JavaEE
Développement d'application web

Aix-Marseille Université

Gaël Guibon, Jean-Luc Massat, Omar Boucelma

2018-2019

Sommaire

Chapitre 1

Glossaire

Voici un glossaire qui sert de rappel rapide aux différentes notions requises et celles abordées dans ce TD et dans le cours.

1.1 Requises

- **HTML** : informations des éléments de l'affichage de la page
- **CSS** : feuilles de style pour gérer l'affichage et son rendu visuel
- **XML** : format de représentation balisée des données (<nom>Gael</nom>)
- **SQL** : langage de requête de bases de données relationnelles (SGBD)
- **Interface Java** : représentation simplifiée de ce que devra contenir une classe Java, les méthodes attendues ect.
- **Annotation** : fait d'ajouter un élement avec @ au dessus d'une classe, d'une méthode, ou d'une propriété afin d'y associer des méta-informations (indiquer cette méthode réécrit celle de l'interface @Override, indiquer que c'est une servlet @WebServlet, indiquer qu'il s'agit de la clé primaire @Id, indiquer qu'il s'agit d'une injection inversée globale @Autowired, etc.)

1.2 Abordées

- **Maven** : Outil de compilation du code Java. Permet de centraliser les dépendance et d'automatiser la compilation et le déploiement (Jar, WAR, etc.).
- **JEE** : Java Enterprise Edition
- **Bean** : représentation des données sous forme de classe Java. La classe qui répond à la question "À quoi ressemblent mes données en fait ?"
- **DAO** : Data Access Object : Classe Java dédiée à accéder aux données. La classe qui répond à la demande "Je veux stocker mes données !"
- **Service** : Classe Java dédiée à la logique de l'application, aux traitements des données. La classe qui répond à la question "Que veux-tu faire ?"
- **JDBC** : Outil d'accès à la base de donnée en Java. Utilisé par le DAO pour faire des requêtes SQL.
- **Servlet** : Classe Java qui gère les requêtes et réponses HTTP selon une URL (/home) et une méthode HTTP (GET, POST). La classe qui répond à la question "Que faire si je vais sur telle URL ?"
- **Controller** : Classe Java qui utilise une ou plusieurs Servlets pour bien aiguiller l'utilisateur vers le bon affichage ou le bon Service en fonction de l'URL et de la méthode HTTP.
- **JSP** : Java Server Page : Page HTML enrichie et compilée dans l'ombre en Java. Permet un affichage de l'interface.
- **EL** : Expression Language : Permet un accès simplifié aux paramètres et attributs pour l'affichage dans la JSP
- **JSTL** : Permet une logique d'affichage dans la JSP. Boucler sur une liste (c :forEach), conditionner tel ou tel affichage (c :if), etc.

— **JPA** : Java Persistence API : Permet d'automatiser le lien entre l'application et la base de données.

Chapitre 2

Notions de bases et environnement

2.1 Notions de bases

2.1.1 Java et JavaEE

Vérification du niveau des étudiants en java standard
Topo sur l'intéret du JEE par rapport au java standard + distinction serveur / navigateur
Quelques rappels :

Java	**JavaEE**
Socle de base	Extension de Java
Programmes locaux	Programmes connectés
Entrée par méthode main()	Entrées par chaque servlet
HTML	**JavaEE**
Site web statique	Site web dynamique
Visible	Opaque et sécurisé

2.1.2 Le modèle MVC

Le principe Modèle Vue Controlleur (MVC) est une bonne pratique de programmation.

Modèle	**Vue**	**Controlleur**
Données	Affichage IHM	Actions / logique du code
Accès BDD	Adaptation de l'UI	Lien entre les éléments
Messages/contacts	Visualisation du message/contacts	Boutons "répondre à tous", envoyer emoji, ...

Exemple d'une répartition MVC :

De nombreux frameworks en différents langages sont désormais MVC :
- Java : Spring [1], Struts [2], Tapestry [3]
- Python : Django, Flask, Pyramid
- Javascript : Angular, ReactJS, EmberJS

1. http://spring.io/
2. http://struts.apache.org/
3. http://tapestry.apache.org/

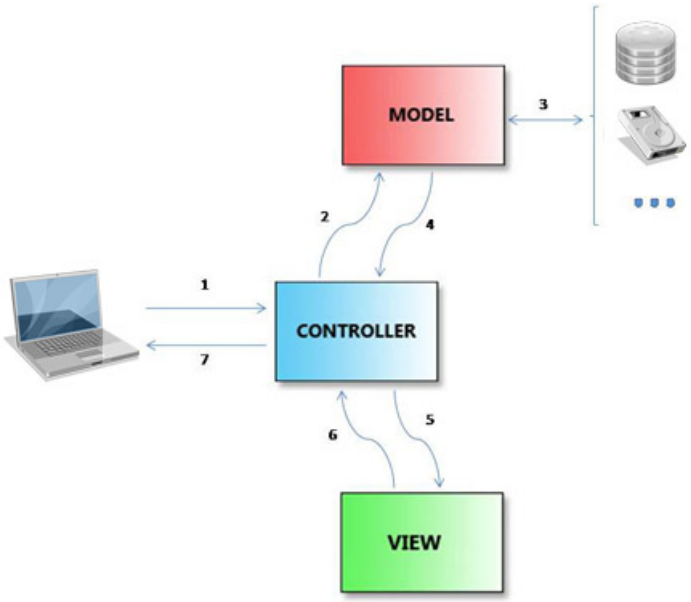


FIGURE 2.1 – Exemple : Messagerie SMS

2.2 Environnement

2.2.1 Installer Java8

Ouvrir un terminal et lancer les commandes suivantes :

1. sudo add-apt-repository ppa :webupd8team/java
2. sudo apt-get update -y
3. sudo apt-get install oracle-java8-installer
4. java -version

La dernière commande devrait vous afficher la version 1.8.x de Java.

2.2.2 Installer Maven

1. cd /opt/
2. wget http ://apache.crihan.fr/dist/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-bin.tar.gz
3. sudo tar -xvzf apache-maven-3.3.9-bin.tar.gz
4. sudo mv apache-maven-3.3.9 maven
5. sudo nano /etc/profile.d/mavenenv.sh
6. Dans ce fichier ajouter les lignes suivantes :

```
export M2_HOME=/opt/maven
export PATH=${M2_HOME}/bin:${PATH}
```

7. sudo chmod +x /etc/profile.d/mavenenv.sh
8. sudo source /etc/profile.d/mavenenv.sh
9. mvn –version

2.2.3 Installer l'IDE : Eclipse Oxygen

1. Télécharger Eclipse : https://www.eclipse.org/downloads/download.php?file=/oomph/epp/oxygen/R/eclipse-inst-linux64.tar.gz
2. Lancer l'installeur
3. Choisir Eclipse for JavaEE
4. Suivre l'installation avec les configurations par défaut

2.3 Comprendre Maven

2.3.1 Qu'est-ce que c'est ? ! !

Un outil d'automatisation :

— Très répandu en entreprise
— Pour la compilation
— Pour le déploiement
— Facilite le partage de code

Format d'usage :

```
usage: mvn [options] [<goal(s)>] [<phase(s)>]
```

2.3.2 Les éléments de base

```
<project>
  <modelVersion/> VERSION DE MON PROGRAMME
  <groupId/> ID DU GROUPE DE MON PROGRAMME : cnrs.amu.lsis ou com.usa.google ou autre
  <artifactId/>  ID DE MON PROGRAMME : monprojet ou minecraft ou autre
  <packaging/> TYPE DE PAQUET EN SORTIE : jar ou war ou autre
  <version/>  VERSION DE MON PROGRAMME : 0.0.9-beta ou 2.1.56
  <name/> NOM EN INTERNE
  <url/> URL DU PROGRAMME : par defaut mettre : http://maven.apache.org
  <dependencies/> DEPENDANCES DU PROGRAMME : jsp, guava, et autres librairies externes
  <build>  EXECUTION LORS DE LA COMPILATION DU PROGRAMME
    <plugins/> PLUGINS DIVERS
  </build>
  </project>
```

2.3.3 Architecture d'un projet maven

Générer un projet maven basique (java SE) :

```
mvn archetype:generate -DgroupId=test.project -DartifactId=superTest -DarchetypeArtifactId=maven-archetype-quickstart
      -DinteractiveMode=false
```

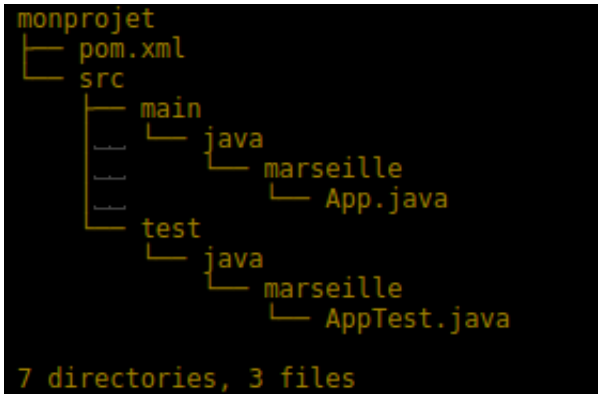


FIGURE 2.2 – Arborescence de base de maven

L'explication officielle de l'arborescence est présente ici : http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

2.3.4 Le fichier POM.xml

C'est le fichier central ! Celui qui permet de guider maven dans la compilation.
Un guide officiel du POM est présent à cette adresse : http://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Les dépendances

```
<dependencies>
  <dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>${tomcat.version}</version>
  </dependency>
  .
  .
```

```
    .
</dependencies>
```

Les plugins

```
<plugins>
    <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
            <source>1.8</source>
            <target>1.8</target>
        </configuration>
    </plugin>
    .
    .
    .
</plugins>
```

2.3.5 Accéder aux dépendances et plugins

Aller chercher sur maven central : https://search.maven.org/

2.4 Hello World

1. Cloner le repository : git clone https ://github.com/gguibon/CoursesJavaEE.git
2. Aller dans le projet de base : cd basic_hello_world
3. Terminal : mvn package
4. Terminal : java -jar target/basicHelloWorld-jar-with-dependencies.jar OU double clic sur le jar
5. Ouvrir un navigateur à l'adresse localhost :8080

Astuce : La commande htop vous indique les processus en cours (sudo apt install htop)

2.5 Sources utiles pour compléter

Documentation officielle de maven : http://maven.apache.org/guides/index.html
Tutoriel officiel : https://docs.oracle.com/javaee/7/tutorial/
Documentations et API officielle : http://docs.oracle.com/javaee/7/index.html
Tutoriel Site du Zero : https://openclassrooms.com/courses/creez-votre-application-web-avec-java-ee
Mémento et exemples d'annotations Servlet 3.0 : http://www.codejava.net/java-ee/servlet/servlet-annotations-

Chapitre 3

Le modèle - beans

3.1 Qu'est-ce que c'est ?

Une manière de stocker les données. Un est :
- Réutilisable ! (objet java)
- Persistant ! (serialization)
- Paramétrable ! (propriétés)

Il respecte donc plusieurs règles :
- C'est une classe publique.
- Implémente Serializable pour pouvoir être sauvegardé.
- AUCUN champ public ! (getter et setters)
- Possède un constructeur par défaut public, sans paramètres ?

3.2 Exo : Agenda

3.2.1 Créer le bean Rendezvous

Créer un bean (un objet java) Rendezvous.java.

1. Ajouter les propriétés "duree"(int), "personnes"(liste de noms), "lieu"(string) et "type"(string)
2. Ajouter des getter et setter pour chaque propriété
3. Ajouter une méthode qui retourne le nombre de personnes

3.2.2 Créer le service RendezvousService

Un service est une classe qui va ouvoir être instancier et gérer un ou plusieurs beans, ainsi que leurs relations.

1. Créer un service RendezvousService.java

```
@Service //annotation qui indique àspring que c'est un service
public class RendezvousService { }
```

2. Ajouter une liste de rendezvous
3. Ajouter un constructeur qui initialise des rendez-vous par défauts
4. Ajouter une méthode d'ajout de rendez-vous
5. Ajouter une méthode qui récupère le nombre de rendez-vous
6. Ajouter une méthode qui récupère les rendez-vous en fonction de plusieurs types de rendez-vous différents

3.2.3 Utiliser le bean

1. Afficher un tableau auto généré pour lequel chaque ligne est un rendez-vous, et chaque colonne une propriété (dans terminal/log). De cette manière :

```
  public void visualizeRdvs() {
    List<String> lines = new ArrayList<String>();
    for (Rendezvous rdv : rendezvousList) lines.add(rdv.toString());
    log.info(String.format("VISUALIZATION\n===== BDD CONTENT ROWS =====\n\%s",
          Joiner.on("\n=====\n").join(lines)) );
}
```

2. Tester l'ajout de chaque fonctionnalité en visualisant au fur et à mesure (manuellement ou non)

Pour rappel, les logs se font ainsi :

```
private static final Logger log = LoggerFactory.getLogger(Application.class);

log.info("Le log fonctionne =) !")
```

Chapitre 4

Le stockage de données - DAO et JDBC Template

4.1 Exo : Stockage de l'agenda

4.1.1 Créer l'interface RendezvousDAO.java

1. Spécifier une méthode de sauvegarde d'un nouveau rendez-vous (ajout) comme ceci :

```
public interface RendezvousDAO {
  void add(Rendezvous rdv);
}
```

2. De la même façon, spécifier une méthode de mise à jour d'un rendez-vous
3. Spécifier une méthode de suppression d'un rendez-vous
4. Spécifier une méthode de requête d'un rendez-vous en fonction de son type
5. Spécifier une méthode de requête des rendez-vous (qui retourne donc une liste de rendez-vous)

```
  List<Rendezvous> findByType( String type );
```

6. Spécifier une méthode de requête des rendez-vous en fonction de leurs types

4.1.2 Créer l'implémentation RendezvousDAOImpl.java

Une classe d'implémentation d'un DAO ressemble à ceci :

```
@Transactional // évite d'avoir àgérer les transactions
@Repository // indique àspring qu'il s'agit d'un accès BDD
public class RendezvousDAOImpl implements RendezvousDAO {
    @Autowired // injection de dépendance pour un jdbctemplate global récupérant ses paramètres dans
         src/main/resources/application.properties
  JdbcTemplate jt;
...
}
```

1. Créer une méthode de sauvegarde d'un nouveau rendez-vous (ajout) avec requête SQL de cette façon :

```
   @Override
  public void add(Rendezvous rdv) {
     String sql = "INSERT INTO rendezvous(duree,lieu,type,personnes) values(?,?,?,?)";
     jt.update(sql, rdv.getDuree(), rdv.getLieu(), rdv.getType(), rdv.getPersonnes());
  }
```

2. De la même façon, créer une méthode de mise à jour d'un rendez-vous avec requête SQL
3. Créer une méthode de suppression d'un rendez-vous avec requête SQL

4. Créer une méthode de requête d'un rendez-vous en fonction de son id unique comme ceci :

```
  @Override
  public Rendezvous findById(Integer id) {
    String sql = "SELECT * FROM rendezvous WHERE id = ?";
    // mapper de lignes des éléments récupérés, automatique liés au bean Rendezvous grâce à
        BeanPropertyRowMapper<T>(T.class)
    RowMapper<Rendezvous> rowMapper = new BeanPropertyRowMapper<Rendezvous>(Rendezvous.class);
    // jdbcTemplate (jt) possède une méthode pour récupérer un seul objet : queryForObject.
    return jt.queryForObject(sql, rowMapper, id);
  }
```

5. Créer une méthode de requête des rendez-vous avec requête SQL avec row Mapper personnalisé :

```
class RDVRowMapper implements RowMapper<Rendezvous>
{
    @Override
    public Rendezvous mapRow(ResultSet rs, int rowNum) throws SQLException {
        Rendezvous rdv = new Rendezvous();
        rdv.setId(rs.getInt("id"));
        rdv.setDuree(rs.getInt("duree"));
        rdv.setLieu(rs.getString("lieu"));
        rdv.setType(rs.getString("type"));
        rdv.setPersonnes(rs.getString("personnes"));
        return rdv;
    }
}
```

6. Créer une méthode de requête des rendez-vous en fonction de leurs types avec requête SQL avec le rowMapper de votre choix.

4.1.3 Modifier le service Rendez-vous

— Adapter RendezvousService pour qu'il utilise le DAO. Par exemple :

```
  public Rendezvous getRDVByID(int id) {
    return rdvdao.findById(id);
  }
```

— Tester les fonctionnalités et visualiser les résultats et changements par terminal/logs

4.1.4 *Optionnel* Ajouter d'autres données

Un agenda a plus que de simples rendez-vous :
— Ajouter des rappels (anniversaires, jours fériés, etc.)
— Ajouter des dates butoir ayant une méthode pour une booléenne propriété *done*

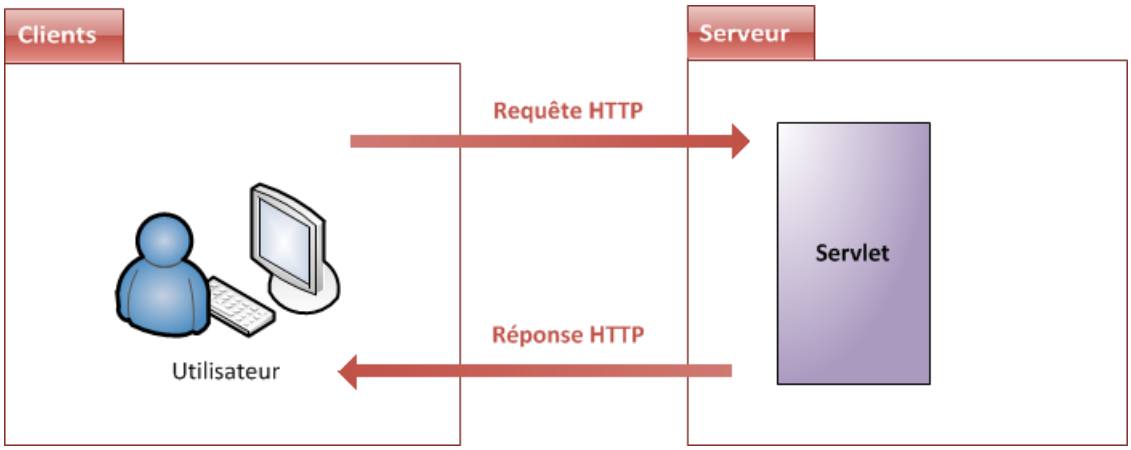
Chapitre 5

Les contrôleurs - servlets

5.1 Méthodes HTTP

GET	POST	HEAD
Pour ressources web	Pour envoi de fichiers	Pour méta-informations web
Répétée	Ponctuelle	Répétée
Taille limitée	Taille non limitée	Taille limitée
doGet()	doPost()	doHead()

5.2 Qu'est-ce qu'une servlet ?



Une servlet est :
- une classe java
- le lien central de l'application : requêtes, réponses, etc.
- un accès aux méthodes HTTP

5.3 Créer une servlet

Ici nous allons créer une servlet de la manière la plus classique qu'il soit. Nous verrons après cela comment avoir une servlet simplifiée en Spring.
Pour les servlets, veuillez utiliser le squelette suivant : https://github.com/gguibon/CoursesJavaEE/tree/master/20182019/servlet_usage

5.3.1 Créer la servlet vide

Imports nécessaires :

```
import javax.servlet.http.HttpServlet;
```

1. Créer une classe src/mainjava/servlet/HelloServlet.java qui étend la classe HttpServlet
2. Sérialiser la classe

5.3.2 Déclarer la servlet à l'aide d'annotation

Imports nécessaires :

```
import javax.servlet.annotation.WebServlet;
```

1. Utiliser l'annotation @WebServlet
2. Par argument d'annotation, nommer la servlet "MaServlet"
3. Par argument d'annotation, lier la servlet aux urls "/hello" et "/bonjour"

Annotations possibles : https://docs.oracle.com/javaee/7/api/ section javax.servlet.annotation
- **@WebServlet** : pour créer une servlet et la déclarer
- **@WebFilter** : pour filtrer des catégories ("/admin/login"), des types de fichiers ("doc, txt, jpg, etc."),
- **@WebInitParam** : pour associer @WebServlet et @WebFilter
- **@WebListener** : pour créer un listener (exemple : listerner sur l'application pour savoir elle s'est arrêtée)
- **@HandlesTypes** : pour déclarer les classes
- **@MultipartConfig** : pour l'upload de fichier(s)
- **@HttpConstraint** : pour les contraintes de sécurité (SSL/TLS par exemple)
- **@HttpMethodConstraint** : pour les contraintes de sécurité sur les méthodes GET POST HEAD
- **@ServletSecurity** : pour les contraintes de sécurité sur toute la servlet

5.3.3 Remplir la servlet

Imports nécessaires :

```
import javax.servlet.http.HttpServletRequest; // pour req
import javax.servlet.http.HttpServletResponse; // pour resp
import javax.servlet.ServletOutputStream; // pour acceder au stream de sortie de la servlet
```

1. Créer une méthode doGet() ayant deux arguments : "req" de type HttpServletRequest et "resp" de type HttpServletResponse
2. Afficher la réponse (resp.getOutputStream() ;) dans le stream de sortie de la servlet (ServletOutputStream)
3. Ecrire "Bonjour le monde" dans le stream de sortie de la servlet
4. Fermer le stream
5. Dans un navigateur aller à "localhost :8080/bonjour"

5.4 Lier une servlet à une jsp (MVC)

1. Créer une nouvelle jsp basique nommée seconde.jsp dans src/main/resources/webapp
2. Ecrire un code html basique dans cette jsp

```
  <html>
<body>
   <h2>This is a superb text</h2>
   <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
```

```
    eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
    minim veniam, quis nostrud exercitation ullamco laboris nisi ut
    aliquip ex ea commodo consequat. Duis aute irure dolor in
    reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
    pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
    culpa qui officia deserunt mollit anim id est laborum.</p>
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad
    minim veniam, quis nostrud exercitation ullamco laboris nisi ut
    aliquip ex ea commodo consequat. Duis aute irure dolor in
    reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
    pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
    culpa qui officia deserunt mollit anim id est laborum.</p>
</body>
</html>
```

3. Créer une nouvelle servlet nommée "servletMVC" et comme url "/lorem"
4. Au lieu d'écrire directement dans la sortie de la servlet, transférer la requête et la réponse à une jsp existante

```
this.getServletContext().getRequestDispatcher( "/[NOM_DE_LA_JSP].jsp" ).forward( [REQUETE], [REPONSE] );
```

5. Aller à localhost :8080/lorem et voyez votre résultat en version MVC

5.5 Rediriger en fonction des paramètres de l'URL

1. Créer une nouvelle servlet nommée "ServletCondition" avec url "/condition"
2. Lire le paramètre "prenom" donné en requête url "/condition ?prenom=John"

```
// l'ensemble des parametres est une Map
Map<String, String[]> parameters = req.getParameterMap();

// il est possible d'acceder directement a un parametre
String paramPrenom = req.getParameter("MON_PARAMETRE");
```

3. Si le paramètre "prenom" est présent et non vide, transférer vers "seconde.jsp". Si non, transférer vers "index.jsp"

Cette solution ne respecte pas le modèle MVC !

5.6 Créer une servlet en Spring (AKA controller)

1. **Reprendre le projet utilisé pour les beans, services et DAO** (le projet Spring fait à partir du "squelette"). Y ajouter les dépendances suivantes :

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
   <optional>true</optional>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!-- JSTL for JSP -->
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>jstl</artifactId>
</dependency>

<!-- Need this to compile JSP -->
<dependency>
   <groupId>org.apache.tomcat.embed</groupId>
   <artifactId>tomcat-embed-jasper</artifactId>
   <scope>provided</scope>
</dependency>
<!-- Need this to compile JSP, tomcat-embed-jasper version is not working,
   no idea why -->
<dependency>
```

```
  <groupId>org.eclipse.jdt.core.compiler</groupId>
  <artifactId>ecj</artifactId>
  <version>4.6.1</version>
  <scope>provided</scope>
</dependency>
```

2. Créer un dossier : src/main/webapp/WEB-INF/jsp dans lequel nous mettrons nos jsp.
3. Dans application.properties ajouter les lignes suivantes :

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

Maintenant votre projet Spring pourra lire les JSP, JSTL et EL et y faire référence dans les controleurs ! En plus de comprendre l'annotation @Controller. Il trouvera également les vues en utilisant les prefix et suffix donnés.
L'approche est beaucoup plus simple, une classe avec l'annotation correspondante devient une servlet :

1. Créer une nouvelle classe "MonControleur" qui sera votre controleur

```
@Controller // annotation qui indique qu'il s'agit d'un controleur (une servlet est crée dans l'ombre)
public class MonControleur {
   @Autowired
  ServletContext context; //injection du servlet context si besoin est pour obtenir la session en cours

  @Autowired
  MonService ms; // injection d'un service pour pouvoir l'appeler et ainsi faire agir la logique de l'app. Par
       exemple avec ms.getAllItems()
...
}
```

2. Dans cette classe, créer une méthode qui possède un mapping par annotation. Cette méthode retourne le nom de la page .jsp à afficher (plus simple que getRequestDispatcher() !)

```
@RequestMapping(value = "/", method = RequestMethod.GET) // indique que cette méthode du contrôleur (qui est une
     servlet cachée par Spring) gère la page racine "/" via la méthode GET. Cela aurait pu être "/bonjour" pour
     gérer localhost/bonjour
   public String add(){
   ...
   return "nom_de_ma_jsp_sans_extension" // ou par exemple return "index" (pour index.jsp)
   }
```

3. Tester que votre appli vous renvoie bien là où vous voulez, selon l'url fournie et la méthode fournie.

Chapitre 6

La vue - EL et JSTL

Vous pouvez faire ce qui suit à partir du projet JEE manuel (servlet_usage) ou en reprenant votre porjet Spring avec les beans.
Nous utilisons ici principalement EL et JSTL combinés dans les jsp. Le langage des jsp est donc évité pour une raison simple : il outrepasse la séparation entre données, vues et modèles en permettant de mettre du java directement dans la jsp. Il n'y a aucune raison, jamais, jamais de faire cela. EL et JSTL sont donc une bonne pratique, plus simple, plus moderne (nombreux sont les framework de vue à s'en être inspiré : AngularJS, Moustache, VueJS, etc.).

6.1 EL : Expression Language

6.1.1 Qu'est-ce que l'EL ?

C'est un langage permettant de connecter facilement le contenu java et l'affichage HTML via une jsp.
Les EL permettent notamment de :
- Conditionner l'affichage
- Faire des calculs simples
- Accéder aux données / méthodes / paramètres de requête

6.1.2 Accéder aux paramètres et attributs de la requête

1. Créer une méthode nommée "ServletEL" avec url "/el" qui transfère vers "el.jsp" - ou par Spring, créer une méthode d'un contrôleur renvoyant gérant "/el" et retournant "el"
2. Créer une jsp nommée "el.jsp" et y placer le HTML suivant

```
  <!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>Test des expressions EL</title>
</head>
<body>
   <p>Je m'appelle ${ A REMPLIR }</p>
</body>
</html>
```

3. Afficher la valeur du paramètre "prenom". ("Je m'appelle XXX")
4. En fonction de la valeur du paramètre "pays" afficher la nationalité à l'aide d'un attribut ("Je m'appelle XXX et je suis de nationalité XXX").

```
// donner un attribut àla requête (servlet clasique)
req.setAttribute("test", "youhou"); // pour une servlet classique

// donner un attribut àla requête (contrôleur Spring)
@RequestMapping(value = "/el", method = RequestMethod.GET)
public String add(HttpServletRequest request, Map<String, Object> model){
model.put("test", "youhou"); // model est donc une map qui représente la requête et ses informations
...
```

```
}

// jsp : acceder àl'attribut
${ test } // cela affichera "youhou"
```

5. Afficher plusieurs prénoms à partir de l'unique paramètre "prénoms". ("Mes prénoms sont XXX et XXX")

```
// donner plusieurs valeurs a un seul parametre
/el?prenoms=XXX&prenoms=XXX

// acceder au parametre àvaleur multiple
${ paramValues.XXX }

// acceder a une de ces valeurs
${ paramValues.XXX[index] }
```

6.2 JSTL

6.2.1 Qu'est-ce que JSTL ?

JSTL est une librairie regroupant plusieurs fonctions à utiliser dans la Vue. Permet de mieux respecter le modèle MVC !

6.2.2 Les principales commandes

Les balises core

Super mémo complet ici –> https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm

Quelques exemples :

```
<%-- Affiche Hello World -->
<c:out value="Hello World">

<%-- Declare une variable nommee "maVar" avec valeur 250 et une portee sur la session -->
<c:set var = "maVar" scope = "session" value = "${50+200}"/>

<%-- Affiche le contenu de la variable "maVar" (i.e. 250) -->
<c:out value="${ maVar }">

<%-- Affiche la balise <p> si "maVar" est superieure à50 -->
<c:if test = "${ maVar > 50 }">
   <p>La variable "maVar" est supérieure à50 ! Wahou!/><p>
</c:if>

<%-- Boucle for sur un indice -->
<c:forEach var = "i" begin = "1" end = "5">
   Item <c:out value = "${i}"/><p>
</c:forEach>

<%-- Iteration sur les elements d'une liste -->
<c:forEach items="${ maListe }" var="element">
     <c:out value="${ element }" />
</c:forEach>
```

Les fonctions JSTL

- fn :replace()
- fn :length()
- fn :join()
- etc.

6.2.3 Configurer JSTL

— Ajouter la librairie jstl en dépendance du projet maven (pom.xml) : https://mvnrepository.com/artifact/jstl/jstl/1.2
— Créer une nouvelle jsp nommée jstl.jsp et y ajouter tout en haut la ligne suivante :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

6.2.4 Un affichage plus intelligent

1. Ajouter les lignes suivantes à jstl.jsp :

```
<html>
<head>
<meta charset="utf-8" />
<title>Test de JSTL</title>
</head>
<body>
   <c:out value="<p>Bonjour inconnu(e)</p>" />
</body>
</html>
```

2. Créer une servlet simple nommée "ServletJSTL" avec url "/jstl", ou par Spring un contrôleur avec RequestMapping et return adéquats.
3. Afficher "Bonjour inconnu(e)" s'il n'y a pas de paramètre "prenom" donné.
4. Afficher le drapeau du pays donné en paramètre "pays"

C'est bien plus propre et respecte mieux le MVC !

Chapitre 7

Une application de vente

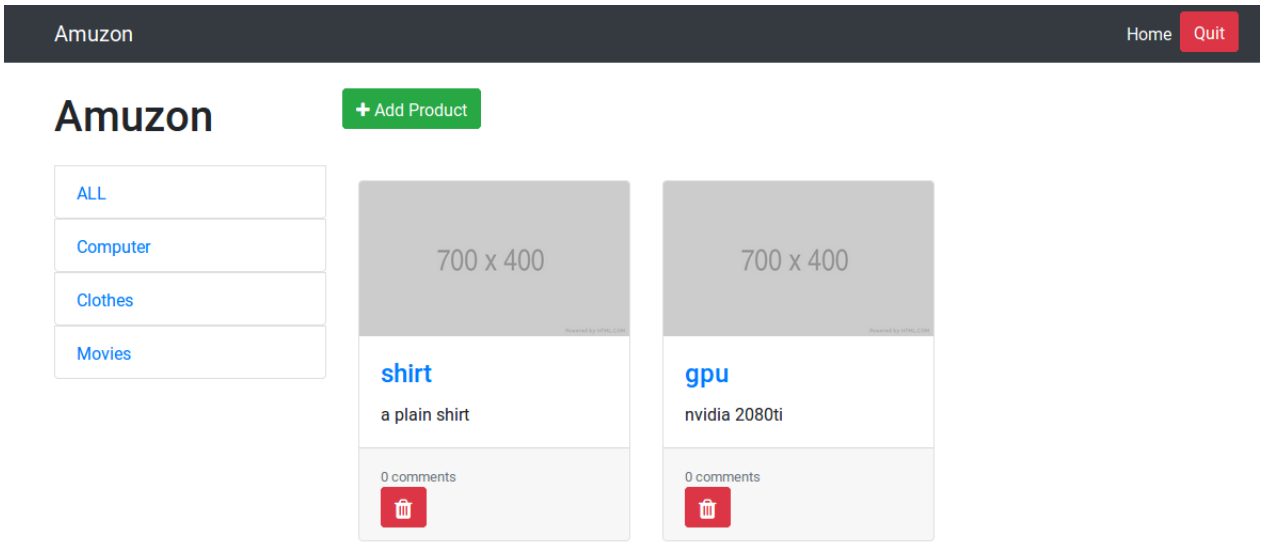


FIGURE 7.1 – Interface de l'application de vente

Vous connaissez :

1. La représentation des données (Bean)
2. Ses traitements (Service)
3. Sa persistance (DAO, JDBC, BDD, SQL)
4. L'aiguillage des pages (Contrôleurs, Servlets)
5. Et enfin l'affiche final (JSP, EL, JSTL)

Vous avez donc désormais le nécessaire pour faire une application ! Suivez donc les étapes.

1. Partez du squelette spring avec toutes les dépendances et les vues préparées mis à disposition à cette adresse (https://github.com/gguibon/CoursesJavaEE/tree/master/20182019/td2)
2. Créez un bean Product qui possède : category, title, description
3. Créez l'implémentation de l'interface suivante :

```
public interface ProductDAO {
  void add(Product product);
  void update(Product product);
  void delete(Product product);
  List<Product> findAll();
```

```
    Product findById( Integer id );
    List<Product> findByCategory( String category );
  }
```

4. Créez l'implémentation du service suivant :

```
  public interface ProductService {
    void addProduct(Product product);
    Product getProduct(int id);
    void removeProduct(int id);
    List<Product> getProducts();
    List<Product> getCategoryProducts(String category);
  }
```

5. Créer un contrôleur pour la racine qui renvoie vers la page principale en donnant la liste des produits en attribut comme ceci :

```
@Controller
public class IndexController {

  @Autowired
  private HttpSession httpSession;

  @Autowired
  ServletContext context;

  @Autowired
  ProductService ps;

   @GetMapping("/") // raccourci pour @RequestMapping( value = "/", method = RequestMethod.GET)
   public String index(Map<String, Object> model) {
     String sessionUser= (String) httpSession.getAttribute("user");
     model.put("products", ps.getProducts() );

     System.out.println("session user = " + sessionUser);
     return "homepage";
   }
}
```

6. Dans ProductController, créer une méthode ("/add") qui ajoute un produit et renvoie en attribut la liste des produits à "homepage"
7. Dans ProductController, créer une méthode ("/remove") qui supprime un produit et renvoie la liste des produits à "homepage"
8. Dans ProductController, créer une méthode ("/category") qui filtre les produits par catégorie et renvoie cette liste à "homepage"
9. Dans la de homepage.jsp, afficher tous les produits avec leurs titre, description
10. Dans la de homepage.jsp, pour chaque produit ajoute un bouton de suppression tel que :

```
  <form action="${pageContext.request.contextPath}/remove" method="POST">
    <input type="hidden" value="${product.id}" name="productId" />
    <button name="buttonRemove" type="submit" class="btn btn-danger btn-xs" >
      <i class="fa fa-trash fa-lg" aria-hidden="true"></i>
    </button>
  </form>
```

11. Dans le modal , adapter le formulaire pour rendre l'ajout de produit effectif
12. Tester les fonctionnalités, l'application est prête !

Chapitre 8

Les tests : Une application de vente testée et approuvée

À la fin du chapitre précédent vous avez testé manuellement votre application : vous avez ouvert un navigateur, cliqué ici et là, regardé, etc. Au lieu de faire tout ça tout le temps, ajoutez-y des tests unitaires et d'intégration !
À partir de votre projet AMUzone bien rempli et fonctionnel, vérifiez que vous avez cette dépendance :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Pour le reste, suivez les étapes suivantes :

1. créez un dossier et sous-dossiers : src/test/java et src/test/resources de sorte d'avoir ceci :

   ```
   src
    main
       java
       resources
       webapp
    test
       java
       resources
   ```

2. créez un package fr.amu dans la partie test du programme (désormais coupé en deux : main et test). Créez le test de la classe Application nommée src/test/java/fr/amu/ApplicationTests.java et qui contient ceci :

   ```
   @RunWith(SpringRunner.class)
   @SpringBootTest
   public class ApplicationTests {

      @Test
      public void contextLoads() {
      }

   }
   ```

3. Dans la racine du projet maven (là où il y a le pom.xml), avec un terminal tapez ceci : "mvn test". Regardez le résultat tout **vert**.

8.1 Tester un contrôleur

Pour tester un contrôleur nous avons deux méthodes. La première (utilisée pour IndexController) est basique et limitée mais rapide. La seconde, créer une sorte de bot plus avancé.

1. Nous allons maintenant tester l'IndexController en recréant les appels urls et autres. Créez la classe IndexControllerTests dans le package fr.amu.controllers (src/test/java/fr/amu/controllers/IndexControllerTests.java). Y mettre ceci :

```
   import static org.assertj.core.api.Assertions.assertThat; // Notez l'import en static !! Plus simple

   ...

   @RunWith(SpringRunner.class)
   @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT) // web environnement choisi un port au hasard
        parmi ceux disponibles
   public class IndexControllerTests {
   ...
   }
```

2. Dans la classe, ajouter les injections suivantes :

```
   @LocalServerPort // prend le port choisi au hasard
   private int port;

  @Autowired
   private IndexController controller;

  @Autowired
   private TestRestTemplate restTemplate; //permet de faire une requête simple, un accès àune url
```

3. Tester le bon lancement du contrôleur :

```
   @Test
   public void contexLoads() throws Exception {
      assertThat(controller).isNotNull(); // on vérifie que le controller est bien lancé
   }
```

4. Faire un "mvn test" et voir si nos deux tests passent bien.
5. Ajouter un test de la méthode index() qui vérifie que celle-ci retourne bien le nom de la vue.

```
   assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/",
              String.class)).contains("homepage"); // cette approche est limitée et n'est pas.
```

6. Tester avec mvn test
7. Cette approche est limitée. Voyons comment vraiment faire un "bot" de test qui va se balader et vérifier un contrôleur. Faisons le en testant ProductController à l'aide de la classe ProductControllerTests. Créez la avec le même principe , avec le MockMvc en plus qui permet plein d'accès à l'url méthodes, paramètres et autres :

```
   @RunWith(SpringRunner.class)
@WebMvcTest(ProductController.class)
public class ProductControllerTests {

  @Autowired
  private ProductController controller;

  @Autowired
  MockMvc mvc;

   ...
}
```

8. Attention aux imports static tels que :

```
   import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;
import static org.springframework.test.web.servlet.setup.MockMvcBuilders.standaloneSetup;
```

9. Ajoutez y un mock bean, c'est à dire une instance factice d'une classe java qui va permettre d'éviter que la méthode testée signale un manque.

```
   @MockBean
  ProductService ps;
```

10. Avant de tester chaque méthode, et donc chaque servlet, nous devons indiquer le contexte, soit dire là où se trouvent les jsp comme donné dans Application.properties :

```
  @Before
public void setup() {

   InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
   viewResolver.setPrefix("/WEB-INF/jsp");
   viewResolver.setSuffix(".jsp");

   mvc = standaloneSetup(controller).setViewResolvers(viewResolver).build();
}

@Test
public void contexLoads() throws Exception {
   assertThat(controller).isNotNull(); // on vérifie que le controller est bien lancé
}
```

11. Ajoutons le test d'une méthode qui retourne une vue, par exemple :

```
  @Test
public void add() throws Exception {
   mvc.perform(post("/add").contentType(MediaType.APPLICATION_FORM_URLENCODED_VALUE).param("title", "value")
         .param("NOM_DU_PARAM", "value"))
   .andExpect(view().name("homepage"));
}
```

12. En suivant ce principe, appliquer les méthodes de tests pour add(), remove() et category(). Attention aux paramètres, à l'url, ainsi qu'au nom de la vue (jsp) retournée. Cette méthode est bien meilleure car elle simule réellement un comportement dans notre appli web.
13. faire mvn test

8.2 Tester un DAO

Voyons maintenant comment tester un DAO en partant de son implémentation.

1. Créer la classe de test ProductDAOTests.java
2. Initialiser cette classe de cette manière :

```
  @RunWith(SpringRunner.class)
@SpringBootTest
public class ProductDAOTests {

  // la base de données est auto remplie grâce àsrc/main/java/resources/data.sql
  @Autowired
  ProductDAO prdao;
```

3. Pour pouvoir tester nous avons besoin de l'ID générer, changer donc l'interface et l'implémentation du DAO comme suit :

```
  Integer add(Product product); // interface

  ...
  // dan l'implémentation
  @Override
public Integer add(Product product) {
   String sql = "INSERT INTO products(category,productTitle,img,description,date) values(?,?,?,?,?)";
   GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
   // nom d ela column générée
   String id_column = "ID";
   // the update method takes an implementation of PreparedStatementCreator which could be a lambda, la
        méthode update a besoin d'un preparedSatementCreator pour modifier sa valeur de sortie
   // 'con' est la connection JDBC cachée par JDBCTemplate
   jt.update(con -> {
       PreparedStatement ps = con.prepareStatement(sql,
       // comme avant on suit l'ordre des ? dans la requete SQL
       new String[]{id_column}); // cet array de String correspond aux clés utilisées pour récupérer les valeur
       //(ordre important, ici en premier car ID est en premier dans CREATE TABLE)
       ps.setString(1, product.getCategory());
       ps.setString(2, product.getProductTitle());
       ps.setString(3, product.getImg());
       ps.setString(4, product.getDescription());
```

```
        ps.setString(5, product.getDate());
        return ps;
      }
      , keyHolder); // ceci va capturer les valeurs de sortie voulues dans une hashmap

     // after the update executed we can now get the value of the generated ID
    // une fois l'update faite on récupère les valeurs grâce àla hashmap de .getKeys().
    Integer id = (Integer) keyHolder.getKeys().get(id_column);
     return id.intValue();
  }
```

4. Ajouter une méthode pour tester la méthode add() comme suit :

```
   @Test // dire que c'est un test (annotation classique de JUnit)
   @Transactional // pour gérer les transactions
   @Rollback(true) // pour remettre la BDD dans son état initial
   public void add() {
     Integer generatedId = prdao.add(product);
     List<Product> products = prdao.findAll();
     Assert.assertEquals(generatedId, Integer.valueOf(products.get(products.size()-1).getId() ));
   }
```

5. Faire des méthodes de test pour les autres méthodes du DAO : findAll, delete, findById, findByCategory. Ne pas oublier les annotations rollback transactional et test.

8.3 Tester un Service

Le test d'un service est simple et très proche d'un test java en JUnit normal.

1. Créer la classe ProductServiceTests comme ceci :

```
   @RunWith(SpringRunner.class)
@SpringBootTest
public class ProductServiceTests {

  @Autowired
  ProductService ps;

  @Autowired
  ProductDAO prdao;

  ...

  }
```

2. Tester les fonctions du service. Voici un exemple :

```
   @Test
  public void getProduct() {
    ps.addProduct(product);
    Product product = ps.getProduct(0);
    Assert.assertTrue(ps.getProduct(0) instanceof Product);
    Assert.assertTrue(product.getId() == 0);
  }
```

3. Vérifier que tous les tests fonctionnent : mvn test

Ça y est, votre application est fonctionnelle et testée !

Chapitre 9

Automatiser la persistance avec JPA

Il est conseilé de faire une sauvegarde de votre AMUzone avec DAO. Faites un fichier zip contenant le src et le pom.xml puis ajoutez-y le psuffixe "_dao_tests" par exemple.

Nous allons aborder ici l'ORM (Object Relational Mapping), soit le fait de lier la base de données avec nos objets. En java, JPA est la spécification standard pour l'ORM.

Jusqu'à présent nous mettions en place un DAO qui s'occupait à lui seul des requêtes SQL et du lien avec la base de données. Ceci fonctionne très bien mais en cas de base de données complexe ou si l'on souhaite changer l'organisation des données alors il est nécessaire de réécrire la requête SQL de création de table ainsi que toutes les requêtes SQL qui l'utilisaient. La spécification Java Persistence API (JPA) permet d'éviter cela en automatisant ces processus.

Comme dit précédemment JPA est une spécification, c'est-à-dire un ensemble de règles, une sorte de guide à suivre pour automatiser la gestion de la base de données. Cet ensemble de règle amène plusieurs annotations et interfaces telles que @Entity, @Id, @ManyToOne, etc.

L'objectif ici est, **en utilisant le même projet d'AMUzone avec tests**, à transformer ce projet pour qu'il accepte les JPA afin de remplacer nos DAO.

Remarque : Les DAO ne deviendront pas nécessairement obsolètes, mais surtout conservés pour les actions complexes et non simplement ajouter, supprimer, modifier, etc.

9.1 Configurer le projet

Nous souhaitons étudier ici JPA uniquement, mais comme dit plus tôt c'est une spécification, il n'est donc possible de l'utiliser qu'à l'aide d'une implémentation existante. Spring Boot utilise Hibernate par défaut car c'est une implémentation populaire qui y ajoute des fonctionnalités mais, pour rester uniquement sur JPA nous allons utiliser l'implémentation Eclipse Link, qui est l'implémentation standard de référence (Hibernate sera vu dans la seconde partie du cours).

Il est nécessaire d'ajouter les imports suivants :

```
<!-- pour les JPA on enlève Hibernate de la configuration classique (car ici on veut faire que du jpa) -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
   <exclusions> <!-- exclusion signifie qu'on enlève hibernate de l'ensemble de dépendance issues de spring boot starter
        data jpa -->
      <exclusion>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-entitymanager</artifactId>
      </exclusion>
      <exclusion>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-core</artifactId>
```

```
      </exclusion>
   </exclusions>
</dependency>
<!-- à la place d'Hibernate nous utilisons l'implémentation JPA de référence : eclipse link -->
<dependency>
   <groupId>org.eclipse.persistence</groupId>
   <artifactId>org.eclipse.persistence.jpa</artifactId>
   <version>2.7.3</version> <!-- dernière version àpartir de maven repository -->
</dependency>
```

Mais ce n'est pas tout, il nous est nécessaire d'ajouter une classe java de configuration [1]. Ce fichier java est à mettre de préférence dans un package "config", par exemple "fr.amu.config.JpaConfiguration.java" contenant ceci :

```
/**
 * JPA est une spécification, nous avons besoin d'une implémentation du PersistenceUnit.
 * Ici nous choisissons celle de eclipse link : import org.eclipse.persistence.config.PersistenceUnitProperties;
 * @author gael
 *
 */
@Configuration // annotation indiquant àSpring une configuration du contexte général. On utilise les outils d'auto
     configuration de spring boot JpaBaseConfiguration etc.
public class JpaConfiguration extends JpaBaseConfiguration {

    protected JpaConfiguration(DataSource dataSource, JpaProperties properties, ObjectProvider<JtaTransactionManager>
         jtaTransactionManager,
        ObjectProvider<TransactionManagerCustomizers> transactionManagerCustomizers) {
        super(dataSource, properties, jtaTransactionManager, transactionManagerCustomizers);
    }

    @Override // permet d'adapter le JPA en fonction de l'implémentation (ici EclipseLink)
    protected AbstractJpaVendorAdapter createJpaVendorAdapter() {
        return new EclipseLinkJpaVendorAdapter();
    }

    @Override // on prend les propriétés propres àl'implémentation du PersistenceUnit
    protected Map<String, Object> getVendorProperties() {
        HashMap<String, Object> map = new HashMap<>();
        map.put(PersistenceUnitProperties.WEAVING, detectWeavingMode());
        map.put(PersistenceUnitProperties.DDL_GENERATION, "drop-and-create-tables");
        return map;
    }

    private String detectWeavingMode() {
        return InstrumentationLoadTimeWeaver.isInstrumentationAvailable() ? "true" : "static";
    }
}
```

Ce n'est pas tout ! Encore un dernier fichier à modifier : application.properties en ajoutant les lignes suivantes :

```
### on lui dit d'afficher en clair les requêtes SQL effectuées (pour plus de compréhension pendant le dévelopement) ###
spring.jpa.show-sql=true
# on lui dit de ne pas utiliser le schema.sql au démarrage
spring.jpa.generate-ddl=false
```

Au passage, LA référence du application.properties est ici : https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html
Voila ! Passons maintenant aux changements dans le code.

9.2 Modifier les beans

Pour qu'un bean soit enregistré et persisté il faut le modifier afin de l'indiquer. Certains utilisent un ficheir XML pour indiquer cela, dans notre cas.... annotations !

Notre bean Product est jusqu'à présent persisté à l'aide du DAO et n'est qu'un Plain Old Java Object (POJO). Pour qu'il soit reconnu automatiquement il faut l'annoter à tout va. Nous avons besoin actuellement de trois annotations.

1. Vous trouverez d'autres alternatives à base de fichier persistence.xml, cette configuration permet d'utiliser directement les interfaces qui auraient lues ce fichier, et ainsi d'éviter encore une fois du XML.

À l'aide des imports suivants faire ceci :

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

1. Indiquer que la classe est une entité à enregistrer (persister) à l'aide de @Entity
2. Indiquer quelle propriété de l'objet est une clé primaire : @Id
3. Indiquer la stratégie de génération automatique de cette clé primaire : @GenerateValue(strategy=GenerationType.A

Comme notre base de données est simple, ces annotations suffisent.

9.3 Créer un dépôt (*Repository*)

Un repository est une encapsulation des méthodes les plus simples (save, delete, findAll, etc.). Finalement, notre DAO le faisait mais nous devions sans cesse le recréer pour chaque bean. Pour faire simple : **le repository rend abstrait l'accès à la base de données**. On peut donc, presque, dire que le repository remplace le DAO.

En fait, pour les actions basiques citées précédemment, nous appelons cela un CRUD repository, et Spring a une interface pour cela. C'est très utile car cette logique perdure même si on change de type de base de données (SQL, NoSQL, etc). Nous allons utiliser l'interface de Spring pour les JPA qui, en plus d'hériter du CRUDRepository, va plus loin en simplifiant certaines choses (counts, findAll retourne une liste typée et non une collection, etc.).

Les marches à suivre sont les suivantes :
— Dans le package dédié aux modèles ou DAO, créer une interface java nommée ProductRepository
— Faire hériter cette interface de JpaRepository<Product, Long>. Qui suit donc la logique JpaRepository<Type de l'objet, Type de la clé primaire>
Voila, c'est tout. Oui, l'interface est vide car elle hérite de beaucoup de choses de JpaRepository qui elle-même hérite des actions basiques de CrudRepository. Il est toutefois toujours possible d'y ajouter des méthodes, ce que nous allons faire par la suite.

Remarque : Vous avez que j'ai spécifié Long comme type de clé primaire. Précédemment nous avions int, vous pouvez le conserver, mais Long est plus approprié pour une vraie utilisation de clé auto générée par la BDD. (ce qui signifie qu'il vous faudra adapter et changer en cascade ces types)

9.4 Supprimer les DAO et DAOImpl

Oui, vous pouvez d'ors et déjà supprimer les DAO. Mais avant cela, vérifiez bien que vous avez sauvegardé votre projet comme indiqué au début de ce chapitre sur les JPA, vous conserverez ainsi vos propres exemples de DAO si besoin est.

9.5 Adapter le code

Vous devez désormais adapter le code en fonction du repository. Comme nous l'avons dit, il remplace le DAO. Donc un @Autowired du DAO deviendra donc un @Autowired du repository.

```
@Autowired
ProductRepository pr;
```

Avant cela, vous rencontrerez un type de Java 8 : Optional<T>. Ce type permet d'éviter les NullPointerException en encapsulant un test de vérification du contenu. Pour accéder directement au type, faites comme ceci :

```
Optional<Product> op = pr.findById(id);
Product p = op.get(); // et voila votre produit. Vous pouvez vérifier qu'il n'est pas null grâce àOptional
```

Ceci bien compris, vous pouvez :

1. Modifiez Controllers et Services afin de bien utiliser le repository
2. Modifiez les tests afin qu'ils fonctionnent bien avec le repository : surtout pour les tests d'intégration

Modifier les test aide beaucoup à comprendre le fonctionnement des repositories ! !
Une fois cela fait, vous pouvez :

1. Tester que cela fonctionne avec : mvn test
2. Si c'est le cas, supprimez votre BDD et testez votre application.

Le code est plus simple, la base de données se créer automatiquement, pas besoin d'un schéma, et tout devient transparent. C'est beaucoup mieux non ? Toutefois gardez à l'esprit que les DAO peuvent être utiles par moment. L'approche de requêtes à la main n'est jamais morte : optimisation des requêtes pour un gain de temps (d'exécution) ou d'argent (dans le cas d'API payantes à distance - bonjour Google Store -), non accès à la structure globale de la BDD à distance (souvent le cas dans les grosses infrastructures), etc. Bref, DAO et ORM avec JPA se combinent aisément.

9.6 Persistance et relations entre objets

À ce stade vous devez posséder l'AMUzone en version JPA avec tests fonctionnels. L'AMUzone de base possède une structure de base de données simple, c'est-à-dire sans JOIN (INNER, LEFT, RIGHT). Nous allons désormais voir comment effectuer ces opérations dans JPA, ainsi que les cascade d'ajout ou de suppression.

9.6.1 Relation Category-Product

Jusqu'à présent nous avions seulement un indicatif de catégorie sous forme de valeur de chaîne de caractères. Ce n'est pas une approche assez robuste si on souhaite ajouter des informations aux catégories, voire y lier des utilisateurs par exemple. Nous allons donc créer un bean category et le lier aux produits

Pour ce faire :

1. Créez un bean Category avec comme attributs (String) "name" et (Set<Product>) "products". Nous utilisons un set afin d'éviter les redondances, utile pour la BDD.

   ```
   private Set<Product> products = new HashSet<Product>(0);
   ```

2. Indiquez que l'Id est la propriété "name"
3. Créez les setters et getters qui en découlent
4. Annotez la propriété "products" en @OneToMany. Pourquoi ? Parce qu'il y a plusieurs produits pour une seule catégorie, et comme nous sommes dans le bean Category le One fait référence au bean actuel/source (category), le Many fait référence au bean cible (ici Product).

Il faut également spécifier le type de récupération : LAZY pour que la requête à la base de données ne soit faite qu'en fonction de ce qui est demandé (le bean et si nécessaire le ou les beans associés), EAGER pour toujours tout récupérer (le bean et tous les beans associés). Quand une liste est suceptible d'être grande, il est recommandé d'utiliser LAZY. Mais alors, comment faire pour indiquer cela dans le code ? Comme ceci :

```
@OneToMany(fetch = FetchType.LAZY)
```

Nous souhaitons également indiquer des ajouts, modifications et suppression en cascade. C'est-à-dire que si j'ajoute un Produit en lui donnant une catégorie, alors il doit apparaître dans la liste des produits de cette catégorie. Il en va de même pour la suppression et la modification. Ici nous allons simplement permettre toutes les cascades comme ceci :

```
@OneToMany(cascade = CascadeType.ALL)
```

L'annotation finale sera donc :

```
@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
```

Voila pour le bean Category.
Pour le bean Product :

1. Modifier la propriété category pour la lier au type Category.
2. Modifier les getters et setters en conséquence
3. Annoter cette propriété category avec @ManyToOne avec une cascade de type ALL et un fetch de type LAZY

9.6.2 Persister et fournir les catégories

Il va falloir créer un dépôt pour les catégories :

1. Créez un CategoryRepository qui hérite de JpaRepository
2. Bien spécifier le type de l'ID : String (et non Long comme pour les produits)

Ces catégories doivent être gérées et servies par un CategoryService :

1. Créer un service pour les catégories qui possède les méthodes suivantes :

```
Category addCategory(Category category);
Optional<Category> getCategory(String name);
void removeCategory(String name);
List<Category> getCategories();
```

Il est également nécessaire d'adapter le service des produits selon cet élément d'interface :

```
List<Product> getCategoryProducts(Category category);
```

9.6.3 Adapter le contrôleur et la vue à ces changements

Modifier le contrôleur ProductController :

1. Injecter ProductService
2. Dans la méthode gérant l'URL "/add", récupérer la catégorie de la BDD et **SI** elle existait **ALORS** on créer un nouveau produit en l'utilisant **SINON** on créer un nouveau produit en utilisant une nouvelle catégorie créée
3. Dans la méthode gérant l'URL "/category", utiliser CategoryService et ProductService pour récupérer une catégorie et ses produits

Pour modifier la vue (JSP), pas de grand changement, il faut juste accéder à la propriété de l'objet product utilisé dans le forEach :

1. On veut afficher le nom de la catégorie comme ceci :

```
<p class="card-text">Type : ${product.category.name}</p>
// au lieu de ${product.category}
```

Voila ! Désormais votre application utilise une logique plus appropriée pour les relations entre différentes tables de la base de données. Le tout sans SQL manuel.

Vous pouvez désormais essayer différents types de cascade pour visualiser les conséquences sur la BDD lors de l'ajout ou de la suppression.

(N'oubliez pas que vous pouvez visualiser la BDD avec hsqldb)

```
java -jar hsqldb.jar
```