Jakarta EE : Persistance de données, authentification et sécurité

Table des matières

CONTEXTE

- Vous allez compléter au fil de votre lecture votre projet Jakarta EE.
- Vous allez persister les données de votre projet dans une base de données.
- MySQL sera utilisé dans ce cours mais vous pouvez utiliser une autre base de données si vous préférez. Vous aurez alors à adapter le paramétrage de l'application ainsi qu'éventuellement certains ordres SQL.

❖ Ce symbole vous dira quel travail à faire

 Celui-ci les liens pour en savoir plus ou des informations à connaître

➔ Enfin celui-là vous indiquera les ressources à lire et/ou à utiliser

1. Le design pattern DAO (Data Access Objet)

- C'est un design pattern utilisé dans les architectures orientée objets et qui est parfaitement adapté au modèle MVC (Modèle-Vue-Contrôleur).
- Les classes DAO font partie du Modèle.
- Il regroupe les accès aux données persistantes (bases de données, fichiers…) dans des classes séparées et dédiées à cette fonctionnalité.
- Ces classes sont en lien avec les classes « contrôleurs » qui **fournissent des objets « métiers »** à la DAO qui en assurent la persistance, et la DAO **fournit des objets « métiers »** à l'application.
- Cette architecture permet de rendre la persistance des données indépendante du reste de l'application. Vous pouvez changer de mode de persistance (fichiers vers bases de données et inversement) ou tout seulement de base de données sans avoir à modifier le reste de l'application.

➢ Dans un premier temps, vous verrez une DAO utilisant une connexion à la base de données avec une DataSource et une classe de persistance de la classe Client utilisant des ordres SQL.

➢ Dans un deuxième temps, vous aborderez JPA (Java Persistence API) et vous ferez évoluer votre diagramme de classe.

2. La DAO classique (DataSource)

1. Connexion à la base de données

Vous allez vous connecter à la base de données avec JDBC (Java DataBase Connectivity)

➔ https://jakarta.ee/specifications/platform/9/apidocs/?jakarta/activation/DataSource.html

Vous allez travailler avec une DataSource.

Vous verrez les avantages de la DataSource en lisant cette ressource

➔ https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html#deploy_datasource

❖ Modifiez votre fichier settings.xml dans <profile><properties>

<mysql.url>**urlBaseDeDonnées**</mysql.url>
Exemple pour MySQL :
<mysql.url>jdbc:mysql://localhost:3306/</mysql.url>
<mysql.user>**nomUserBdd**</mysql.user>
<mysql.password>**MotDePasseBdd**</mysql.password>

❖ Ajoutez à votre fichier pom.xml le connector de votre base de données :

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j

- Modifier votre plugin artifactId maven-war-plugin de votre fichier pom.xml en ajoutant cette configuration :

```
<plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.2</version>
        <configuration>
          <webResources>
            <resource>
              <directory>src/main/webapp/META-INF</directory>
              <!--
                We ask for filtering as we inject the database connection
                properties
              -->
              <filtering>true</filtering>
              <targetPath>META-INF</targetPath>
            </resource>
          </webResources>
        </configuration>
    </plugin>
```

- Créez un dossier META-INF dans le dossier webapp (il doit être au même niveau que le dossier WEB-INF, c'est-à-dire à la racine du classpath). Le dossier META-INF contient les fichiers de configuration/paramétrage.

- Avec votre IDE, créez dans votre dossier META-INF un fichier context.xml contenant les informations suivantes

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <Resource
            name="jdbc/ nomDeLaRessource "
            auth="Container"
            type="javax.sql.DataSource"
            driverClassName="com.mysql.cj.jdbc.Driver"
            username="${mysql.user}"
            password="${mysql.password}"
            url="${mysql.url}NomBDD?serverTimezone=UTC"

    />
</Context>
```

Jakarta EE Persistance de données, authentification et sécurité

❖ Création de la connexion avec la DataSource dans le Front Controller (Servlet)

- Créez un attribut de la classe DataSource avec l'annotation :

@Resource(name="jdbc/ **nomDeLaRessource** ") public static DataSource datasource;

L'annotation @Ressource va chercher les informations dans le fichier Context.xml

⚠ Remarque : l'annotation @Resource est dédiée aux « Components class » de Jakarta EE

➔ https://docs.oracle.com/cd/E19575-01/819-3669/bnabb/index.html

Si vous voulez instancier une DataSource ailleurs que dans une Components class, vous devrez inclure le code ci-dessous :

```
public static DataSource datasource ;
try {
        Context initCtx = new InitialContext();
        Context envCtx = (Context) initCtx.lookup("java:comp/env");

        // Look up our data source
        datasource = (DataSource) envCtx.lookup("jdbc/bddappweb");
}
catch (Exception e) {
        new ServletException("Couldn't find the Datasource", e);
}
```

- Créez un attribut de classe connection de la classe Connection
- Dans la méthode init(), créez votre connexion

```
connection = datasource.getConnection();
```

La méthode getConnection peut lever une SQLException qui doit être redirigée vers une ServletException

- Créez une méthode : public void destroy() (cf chapitre 6 du cours jakartaEE) dans laquelle vous allez fermer la connexion : connection.close()

3. L'authentification

- Gérer une authentification pour les pages de saisie, modification et suppression des Clients.
- Ajouter des token CSRF à votre application
- Ajouter une authentification par cookies

❖ Création du user admin et de son mot de passe dans la base de données

La création du user « admin » et de son mot de passe associé crypté se fera à l'aide d'un contrôleur dédié et l'une adresse url que vous lancerez une fois.

- Créez un JavaBean User contenant les attributs identifiant, user, pwd avec les annotations correspondantes. La table user sera créé à la première instanciation du JavaBean
- Utiliser la méthode de hachage Argon2

  Dépendance : https://mvnrepository.com/artifact/de.mkammerer/argon2-jvm
- En option : Ajouter un sel.

  Il doit se trouver dans une variable d'environnement que vous aurez créé. Exemple : APP_SECRET

  Pour la lire :

  String appSecret = System.getenv("APP_SECRET");
- Codez le contrôleur pour crypter le mot de passe et sauvegarder dans la base de données. Modifiez votre Front Controller pour inclure l'adresse url que vous lancerez directement :
  http://localhost:**numéroPort**/**nomProjet/**?cmd=**adresseUrl**
  Une fois la table et l'enregistrement crées, enlevez le lien dans le Front Controller

❖ Créez un lien (ou un onglet) connexion qui accèdera à une page de connexion dans votre barre de navigation (dans le header.jsp).

❖ Créez une JSP de connexion avec user et mot de passe

Jakarta EE Persistance de données, authentification et sécurité

❖ Créez une classe UserForm de validation du formulaire

- Accédez à la table user de la base de données

- Si les identifiants sont bons (accès à la base de données), créez une variable de session qui indiquera que l'utilisateur est autorisé à créer, modifier ou supprimer des Clients

- Sinon, envoyez un message d'erreur

➢ Choix 1 : dans vos contrôleurs de saisie, modification et suppression du client

- Testez si la variable de session de connexion existe.
- Sinon, redirigez vers la JSP de connexion via le Front Controller et le contrôleur associé.

➢ Choix 2 : dans votre FrontController

- Créez une Hashmap contenant les url des pages « admin »
- Si l'URL est celle d'une page admin et la variable de session indiquant que l'utilisateur est un admin n'est pas présente, redirigez vers la JSP de connexion via le Front Controller et créer un paramètre de session contenant la page souhaitée par l'utilisateur.
- Dans le contrôleur de Connexion, si un paramètre de session contenant la page souhaitée par l'utilisateur existe, si l'authentification est ok, redirigez vers la JSP souhaitée par l'utilisateur via le Front Controller et le contrôleur associé.

❖ Gestion de la déconnexion

- Créez un lien (ou un onglet) déconnexion dans votre barre de navigation (dans le header.jsp).

- Créez un contrôleur pour la déconnexion, codez dedans la suppression de la session : session.invalidate() et rediriger vers la page d'accueil de votre site.

❖ Authentification par cookies

- Ajouter un « remember me » à l'aide d'un token d'identification et gérer avec un filtre

4. Sécurité : Les tokens CSRF

- Ajouter un token CSRF à toutes les pages ayant un formulaire

  Dans le contrôleur ou directement dans la JSP :

  - ✓ Générer un nombre aléatoirement et le mettre dans une variable
  - ✓ Dans la JSP, créer un champ caché (type : hiden) avec la valeur de cette variable
  - ✓ Créer une variable de session avec la valeur de cette variable
  - ✓ Au retour du formulaire, vérifier que les 2 valeurs sont identiques directement dans le contrôleur ou avec un filtre (https://jakarta.ee/learn/docs/jakartaee-tutorial/current/web/servlets/servlets006.html)

5. JPA (Java Persistance API)

➔ https://fr.wikipedia.org/wiki/Java_Persistence_API

A retenir :

- L'API JPA est une spécification définissant les règles de la gestion de la persistance de données. JPA se base sur le Mapping Objet Relationnel (ORM)
- L'API JPA se trouve dans le jar jakarta.persistence-api

❖ Ajouter la dépendance

https://mvnrepository.com/artifact/jakarta.persistence/jakarta.persistence-api

Prenez la version 3.1.0

- Le langage utilisé par JPA est JPQL (Java Persistence Query Language)
- JPA repose sur l'utilisation des annotations Java (@)
- JPA ne peut fonctionner sans une implémentation sous la forme d'un Framework.
  Framework utilisant le standard JPA : Hibernate, EclipseLink, OpenJPA, TopLink…

JPA utilise les EJB (Enterprise JavaBean) qui sont gérés par le conteneur EJB (ne pas confondre avec le conteneur des servlets) qui gère entièrement le cycle de vie des EJB. Les EJB actuels sont les **EJB3**.

⚠ Attention, de nombreuses ressources sur le web parlent des EBJ 1 et 2 qui sont complétement obsolètes

Il existe 3 types d'EJB :

- Les EBJ
- Les EJB Session qui sont des EBJ proposant des services à des « clients
- Les EJB Message (Message-Driven Bean) c'est un composant coté serveur qui traite des messages venant d'autres applications de manière asynchrone.

Par défaut, tout serveur d'applications d'entreprise Jakarta EE contient un conteneur EJB. En revanche, pour ce qui est des serveurs légers comme Tomcat, ce n'est pas le cas. Vous ne pourrez donc pas manipuler certaines fonctionnalités des EJB depuis Tomcat sans y ajouter un conteneur EJB.

Par exemple, certaines annotations comme @PersistenceContext et @PersistenceUnit ne pourrons pas également être utilisées.

➔ https://javaee.github.io/tutorial/persistence-intro.html

Vous allez utiliser Hibernate qui est un Framework qui gère la persistance des objets en base de données relationnelles. Il implémente les spécifications de JPA.

➔ https://hibernate.org/orm/

a) Mise en place du nouveau projet

- Clonez votre projet et appeler le différemment
- Enlevez de votre pom.xml les properties liées au fichier context.xml
- Ajoutez à votre fichier pom.xml la dépendance d'Hibernate

  https://mvnrepository.com/artifact/org.hibernate/hibernate-core-jakarta

  Pour Jakarta/Tomcat 10, prenez la version 5.6.15.Final

  Regardez toutes les dépendances ajoutées par cette dépendance

  Dans le Front Controller, enlevez le code spécifique à la connexion et la DataSource

- Supprimez le fichier context.xml

b) Connection à la base de données avec JPA

La connexion à la base de données a besoin d'un fichier de configuration persistence.xml qui doit être localisé dans un sous package META-INF du package src/main/resources. Ce fichier contient les propriétés de la partie JPA et de la partie Hibernate.

➔ https://examples.javacodegeeks.com/enterprise-java/jpa/java-persistence-xml-example/

➔ Ci-dessous un exemple de fichier persistence.xml pour MySql version 8

```
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
    <persistence-unit name="nomDeLaRessource" transaction-
type="RESOURCE_LOCAL">
        <properties>
            <!-- partie JPA générique -->
            <property name="javax.persistence.jdbc.driver"
                value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://127.0.0.1/NomBaseDeDonnées?serverTimezone=UTC"
/>
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="root" />

            <!-- partie spécifique Hibernate -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL8Dialect" />
        </properties>
    </persistence-unit>
</persistence>
```

Rq : Si votre base de données est de type « InnoDb », remplacez MySQL8Dialect par le dialect adapté à votre base (ex : MySQLInnoDBDialect)
https://docs.jboss.org/hibernate/stable/core/javadocs/org/hibernate/dialect/

❖ Adaptez ce fichier à votre base de données et votre version de MySQL

- property name="hibernate.hbm2ddl.auto" value :

  hbm2ddl.auto est utilisé pour valider et exporter les schéma DDL vers la base de données
  - create : Hibernate supprime en premier les tables existantes (structure et données) et recrée de nouvelles tables
  - validate : Hibernate valide seulement les structures des tables et lève une exception si problème
  - update : Hibernate vérifie la structure des tables et colonnes. Il met à jour le schéma si nécessaire
  - create-drop : Hibernate supprime le schéma lorsque l'application est arrêtée

A savoir :

Le concept de cache de base de données est un concept important à connaître.

Sans une copie des données en mémoire (c'est-à-dire un cache), lorsque vous appelez une méthode de lecture, le fournisseur de persistance devrait aller lire la base de données. L'appel de cette même méthode de lecture plusieurs fois entraînerait plusieurs accès vers la base de données. Ce serait évidemment un gros gaspillage de ressources.

L'autre avantage d'avoir un cache est que lorsque vous appelez une méthode de mise à jour de la base de données, celle-ci, en règle générale, ne modifiera pas immédiatement la base. Lorsque le cache est "vidé", les données qu'il contient sont envoyées à la base de données via autant de mises à jour, d'insertions et de suppressions SQL que nécessaire.

➔ https://javaee.github.io/tutorial/persistence-intro004.html

En résumé :

- Un **PersistenceContext** est essentiellement un cache. Il a également tendance à avoir sa propre connexion à la base de données non partagée.
- Un **EntityManager** représente un PersistenceContext (et donc un Cache)
- Un **EntityManagerFactory** crée un EntityManager (et donc un PersistenceContext / Cache)

- transaction-type :

Avec l'option " RESOURCE_LOCAL ", vous êtes responsable de la création et du suivi de l'EntityManager (PersistenceContext / Cache) ...

- Vous **devez** utiliser **EntityManagerFactory** pour obtenir un EntityManager
- L'instance **EntityManager** résultante **est** un PersistenceContext / Cache
- Un **EntityManagerFactory** peut être injecté via l'annotation **@PersistenceUnit** uniquement (pas @PersistenceContext) si vous avez un conteneur EJB
- Vous **devez** utiliser l'API **EntityTransaction** pour commencer / valider **chaque** transaction
- L'appel de entityManagerFactory.createEntityManager () deux fois entraîne **deux** instances EntityManager distinctes et donc **deux** PersistenceContexts / Caches distincts.
- Ce n'est **presque jamais** une bonne idée d'avoir plus d'une **instance** d'un EntityManager en cours d'utilisation (n'en créez pas une seconde à moins d'avoir détruit la première)

L'autre option est "JTA" mais elle demande un conteneur EJB.

L'API Java Transaction (JTA) permet entre autres d'effectuer des transactions distribuées, c'est-à-dire des transactions qui accèdent et mettent à jour des données sur plusieurs ressources informatiques en réseau.

❖ Modifiez votre Front Controller

- Créez un attribut de classe de la classe EntityManagerFactory et un attribut de classe de la classe EntityManager
- Dans la méthode init(), donnez la valeur Persistence.createEntityManagerFactory("**nomDeLaRessource**") à votre objet EntityManagerFactory (nom de la ressource de votre fichier persistence.xml
- Donnez la valeur nomEntityManagerFactory.createEntityManager() à votre objet EntityManager
- Dans la méthode destroy(), fermez votre EntityManagerFactory avec nomEntityManagerFactory.close() ;
- Pensez à gérer les exceptions

c) Les annotations des JavaBeans

Les JavaBeans peuvent être directement liés à la base de données via un ORM utilisant l'API JPA. Ce mapping est défini soit dans un fichier de configuration XML, soit directement dans le code Java en utilisant des annotations. Nous utiliserons les annotations dans ce cours qui est la méthode actuelle. Attention, de nombreuses ressources sur le Web utilisent encore le fichier configuration.

❖ Dans cette ressource, un exemple complet d'annotations vous est fourni avec la classe « contact »

➔ https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm

❖ Modifiez votre classe Client

- Rajoutez les annotations dans votre classe Client en générant l'identifiant de votre table MySQL en auto-generate
- Rajoutez l'annotation @Table(name="client") qui fait le lien entre votre JavaBean et votre table de base de données si vous voulez que le nom de la table soit différente du nom de vote classe
- Rajoutez l'annotation @Column(name = "**nomColonneTable**") à tous vos attributs si vous voulez des noms de colonne différents du nom de l'attribut. Cherchez les options pour cette annotation

d) Utilisation de JPA (Java Persistence API)

En fait, tout va changer dans la classe DaoClient.
L'accès à la base de données ne se fera plus à l'aide de la connexion mais avec l'EntityManager instancié dans le Font Controller.

❖ Modification de la Classe Dao

- Créez un attribut de classe de la classe EntityManager ayant pour valeur celui du Front Controller

➔ https://eclipse-ee4j.github.io/jakartaee-tutorial/#the-jakarta-persistence-query-language

➔ https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/entitymanager

➔ https://jakarta.ee/learn/docs/jakartaee-tutorial/current/persist/persistence-querylanguage/persistence-querylanguage.html

- Codez vos méthodes findAll() et findPersonById(int id) à l'aide des 2 ressources ci-dessus en utilisant la méthode createQuery de JPA pour votre méthode findAll() et la méthode find de JPA pour votre méthode findPersonById(int id).
- Utilisez des paramètres (setParameter) pour éviter les injections SQL
- Codez votre méthode delete avec la méthode remove de JPA.
- Codez la méthode save pour la création et la modification du client avec les méthodes JPA persist et merge

❖ Gestion des transactions

La gestion des transactions sera faite avec un objet de la classe EntityTransaction et grâce la méthode getTransaction() de l'EntityManager.

Les méthodes de l'objet de la classe EntityTransaction sont : Begin(), commit() et rollback()

e) Les relations entre les classes

- https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/manytomany
- https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/manytoone
- https://jakarta.ee/specifications/persistence/2.2/apidocs/javax/persistence/onetoone

Jakarta EE Persistance de données, authentification et sécurité

Java EE 8

DATE DE MISE A JOUR

03/2025

Agence nationale pour la Formation Professionnelle des Adultes
3 rue Franklin – 93100 Montreuil

www.afpa.fr