Université Mohamed Premier
Ecole Supérieure de Technologie de Nador

Département d'Innovation Technologique et Ingénierie Informatique

Filière d'Ingénierie Logicielle et Cybersécurité

Rapport des Travaux Pratique

Réalisé par :
NORHANE RAMZI
MOHAMED AYMANE JLIDA

Demander par :
Mr. REDOUANE ESBAI

Année universitaire : 2025/2026

المدرسة العليا للتكنولوجيا الناظور
École Supérieure de Technologie de Nador

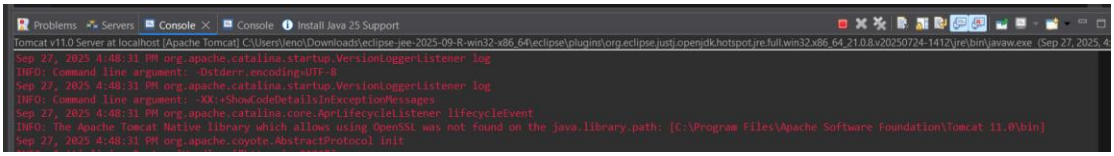
TP 0 : Prise en main de l’environnement JEE

Dans ce premier TP, nous avons mis en place l’environnement de développement nécessaire pour créer des applications JEE. L’objectif principal était d’installer et configurer les outils indispensables, puis de créer notre premier projet dynamique simple.

Nous avons commencé par installer le serveur Apache Tomcat, qui joue le rôle de conteneur JEE. Ce serveur permet d’héberger et d’exécuter des applications web basées sur Java.
Une fois Tomcat installé, nous avons :

- Vérifié son bon fonctionnement en le démarrant depuis le navigateur.
- Configuré Eclipse IDE pour qu’il reconnaisse automatiquement le serveur Tomcat et puisse déployer les projets JEE.

Cette étape garantit que l’environnement de développement est correctement opérationnel.



Premier projet JEE

Nous avons ensuite créé un nouveau Dynamic Web Project intitulé MonProjet. À l’intérieur de ce projet :

- Nous avons ajouté un fichier debut.html contenant un simple Hello World.
- Nous avons configuré le Projet pour que debut.html soit la page de démarrage par défaut.

nous accédons à l’adresse : http://localhost:8080/MonProjet/
Tomcat affiche automatiquement debut.html, même si aucun fichier n’est spécifié dans l’URL.

Hello world

Ma première Servlet

Nous avons Commencé d'abord par créér le fichier java : MaServlet.java. On récupère le flot de sortie : PrintWriter, et on écrit dedans le fichier HTML qui sera généré par le serveur dans ce le code html nous avons récupère la date actuelle par ''LocalDate.now()'' .

Dans un premier temps, nous avons configuré manuellement le mapping entre l'URL et la servlet via le fichier :

```
<servlet>
    <servlet-name>MaServlet</servlet-name>
    <servlet-class>com.MaServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>MaServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Nous avons ensuite accédé à la servlet via : http://localhost:8080/MonProjet/hello

Afin de simplifier la configuration et éviter de modifier le fichier web.xml à chaque servlet, nous avons remplacé le mapping par une annotation (@WebServlet("/hello")) directement dans le code. :

```
14  */
15 @WebServlet("/hello")
16 public class MaServlet extends HttpServlet {
```

TP 01 : Servlet

Dans ce TP, l'objectif était de créer une servlet capable d'afficher plusieurs informations concernant le client ainsi que les paramètres envoyés via la méthode GET.

Nous avons créé une servlet dont le rôle est d'afficher diverses informations sur le

client, telles que :

- Le type de requête L'adresse IP du client
- Le nom du serveur
- Le port utilisé
- Le protocole
- Le schéma de la requête (HTTP/HTTPS)
- Et l'ensemble des paramètres transmis dans l'URL

Pour récupérer ces informations, nous avons utilisé les méthodes suivantes :

- getProtocol();
- getContentType();
- getRemoteAddr();
- getRemoteHost();
- getServerName();
- getServerPort();
- getScheme();

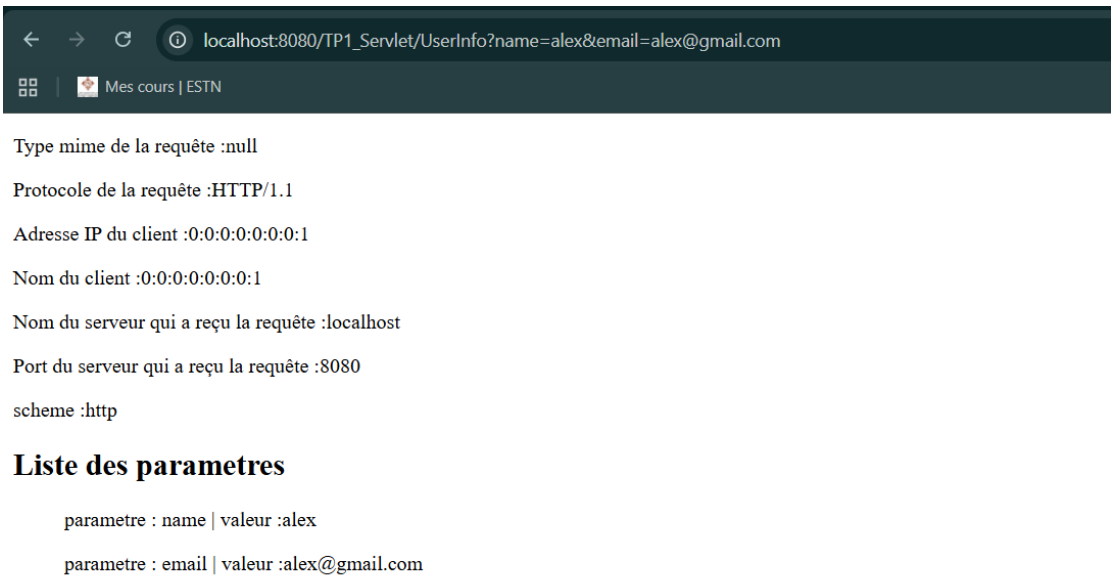
Afin de traiter les paramètres envoyés via l'URL, nous avons récupéré :

La liste des noms de paramètres à l'aide de : request.getParameterNames();
Les valeurs associées via : request.getParameter(nom);

Les noms des paramètres ont été stockés dans une énumération (Enumeration), puis parcourus grâce à une boucle while, ce qui permet d'afficher dynamiquement tous les paramètres reçus, quelle que soit leur quantité.

```
Enumeration<String> paramNames = request.getParameterNames();

while (paramNames.hasMoreElements()) {
    String paramName = paramNames.nextElement();
    String paramValue = request.getParameter(paramName);
    out.println("<p> nom : " + paramName + " valeur :" + paramValue + "</p>");
}
```

Nous avons accédé à notre servlet par l'URL
http://localhost:8080/TP1_Servlet/InfoServlet?name=alex&email=alex@gmail.com



TP 02 : Servlet

EXERCICE 1 :

Dans un premier temps, nous avons créé une classe SimpleServlet qui implémente l'interface : jakarta.servlet.Servlet
Nous avons redéfini la méthode service() pour envoyer une réponse qui contient la date actuelle au navigateur.
Cette méthode est pour créér une page HTML ,Nous avons ajouté un fichier web.xml dans le répertoire WEB-INF
On déclare avec <servlet > le nom et l' emplacement de servlet , et après on définit son chemin d'accès dans l'URL (<servlet-mapping>).

Nous avons accédé à notre servlet par l'URL : http://localhost:808/TP2ServletSimp/simple

EXERCICE 2 :

Nous avons ajouté de paramètres dans web.xml :

```
<context-param>
  <description> nom webmaster </description>
  <param-name> webmaster</param-name>
  <param-value>norhane</param-value>
/context-param>
context-param>
  <description> email webmaster</description>
  <param-name> email </param-name>
  <param-value>norhaneramzi22@gmail.com</param-value>
/context-param>
```

Dans la méthode doGet , Nous avons utilisé ServletContext pour lire les paramètres définis dans web.xml.

```
        String nom = content.getInitParameter("nom");
        String email = content.getInitParameter("email");
```

Et avec la méthode getInitParameter() pour récupérer les paramètres webmaster qui a valeur de mon nom et email pour mon email.
Nous avons accédé à notre servlet par l'URL :
http://localhost:808/TP2ServletSimp/ServletEx2

liste des parametres

nom : norhane!

email: norhaneramzi22@gmail.com

EXERCICE 3 :

Nous avons créé une classe AdditionServlet qui hérite de HttpServlet.
La servlet est accessible par l'URL /add avec l'annotation @WebServlet("/add").

```
@WebServlet("/add")
```

Nous avons créé les méthodes doGet et doPost
doGet : pour envoyer qui affiche une page html avec un formulaire qu'on peut additionner deux paramètres .
doPost : La servlet récupère les paramètres param1 et param2, calcule leur somme, puis affiche : le résultat de l'addition et à nouveau le formulaire.
Nous avons ajouter une méthode formulaire(PrintWriter out) pour afficher et appeler la fonction formulaire(out) dans doGet et doPost sans repetition du code formulaire .

```
public void formulaire(PrintWriter out) {
        out.println("<form method='POST' action='add'> ");
        out.println("<input type='number' name='param1'>");
```

Nous avons accédé à notre servlet par l'URL :
http://localhost:808/TP2ServletSimp/add

Addition SERVLET

Additionner

4

TP 03 : COOKIES

Dans ce TP, nous avons développé une servlet permettant de gérer des préférences

utilisateur à l'aide de cookies, ainsi qu'un système personnalisé de gestion des erreurs HTTP (404 et 500).

Nous avons développé une servlet personnalisée qui affiche un formulaire contenant plusieurs choix permettant à l'utilisateur de sélectionner :

- Sa couleur de fond préférée
- Sa couleur de police
- Son type de livres préféré
- Son nombre favori

Ce formulaire est envoyé via la méthode **POST** à la même servlet.

Après la soumission du formulaire, la servlet : Récupère les valeurs choisies par l'utilisateur Crée un cookie pour chaque préférence Envoie ces cookies au navigateur grâce à : response.addCookie(cookie);
Ces cookies permettent de mémoriser les préférences de l'utilisateur.

```
response.addCookie( new Cookie("couleurFond",couleurFond));
response.addCookie( new Cookie("couleurPolice",couleurPolice));
response.addCookie(new Cookie("typeLivre",typeLivre));
response.addCookie(new Cookie("nb",nb));
response.sendRedirect("FormServlet");
```

Lorsqu'un utilisateur revient sur la page

- La servlet récupère tous les cookies existants via : request.getCookies()
- Une boucle parcourt les cookies pour retrouver ceux correspondant aux préférences précédemment enregistrées
- Les valeurs trouvées sont automatiquement appliquées dans le formulaire
- L'utilisateur retrouve ainsi ses préférences sans avoir à les ressaisir

Ce mécanisme garantit une personnalisation persistante de l'interface.

- La gestion des erreurs (404 et 500)

On a crée deux page d'erreur pour erreur :

- **404 (page non trouvée)** redirection vers une page HTML dédiée (error404.html).
- **500 (erreur interne du serveur)** redirection vers une autre page personnalisée (error500.html).

```
    <error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>

<error-page>
    <error-code>500</error-code>
    <location>/500.html</location>
</error-page>
```

TP_JSP

Exercice 1 - Infos du client

L'objectif est de créer une page info.jsp qui affiche les informations du client : type de requête, adresse IP ,nom du client , nom du protocole ... et une liste de paramètres que l'on passe par la méthode GET

On a utiliser ces fonctions :

- getProtocol();
- getContentType();
- getRemoteAddr();
- getRemoteHost();
- getServerName();
- getServerPort();
- getScheme();

Pour l'affichage des paramètres GET, nous avons utilisé :

- request.getParameterNames() ; pour récupérer la liste des noms des paramètres.
- request.getParameter(nom) ; pour récupérer leurs valeurs des paramètres.

Nous avons utilisé une énumération pour obtenir la liste des noms des paramètres, puis une boucle while pour afficher chaque paramètre avec sa valeur.

```jsp
<%
    java.util.Enumeration<String> paramNames = request.getParameterNames();

        while (paramNames.hasMoreElements()) {
            String paramName = paramNames.nextElement();
            String paramValue = request.getParameter(paramName);
%>
            <li>Nom : <b><%= paramName %></b>  Valeur : <b><%= paramValue %></b></li>
```

Nous avons accédé à notre page via l'URL :
http://localhost:808/Tp1JSP/index.jsp?ali=vid&ahmed=auto

Informations du requete :

- Type mime de la requête : null
- Protocole de la requête : HTTP/1.1
- Adresse IP du client : 0:0:0:0:0:0:0:1
- Nom du client : 0:0:0:0:0:0:0:1
- Nom du serveur qui a reçu la requête : localhost
- Port du serveur qui a reçu la requête : 808
- scheme : http

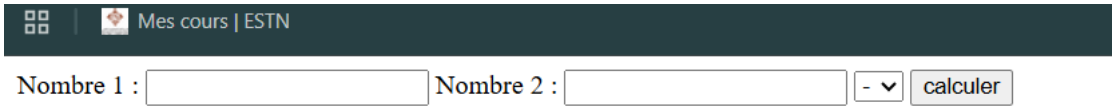
Liste des paramètres:

- Nom : **ali** — Valeur : **vid**
- Nom : **ahmed** — Valeur : **auto**

Exercice 2 - Calculette en JSP

Nous avons créé la page calculette.jsp qui permet d'effectuer des opérations arithmétiques entre deux nombres saisis .
La page contient :

- un formulaire HTML permettant de saisir deux nombres (nb1, nb2)
- un menu de sélection permettant de choisir l'opération (+, -, *, /)



Les valeurs reçues (nombres et opérations) sont récupérées avec request.getParameter(). Ensuite, les chaînes sont converties en valeurs numériques avec la méthode (Float.parseFloat()).

```
if(param1 != null && param2 != null){
    float nb1 = Float.parseFloat( param1);
    float nb2 = Float.parseFloat(param2);
```

Une vérification a été ajoutée pour éviter la division par zéro, ce qui affiche un message d'erreur.

```
}else if("div".equals(operation)){

        if(nb2 == 0){
            out.println( "le nombre doit être différent de  ");
            res = nb2;

        }else{
          res = nb1 / nb2;
```

Voici le resultat :

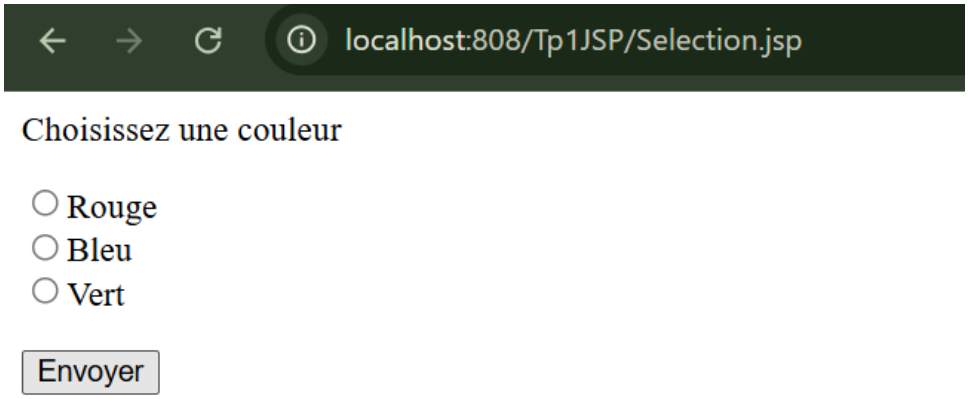
On a accédé à notre page jsp par l'URL : http://localhost:808/Tp1JSP/Calculette.jsp

Exercice 3 - JSP Sélection

Nous avons récupéré la page Selection.jsp qui permet d'afficher la selection
Nous avons ajouté un scriptlet permettant de vérifier si la requête reçue est de type **POST** grâce à la condition suivante :

```
<%
String methode = request.getMethod();
if ("POST".equals(methode))  {
    String couleur=request.getParameter("couleur");%>
    <h2>Vous avez choisis  la couleur :<%= couleur %></h2>
```

Le formulaire envoie la requête avec la méthode POST vers la même page (Selection.jsp) qui affiche le texte "Vous avez choisi la couleur <couleur>" d'où la valeur du paramètre couleur sera la couleur saisi .



Nous avons accédé à notre page jsp par l'URL :http://localhost:808/Tp1JSP/Selection.jsp

Exercice 4 - Redirection des exceptions

Nous avons créé un fichier jsp selection2.jsp permet aux utilisateur de choisir un couleur
Nous verifions si la requete est POST lorsque la couleur choisi est orange c'est une exception l'utilisateur va redirigé vers la page erreur.jsp

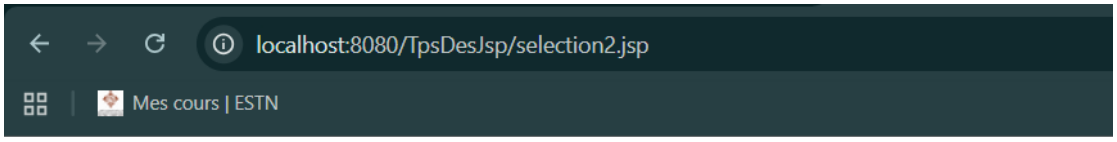
```
18
19      <%
20          String methode = request.getMethod();
21
22          if ("POST".equals(methode)) {
23              String color = request.getParameter("couleur");
24               if ("orange".equals(color)){
25
26                   throw new Exception("Cette couleur n'est pas belle");
27
```

Nous avons défini la page d'erreur en haut du jsp en utilisant errorPage=erreur.jsp .

```
1   <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2       pageEncoding="ISO-8859-1" errorPage="erreur.jsp"%>
```

Nous avons définit dans le fichier erreur.jsp le message d'exception d'erreur qui va s'afficher.

```
 8  </head>
 9  <body>
10  <p>
11              Le message de l'erreur est : <%= exception.getMessage() %>
12          </p>
13  </body>
```

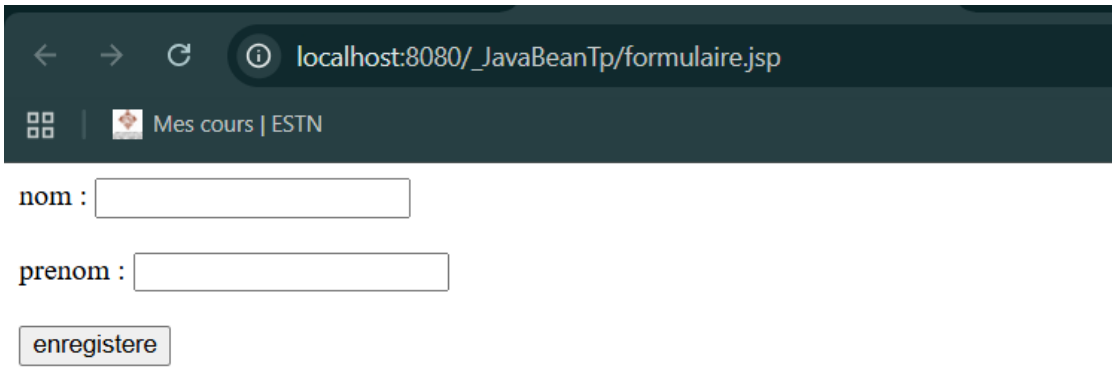


Le message de l'erreur est : Cette couleur n'est pas belle

Nous avons accédé à notre page via l'url :
http://localhost:8080/TpsDesJsp/selection2.jsp

TP : Chaînage JSP-Servlet-JavaBean

Nous avons créé une page statique contenant un formulaire à remplir (le nom et le prénom) : formulaire.jsp.



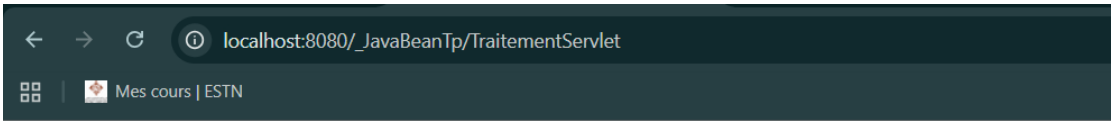
Ensuite, nous avons créé une classe Java Etudiant.java avec trois attributs : nom, prénom et note, ainsi que deux constructeurs et les getters et setters de ces attributs. Lorsque l'utilisateur remplit le formulaire, la page traitementDemande.jsp récupère le nom et le prénom via un JavaBean, puis les envoie à TraitementServlet.java.

```
<jsp:useBean id="etu" class="com.Etudiant" scope="request"/>
 <jsp:setProperty name="etu" property="nom"/>
    <jsp:setProperty name="etu" property="prenom"/>

    <jsp:forward page="/TraitementServlet" />
</body>
```

Le fichier TraitementServlet.java permet de récupérer les attributs nom et prénom, d'attribuer une note aléatoire à chaque étudiant, puis de rediriger vers reponse.jsp.

```
        Etudiant etu = (Etudiant) request.getAttribute("etu");
        double note = (float) Math.random() * 10;
         note = note + 10 ;
           note = Math.round(note * 100.0) / 100.0;
           etu.setNote(note);
           request.getRequestDispatcher("reponse.jsp").forward(request, response);
```

La page reponse.jsp permet d'afficher le nom, le prénom et la note de l'étudiant en utilisant le JavaBean.



Informations de l'étudiant :

Nom : **elon**

Prénom : **musk**

Note : **10.2**

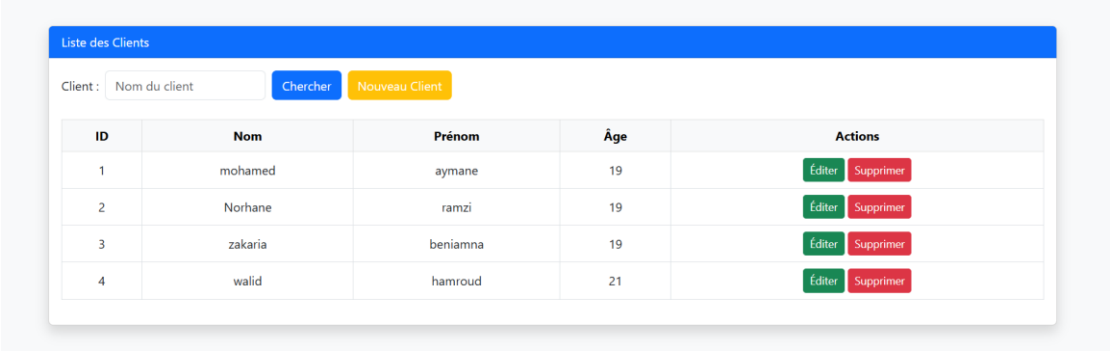
Application Web JEE CRUD avec JSP et JBDC

Nous avons créé une classe Java Client.java qui contient quatre attributs : idClient, nom, prénom et âge, ainsi que les getters, setters et un constructeur.

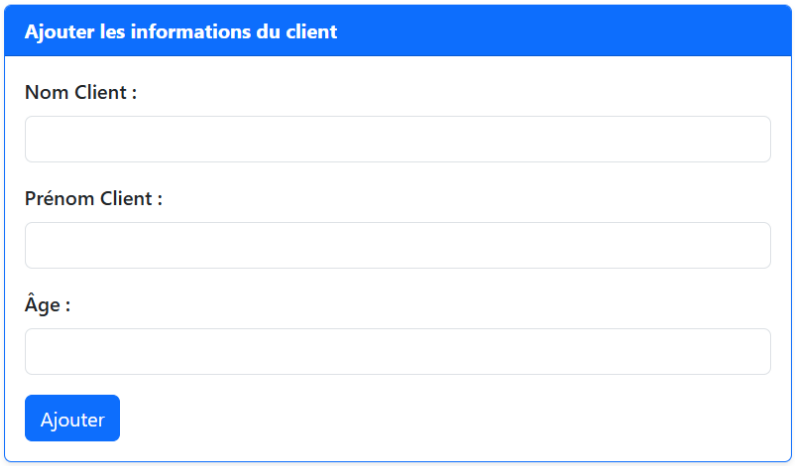
Ensuite, nous avons créé un fichier JSP dashboard.jsp qui permet d'afficher la liste complète des clients. Cette page contient une barre de recherche par nom, ainsi qu'un bouton pour ajouter, modifier ou supprimer un client.

```
try{
    Class.forName("com.mysql.cj.jdbc.Driver");
    Connection conn = DriverManager.getConnection(url_db, user_db,pwd_db);

    PreparedStatement ps = conn.prepareStatement("select * from client where nom like ?");
    ps.setString(1,"%"+moteCle+ "%");
    ResultSet rs = ps.executeQuery();

    while(rs.next()){
        Client c = new Client();
        c.setIdClient(rs.getInt("idClient"));
        c.setNom(rs.getString("nom"));
        c.setPrenom(rs.getString("prenom"));
        c.setAge(rs.getInt("age"));
        clients.add(c);
    }
```

Pour la récupération des données, nous avons utilisé des blocs try/catch afin d'assurer la gestion des exceptions, et une liste de clients pour stocker les clients récupérés depuis la base de données.



Lorsque l'utilisateur clique sur "Nouveau client", il est redirigé vers la page JSP addClient.jsp, qui permet de créer un nouveau client.

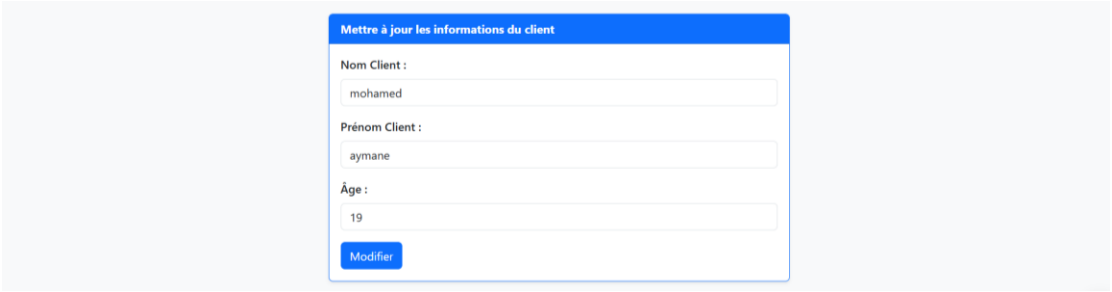


Nous avons créé un formulaire en méthode POST permettant de saisir le nom, le prénom et l'âge du client. Ce formulaire envoie les données au contrôleur

Ensuite nous avons récupéré ces trois variables et le stocker dans la table client en utilisant la requête suivante.

```
PreparedStatement ps = conn.prepareStatement("insert into client (nom, prenom, age) values(?,?,?) ");
```

Mais si utilisateur cliquer sur modifier il vas rediriger vers la page jsp EditClient.jsp avec idClient comme paramètre qui on vas le récupère et Nous utilise 2 requête la premier pour récupérer les donner de client par le id Client et la deuxième pour modifier utilisateur.



Nous avons utilisé getParameter pour récupérer le id client , par ce variable nous avons récupéré les données actuelle de cette client

```
PreparedStatement ps1 = conn.prepareStatement("select * from client where idClient = ?");
ps1.setInt(1,id);
ResultSet rs1 = ps1.executeQuery();
```

Lorsque le client modifie les données nous avons récupère cette donner modifier et nous le modifier dans la table client par la requête suivant enfin il vas le rediriger vers le Dashboard page

```
PreparedStatement ps = conn.prepareStatement(
    "UPDATE client SET nom=?, prenom=?, age=? WHERE idClient=?"
);
ps.setString(1, newNom);
ps.setString(2, newPrenom);
ps.setInt(3, newAge);
ps.setInt(4, id);

int n = ps.executeUpdate();
conn.close();

 response.sendRedirect("Dashbord.jsp");
return;
```

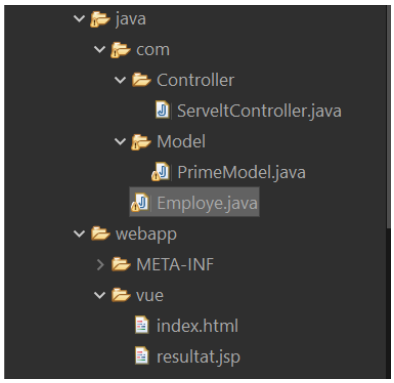
Enfin si utilisateur cliquer sur supprimer il vas le rediriger ver le fichier jsp DeleteClient.jsp avec id Client comme paramètre qui on vas le récupérer pour supprimer utilisateur Avec une requête de supprimer par id de client avec utilisation de try catch pour la gestion des exception

On a utilisé 2 requêtes une pour récupère le nom de client par son id et si utilisateur cliquer sur supprimer ce client vas supprimer à travers son id mais si utilisateur cliquer sur annuler il vas rediriger vers le Dashboard

```
PreparedStatement ps = conn.prepareStatement("DELETE FROM client where idClient = ?");
ps.setInt(1,clientId);
int n = ps.executeUpdate();
if (n>0){
    response.sendRedirect("Dashbord.jsp");
    return;
}
```

TP MVC Gestion des Primes

Nous avons créé une mini-application Gestion des Primes permettant de calculer le prime d'un employé à partir de son ancienneté. L'utilisateur saisit l'id d'un employé, et il affiche son nom, prénom et le montant de sa prime.
Nous utilisons une architecture **MVC** pour une séparation claire du données logique métier de l'application(model) ,l'interface utilisateur(la vue) ,et le controller qui est l'intermédiaire entre le modèle et la vue. Il reçoit les requêtes de l'utilisateur, exécute la logique nécessaire, fait appel au modèle et transmet les données à la vue:



Nous avons créé les classes Java Employe.java et PrimeModel.java dans un dossier

Model.

-Employe.java : contient cinq attributs privés : id, nom, prénom et date_debut, salaire ainsi que deux constructeurs et les getters et setters de ces attributs.

- PrimeModel.java : contient des attributs privés ; nom, prenom, prime avec le constructeur et les getters et setters de ces attributs .

Nous avons également créé des pages .jsp dans un dossier Vue pour le formulaire de saisie d'id et le résultat d'affichage.

Et pour le dossier Controller nous avons créé une ServletControlller.java on a modifier le chemin de @WebServlet au (*.do) afin de récupérer tous les chemins terminant par .do

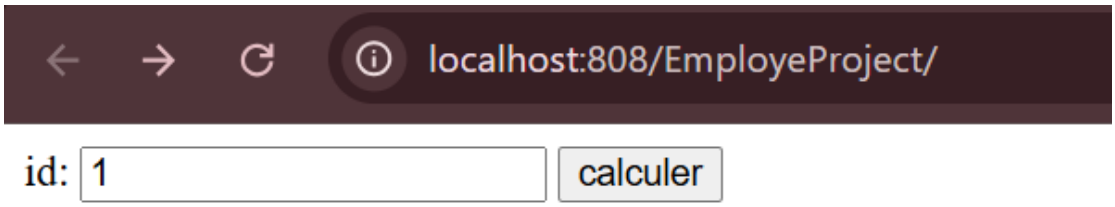
- **Ce qui gère deux actions :**

→ **(/acceuil.do)** qui est responsable de l'affichage de la page d'acceuil

d'application index.html nous avons gèrer la redirection du page avec la requête suivante : getRequestDispatcher("/vue/index.html").forward(request, response);

```
String path = request.getServletPath();
if ("/acceuil.do".equals(path)) {
    request.getRequestDispatcher("/vue/index.html").forward(request, response);
```

La page index.html contient le formulaire de saisie d 'id :



→**(/calculer.do)** qui est responsable de calculer la prime :

```
}else if("/calculer.do".equals(path)) {
```

Si le chemin égale /calculer.do nous suivons ces étapes:

- Récupération de l'id envoyé dans le formulaire et le convertir en valeur numérique (int).
- Connexion à la base de données employe_db.
- La récupération de nom, prénom, salaire et date debut avec la requête SQL (select * from employe where id = ?)

- Calculer la prime d'où La date début (date_debut) est récupérée depuis la base de donnée et il est convertie en LocalDate et l'ancienneté est calculée avec :

Period.between(LocalDate.parse(date_debut.toString()), LocalDate.now())

Puis nous avons calculer le prime selon le salaire+(l'ancienneté x500)

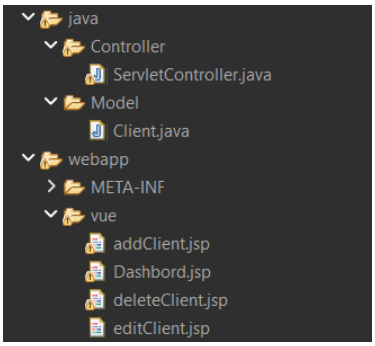
- La création d'un primeModel où la valeur du prime est stocké
-Enfin la redirection en page resultat.jsp avec la requête suivante :

```
request.setAttribute("pm", primeModel1);
 request.getRequestDispatcher("/vue/resultat.jsp").forward(request, response);
```

La page resultat.jsp permet d'afficher le résultat du calcul de prime avec le nom et prenom d'employe . Les valeurs ${pm.nom}, ${pm.prenom}et ${pm.prime} sont récupérées depuis le contrôleur.

Application CRUD Client MVC

L'application crud client doit respecter la séparation de architecture MVC ,le Modele contient les données qui sont affichées dans la vue qui contient les fichier jsp responsable du présentation de l'interface utilisateur et le Controller qui contrôle entre le modèle et la vue ce qui facilite la maintenance et rend le code plus claire.



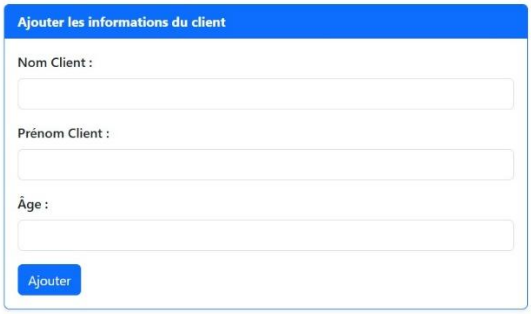
Nous avons créé la classe Java Client.java dans un dossier Model. Elle contient quatre attributs privés : idClient, nom, prénom et âge, ainsi que deux constructeurs et les getters et setters de ces attributs.
Nous avons également créé des pages .jsp dans un dossier Vue pour afficher, ajouter, modifier et supprimer les clients.

Dashbord.jsp :

Nous avons créé un tableau contenant cinq colonnes : ID, nom, prénom, âge et action, qui contient deux boutons : supprimer et modifier. En haut de la page, nous avons également ajouté une barre de recherche par nom ainsi qu'un bouton pour créer un nouveau client.

addClient.jsp:

Nous avons créé un formulaire pour ajouter les informations du client : le nom, le prénom et l'âge.



editClient.jsp:

Nous avons créé un formulaire pour modifier les informations du client : le nom, le prénom et l'âge.

deleteClient.jsp:

Nous avons créé une page pour confirmer la suppression d'un client avec deux boutons : un pour confirmer la suppression et l'autre pour annuler l'action.

Puis nous avons créé un fichier Servlet jouant le rôle de contrôleur dans un dossier Controller. Ce contrôleur va récupérer tous les chemins se terminant par .do.
@WebServlet("*.do")

Ensuite, nous avons établi la connexion à la base de données. Nous avons récupéré le chemin demandé et l'avons stocké dans une variable *path*, puis nous avons créé des

conditions *if* pour traiter chaque chemin :

```
if("/clients.do".equals(path)) { ... }
else if("/addClient.do".equals(path)) { ... }
else if("/DeleteClient.do".equals(path)) { ... }
else if("/EditClient.do".equals(path)) { ... }
```

Pour chaque condition, nous avons créé une requête SQL puis nous l'avons exécutée.

Pour clients.do, nous avons vérifié s'il existe un mot clé saisi dans la barre de recherche ; sinon, il reste null. Ensuite, nous avons préparé un statement, inséré le paramètre (mot clé), puis exécuté la requête. Le résultat a été stocké dans une variable, puis nous avons utilisé une boucle while pour parcourir ce résultat.
À chaque itération, nous avons créé un objet Client, rempli ses attributs avec les données de la base, puis ajouté cet objet à une liste de clients. Enfin, nous avons envoyé cette liste au fichier dashboard.jsp à l'aide de setAttribute.

```java
try{
    Class.forName("com.mysql.cj.jdbc.Driver");
    Connection conn = DriverManager.getConnection(url_db, user_db,pwd_db);

    PreparedStatement ps = conn.prepareStatement("select * from client where nom like ?");
    ps.setString(1,"%"+moteCle+ "%");
    ResultSet rs = ps.executeQuery();

    while(rs.next()){
        Client c = new Client();
        c.setIdClient(rs.getInt("idClient"));
        c.setNom(rs.getString("nom"));
        c.setPrenom(rs.getString("prenom"));
        c.setAge(rs.getInt("age"));
        clients.add(c);
    }

    request.setAttribute("pm", clients);
    request.getRequestDispatcher("/vue/Dashbord.jsp").forward(request, response);
```

Pour addClient.do, lorsque la méthode est POST, nous récupérons les valeurs du formulaire (nom, prénom, âge), puis nous préparons et exécutons une requête SQL d'insertion pour ajouter un nouveau client dans la base de données. Après l'exécution, nous redirigeons vers clients.do.
Si la méthode est GET, nous redirigeons simplement vers la page addClient.jsp pour afficher le formulaire.

```java
try{

    Class.forName("com.mysql.cj.jdbc.Driver");
    Connection conn = DriverManager.getConnection(url_db, user_db, pwd_db);

    String nom = request.getParameter("nom");
    String prenom = request.getParameter("prenom");
    int age = Integer.parseInt(request.getParameter("age"));

    PreparedStatement ps = conn.prepareStatement("insert into client (nom, prenom, age) values(?,?,?)");
    ps.setString(1,nom);
    ps.setString(2,prenom);
    ps.setInt(3,age);
    int n = ps.executeUpdate();
    response.sendRedirect("clients.do");
    return;
```

Pour DeleteClient.do, si la méthode est GET, nous récupérons l'identifiant du client, chargeons ses informations depuis la base, créons un objet Client, puis l'envoyons à deleteClient.jsp pour afficher la confirmation de suppression.
Si la méthode est POST, nous préparons une requête DELETE avec l'id du client, l'exécutons, puis redirigeons vers clients.do.

Pour EditClient.do, si la méthode est GET, nous récupérons l'id du client, chargeons ses informations dans la base, créons un objet Client et l'envoyons à editClient.jsp pour afficher le formulaire pré-rempli.
Si la méthode est POST, nous récupérons les nouvelles valeurs du formulaire, préparons une requête UPDATE, l'exécutons, puis redirigeons vers clients.do.

Application CRUD Client MVC et Spring IoC

L'application crud doit respecter la séparation des couches **DAO** (Data Access Object), **Métier** (Logique applicative), et **Présentation** (Interface utilisateur), l'utilisation du framework Spring pour la gestion de l'injection des dépendances (IoC) et l'utilisation du Singleton pour gérer une seule connexion à la base de données.

```
dao
    DaoImpl.java
    IDao.java
metier
    IMetier.java
    MetierImpl.java
presentation
    Client.java
    ServletController.java
unite
    DbConnection.java
    SingleTonConnection.java
```

Dans le répertoire dao nous avons crée une interface IDao avec les fonctions nécessaires pour l'application comme getClients (la récuperation des clients) ajouter , chercher , modifier , supprimer) car la couche DAO Est responsable de toute la

communication avec la base de données (requêtes SQL pour l'ajoute, la modification, la suppression et la recherche).

```
public List<Client> getClients(String moteCle) throws SQLException;
public boolean AddClient(String nom , String prenom , int age);
public boolean DeleteClient(int id);
public Client FindClientById(int id );
public boolean EditClient(int id, String nom , String prenom , int age);
```

Puis nous avons crée dans le mème répertoire Dao une classe DaoImpl qui implémente de l' interface IDao, nous avons créé un variable conn pour la connexion avec base de donné avec la classe SingleTonConnection pour éviter la création multiple de connexion , et l'utilisation de preparedStatement pour les requêtes sql de l'ajoute, la modification, la suppression et la recherche).

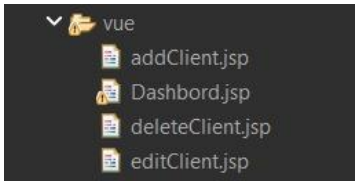
Pour la couche métier qui agit comme une interface intermédiaire entre la couche DAO ET présentation.

Nous avons créé le répertoire métier avec l'interface IMetier.java et la classe MetierImpl.java
Et nous posons les mêmes méthodes utilisé dans IDao .

```
public interface IMetier {
    public List<Client> getClients(String moteCle) throws SQLException ;
    public boolean AddClient(String nom , String prenom , int age);
    public boolean DeleteClient(int id);
    public Client FindClientById(int id );
    public boolean EditClient(int id, String nom , String prenom , int age);
```

Pour la classe MetierImpl qui implémente de l'interface IMetier et il encapsule les méthodes de gestions de clients et on instancie le variable de IDao et création d'un Constructeur et injection du DAO via un setter pour une injection automatique avec Spring IoC.

Pour la Vue :



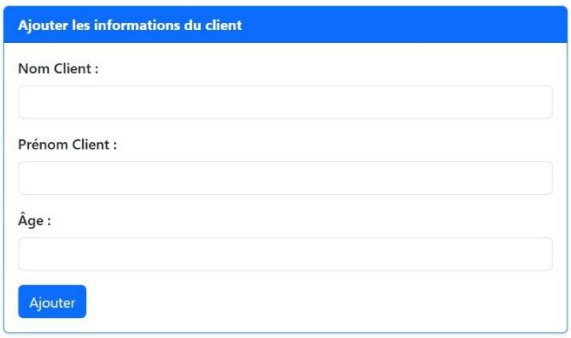
Nous avons créé des pages .jsp dans un dossier Vue pour afficher, ajouter, modifier et supprimer les clients.

Dashbord.jsp :

Nous avons créé un tableau contenant cinq colonnes : ID, nom, prénom, âge et action, qui contient deux boutons : supprimer et modifier. En haut de la page, nous avons également ajouté une barre de recherche par nom ainsi qu'un bouton pour créer un nouveau client.

addClient.jsp:

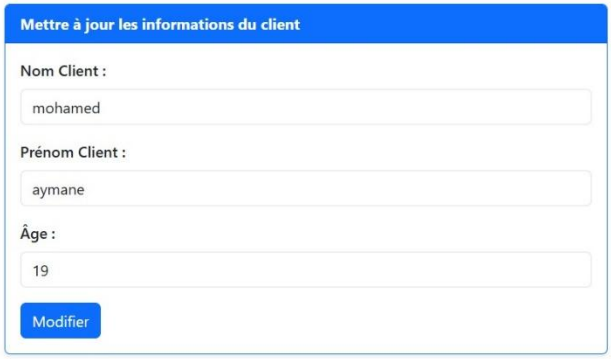
Nous avons créé un formulaire pour ajouter les informations du client : le nom, le prénom et l'âge



.

editClient.jsp:

Nous avons créé un formulaire pour modifier les informations du client : le nom, le prénom et l'âge



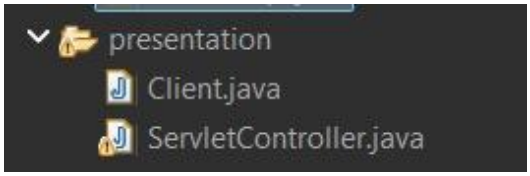
.

deleteClient.jsp:

Nous avons créé une page pour confirmer la suppression d'un client avec deux boutons : un pour confirmer la suppression et un autre pour annuler l'action.

La couche Presentation: est la partie interface utilisateur et contrôle de l'application.

Nous avons créé une classe Client.java avec les instanciations des variables ,constructeurs getters et setters et la classe ServletController.java qui est le lien entre les pages de vue et les classes MetierImpl et DaoImpl.



et la classe ServletController.java qui est le lien entre les pages de vue et les classes MetierImpl et DaoImpl

```
String path = request.getServletPath();


if("/clients.do".equals(path)) {
    String moteCle = request.getParameter("motCle") ;
    List<Client> clients ;
    try {
          clients = metier.getClients(moteCle);
          request.setAttribute("pm", clients);
```

Le contrôleur vérifie si l'URL demandée est bien **/clients.do**, et récupérer le mot-clé que l'utilisateur a saisi et il les stocke dans une liste clients et on fait l'appel à l'objet métier pour récupérer la liste des clients en fonction du mot-clé saisie par la méthode : **metier.getClients(motCle);** ,puis on les stockent dans l'objet requête sous le nom pm .

```
if("/clients.do".equals(path)) { ... }
else if("/addClient.do".equals(path)) { ... }
else if("/DeleteClient.do".equals(path)) { ... }
else if("/EditClient.do".equals(path)) { ... }
```

→ ici le contrôleur interprète l'URL, exécute la logique pour appeler la **couche métier**, puis redirige l'utilisateur vers la bonne **vue ("/clients.do"** avec methode **"GET"** et le reste avec la methode **" POST").**

Dans la methode init() on écrit les injections :
Elle permet d'initialiser les dépendances et il exécute une seul fois lors de démarrage du servlet.

Il y a 4 injections :

- **Static :**

```
DaoImpl dao = new DaoImpl();

this.metier = new MetierImpl();
metier.setDao(dao);
```

On crée manuellement une instance de la connexion, dao , métier.

dao.setConn(conn);
metier.setDao(dao);

On injecte manuellement la connexion dans le Dao et le Dao dans métier en utilisant une méthode "setter".

- **Dynamique**

```
try {
    String configFilePath = getServletContext().getRealPath("/WEB-INF/config.text");
    File configFile = new File(configFilePath);

    Scanner scanner = new Scanner(configFile);

    String daoClasseName = scanner.nextLine();
    String metierClasseName = scanner.nextLine();

    Class cdao = Class.forName(daoClasseName);
    IDao dao = (IDao) cdao.getDeclaredConstructor().newInstance();


    Class cmetier = Class.forName(metierClasseName);
    this.metier = (MetierImpl) cmetier.getDeclaredConstructor().newInstance();


    Method meth = cmetier.getMethod("setDao", new Class[] {IDao.class});
    meth.invoke(this.metier, new Object[] {dao});
```

-

- L'utilisation de Class.forName() pour l' appel des noms des classes et invoke() pour l'execution d'une methode par un objet
- Il lit le fichier config.text pour trouvé le nom de les classes montré dans le

fichier config.text de connexion, Dao , et metier

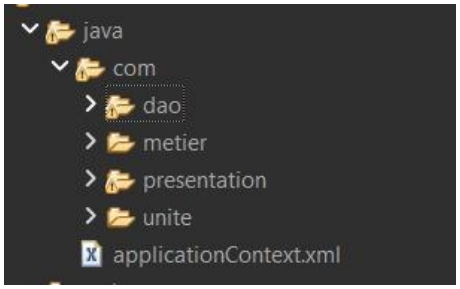
On utilise un fichier config.text qui est dans : **webapp/WEB-INF/config.text**

```
com.dao.DaoImpl
com.metier.MetierImpl
```

-
```
ApplicationContext springContext =
   new ClassPathXmlApplicationContext("applicationContext.xml");

this.metier = springContext.getBean("metierImpl", MetierImpl.class);
```

Par xml :

Et on est besoin d'un fichier xml




```
<bean id="daoImpl" class="com.dao.DaoImpl"> </bean>

<bean id="metierImpl" class="com.metier.MetierImpl">
    <property name="dao" ref="daoImpl" />
</bean>

</beans>
```

- **Par annotaion :**

```
@Component("metier")
public class MetierImpl implements IMetier {
        @Autowired
```

```
@Component
public class DaoImpl implements IDao {
```

Nous utilise **@component** pour la création des objets (métier, dao). Et nous ajoute **@Autowired** dans metier pour trouver l'objet dao et comme ça Spring connecte les différentes couches de votre application automatiquement.
Nous avons ensuite modifié la méthode init() en y ajoutant le code suivant, avec un bloc try/catch

```
ApplicationContext ctx = new AnnotationConfigApplicationContext("com");
this.metier = ctx.getBean(IMetier.class);
```

Cela permet à Spring de charger le contexte, de détecter automatiquement les classes annotées et de récupérer l'objet métier.

Le lien GitHub : https://github.com/M-Aymane-404/Tps-javaEE