Mise en oeuvre des Servlets et des JSP

1 Le conteneur WEB Tomcat d'Apache (5m)

- Avec les commandes ci-dessous (à copier-coller dans un terminal), vous allez récupérer et désarchiver le conteneur WEB Tomcat [1] :

```
cd "$HOME"
wget http://tinyurl.com/jlmassat2/jee/ress/apache-tomcat-10.1.19.zip
unzip apache-tomcat-10.1.19.zip
chmod u+x apache-tomcat-10.1.19/bin/*.sh
export TOMCAT=$HOME/apache-tomcat-10.1.19
```

- **Tester Tomcat (travail à faire plus tard)** :
  - ▷ Lancez le serveur avec le script `$TOMCAT/bin/startup.sh`.
  - ▷ Testez ensuite les exemples présents en vous connectant à l'adresse `http://localhost:8080`. **Attention** : commencez par les exemples de servlets puis les exemples en JSP 1.2 et laissez les exemples JSP 2.0 pour plus tard.
  - ▷ **N'oubliez pas**, à la fin de cet exercice, de stopper Tomcat (commande `$TOMCAT/bin/shutdown.sh`).

2 Eclipse JEE et les applications WEB (20m)

Nous allons utiliser le plugin `WTP` [2] qui est intégré par défaut dans Eclipse pour JEE. Pour ce faire, suivez ces étapes **les unes après les autres** :

1. Lancer la version JEE de Eclipse [3].
2. Dans le menu « **Windows/Preferences** » choissisez l'onglet « **Server / Runtime Environments** » et ajoutez un nouveau **Runtime** de type **Apache Tomcat 10.1**.
3. Repérez la vue **Server** (menu « **Windows / Show view** ») et créez un nouveau serveur (avec le menu contextuel dans la vue **Server**) basé sur le **Runtime Tomcat 10** précédemment créé.
4. Créez un nouveau projet de type « **Web / Dynamic Web project** » :
   - **Écran 1** : Fixez le nom `myapp`, choisissez le **Target Runtime** Apache Tomcat 10.1 et la version 6.0 du module.
   - **Écran 2** : Configuration du répertoire source (rien à faire).
   - **Écran 3** : Configuration du contexte et du répertoire contenant les ressources WEB (rien à faire). **À ce stade, il faut** choisir l'option « Generate web.xml » afin qu'eclipse prépare automatiquement le fichier de configuration.
5. Convertissez votre projet à Maven : **Sélectionnez votre projet** / **Bouton-droit** / **Configure** / **Convert to Maven Project**. Vous devez à cette étape donner une numéro de version à votre projet. Laissez les valeurs par défaut.
6. Ajoutez les dépendances ci-dessous dans le fichier `pom.xml` après `</build>` :

1. http://tomcat.apache.org/
2. http ://www.eclipse.org/webtools/
3. http://www.eclipse.org/

```xml
<dependencies>
    <!-- Pour utiliser Lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.30</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

7. À ce stade, votre projet doit ressembler à ceci :

```
myapp
  | Deployment Descriptor     version agréable de web.xml
  | JAX-WS Web Services       pas nécessaire dans l'immédiat
  | Java Ressources
  |   | src/man/java          les sources Java
  |   | Librairies
  |   |   | JRE System...     la JRE
  |   |   | Maven Dependencies les librairies gérées par Maven
  |   |   | Server Runtime (Tomcat...) les librairies de Tomcat
  | Deployed Resources
  |   | webapp                votre application WEB
  |   |   | META-INF/         le manifest
  |   |   | WEB-INF           configuration de votre app.
  |   |   |   | lib/          les librairies de votre app
  |   |   |   | web.xml       configuration de votre app
  |   | web-resources         les ressources
  | build/                    zone de travail
  | src/                      une autre vue des sources
  | target/                   zone de travail maven
  | pom.xml                   fiche de configuration maven
```

8. Créez dans le répertoire `webapp` une page JSP et nommez-la `index.jsp`. Elle doit ressembler à ceci

Fichier `webapp/index.jsp`

```jsp
<%@ page language="java"
    contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Title</title>
</head>
<body>
  <p>Hello.</p>
</body>
</html>
```

9. Sélectionnez votre projet et exécutez-le sur le serveur Tomcat (menu « **Run as... / Run on server** »). Faites en sorte d'associer définitivement votre projet et le serveur Tomcat (une petite case à cocher avant le lancement).

10. À ce stade, vous devez pouvoir accéder à votre application via l'URL (`http://localhost:8080/myapp/`). Vous pouvez maintenant ajouter des pages HTML, JSP et/ou des sources Java dans votre projet.

3 Mon premier bean (20m)

- Modifiez la page précédente pour qu'elle affiche la date du jour et l'heure à chaque exécution (créez pour cela une instance de la classe `java.util.Date` et affichez la).

```
<%@page import="java.util.Date" %>

<%!
Date now = new Date();
%>

<p>Aujourd'hui : <%= now %></p>
```

- Quel comportement anormal observez-vous ? Essayez cette nouvelle version :

```
<%@page import="java.util.Date"%>

<p>Aujourd'hui : <%= new Date() %></p>
```

- Finalement, l'introduction de code Java dans les JSP est maintenant **fortement** déconseillée. Nous allons préférer la forme ci-dessous basée sur les EL (**langage d'expressions**) :

```
<jsp:useBean id="now" class="java.util.Date" />

<p>Aujourd'hui : ${now}</p>
```

- Faites tourner les exemples présentés en cours [4] d'affichage et de manipulation d'un produit.
- Faites varier la portée (le `scope`) du `<jsp:useBean>` et observez les différences dans les trois cas [5] (`request`, `session`, `application`).
  - ▷ **Conseil 1** : Vous pouvez placer du code JSP à l'intérieur de l'action `<jsp:useBean>`. Ce code est exécuté lorsque le bean est créé.
  - ▷ **Conseil 2** : Pour bien manipuler les *beans* de portée `session`, utilisez plusieurs navigateurs (firefox ou chrome) ou une fenêtre de navigation privée.
  - ▷ **Conseil 3** : Faites en sorte que votre bean implante l'interface `HttpSessionBindingListener` [6] et mettez en place des traces pour suivre la vie de ce bean. Dans un deuxième temps, diminuez la durée de vie des sessions (ajoutez le code ci-dessous dans le fichier `web.xml`) afin d'observer la suppression automatique des beans (**attention** : c'est loin d'être systématique).

```
...
<session-config>
   <session-timeout>1</session-timeout><!-- une minute -->
</session-config>
...
```

4. jsp.html#javabean
5. servlet.html#scope
6. servlet.html#HttpSessionBindingListener

4 Une petite application (1h40)

4.1 Étape 1 : La servlet

- Créez une servlet `PersonServlet` que ne réalise aucune action (pour l'instant) et qui est associée à l'URL `/myapp/person` (code ci-dessous).

```
package myapp;

import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

/**
 * Une servlet pour les actions sur les personnes.
 */
@WebServlet(//
        description = "Les actions sur les personnes", //
        urlPatterns = { "/person" })
public class PersonServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * Requêtes GET
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) //
            throws IOException {
        var writer = response.getWriter();
        writer.printf("<p>");
        // afficher les informations sur la requête
        writer.printf("method = %s</br>", request.getMethod());
        writer.printf("contextPath = %s</br>", request.getContextPath());
        writer.printf("servletPath = %s</br>", request.getServletPath());
        // afficher les paramètres et leurs valeurs
        writer.printf("'a' parameter = %s</br>", request.getParameter("a"));
        request.getParameterMap().forEach((param, values) -> {
            writer.printf("parameter '%s' = %s</br>", param, String.join(", ", values));
        });
    }

    /**
     * Requêtes POST (la même chose que GET)
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response) //
            throws ServletException, IOException {
        doGet(request, response);
    }

}
```

- Testez ce composant avec plusieurs requêtes `/myapp/person`, `/myapp/person?aa=Hello&bb=Salut&aa=Bye` afin de bien comprendre son fonctionnement.

4.2 Étape 2 : les données

- Créez un « bean » `Person` représentant une personne :

```java
package myapp;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Person {

    private String id;
    private String name;
    private String mail;

    public Person(Person p) {
        this(p.id, p.name, p.mail);
    }

}
```

- Modifiez votre projet afin d'ajouter une classe métier orientée vers le traitement des personnes. Préparez ensuite une instance de `PersonManager` dans la servlet. Si nous avions Spring, nous aurions pu demander une injection.

```java
package myapp;

import java.util.Collection;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class PersonManager {

    final private Map<String, Person> persons;

    public PersonManager() {
        persons = Collections.synchronizedMap(new HashMap<>());
        save(new Person("100", "Paul", "paul@hello.fr"));
        save(new Person("200", "Laure", "laure@univamu.fr"));
    }

    public Person find(String id) {
        var p = persons.get(id);
        if (p == null) throw new IllegalArgumentException();
        return new Person(p);
    }

    public Collection<Person> findAll() {
        return persons.values().stream().map(p -> new Person(p)).toList();
    }

    public void save(Person p) {
        persons.put(p.getId(), new Person(p));
    }

    public void remove(String id) {
        persons.remove(id);
    }

    public Map<String, String> validate(Person p) {
        var errors = new HashMap<String, String>();
        if (p.getId() == null || p.getId().isBlank()) {
            errors.put("id", "ID␣incorrect");
        }
        if (p.getMail() == null || !p.getMail().matches("^[a-z0-9.]+@[a-z0-9.]+")) {
            errors.put("mail", "e-mail␣incorrect");
        }
        if (p.getName() == null || p.getName().isBlank()) {
            errors.put("name", "Le␣nom␣est␣obligatoire");
        }
        return errors;
    }

}
```

4.3 Étape 3 : lister

Nous allons traiter la requête `GET /myapp/person` afin de lister les personnes.

- Créer une page JSP `lister.jsp` qui a pour but de lister une collection de personnes rangée dans la zone `request` :

```jsp
<%@page import="java.util.List"%>
<%@page import="myapp.Person"%>

<jsp:useBean id="persons" type="List<Person>" scope="request" />

<html>
<body>

<h1>Liste des personnes</h1>

<table border='1'>
    <%
    for (Person person: persons) {
        pageContext.setAttribute("person", person);
        %>
        <tr>
            <td>${person.id}</td>
            <td>${person.name}</td>
            <td>${person.mail}</td>
        </tr>
        <%
    }
%>
</table>
</body>
</html>
```

- Dans la servlet, modifier `doGet` afin de récupérer la liste des personnes, la mettre dans la zone `request` et appeler la page JSP qui va l'afficher :

```java
request.setAttribute("persons", manager.findAll());
// Appeler une page JSP depuis une servlet
request.getRequestDispatcher("lister.jsp").forward(request, response);
```

- Essayez sur un navigateur la requête `http://localhost:8080/myapp/person`.
- **Moralité** : Les requêtes sont systématiquement traitées par les servlets et les pages JSP affichent les données préparées par les servlets.

4.4 Étape 4 : modifier

Nous allons maintenant traiter la requête `GET /myapp/person?id=100` afin de modifier la personne en question.

- Modifier `lister.jsp` afin d'ajouter un lien de modification :

```jsp
<td><a href="person?id=${person.id}">Modifier</a></td>
```

- Créez une page JSP `edition.jsp` afin de produire un formulaire HTML d'édition des caractéristiques d'une personne placée dans la zone `request`. La soumission de ce formulaire va générer la requête `POST /myapp/person`.

```
<jsp:useBean id="person" type="myapp.Person" scope="request" />

<html><body>
<h1>Modifier une persone</h1>

<form method="post" action="person">
    <label>ID :</label>
    <input name="id" type="text" value="${person.id}"/> <br/>
    <label>Name : </label>
    <input name="name" type="text" value="${person.name}"/> <br/>
    <label>Mail : </label>
    <input name="mail" type="text" value="${person.mail}"/> <br/>
    <label>Validation :</label>
    <input name="ok" type="submit" value="Ok"/>
</form>
</body></html>
```

- Dans la Servlet : modifier `doGet` afin de
  - ▷ tester la présence du paramètre `id`,
  - ▷ charger la personne,
  - ▷ la placer en zone `request`,
  - ▷ appeler la JSP `edition.jsp`.
- Tester le bon fonctionnement du lien de modification.

4.5 Étape 6 : traiter

Nous allons maintenant traiter la requête `POST /myapp/person`.

- Modifier la méthode `doPost` afin de
  - ▷ créer une instance de `Person`,
  - ▷ affecter cette instance avec les paramètres de la requête HTTP (`id`, `name`, `mail`),
  - ▷ sauver cette instance avec le manager,
  - ▷ effectuer une redirection vers `GET /myapp/person` avec `response.sendRedirect("person");`
- Tester le bon fonctionnement.
- Vérifier que les injections JavaScript fonctionnent (malheureusement). Nous réglerons ce problème plus tard.

Problème : Vérifier que le changement d'ID provoque la création d'une nouvelle personne. Afin d'éviter ce problème, faites en sorte de placer en session (avec `request.getSession().setAttribute("name",value)` la valeur de l'ID sur la requête `GET` pour pouvoir le vérifier dans la requête `POST`.

4.6 Étape 7 : valider et enregistrer

- Dans la Servlet : Ajoutez une phase de validation des données (méthode `validate` du manager).
  - ▷ Si les données ne sont pas valides, faites en sorte de revenir au formulaire en proposant les anciennes valeurs.
  - ▷ Si les données sont valides, enregistrez l'instance indexée par l'ID de la personne.
- Modifiez la page `edition.jsp` et la servlet afin de faire apparaitre les messages d'erreur (en rouge) à coté des champs fautifs. Pour ce faire, vous devez placer dans la zone `request` la `Map` des erreurs.

4.7 Étape 8 : ajouter

- Ajoutez à votre page `lister.jsp` le lien ci-dessous afin de pouvoir ajouter une nouvelle personne :

```
<p><a href="person?ajout=1">Ajouter une personne</a></p>
```

- Modifier la méthode `doGet` pour traiter ce cas.
- **Important** : Nous avons déjà placé en session l'ID de la personne modifiée. Faites en sorte de ne rien mettre en session si il s'agit d'un ajout. Modifier la méthode `doPost` pour traiter les ajouts.

4.8 Étape 9 : supprimer

Modifiez votre page `lister.jsp` et votre servlet pour ajouter et traiter le cas de la suppression :

```
<a href="deletePerson?id=123456">Supprimer cette personne</a>
```

Prévoyez que votre servlet traite également l'URL `/myapp/deletePerson` en modifiant l'annotation :

```
@WebServlet(//
        description = "Les␣actions␣sur␣les␣personnes", //
        urlPatterns = { "/person", "/deletePerson" })
```

La servlet utilisera la méthode `request.getServletPath()` pour savoir sous quel nom elle a été appelée.

4.9 Étape 10 : utiliser les routes

Afin de rendre les URL plus simples, remplacer `/person?ajout=1` par `/addPerson`.