!-- Listes -->
<form></form> <!-- Formulaire -->
```

1.3 - Mise en page avec CSS

(Cascading Styling Sheet)

➡️ Sert à définir la présentation d'une page

- Principe : donner des propriétés (taille, couleur, ...) aux balises
- Exemple de fichier CSS :

```
h1 {  /*Appliqué aux balises <h1>*/
  color: blue;
  text-align: center;
}

/*Appliqué aux balises portant l'attribut class="ma_classe"*/
/*Exemple : <p class="ma_classe une_deuxieme_classe_why_not"></p
>*/
.ma_classe {
  font-family: Arial;
}

/*Appliqué aux balises portant l'attribut id="mon_id"*/
/*Exemple : <h3 id="mon_id"></h3>*/
.mon_id{
  font-weight: bold;
}
```

- Penser à importer la feuille CSS au début du fichier HTML :

```
<head>
  <link rel="stylesheet" type="text/css" href="./style.css"/>
</head>
```

1.4 - Le protocole HTTP

(HyperText Transfer Protocol), basé sur TCP/IP

- Utilisé quasiment tout le temps pour le web
- Composition d'une URL :

- Transaction HTTP = requête + réponse

1.5 - Formulaires HTML

➡️ But : envoyer des données saisies par l'utilisateur, du client vers le serveur

- Balise form

```
<form action="/getComptes" method="GET">
<!--     URL d'envoi       GET ou POST -->
</form>
```

- Balise de saisie

```
<input type="text" name="prenom" value="Johan">
```

- Méthodes HTTP :
  - Principales méthodes :

    `GET` récupérer une ressource (ex: page HTML, données JSON)
    ➡️ Paramètres inclus dans l'URL : visibles et limités en taille (255 c)
    exemple : `http://127.0.0.1:8080/ma_page?prenom=mathis&nom=blanc`

    `POST` envoyer des données (ex: formulaire)
    ➡️ Paramètres dans le corps de la requête : confidentialité, pas de limite de taille
  - Autres méthodes (peu utilisées) :

    HEAD, OPTIONS, TRACE, CONNECT, PUT, DELETE
- Principaux codes de retour :

  200 (OK), 301 (moved permanently), 400 (bad request), 401 (unauthorized), 404 (not found), 500 (internal server error), 503 (service unavailable)

- Types possibles : TEXT, PASSWORD, SUBMIT (bouton de soumission du formulaire), RADIO (unique case à cocher), CHECKBOK, HIDDEN
- A l'envoi, le serveur récupère un couple name=value. Si le champ est modifiable par l'utilisateur, l'attribut **value** va changer au fur et à mesure de la saisie. Sur cet exemple, si l'utilisateur inscrit "Mathis" dans le champ, le serveur va recevoir *prenom=Mathis*

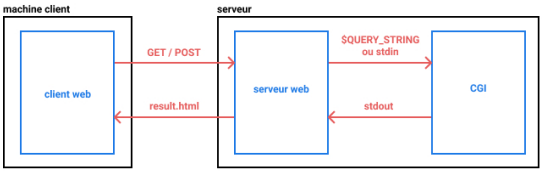
- Exemple

```
<html>
  <head>
    <title>MaBanque</title>
  </head>
  <body>
    <form method="post" action="/servlet/BanqueAccount">
      <p>numero de compte<input type="text" name= "num"></p>
      <p>montant<input type="text" name= "val"></p>
      <input type="submit" name="operation" value="solde">
      <input type="submit" name="operation" value="debit">
      <input type="submit" name="operation" value="credit">
    </form>
  </body>
</html>
```

1.6 - Scripts CGI

➡️ Programme qui génère un contenu en réponse à une requête

- Programmé dans n'importe quel langage
- Réponse passe par la sortie standard *stdout*



- Requête `GET` : paramètres passent par la variable d'environement *$QUERY_STRING*
- Requête `POST` : paramètres passent par l'entrée standard *stdin*

1.7 - Cookies HTTP

➡️ Mécanisme de stockage d'informations chez le client

- Utilisation : sauvegarde d'options, session utilisateur
- Créé par le serveur, stocké chez le client
- Instruction à mettre dans l'en-tête d'une requête HTTP

```
// obligatoire
name=value     // associer une valeur à une clé
// Optionnel
expires        // date d'échéance du cookie
domain         // le serveur émetteur du cookie
path           // association à un ensemble de ressources du ser
veur
secure         // nécessite une connexion https
```

- Gestion par le client :
  - Mémorise les cookies qu'il reçoit
  - Peut insérer des cookies dans une requête au serveur
  - Pour effacer un cookie, mettre une expiration courte
  - Si pas d'expiration, le cookie est supprimé à la fermeture du navigateur

---

2 - Web dynamique

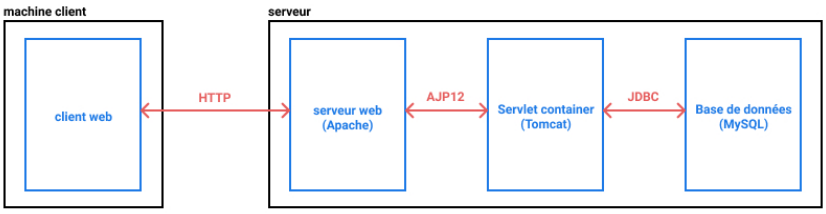
2.1 - Servlet

2.1.1 - Fonctionnement

➡️ Programme Java exécuté côté serveur, répond à une requête (généralement en générant une page HTML)

⚠️ Ne pas confondre avec Applet : programme Java chargé et exécuté par le client web

- S'exécute dans un *Servlet container* sur une JVM (Java Virtual Machine) : généralement *Tomcat* ou *Jetty*



- API Servlet HTTP

```
public void init()
protected void doGet(HttpServletRequest request, HttpServletResponse respo
nse)
protected void doPost(HttpServletRequest request, HttpServletResponse resp
onse)
```

➡️ 3 méthodes principales à implanter
➡️ Méthodes :
- `init()` : exécuté à l'initialisation de la servlet
- `doGet()` : exécuté à la réception d'une requête `GET`
- `doPost()` : exécuté à la réception d'une requête `POST`

➡️ Paramètres :
- `HttpServletRequest` : manipuler la requête reçue
- `HttpServletResponse` : générer la réponse

2.1.2 - Exemple simple

```
<html>
  <head>
    <title>Directory</title>
  </head>
  <body>
    <form action="Directory" method="post">
      Prénom : <input type="text" name="prenom">
      Nom: <input type="text" name="prenom">
      <input type="submit" name="op" value="afficher_prenom">
      <input type="submit" name="op" value="afficher_nom">
    </form>
  </body>
</html>
```

- Ici le serveur web se contente de retourner les données insérées par l'utilisateur dans le formulaire. Il est possible d'interagir avec une base de données pour stocker et récupérer des données
- Il y a 2 boutons submit : chacun déclenche une action différente dans la Servlet grâce au champ `value`

```
@WebServlet("/Directory")
public class Directory extends HttpServlet {

  protected void doPost(HttpServletRequest request, HttpServletResponse response) {

    response.setContentType("text/html");
    response.getWriter().print("<html><body><h1>Directory</h1>");

    String op = request.getParameter("op");
    if (op.equals("afficher_prenom")) {
      resonse.getWriter().print("<p>" + request.getParameter("prenom") + "</p>");
    }
    else if (op.equals("afficher_nom")) {
      resonse.getWriter().print("<p>" + request.getParameter("nom") + "</p>");
    }
  }
}
```

- Pour écrire du HTML, il faut passer par `response.getWriter()` puis faire des `print()` ou `println()` comme d'habitude.
  ⚠️ Les balises HTML doivent être écrites dans les print, voir l'exemple ci-contre

---

2.1.3 - Interaction avec une base de données

```
// Connexion à la BDD
String db_url = "jdbc:hsqldb:hsql://localhost/xdb";
String db_user = "sa";
Class.forName("org.hsqldb.jdbcDriver");
Connection con = DriverManager.getConnection(db_url, db_user, null);

// Récupération de données
Statement stmt = con.createStatement();
ResultSet res = stmt.executeQuery("SELECT * FROM directory");
System.out.println(res.getString("firstname")+" "+res.getString("las
tname"));
```

- Une Servlet peut échanger avec une base de données. Il faut l'initialiser (4 premières lignes) puis effectuer des requêtes SQL. Enfin, on peut extraire les données de la réponse SQL

2.1.4 - Session

➡️ Principe : une requête dépend du résultat des requêtes précédentes

➡️ Couramment implanté avec des cookies

```
// Création de session
HttpSession HttpServletRequest.getSession()
HttpSession HttpServletRequest.getSession(boolean create)

// Utilisation de la session
Object getAttribute(String name)
Enumeration getAttributeNames()
long getCreationTime()
int getMaxInactiveInterval()
void invalidate()
void removeAttribute(String name)
void setAttribute(String name, Object value)
void setMaxInactiveInterval(int interval)
```

2.1.5 - Cookies

```
// Création, initialisation
Cookie(java.lang.String name, java.lang.String value)
void setValue(java.lang.String newValue)
void setMaxAge(int expiry)
void setDomain(java.lang.String pattern)
java.lang.String getValue()
java.lang.String getDomain()
int getMaxAge()

// A l'exécution
Cookie[] HttpServletRequest.getCookies()
HttpServletResponse.addCookie(javax.servlet.http.Cookie)
```

2.1.6 - Packaging et déploiement

- 1 appli = 1 dossier
- Répertoire *WEB-INF*
  - Répertoire *classes* : classes et servlets
  - Répertoire *lib* : les jar externes
- Déploiement : création d'un fichier *WAR* à placer dans le Servlet container

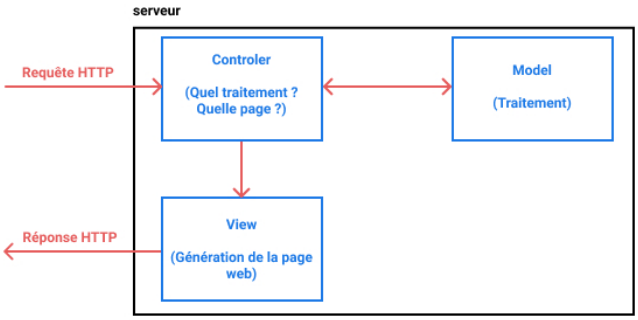
2.1.7 - Bilan

- Facile à programmer
- Mélange entre le front-end (HTML) et le back-end (Traitement des données, BDD)
  ➡️ Prochain objectif : séparer ces aspects

2.2 - Modèle MVC

(Model View Controler)

➡️ Séparation entre :
- Le contrôleur : Servlet qui aiguille les requêtes
- Le modèle : les classes (beans) qui traitent les données
- La vue : les pages JSP pour l'affichage en HTML



2.3 - JSP

(Java Server Page)

2.3.1 - Fonctionnement

➡️ But : générer des pages dynamiques

- Page HTML dans laquelle on introduit des scripts Java

2.3.2 - Exemple simple

```
<%@ page language="Java" %>
<html>
  <head>
    <title>First.jsp</title>
```

- Compilé dynamiquement en Servlet


```
  </head>
  <body>
    <h1>Nombres de 1 à 10</h1>
      <% for(int i=1; i<=10; i++) {
      out.println(i + "<br>");  // avec out = response.getWriter();
      }
      %>
  </body>
</html>
```

- Cet exemple affiche les entiers de 1 à 10. La balise `<br>` effectue un retour à la ligne en HTML
- Quand le client appelle la JSP, le script est exécuté et le client reçoit une page HTML complète sans java

---

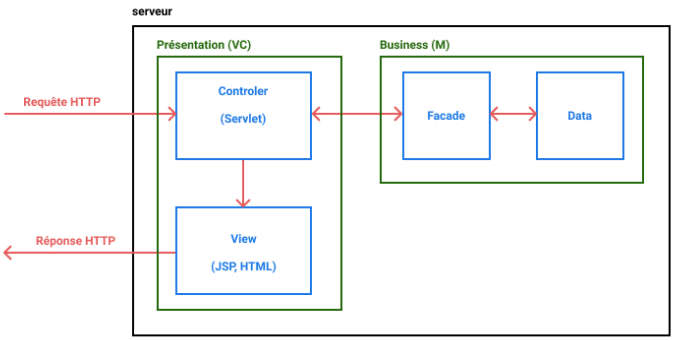
2.3.3 - Directives, déclarations et scripts

- **Directive** = paramétrage du comportement d'exécution de la JSP
  Écriture : `<%@ attribut1="valeur" attribut2="valeur" %>`
  Exemple : `<%@ page language="Java" %>`
- **Déclaration** = variables et méthodes globales à la page
  Écriture : `<%! declaration %>`
  Exemple : `<%! String chaine = "bonjour"; int num = 10; %>`
- **Script** = instructions java
  - **Code java**
    Écriture : `<% code %>`
    Exemple : `<% for(int i=0; i<10; i++) { %>` ou `<% response.getWriter().println(i); %>`
  - **Évaluation d'expression**
    Écriture : `<%= expression %>`
    Exemple : `<%= i %>` affiche i, pas besoin d'utiliser `response.getWriter()`

2.3.4 - Lien HTML - Servlet - JSP

- Page HTML référence une servlet dans un formulaire
  `<form action="ServletURL" method="post"></form>`
- Servlet référence une page JSP
  `RequestDispatcher disp = request.getRequestDispatcher(”page.jsp”); disp.forward(request, response);`
- Passage de paramètres entre servlet et JSP
  - Côté servlet :
    `request.setAttribute("key", value);`
  - Côté JSP :
    `value = (ValueType) request.getAttribute("key");`

2.3.5 - Architecture avec façade



- A gauche : "front-end" : Servlets et contrôleurs de vue
- A droite : "back-end" : Modèle, code métier ➡️ Implante la logique de l'application, indépendant du web
  - Facade : interface d'utilisation de la partie métier
  - Data : objets que l'on stocke

2.3.6 - Exemple complet

Objectif : application simplifiée de banque

- Servlet

```
@WebServlet("/Serv")
public class Serv extends HttpServlet {
```

- Facade

```
public class Facade {

  private Map<Integer, Compte> comptes = new Hashtable<
```

```
  private Facade facade = new Facade();

  protected void doGet(HttpServletRequest request, HttpServletResponse res
ponse)
      throws ServletException, IOException {
    try {

      String action = request.getParameter("action");

      if (action.equals("consulter")) {

        int num_compte = Integer.parseInt(request.getParameter("num_compt
e"));
        request.setAttribute("num_compte", num_compte);  // pour la JSP
        request.setAttribute("compte", facade.getCompte(num)); // pour la
 JSP
      }

      else if (action.equals("ajouter_soustraire")) {

        int num=Integer.parseInt(request.getParameter("num"));
        int montant=Integer.parseInt(request.getParameter("montant"));
        request.setAttribute("num", num);
        facade.credit_debit(num, montant);  // Opération sur les données
    }
    catch (Exception e) {

      request.setAttribute("exception", e.getMessage());
    }

    // Envoi d'une requête au JSP, redirection de la réponse vers le web
    request.getRequestDispatcher("Banque.jsp").forward(request, response);
  }
}
```

- JSP

```
<%@ page language="java" import="bk.*, java.util.*"
  contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <form action="Serv" method="get">
    Numéro : <input type="text" name="num_compte"><br>
    Montant : <input type="text" name="montant"><br>
    <input type="submit" name="action" value="consulter">
    <input type="submit" name="action" value="ajouter_soustraire">
    </form>

    <!-- Affichage du compte -->
    <% Compte compte = request.getAttribute("compte");
      if (compte) != null) { %>

      <p>Compte numéro <%= compte.getNum() %> : nom=<%= compte.getNom() %>
         solde=<%= compte.getSolde() %>
      </p>

    <% } %>

  </body>
</html>
```

```
>();

  public Facade() {

    comptes.put(1, new Compte(1, "Clément", 1000));
    comptes.put(2, new Compte(2, "Léo", 4000));
  }

  public Compte getCompte(int num) throws RuntimeExcept
ion {

    Compte c = comptes.get(num);
    if (c == null) throw new RuntimeException("Not foun
d");
  }

  public void credit_debit(int num, int montant) {

    Compte c = this.getCompte(num);
    c.modifierSolde(montant);
  }
}
```

- Classe Compte

```
public class Compte {

  private int num;
  private String nom;
  private int solde;

  public Compte (int num, String nom, int solde) {

    this.num = num;
    this.nom = nom;
    this.solde = solde;
  }

  public void modifierSolde(int montant) {

    this.solde += montant;
  }

  // getters
}
```

---

3 - EJB

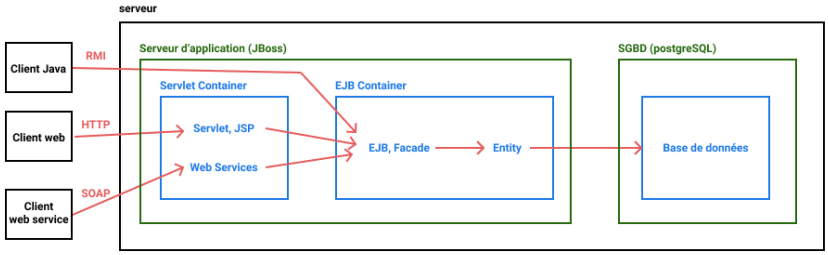
(Entreprise Java Beans)

➡️ Objectif : structurer la partie Modèle (métier)

3.1 - Architecture

Par rapport à l'architecture au dessus :

- On a divisé *Data* en 2 parties : un *Entity Bean* et une base de données
  ➡️ On ne manipule plus la BD avec des requêtes SQL, grâce aux EJB



Entities on "voit" des objets Java dans la BD

- L'application est accessible depuis d'autres types de client (par ex: client Java en RMI)

3.2 - Définitions et caractéristiques

- EJB = framework qui permet de gérer des :
  - Entity Beans : Représentation en objets Java de données stockées en BD
    ➡️ Évite d'accéder à la BD avec JDBC et avec des requêtes SQL
    ➡️ On manipule des objets Java "comme d'habitude" et ils sont automatiquement convertis dans la BD
  - Session Beans : Implantation de l'interface, code de l'application qui utilise les données
    ➡️ Manipulation des Entity Beans
  - Message Driven Beans : Traitements asynchrones (pas vu dans le cours)
- EJB propose de nombreux services sans effort d'implantation : répartition sur plusieurs machines, transactions (éviter des états incohérents en cas de crash), persistance des données automatique, chargement / déchargement des données en mémoire, gestion de la concurrence, sécurité

3.3 - EJB Session

- Accessibles en local, à distance (ex: RMI) ou depuis la servlet
- 3 manières d'implanter une Session Bean :
  1. Stateless : ne conserve aucune donnée sur son état ; une instance pour plusieurs connexions de clients
  2. Statefull : conserve les données entre les échanges avec le client ; une instance par connexion de client
  3. Singleton : une unique instance quelque soit le nombre de connexions ; on peut gérer un état partagé par tous les clients
     ➡️ C'est cette implantation qu'on utilisera le plus
- Implantation en utilisant des annotations (exemple : `@Singleton`)
  ➡️ Déclarer une interface avec `@Remote` ou `@Local`
  ➡️ Déclarer la classe avec `@Singleton` ou `@Satefull` ou `@Stateless`

```
@Remote ou @Local
public interface ma_classe {}

@Singleton ou @Satefull ou @Stateless
public class ma_classe_impl {}
```

3.4 - Exemple complet

Reprise de l'application de banque*

⚠️ Ce code fonctionne uniquement si la servlet se situe dans la même JVM que le serveur d'application

- Servlet

```
@WebServlet("/Serv")
public class Serv extends HttpServlet {

  @EJB
  private Facade facade = new Facade();

  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    try {

      String action = request.getParameter("action");

      if (action.equals("consulter")) {

        int num_compte = Integer.parseInt(request.getParameter("num_compte"));
        request.setAttribute("num_compte", num_compte);  // pour la JSP
        request.setAttribute("compte", facade.getCompte(num)); // pour la JSP
      }

      else if (action.equals("ajouter_soustraire")) {

        int num=Integer.parseInt(request.getParameter("num"));
```

- Classe Compte
  Pas grand chose qui change à part le *implements*

```
// Serializable pour que ça marche en réparti
public class Compte implements Serializable {

  private int num;
  private String nom;
  private int solde;

  public Compte (int num, String nom, int sol
de) {

    this.num = num;
    this.nom = nom;
    this.solde = solde;
  }

  public void modifierSolde(int montant) {

    this.solde += montant;
  }
```

```
        int montant=Integer.parseInt(request.getParameter("montant"));
        request.setAttribute("num", num);
        facade.credit_debit(num, montant);  // Opération sur les données
    }
    catch (Exception e) {

      request.setAttribute("exception", e.getMessage());
    }

    // Envoi d'une requête au JSP, redirection de la réponse vers le web
    request.getRequestDispatcher("Banque.jsp").forward(request, response);
  }
}
```

```
  // getters
}
```

- JSP

```
<%@ page language="java" import="bk.*, java.util.*"
  contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <form action="Serv" method="get">
    Numéro : <input type="text" name="num_compte"><br>
    Montant : <input type="text" name="montant"><br>
    <input type="submit" name="action" value="consulter">
    <input type="submit" name="action" value="ajouter_soustraire">
    </form>

    <!-- Affichage du compte -->
    <% Compte compte = request.getAttribute("compte");
      if (compte) != null) { %>

      <p>Compte numéro <%= compte.getNum() %> : nom=<%= compte.getNom() %>
         solde=<%= compte.getSolde() %>
      </p>

    <% } %>

  </body>
</html>
```

- Interfaces de la façade

```
@Local
public interface BankLocal {

  public Compte getCompte(int num) throws RuntimeExcept
ion;
  public void credit_debit(int num, int montant);
}

@Remote
public interface BankRemote {

  public Compte getCompte(int num) throws RuntimeExcept
ion;
  public void credit_debit(int num, int montant);
}
```

- Facade

```
@Singleton
public class Facade implements BankLocal, BankRemote {

  private Map<Integer, Compte> comptes = new Hashtable<>();

  public Facade() {}

  // Cette méthode sert de "constructeur" dans EJB car JBoss
  // finalise la création des instances après l'appel du constructeur
  @PostConstruct
  public void initialisation() {

    comptes.put(1, new Compte(1, "Clément", 1000));
    comptes.put(2, new Compte(2, "Léo", 4000));
  }

  public Compte getCompte(int num) throws RuntimeException {

    Compte c = comptes.get(num);
    if (c == null) throw new RuntimeException("Not found");
  }

  public void credit_debit(int num, int montant) {

    Compte c = this.getCompte(num);
    c.modifierSolde(montant);
  }
}
```

3.5 - Particularités en réparti

Comme dit plus haut, le système EJB fonctionne aussi en réparti (un projet Java sur une autre JVM peut accéder aux données en RMI par exemple) et en web service

- Réparti (RMI) : voir intergiciels

- Web Service : rajouter `@WebService` avant la déclaration de l'interface, puis ajouter `@WebMethod` avant chaque méthode qu'on rend accessible. Enfin, ajouter `@WebService(endpointInterface="nom_de_l'interface", serviceName="nom_du_service_souhaité")` avant la déclaration de la classe *Facade*

3.6 - Bilan

- Gros problème : les données ne sont pas persistantes : quand on arrête JBoss, on perd les données. En effet, les données sont stockées dans une table dans la Session Bean. On aimerait les stocker dans une vraie base de données

---

4 - JPA

4.1 - Entity Beans

- Même concept que EJB
  ➡️ Une classe = une table dans la BD
  ➡️ Une instance = une ligne dans la table
  ➡️ Un champ = une colonne dans la table
- Programmation avec des objets Java "comme d'habitude" (POJO, Plain Old Java Object)
  ➡️ Pas d'héritage ou quoi, implantation très simple
- Échangés avec les clients (application ou servlet) ; Si échange avec d'autres JVM, doivent être sérialisables
- ⚠️ Ils doivent posséder un constructeur vide et des setter / getter
- La classe doit être annotée avec `@Entity`
- La classe doit contenir un champ annoté avec `@Id` → clé primaire de la BD

4.2 - Mapping entre le Bean et la table

- `@Table` : précise le nom de la table associé à la classe
  ➡️ Par défaut, c'est le nom de la classe
- `@Column` : précise le nom de la colonne associée à un champ
  ➡️ Par défaut, c'est le nom du champ
- `@GeneratedValue` : génération automatique de la clé primaire
- `@Transient` : Demander à ne pas tenir compte d'un champ
  ➡️ Pas d'existence dans la BD

4.3 - Entity Manager (façade)

- Permet de gérer la persistance

```
// [1]
@PersistenceContext
private EntityManager em;

// [2]
Compte c = new Compte();
em.persist(c);

// [3]
Compte c = em.find(Compte.class, num);

// [4]
TypedQuery<Compte> req = em.createQuery("select c from Compte c", Compte.clas
s);
Collection<Compte> liste_comptes = req.getResultList();
```

1️⃣ Création de l'entity manager. Pas besoin de l'initialiser (automatique)

2️⃣ `em.persist()` copie l'objet dans la base de données, dans la table associée à la classe Compte

3️⃣ `em.find()` retourne une référence à une instance d'un objet à partir de sa clé primaire. Si on modifie l'objet, la BD sera mise à jour.

4️⃣ On peut récupérer une collection d'objets en écrivant une requête SQL

4.4 - État des objets

- Objet attaché : géré dans le container EJB et associé à une image dans la BD
  ➡️ Toute modification est répercutée dans la BD
- Objet détaché : associé à une image dans la BD, mais plus géré par le contiainer EJB
  ➡️ Pour "push" les modifications en BD : `em.merge(c);`

4.5 - Associations entre les Entity

Pour expliquer cette partie, on va prendre pour exemple les relations entre les classes *Etudiant* et *Ordinateur*

4.5.1 - @OneToOne

➡️ Un étudiant possède un unique ordinateur. Un ordinateur est associé à un unique étudiant.

- Configuration par défaut :

```
public class Etudiant {

  @OneToOne
  private Ordinateur ordinateu
r;
}
```

```
public class Ordinateu
r{

  @OneToOne
  private Etudiant etu
diant;
}
```

ETUDIANT

ID
1

ORDINATEUR

ID
2

ETUDIANT_ORDINATEUR

ETUDIANT_ID	ORDINATEUR_ID
1	2

Dans cette configuration, on ne spécifie pas de règle pour le mapping dans la DB. Ainsi une 3ème table est créée et référence les couples Etudiant - Ordinateur.

- Configuration avec règle de mapping :

```
@Entity
public class Etudiant {

  // @Id, @GeneratedValue

  @OneToOne
  private Ordinateur mon_ordinateur;
}
```

```
public class Ordinateur{

  // @Id, @GeneratedValue

  @OneToOne(mappedBy="mon_ordinateu
r")
  private Etudiant etudiant;
}
```

ETUDIANT

ID	MON_ORDINATEUR
1	2

ORDINATEUR

ID
2

Dans cette configuration, on dit que le mapping est effectué par le champ *mon_ordi* de la classe *Etudiant*. Ainsi il n'y a pas de table supplémentaire créée.

4.5.2 - @OneToMany, @ManyToOne

➡️ Un étudiant peut posséder plusieurs ordinateurs mais un ordinateur ne peut appartenir qu'à un unique étudiant.

➡️ Même réflexion dans l'autre sens pour l'inverse (plusieurs étudiants par ordi, un seul ordi par étudiant)

💡Pour savoir lequel mettre : `@<< un ou plusieurs étudiant(s) ? >> To << un ou plusieurs ordinateur(s) ? >>`

- Configuration unidirectionnelle :

```
@Entity
public class Etudiant {

  // @Id, @GeneratedValue

  // Un étudiant pour plusieurs ordinateurs
  @OneToMany
  private Ordinateur mon_ordinateur;
}
```

```
@Entity
public class Ordinateu
r {

  // @Id, @GeneratedVa
lue
}
```

ETUDIANT

ID
1

ORDINATEUR

ID
2
3

ETUDIANT_ORDINATEUR

ETUDIANT_ID	ORDINATEUR_ID
1	2
1	3

Dans cette configuration, la classe *Ordinateur* ne référence pas l'étudiant associé. Une table de liaison est créée (pas toujours mais y a des chances, sinon le mapping est fait dans la table *Etudiant* par JBoss)

- Configuration bidirectionnelle :

```
@Entity
public class Etudiant {

  // @Id, @GeneratedValue

  // Un étudiant pour plusieurs ordinateurs
  @OneToMany(mappedBy="owner")
  private Ordinateur mon_ordinateur;
}
```

```
@Entity
public class Ordinateur {

  // @Id, @GeneratedValue

  // Plusieurs ordinateurs pour un étudia
nt
  @ManyToOne
  Etudiant owner;
}
```

ETUDIANT

ID
1

ORDINATEUR

ID	OWNER
2	1
3	1

4.5.3 - @ManyToMany

➡️ Plusieurs étudiants peuvent avoir plusieurs ordinateurs

- Configuration unidirectionnelle : comme un @OneToMany unidirectionnel, obligatoirement une table de liaison.
  ⚠ Si il y a un @ManyToMany de chaque côté sans *mappedBy*, alors ce sera considéré comme 2 relations indépendantes (→ 2 tables de liaison)

```
@Entity
public class Etudiant {

  // @Id, @GeneratedValue

  @ManyToMany
  private List<Ordinateur> ordinateur
s;
}
```

```
@Entity
public class Ordinateu
r {

  // @Id, @GeneratedVa
lue
}
```

ETUDIANT

ID
1
4

ORDINATEUR

ID
2
3

ETUDIANT_ORDINATEUR

ETUDIANT_ID	ORDINATEUR_ID
1	2
1	3
4	2

- Configuration bidirectionnelle : @ManyToMany des 2 côtés avec un *mappedBy*, propagation des mises à jour des 2 côtés, obligatoirement une table de liaison

```
@Entity
public class Etudiant {

  // @Id, @GeneratedValue

  @ManyToMany
  private List<Ordinateur> ordinateurs;
}
```

```
@Entity
public class Ordinateur {

  // @Id, @GeneratedValue

  @ManyToMany(mappedBy="mon_ordinateur")
  List<Etudiant> owners;
}
```

Ca donne exactement la même table que au dessus. La différence, c'est qu'on peut accéder aux étudiants qui possèdent un ordinateur depuis la classe Ordinateur

---

4.6 - Cascades, Fetch

- Quand on enlève un objet d'une liste, si celui-ci n'est référencé nulle-part, il ne sera pas supprimé automatiquement de la BD.
  ➡️ Si on veut une suppression automatique : `@OneToMany( cascade=CascadeType.REMOVE )`
- Quand on récupère une entité, par défaut on ne récupère pas les sous-entités, c'est à dire celles qui sont référencées dans des listes par exemple.
  ➡️ Si on veut tout récupérer : `@OneToMany(mappedBy="owner", fetch=FetchType.EAGER )`

---

5 - Correction du partiel 2010

⚠ **ATTENTION : Les réponses ne viennent pas du prof mais de moi-même. Je ne peux pas garantir qu'elles soient correctes.**

Question 1

> La séparation des préoccupations est un principe implanté à différents endroits dans une architecture J2EE. Analysez, expliquez.

Voir les schémas dans le cours, on essaye au fur et à mesure d'avoir des techno séparées pour bien traiter chaque partie de l'appli : front-end (contrôleur, vue), back-end (façade, entités, base de données)

Question 2

> Afin de comparez les technologies CGI, PHP, Applet, Servlet, JavaScript et JSP, veuillez compléter le tableau suivant avec des croix dans les bonnes cases.

Comparaison des technologies

Techno	Exécution côté client	Exécution côté serveur	Intégration à HTML	Programmation en Java
CGI	☐	☑	☐	☐
PHP	☐	☑	☑	☐
Applet	☑	☐	☐	☑
Servlet	☐	☑	☐	☑
JSP	☐	☑	☑	☑
Javascript	☑	☐	☐	☐

Problème

> Vous avez à implanter en EJB3 une application qui gère des imprimantes et des ordinateurs.
>
> On gère 2 beans entity :
>
> Computer avec les attributs :
> – String hostname; // la clé primaire
> – Info otherinfo;
>
> Printer avec l'attribut :
> – String printername; // la clé primaire
> – Info otherinfo;

Question 3

> On a une relation entre ces 2 beans entity telle que :
> – un ordinateur ne peut être associé qu'à une seule imprimante
> – une imprimante ne connait pas les ordinateurs qui l'utilisent
> – une imprimante peut être partagée par plusieurs ordinateurs
> – la relation est uni-directionnelle
>
> Donnez l'implantation de ces 2 beans. Dans la suite, ne vous occupez pas des imports, ni des cascades au niveau des annotations.

Analyse de l'enoncé :

- un ordinateur ne peut être associé qu'à une seule imprimante ➡️ `@<<?>>ToOne` côté ordinateur
- une imprimante ne connait pas les ordinateurs qui l'utilisent ➡️ Le mapping se fait uniquement dans la classe Ordinateur
- une imprimante peut être partagée par plusieurs ordinateurs ➡️ `@OneToMany` côté imprimante donc `@ManyToOne` côté ordinateur
- la relation est unidirectionnelle ➡️ Mapping que d'un côté

```
@Entity
public class Computer {
```

```
@Entity
public class Printer {
```

```
  // Ma stratégie : le hostname devient la clé principale
  @Id
  private String hostname;

  private Info otherinfo;

  @ManyToOne
  Printer associated_printer;

  public Computer(){}

  public void setHostname(String hostname){
    this.hostname = hostname;
  }

  public String getHostname(){
    return this.hostname;
  }

  public void setOtherinfo(Info otherinfo){
    this.otherinfo = otherinfo;
  }

  public Info getOtherinfo(){
    return this.otherinfo;
  }

  public void setPrinter(Printer printer){
    this.associated_printer = printer;
  }

  public Printer getPrinter(){
    return this.associated_printer;
  }
}
```

```
  // Ma stratégie : le printername devient la clé principale
  @Id
  private String printername;

  private Info otherinfo;

  public Printer(){}

  public void setPrintername(String printername){
    this.printername = printername;
  }

  public String getPrintername(){
    return this.printertname;
  }

  public void setOtherinfo(Info otherinfo){
    this.otherinfo = otherinfo;
  }

  public Info getOtherinfo(){
    return this.otherinfo;
  }
}
```

⚠ Penser au constructeur vide
⚠ Penser aux setters et getters

Question 4

> On suppose maintenant que cette relation est bi-directionnelle :
> – un ordinateur ne peut être associé qu'à une seule imprimante
> – une imprimante connait les ordinateurs qui l'utilisent
>
> Indiquez le changement à apporter à ce qui précède (juste les lignes qui changent, ne ré-écrivez pas tout).

Analyse de l'ennoncé :

- un ordinateur ne peut être associé qu'à une seule imprimante ➡️ Rien ne change
- une imprimante connait les ordinateurs qui l'utilisent ➡️ Relation bidirectionnelle, il faut mapper des 2 côtés

```
@Entity
public class Computer {

  @ManyToOne
  Printer associated_printer;
  // Rien ne change

}
```

La mention *mappedBy* n'est pas obligatoire mais plutôt conseillée. Sans elle, une table de liaison *PRINTER_COMPUTER* va être créée dans la BD. Avec elle, c'est juste une colonne *PRINTER_ID* qui sera créée dans la table *COMPUTER*.

```
@Entity
public class Printer {

  @OneToMany(mappedBy="associated_printer")
  private List<Computer> computers = new Arraylist<>();

  public void setComputers(List<Computer> computers){
    this.computers = computers;
  }

  public List<Computer> getComputers(){
    return this.computers;
  }
}
```

Question 5

> Indiquer rapidement les changements à effectuer (notamment les annotations à utiliser) dans les cas suivants :
> – un ordinateur peut être associé à plusieurs imprimantes
> – un ordinateur ne peut être associé qu'une seule imprimante et une imprimante à un seul ordinateur

Question 5.1

"un ordinateur peut être associé à plusieurs imprimantes" ➡️ Open bar, on passe en `@ManyToMany`

⚠️ Voir la section de cours correspondante : il faut faire un choix entre unidirectionnelle et bidirectionnelle (attention au *mappedBy*)

Question 5.2

"un ordinateur ne peut être associé qu'une seule imprimante et une imprimante à un seul ordinateur" ➡️ Pas du tout open bar, on passe en `@OneToOne`

⚠️ Voir la section de cours correspondante : il faut choisir de spécifier une règle de mapping ou non (*mappedBy*)

Question 6

> On s'appuie sur la version de la question 4. On veut maintenant implanter un session bean qui permet l'ajout d'une imprimante, son association à un ordinateur, et la recherche des ordinateurs associés à une imprimante.
> On veut fournir l'interface suivante :

```
public interface PrinterManager {
  public void AddPrinter(String printername, Info printerinfo);
  public void AssociatePrinter (String printername, String hostname);
  public ArrayList<Computer> getComputersForPrinting(String printername);
}
```

> Donnez l'implantation de ce bean session.

Analyse de l'ennoncé :

- "On veut maintenant implanter un session bean" ➡️ On va créer une façade

```
@Singleton
public class PrinterManagerImpl implements PrinterManager {

  @PersistenceContext
  private EntityManager em;

  public PrinterManagerImpl(){}

  public void AddPrinter(String printername, Info printerinfo){

    // On crée l'objet printer
    Printer printer = new Printer();
    printer.setPrintername(printername);
    printer.setPrinterinfo(printerinfo);

    // On push l'objet dans la base de données
    em.persist(printer);
  }

  public void AssociatePrinter (String printername, String hostname) {

    // On récupère le printer. On peut le faire avec le printername car c'est la clé principale
    Printer printer = em.find(Printer.class, printername);

    // On récupère le computer. Même remarque
    Computer computer = em.find(Computer.class, hostname);

    // On récupère la liste des computers du printer.
    // ATTENTION : Pour que cette opération fonctionne, il faut absolument
    // indiquer fetch=FetchType.EAGER dans la classe Printer au dessus de
    // la liste de computers
    List<Computer> computers = printer.getComputers();

    // On associe
    computers.add(computer);
    printer.setComputers(computers);

    // On push les modifications de printer dans la BD
    em.merge(printer);
  }

  public ArrayList<Computer> getComputersForPrinting(String printername) {

    // On récupère le printer. On peut le faire avec le printername car c'est la clé principale
    Printer printer = em.find(Printer.class, printername);

    // On renvoie la liste des computers du printer.
    // ATTENTION : Pour que cette opération fonctionne, il faut absolument
    // indiquer fetch=FetchType.EAGER dans la classe Printer au dessus de
    // la liste de computers
    return (ArrayList<Computer>) printer.getComputers();
  }
}
```

Question 7

> Dans le bean session précédent, on veut ajouter une méthode qui retourne le nombre d'imprimantes disponibles. Donnez une solution et discutez son efficacité

```
@Singleton
public class PrinterManagerImpl implements PrinterManager {

  public int getNumberOfPrinters(){

    TypedQuery<Printer> req = em.createQuery("select p from Printer p", Printer.class);
    Collection<Printer> liste_printers = req.getResultList();

    return liste_printers.size();
  }
}
```