



TECHNOLOGIES

- Servlets

COMPETENCES

- Structurer une application web java
- Mettre en place une Servlet
- Utiliser la classe HttpServlet
- Découvrir les différents contextes d'une application Java
- Lire une réponse http
- Ecrire une réponse http

SOMMAIRE

Table des matières

3.1. Introduction aux servlets	2
3.2. La structure d'une application web	2
3.3. Le projet	6
3.4. Les servlets.....	9
3.5 La classe HttpServlet.....	18
3.6 Les différents contextes	18
3.7 La lecture de la requête	20
3.8. La creation de la réponse	27

3.1. INTRODUCTION AUX SERVLETS

Dans l'environnement **Java**, une **application web** est une collection de servlets, de pages **HTML**, de classes et de toutes autres ressources utiles à la bonne exécution de l'application (des fichiers **CSS**, des fichiers **JavaScript**, des fichiers images...).

Une servlet est un composant web de la technologie **Java**. Une **servlet** permet de générer un contenu dynamique. Elle est gérée par un conteneur appelé généralement **conteneur web**, **conteneur de servlet** ou **moteur de servlets**.

Ce conteneur est une extension à un **serveur web** et propose les fonctionnalités indispensables au fonctionnement des servlets (décoder les requêtes, générer les réponses, gérer le **cycle de vie des servlets**).

La servlet est l'élément central. Il ne faut cependant pas oublier d'autres composants importants :

- Les filtres,
- Les événements,
- Les cookies,
- Les sessions.

3.2. LA STRUCTURE D'UNE APPLICATION WEB

3.2.1. PRESENTATION

Une application web, quelle que soit la technologie serveur utilisée, permet de délivrer des réponses à des requêtes HTTP. Un ensemble de langages est commun à tous les environnements :

- **HTML** (HyperText Markup Language) : langage incontournable dans le développement web. Il permet de présenter le résultat à l'utilisateur.
- **CSS** (Cascading Style Sheet) : langage permettant d'améliorer la présentation HTML en définissant des règles de représentation (coloration, police d'écriture, disposition...).
- **JavaScript** : langage permettant d'améliorer l'expérience utilisateur en apportant des traitements plus ou moins complexes côté client. Il peut, aujourd'hui, aussi être utilisé côté serveur à l'instar du Java.
- **SQL** (Structured Query Language) : langage permettant d'accéder aux données si la base de données sous-jacente est une base de données relationnelles.

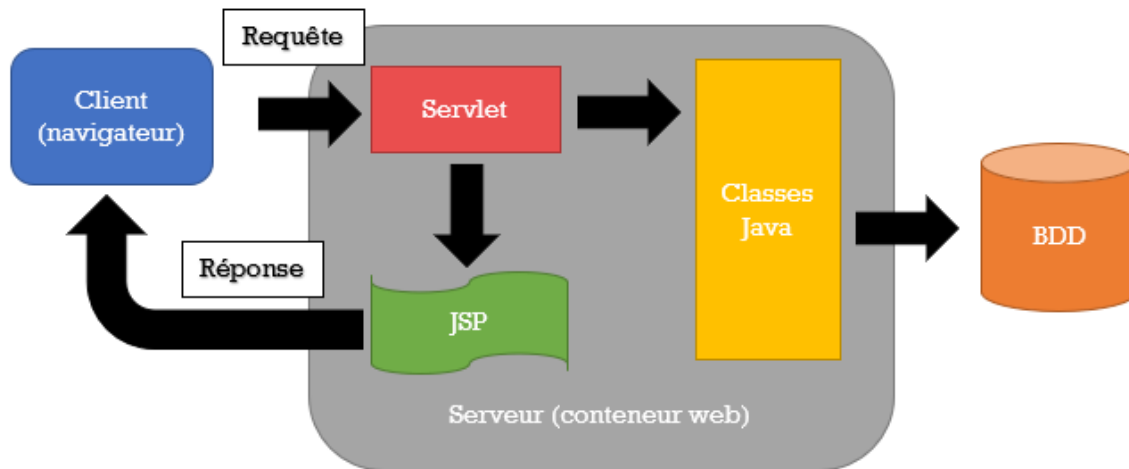
D'autres langages/technologies peuvent être utilisés comme le **XML**...

Pour rendre la génération des réponses dynamiques, il est nécessaire d'utiliser une technologie prévue à cet effet. Parmi les plus connues, citons :

- Le **PHP**,
- Le **C#** et le **VB.NET** de l'environnement Microsoft,
- Le **Java**.

3.2.2. STRUCTURE LOGIQUE D'UNE APPLICATION WEB

Ci-dessous, un schéma montrant la communication entre le client et le serveur, et la manière classique utilisée par le serveur pour restituer une réponse générée dynamiquement :



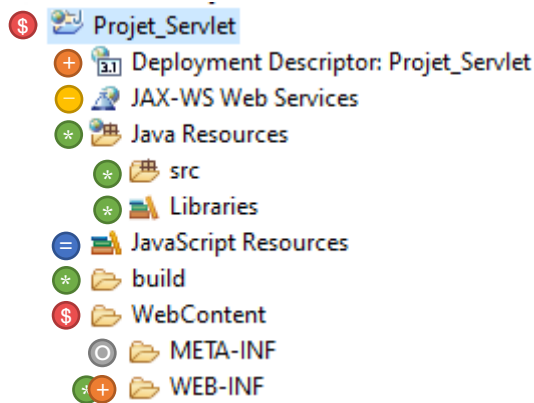
Une servlet répond à une requête **HTTP** émise par un client. Elle utilise des classes **Java** pour calculer la réponse. Lorsque toutes les informations sont obtenues, la servlet demande à une page **JSP (Java Server Page)** de constituer, mettre en forme, une réponse exploitable par le client.

Cette architecture mise en place est classique. Il s'agit d'une architecture **MVC (modèle, vue, contrôleur)**. Le contrôleur est la servlet, la vue est la page **JSP** et le modèle est les classes Java. Le rôle de la servlet est donc de réceptionner et de comprendre la demande de l'utilisateur. La **servlet** demande ensuite aux classes du modèle de faire le traitement attendu. Pour finir, la servlet demande à la vue adaptée de générer la réponse.

Nous allons commencer par mettre en forme nous même la réponse, via les **servlets**, pour comprendre le fonctionnement global.

3.2.3 STRUCTURE PHYSIQUE D'UNE APPLICATION WEB

Pour être exécutable sur un serveur, une application web doit respecter une certaine structure, en java ou n'importe quel autre langage. Nous retrouverons les répertoires, obligatoires et inaccessibles directement par le client, **META-INF** et **WEB-INF**, ainsi que le dossier racine de l'application (dans l'exemple **Projet_Servlet**). Voici l'arborescence d'un projet **Java** :



Le répertoire racine de l'application est généralement représenté par le répertoire

- \$ **WebContent**, en **Java**, c'est un dossier à part. Il possède les deux sous-répertoires **META-INF** et **WEB-INF**. Au moment du déploiement, ce répertoire est renommé avec le nom du projet (**Projet_Servlet** dans l'exemple).

- + **Deployment Descriptor** : ce nœud permet d'avoir une vue récapitulative du descripteur de déploiement. Le descripteur de déploiement est un fichier de paramétrage nommé **web.xml** situé dans le répertoire **WEB-INF**.

- **JAX-WS Web Services** : ce nœud permet d'administrer les services web.

- * **Java Resources** possédant deux sous-éléments.

- * ► Un répertoire **src** dans lequel les classes **Java** et notamment les servlets sont écrites. Le résultat de la compilation de ce répertoire est écrit dans le sous-répertoire classes du répertoire **WEB-INF**, ainsi que dans le répertoire **build**.

- * ► Un nœud **Libraries** décrivant l'ensemble des librairies utilisées par l'application. Les librairies qui ne sont pas dans le classpath du **JDK** ou dans le répertoire **lib** de **Tomcat** sont automatiquement copiées dans le sous-répertoire **lib** du répertoire **WEB-INF**.

- = **JavaScript Resources** : ce nœud permet de gérer les scripts... JavaScript.

- Le répertoire **META-INF** contient le fichier **MANIFEST.MF** décrivant les caractéristiques de l'application. Ce répertoire peut contenir d'autres fichiers de paramétrage notamment pour le déploiement de l'application.

3.2.4 LE DESCRIPTEUR DE DEPLOIEMENT

Une application web nécessite d'être paramétrée. Jusqu'à la version 2.5 de la spécification des servlets, ce paramétrage, ne passait que par un fichier xml, le fichier web.xml (nommé descripteur de déploiement). Depuis, la version 3.0, le paramétrage par annotation est devenu possible.

Le fichier web.xml contient initialement le code suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID" version="3.1">
  <display-name>Projet_Servlet</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Le nœud racine <web-app> montre que ce fichier doit respecter une structure décrite par un schéma XSD (XML Schema Definition). Ce schéma, comme le namespace l'indique (http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd), est lié à la version 3.1 de la spécification des servlets.

Ce fichier ne peut donc être copié tel quel dans un projet de version inférieure.

Le nœud <welcome-file-list> permet d'indiquer le fichier à utiliser par ordre de préférence si une requête HTTP n'indique aucune ressource particulière à retourner. Cela correspond à la page par défaut. Voici les principaux éléments de paramétrage disponibles :

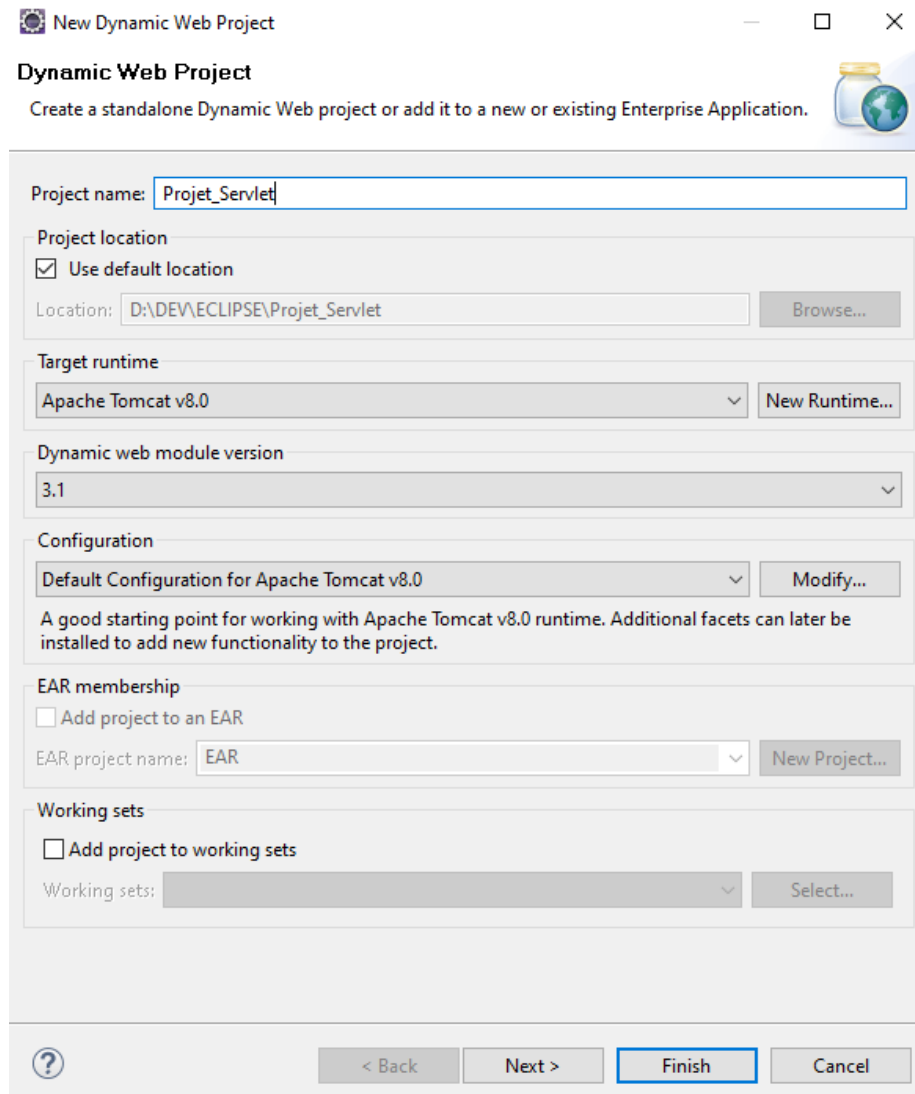
- La gestion du cycle de vie de l'application,
- La gestion des paramètres globaux,
- La gestion des servlets,
- La gestion des paramètres des servlets,
- La gestion des filtres,
- La gestion des sessions,
- La gestion des pages d'accueil,
- La gestion des pages d'erreur,
- La gestion de la sécurité...

Les annotations permettent de limiter l'usage du descripteur de déploiement notamment pour la gestion des servlets et des filtres. Les annotations sont présentées au moment où leur usage est possible.

3.3. LE PROJET

Commencez par créer un **Dynamic Web Project** avec les mêmes caractéristiques que dans le chapitre précédent :

- Apache Tomcat v8.0 comme Target runtime
- jdk1.8.x comme JRE
- La version 3.1 pour le Dynamic web module version

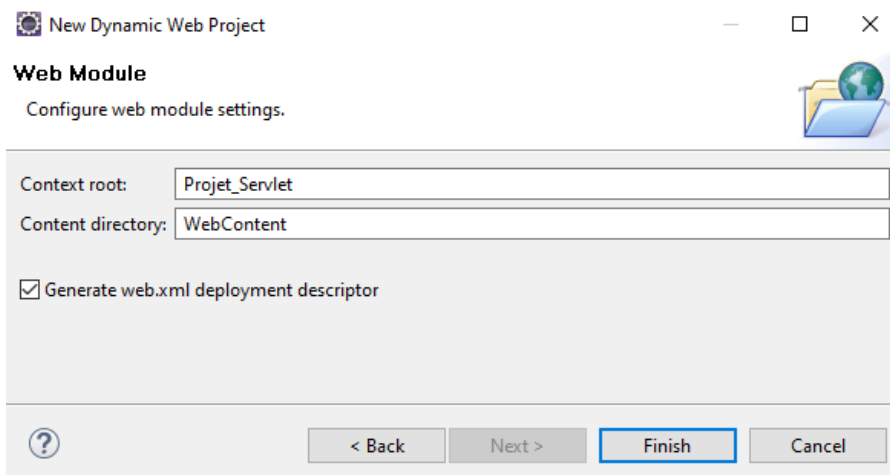


The screenshot shows the 'New Dynamic Web Project' dialog box in Eclipse. The title bar reads 'New Dynamic Web Project'. Below the title bar, the text 'Dynamic Web Project' is followed by a description: 'Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.' and a small icon of a globe with a jar. The dialog is divided into several sections:

- Project name:** A text field containing 'Projet_Servlet'.
- Project location:** A section with a checked checkbox 'Use default location' and a text field 'Location:' containing 'D:\DEV\ECLIPSE\Projet_Servlet'. A 'Browse...' button is to the right.
- Target runtime:** A dropdown menu showing 'Apache Tomcat v8.0' and a 'New Runtime...' button.
- Dynamic web module version:** A dropdown menu showing '3.1'.
- Configuration:** A dropdown menu showing 'Default Configuration for Apache Tomcat v8.0' and a 'Modify...' button. Below this, a note states: 'A good starting point for working with Apache Tomcat v8.0 runtime. Additional facets can later be installed to add new functionality to the project.'
- EAR membership:** A section with an unchecked checkbox 'Add project to an EAR' and a text field 'EAR project name:' containing 'EAR'. A 'New Project...' button is to the right.
- Working sets:** A section with an unchecked checkbox 'Add project to working sets' and a text field 'Working sets:' which is empty. A 'Select...' button is to the right.

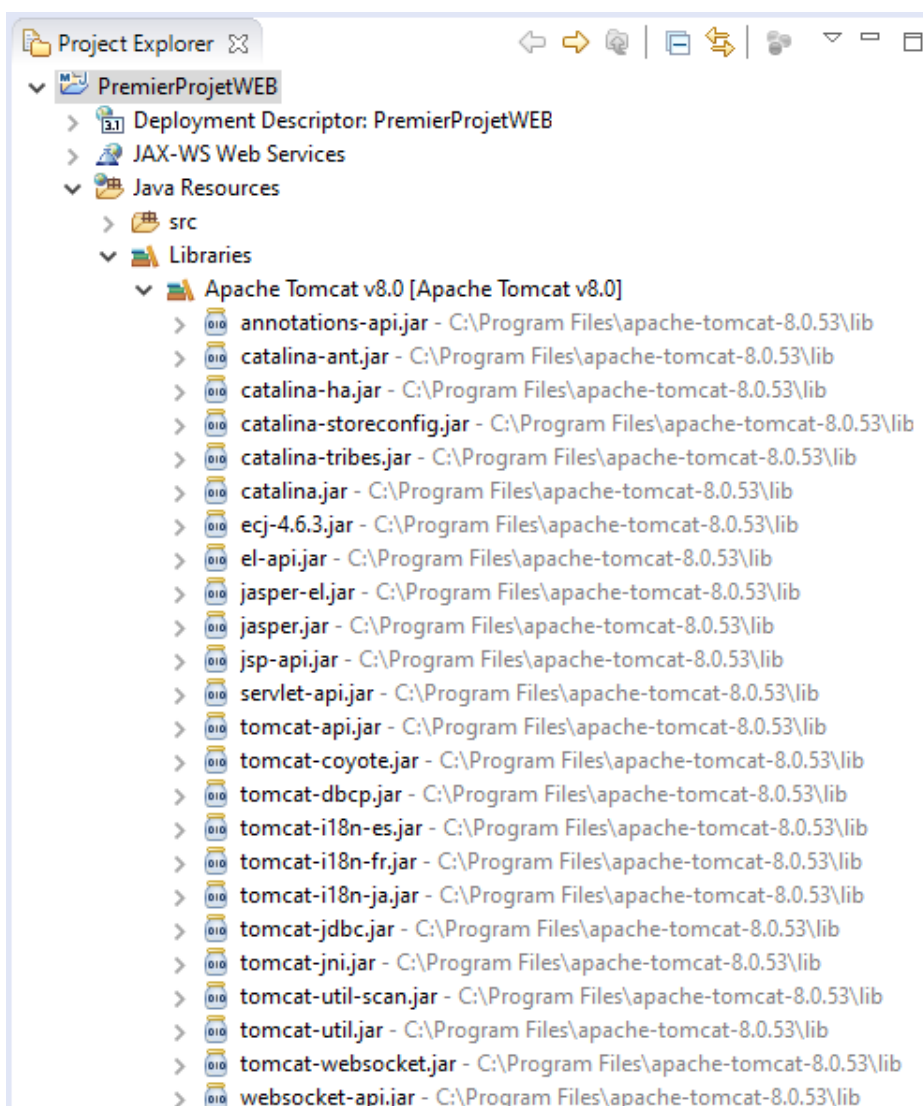
At the bottom of the dialog, there is a question mark icon, and four buttons: '< Back', 'Next >', 'Finish' (which is highlighted with a blue border), and 'Cancel'.

Cliquez deux fois sur **Next** pour passer à l'écran suivant :



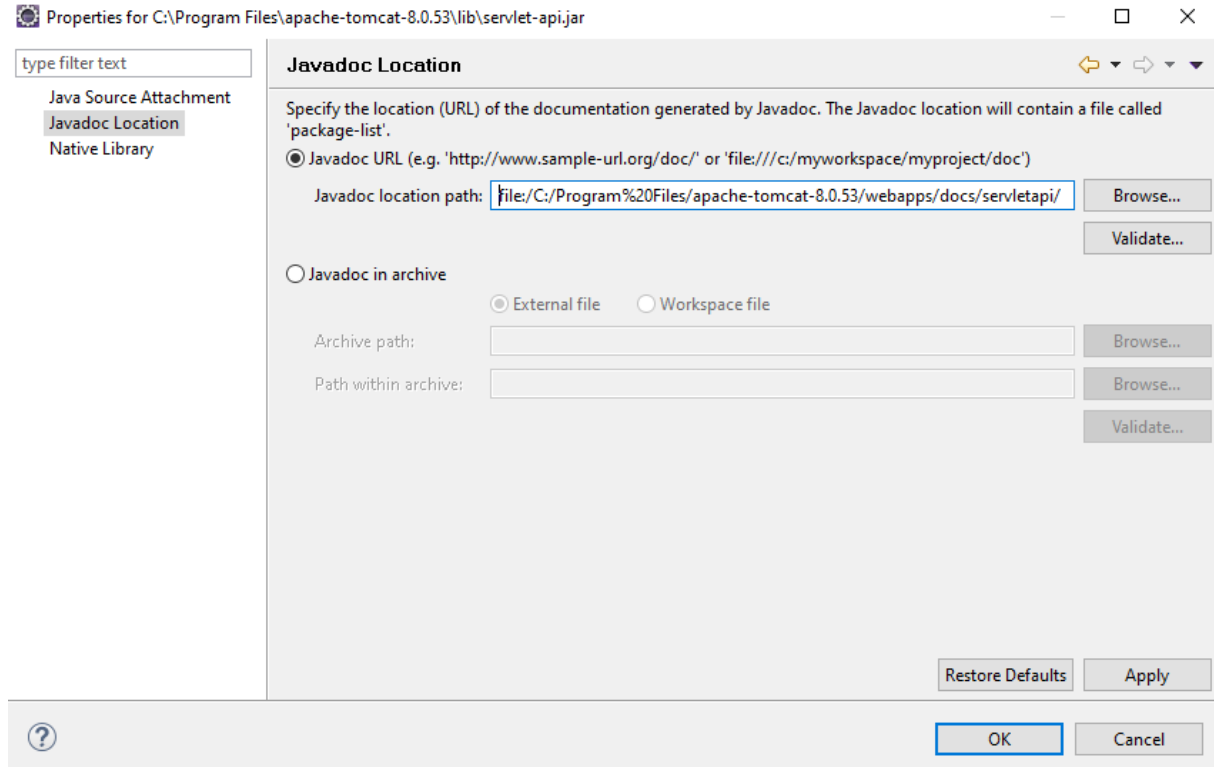
Cochez la case **Generate web.xml deployment descriptor** et cliquez sur **Finish**. Le projet est prêt.

Il ne reste plus qu'à référencer la Javadoc au travers d'Eclipse pour avoir une bonne expérience de développeur. Pour cela, veuillez suivre la procédure suivante :
Déployez le nœud Java Resources - Libraries - Apache Tomcat v8.0 du projet :



Visualisez le jar **servlet-api.jar**.

Faites un clic droit sur ce fichier pour accéder au menu **Properties**. Sélectionnez le nœud **Javadoc Location** et définissez le répertoire contenant la Javadoc de l'API. Ce répertoire est le sous-répertoire `servletapi` de la documentation Tomcat.



Cliquez sur **OK**.

Maintenant, lorsque vous développerez des servlets, la Javadoc vous permettra d'avoir la documentation nécessaire pour comprendre l'utilisation des différentes méthodes.

3.4. LES SERVLETS

3.4.1 INTRODUCTION

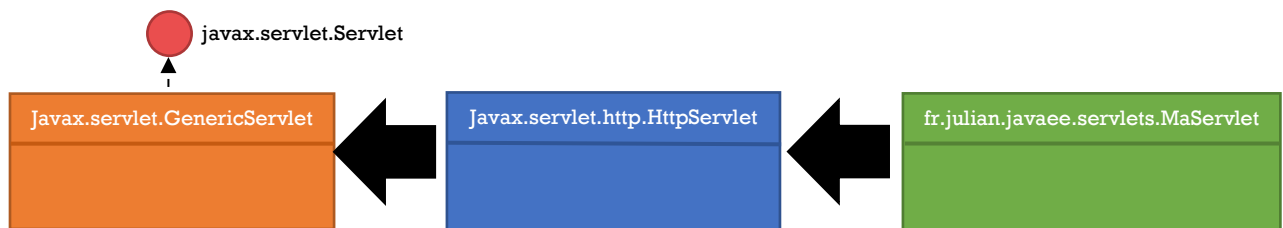
Une servlet est tout simplement une classe Java utilisée pour étendre les capacités d'un serveur qui héberge des applications accessibles au travers d'un modèle de programmation requête/réponse. Les servlets peuvent donc répondre à n'importe quel type de requêtes cependant, le cas le plus courant est le traitement de requêtes HTTP.

Dans le cadre d'une application web, une servlet est une classe Java accessible au travers d'une ou plusieurs URL HTTP.

Les packages `javax.servlet` et `javax.servlet.http` fournissent les interfaces et les classes nécessaires à l'écriture des servlets. Une servlet doit implémenter l'interface `Servlet` définissant les méthodes du cycle de vie d'une servlet. La classe abstraite `GenericServlet` implémente cette interface. La classe `HttpServlet` hérite de cette classe et ajoute les méthodes permettant de traiter les requêtes HTTP comme les méthodes `doGet(...)` et `doPost(...)` pour traiter les requêtes de type GET et de type POST.

La création d'une servlet consiste à créer une classe Java héritant de la classe `HttpServlet` et à substituer les méthodes nécessaires pour obtenir le comportement attendu.

Voici un schéma résumant cette architecture :



La classe `fr.julian.javaee.servlets.MaServlet` est un servlet permettant de traiter une requête http et de retourner une réponse HTTP.

3.4.2. CYCLE DE VIE

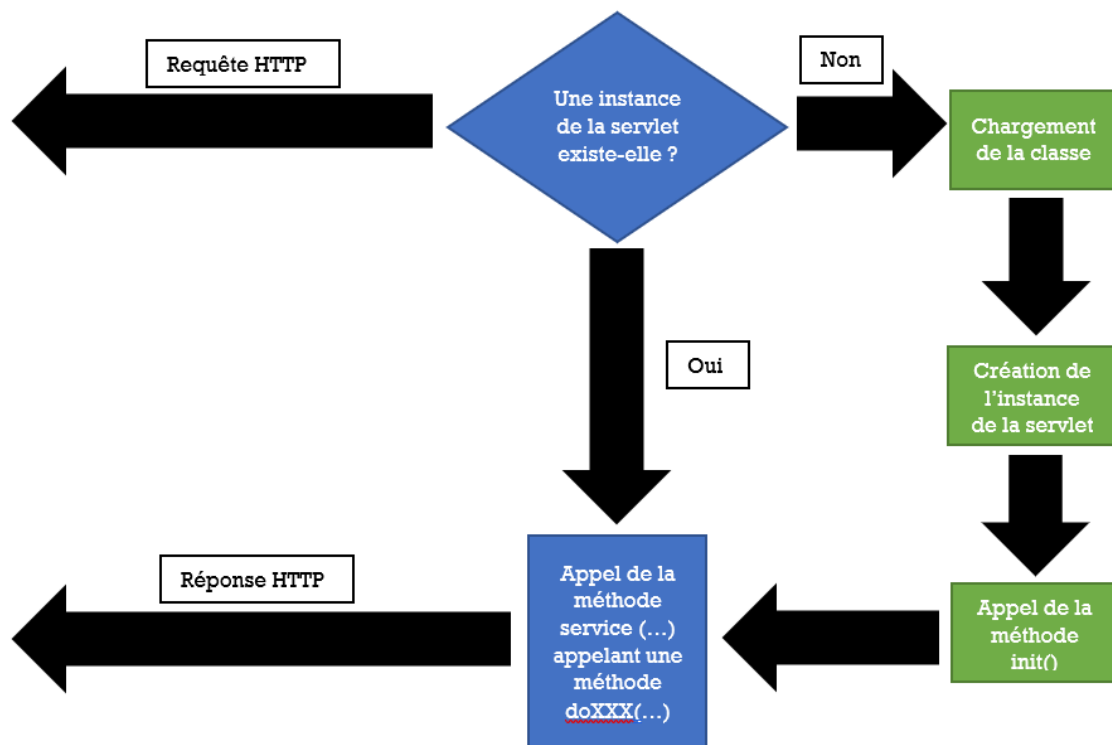
Le développeur n'a pas à créer lui-même d'instances de la servlet. Le cycle de vie d'une servlet est contrôlé par le conteneur dans lequel la servlet est déployée. Lorsqu'une requête HTTP nécessite l'exécution d'une servlet, le conteneur effectue les traitements suivants :

- Si aucune instance de la servlet n'existe, alors le conteneur :
 - Charge la servlet.
 - Crée une instance de la servlet.
 - Appelle la méthode `init()` pour initialiser d'éventuels paramètres.
- Le conteneur appelle la méthode `service(...)` prenant en paramètre deux objets représentant la requête (`HttpServletRequest`) et la réponse (`HttpServletResponse`) HTTP. Le rôle principal de cette méthode est de définir le type de la requête HTTP et d'appeler la méthode `doXXX(...)` adaptée :
 - `doGet(...)` pour les requêtes de type GET,
 - `doPost(...)` pour les requêtes de type POST,
 - `doPut(...)` pour les requêtes de type PUT,
 - `doDelete(...)` pour les requêtes de type DELETE,
 - `doHead(...)` pour les requêtes de type HEAD,
 - `doOptions(...)` pour les requêtes de type OPTIONS,
 - `doTrace(...)` pour les requêtes de type TRACE.

Lorsque le conteneur n'a plus besoin de la servlet (à l'arrêt du serveur par exemple) alors celle-ci est déchargée avec l'appel de la méthode `destroy()`.

Le constat de cette description est qu'une servlet n'est instanciée qu'une seule fois. Le conteneur utilise alors des threads différents pour traiter en parallèle les requêtes HTTP utilisant la même servlet. C'est un point important à retenir dans le cadre du développement d'une servlet.

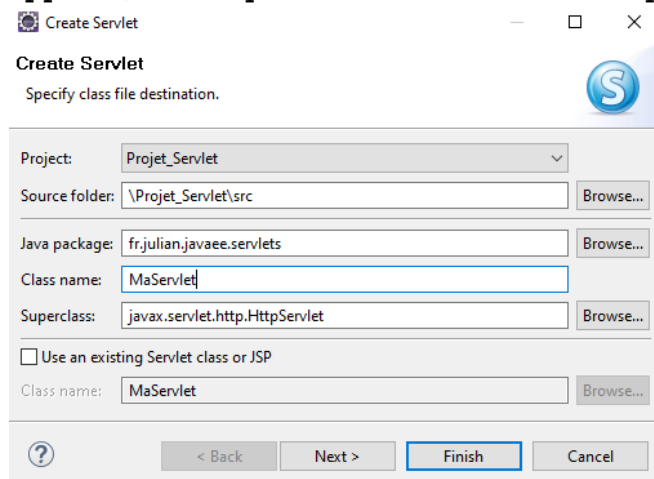
Voici un schéma récapitulant le traitement d'une requête HTTP par une servlet :



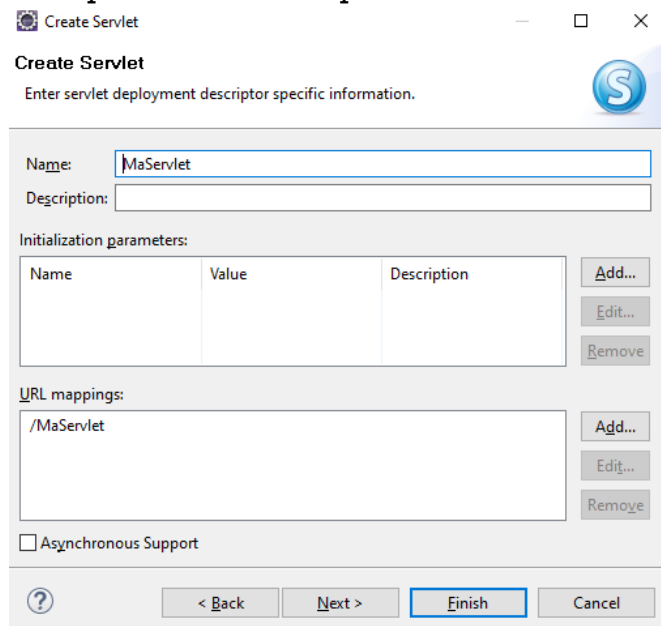
3.4.2. CREATION DE LA PREMIERE SERVLET

Il est temps de créer une première servlet. Voici la marche à suivre :

Faites un clic droit sur le projet et cliquez sur le menu **New - Servlet**. L'écran suivant apparaît, dans lequel vous saisissez un nom de package et un nom de classe :



Cliquez sur le bouton **Next** pour arriver sur l'écran suivant permettant de définir des paramètres d'initialisation et les URL pour lesquelles la servlet est utilisée. Par défaut l'URL paramétrée correspond au nom de la servlet.



Cliquez sur **Next** pour décider des méthodes de la classe mère à substituer.

Pour rappel, la classe mère est la classe `javax.servlet.http.HttpServlet`. Vous retrouvez les méthodes de l'interface `javax.servlet.Servlet` (`init`, `destroy`, `getServletConfig`, `getServletInfo`, `service`) et les méthodes spécifiques de la classe `HttpServlet` (`doGet`, `doPost`, `doPut`, `doDelete`, `doHead`, `doOptions` et `doTrace`).

Les méthodes `doGet` et `doPost` sont cochées par défaut. Elles correspondent aux types de requêtes traditionnellement manipulées dans une application web.

Create Servlet

Specify modifiers, interfaces to implement, and method stubs to generate.

Modifiers: ☒ public ☐ abstract ☐ final

Interfaces: Add... Remove

Which method stubs would you like to create?

☒ Constructors from superclass

☒ Inherited abstract methods

☐ init ☐ destroy ☐ getServletConfig

☐ getServletInfo ☐ service ☒ doGet

☒ doPost ☐ doPut ☐ doDelete

☐ doHead ☐ doOptions ☐ doTrace

? < Back Next > Finish Cancel

Cliquez sur **Finish** pour terminer. Votre première servlet est créée. Voici le code généré :

```
MaServlet.java
1 package fr.julian.javaee.servlets;
2
3 import java.io.IOException;
4
5 /**
6  * Servlet implementation class MaServlet
7  */
8 @WebServlet("/MaServlet")
9 public class MaServlet extends HttpServlet {
10     private static final long serialVersionUID = 1L;
11
12     /**
13      * @see HttpServlet#HttpServlet()
14      */
15     public MaServlet() {
16         super();
17         // TODO Auto-generated constructor stub
18     }
19
20     /**
21      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
22      */
23     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
24         // TODO Auto-generated method stub
25         response.getWriter().append("Served at: ").append(request.getContextPath());
26     }
27
28     /**
29      * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
30      */
31     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
32         // TODO Auto-generated method stub
33         doGet(request, response);
34     }
35 }
36
37
38
39
40
41 }
```

Voici les caractéristiques importantes de cette classe :

- La classe `MaServlet` hérite de la classe `HttpServlet`.
- Deux méthodes sont réécrites : `doPost(...)` et `doGet(...)`. Elles prennent en paramètre un objet de type `HttpServletRequest` et un objet de type `HttpServletResponse`. Ces objets sont créés par le conteneur pour faciliter la manipulation de la requête et la génération de la réponse. Dans l'exemple, la méthode `doPost(...)` appelle la méthode `doGet(...)` pour faire en sorte que le traitement soit le même pour les deux types de requêtes.
- Le corps de la réponse est écrit dans la méthode `doGet(...)` avec l'utilisation d'un objet de type `PrintWriter` obtenu à partir du paramètre `response` de type `HttpServletResponse`.
- Pour finir, la classe est annotée avec l'annotation `@WebServlet`. Cette annotation permet d'indiquer l'URL (relative à l'application) permettant d'accéder à cette servlet.

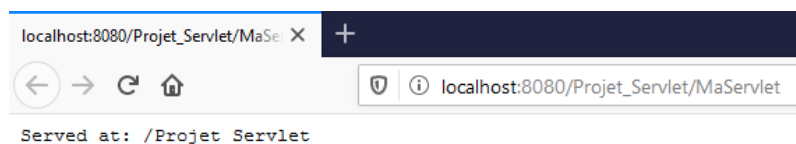
Pour accéder à cette servlet, l'URL d'accès est la suivante dans un environnement de développement :

http://localhost:8080/Projet_Servlet/MaServlet

Cette URL est composée des éléments suivants :

- `localhost` : un nom de domaine standard correspondant à la machine locale. L'adresse IP associée est `127.0.0.1`.
- `8080` : le port HTTP paramétré par défaut dans Tomcat.
- `Projet_Servlet` : le nom de l'application. Il correspond au nom du projet en phase de développement.
- `/MaServlet` : la ressource demandée.

À l'exécution, voici le résultat obtenu pour une requête de type GET :



3.4.3. LA DECLARATION D'UNE SERVLET

Pour utiliser une servlet dans une application, il faut réaliser deux opérations :

- Déclarer l'existence de la servlet.
- Déclarer les URL pour lesquelles la servlet est utilisée.

Ces deux opérations peuvent se faire dans le descripteur de déploiement ou par annotations. Eclipse propose un paramétrage automatique par annotations à partir de la version 3.0 de la spécification des servlets.

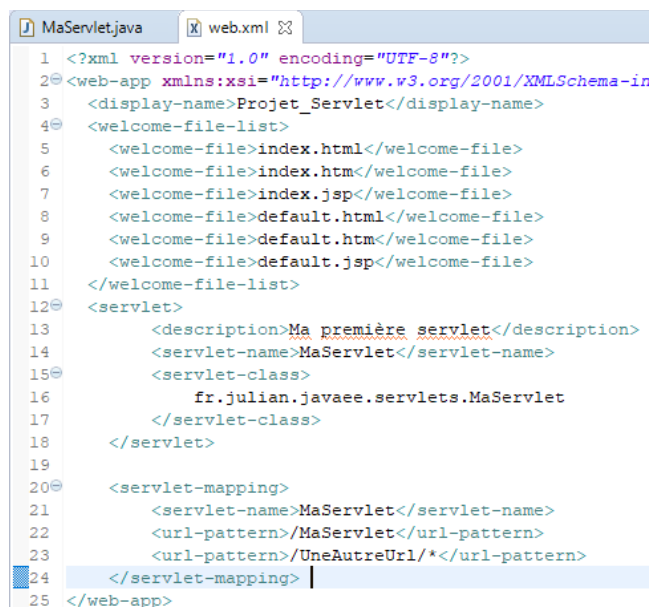
Paramétrage dans le descripteur de déploiement

La déclaration de l'existence de la servlet se fait avec l'aide de la balise `<servlet>`. Cette balise accueille plusieurs sous-balises permettant de décrire la servlet. Les balises présentées dans l'exemple ci-dessous sont les suivantes :

- `<description>` : cette balise permet de décrire succinctement le rôle de la servlet. C'est l'équivalent d'une Javadoc.
- `<servlet-name>` : cette balise permet de donner un nom logique à la servlet. Ce nom doit être unique.
- `<servlet-class>` : cette balise permet de déclarer la classe correspondant à la servlet.

La déclaration des URL associées se fait à l'aide de la balise `<servlet-mapping>`. Cette balise accueille les sous-balises suivantes :

- `<servlet-name>` : cette balise permet de faire référence à la balise `<servlet>` contenant une balise `<servlet-name>` avec la même valeur.
- `<url-pattern>` : cette balise peut être présente plusieurs fois si la servlet doit être associée à plusieurs URL. Le pattern commence par un slash (/). Attention, l'URL est sensible à la casse. La séquence `/*` est une séquence joker que l'on peut placer à la fin d'un pattern pour indiquer que la servlet est associée à toutes les URL commençant par `/UneAutreUrl/`.



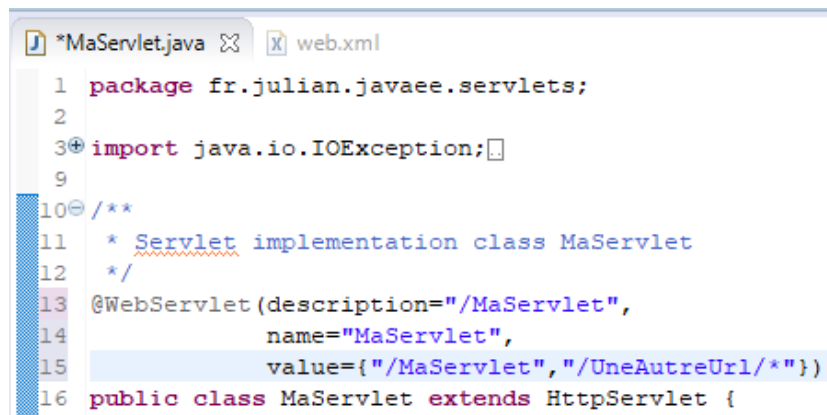
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
3 <display-name>Projet_Servlet</display-name>
4 <welcome-file-list>
5 <welcome-file>index.html</welcome-file>
6 <welcome-file>index.htm</welcome-file>
7 <welcome-file>index.jsp</welcome-file>
8 <welcome-file>default.html</welcome-file>
9 <welcome-file>default.htm</welcome-file>
10 <welcome-file>default.jsp</welcome-file>
11 </welcome-file-list>
12 <servlet>
13 <description>Ma première servlet</description>
14 <servlet-name>MaServlet</servlet-name>
15 <servlet-class>
16 fr.julian.javaee.servlets.MaServlet
17 </servlet-class>
18 </servlet>
19
20 <servlet-mapping>
21 <servlet-name>MaServlet</servlet-name>
22 <url-pattern>/MaServlet</url-pattern>
23 <url-pattern>/UneAutreUrl/*</url-pattern>
24 </servlet-mapping>
25 </web-app>
```

Attention, les URL sont sensibles à la casse.

Paramétrage par annotations

Le paramétrage par annotations remplace le paramétrage décrit précédemment. Ce paramétrage est fait directement sur la classe constituant la servlet. L'annotation nécessaire est `@WebServlet`. Cette annotation prend plusieurs paramètres. Les paramètres nécessaires pour obtenir un paramétrage équivalent au paramétrage précédent sont les suivants :

- `description` : ce paramètre est l'équivalent de la balise `<description>`.
- `name` : ce paramètre est l'équivalent de la balise `<servlet-name>`.
- `value` : ce paramètre est l'équivalent des balises `<url-pattern>`. Ce paramètre attend un tableau de patterns.



```
1 package fr.julian.javaee.servlets;
2
3 import java.io.IOException;
4
5
6
7
8
9
10 /**
11  * Servlet implementation class MaServlet
12  */
13 @WebServlet(description="/MaServlet",
14             name="MaServlet",
15             value={"/MaServlet", "/UneAutreUrl/*"})
16 public class MaServlet extends HttpServlet {
```

3.4.4. LES PARAMETRES D'INITIALISATION D'UNE SERVLET

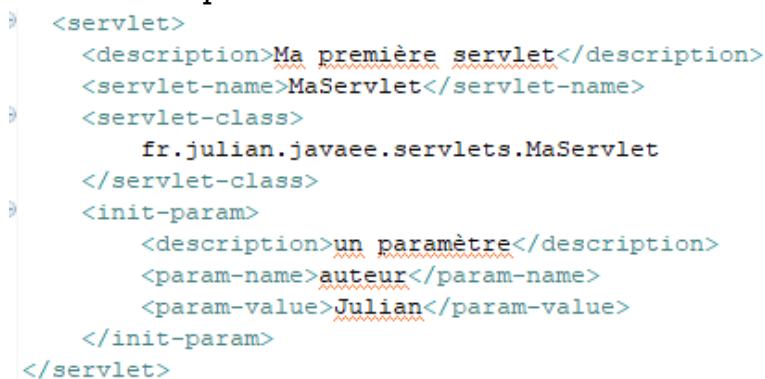
Une servlet peut avoir besoin d'informations complémentaires pour fonctionner. Au lieu d'écrire en dur ces informations dans le programme, il est plus judicieux de les rendre paramétrables. Ces paramètres peuvent être définis au travers du descripteur de déploiement ou au travers de l'annotation `@WebServlet`.

Paramétrage dans le descripteur de déploiement

La balise `<servlet>` accepte une sous-balise complémentaire `<init-param>`. Cette balise peut être présente plusieurs fois en cas de paramètres multiples. La balise `<init-param>` dispose de trois sous-balises :

- `<description>` : cette balise permet de décrire le paramètre.
- `<param-name>` : cette balise permet de nommer le paramètre.
- `<param-value>` : cette balise permet de valoriser le paramètre.

Voici un exemple de mise en œuvre :



```
<servlet>
  <description>Ma première servlet</description>
  <servlet-name>MaServlet</servlet-name>
  <servlet-class>
    fr.julian.javaee.servlets.MaServlet
  </servlet-class>
  <init-param>
    <description>un paramètre</description>
    <param-name>auteur</param-name>
    <param-value>Julian</param-value>
  </init-param>
</servlet>
```

Paramétrage par annotations

Il est possible de faire ce paramétrage au travers de l'annotation `@WebServlet` avec le paramètre `initParams`. Ce paramètre attend un tableau d'annotations `@WebInitParam`. Cette annotation possède trois paramètres `description`, `name` et `value` pour définir un paramètre.

```
@WebServlet(description="/MaServlet",
            name="MaServlet",
            value={"/MaServlet", "/UneAutreUrl/*"},
            initParams={@WebInitParam(description="un paramètre", name="auteur", value="Julian")})
public class MaServlet extends HttpServlet {
```

Utilisation du paramètre dans la servlet

La servlet possède deux méthodes permettant d'exploiter les paramètres d'initialisation d'une servlet :

- `getInitParameter(String name)` : cette méthode prend en paramètre le nom d'un paramètre et retourne la valeur sous la forme d'une chaîne de caractères.
- `getInitParameterNames()` : cette méthode retourne une collection de chaînes de caractères correspondant au nom des paramètres d'initialisation de la servlet.

La lecture des paramètres peut être faite uniquement lors de l'instanciation de la servlet. L'endroit privilégié est la méthode `init()` car elle est automatiquement appelée à la création de la servlet.

3.4.5. MISE EN EVIDENCE DU CYCLE DE VIE D'UNE SERVLET

Pour mettre en évidence le cycle de vie d'une servlet, il suffit de surcharger les méthodes `init()`, `service(...)` et `destroy()` de la servlet comme dans l'exemple suivant :

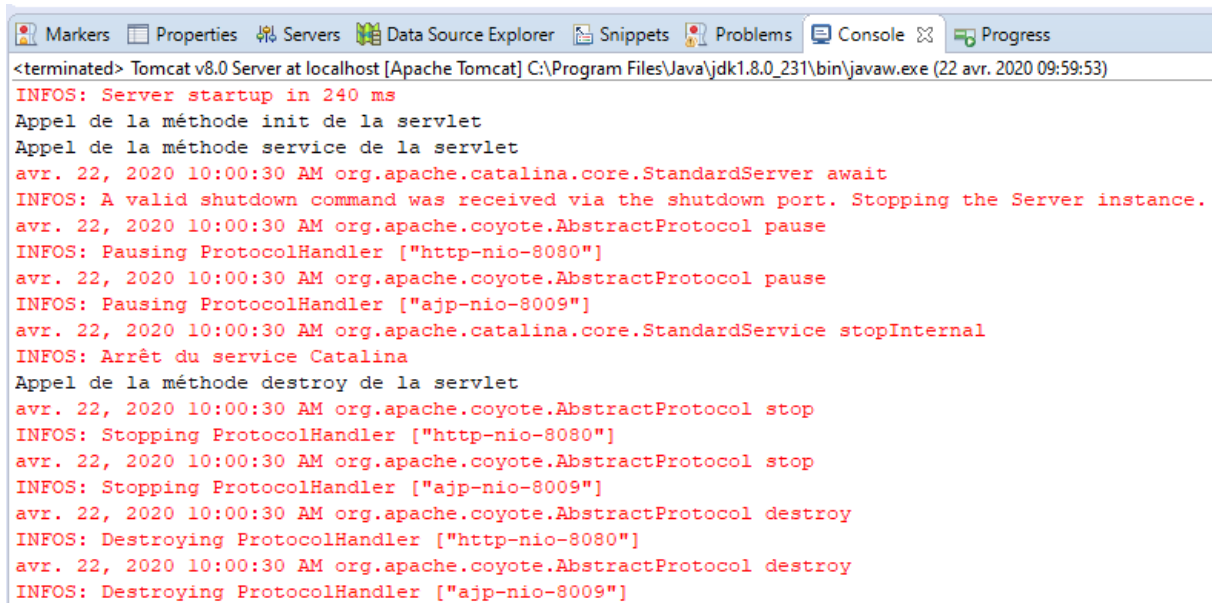
```
@WebServlet(description = "/MaServlet", name = "MaServlet", value = { "/MaServlet", "/UneAutreUrl/*" }, initParams = {
    @WebInitParam(description = "un paramètre", name = "auteur", value = "Julian") })
public class MaServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    //Attention les variables membres sont partagées par tous les threads
    private String auteur;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public MaServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
    @Override
    public void init() throws ServletException {
        System.out.println("Appel de la méthode init de la servlet");
        this.auteur = this.getInitParameter("auteur");
    }

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        System.out.println("Appel de la méthode service de la servlet");
        super.service(req, resp);
    }

    @Override
    public void destroy() {
        System.out.println("Appel de la méthode destroy de la servlet");
        super.destroy();
    }
}
```

Les différentes méthodes écrivent une ligne dans la console pour observer les appels. La méthode `init()` s'occupe aussi de lire le paramètre d'initialisation `auteur` pour le placer dans une variable membre de même nom.

À l'exécution, voici la trace obtenue sur la console :



```
<terminated> Tomcat v8.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (22 avr. 2020 09:59:53)
INFOS: Server startup in 240 ms
Appel de la méthode init de la servlet
Appel de la méthode service de la servlet
avr. 22, 2020 10:00:30 AM org.apache.catalina.core.StandardServer await
INFOS: A valid shutdown command was received via the shutdown port. Stopping the Server instance.
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol pause
INFOS: Pausing ProtocolHandler ["http-nio-8080"]
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol pause
INFOS: Pausing ProtocolHandler ["ajp-nio-8009"]
avr. 22, 2020 10:00:30 AM org.apache.catalina.core.StandardService stopInternal
INFOS: Arrêt du service Catalina
Appel de la méthode destroy de la servlet
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol stop
INFOS: Stopping ProtocolHandler ["http-nio-8080"]
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol stop
INFOS: Stopping ProtocolHandler ["ajp-nio-8009"]
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol destroy
INFOS: Destroying ProtocolHandler ["http-nio-8080"]
avr. 22, 2020 10:00:30 AM org.apache.coyote.AbstractProtocol destroy
INFOS: Destroying ProtocolHandler ["ajp-nio-8009"]
```

Le premier appel de la servlet provoque successivement l'appel des méthodes `init()`, `service(...)` et `doGet(...)`. Les appels suivants provoquent l'appel des méthodes `service(...)` et `doGet(...)` mais plus `init()` car la servlet a déjà été créée. Lorsque le serveur est arrêté, l'appel de la méthode `destroy()` est réalisé. C'est donc bien la même instance de servlet qui est utilisée pour répondre aux différentes requêtes.

3.5 LA CLASSE HTTPSERVLET

La classe `HttpServlet` est la classe de base pour créer les servlets. Elle met à disposition un certain nombre de méthodes. Quelques-unes ont déjà été présentées dans la section précédente :

- Les méthodes pour gérer le cycle de vie (`init()`, `service(...)` et `destroy()`).
- Les méthodes pour traiter les requêtes et apporter une réponse en fonction du type de la requête : `doGet(...)`, `doPost(...)`...
- Les méthodes pour exploiter les paramètres d'initialisation (`getInitParameter(...)` et `getInitParameterNames()`).

D'autres méthodes sont disponibles :

- `getServletConfig()` : cette méthode retourne un objet de type `ServletConfig`. Cet objet permet d'accéder aux mêmes éléments que les méthodes suivantes ainsi qu'aux paramètres d'initialisation abordés précédemment.
- `getServletContext()` : cette méthode retourne un objet de type `ServletContext`. Cet objet représente l'environnement dans lequel évolue la servlet. Cet objet permet d'accéder aux informations globales de l'application. On parle généralement de contexte d'application. La section suivante, Les différents contextes, apporte des détails sur son utilisation.
- `getServletInfo()` : cette méthode retourne une chaîne de caractères donnant des informations sur la servlet comme l'auteur et la version. Par défaut, la méthode retourne une chaîne vide. Pour obtenir un résultat différent, il suffit de la substituer.
- `getServletName()` : cette méthode permet d'obtenir le nom de la servlet défini par le paramètre `name` de l'annotation `@WebServlet`.

3.6 LES DIFFERENTS CONTEXTES

Lorsqu'une requête est prise en charge par une servlet, le traitement peut demander l'usage d'informations se situant dans différents contextes.

Le premier contexte est l'application. Les informations définies au niveau de l'application sont accessibles pour traiter toutes les requêtes de tous les clients. Les informations présentes au niveau de l'application ne doivent jamais être spécifiques à un client.

Le deuxième contexte est la session. Les informations contenues dans la session sont accessibles pour traiter toutes les requêtes provenant du même client.

Le troisième contexte est la requête. Les informations de la requête sont utilisées pour apporter une réponse adaptée au client. Les informations saisies dans un formulaire sont des informations liées à une requête.

3.6.1 LE CONTEXTE D'APPLICATION

Dans une servlet, le contexte d'application est accessible au travers de la méthode `getServletContext()`. Cette méthode retourne un objet de type `ServletContext`. Voici les méthodes les plus utilisées de cette interface :

- `getContextPath()` : cette méthode retourne une chaîne de caractères correspondant à l'URL permettant d'accéder à l'application. Dans le cadre de l'exemple, la méthode retourne `/Projet_Servlet`. Cette méthode est très importante pour variabiliser les URL utilisées afin qu'elles soient indépendantes du nom du projet car ce ne sera pas forcément le nom de l'application en production.
- `getInitParameter(...)` : cette méthode permet de lire la valeur d'un paramètre lié à l'application. Ce paramètre est défini au niveau du fichier `web.xml` dans une balise `<context-param>` comme le montre l'exemple suivant :

```
12 <context-param>
13     <param-name>nomSociete</param-name>
14     <param-value>Complexe Sportif SA</param-value>
15 </context-param>
```

Cette méthode ne doit pas être confondue avec la méthode de même nom disponible directement sur la servlet qui sert à lire un paramètre d'initialisation de la servlet.

- `getAttribute(...)` et `setAttribute(...)` : ces méthodes permettent de lire et d'écrire un attribut au niveau du contexte d'application. Un attribut peut s'apparenter à un paramètre sauf qu'il est défini programmatiquement.
- `getNamedDispatcher(...)` : cette méthode retourne un objet de type `RequestDispatcher` permettant d'accéder à une autre servlet ou une JSP. Le rôle de cette méthode sera expliqué plus en détail dans la section La création de la réponse.

3.6.2 LE CONTEXTE DE SESSION

Dans une servlet, la session est accessible au travers de la méthode `getSession()` de l'objet de type `HttpServletRequest` passé en paramètre des méthodes `doXXX()`.

3.6.3 LE CONTEXTE DE REQUETE

Dans une servlet, l'objet de type `HttpServletRequest` passé en paramètre des méthodes `doXXX()` permet d'accéder aux informations de la requête.

3.7 LA LECTURE DE LA REQUETE

Lorsqu'une requête HTTP arrive sur le serveur web, celle-ci est au format texte. Si cette requête doit être prise en charge par une servlet, le conteneur web s'occupe de créer un objet de type `HttpServletRequest` afin de faciliter sa manipulation. Le conteneur web crée un autre objet de type `HttpServletResponse` pour permettre la création de la réponse. À l'issue du traitement, le conteneur web utilise le contenu de l'objet `HttpServletResponse` pour générer la réponse HTTP.

3.7.1. LECTURE DES INFORMATIONS DE L'URL

La classe `HttpServletRequest` possède plusieurs méthodes permettant d'accéder aux différents composants de l'URL. Les méthodes sont présentées dans l'ordre des composants de l'URL :

- `getScheme()` : cette méthode retourne une chaîne de caractères correspondant au protocole réellement utilisé (http, https).
- `getServerName()` : cette méthode retourne le nom d'hôte de la machine vers laquelle la requête a été émise. Le nom d'hôte correspond à l'information située sur l'en-tête de requête `Host` sans prendre en compte le port si celui-ci est renseigné. Le nom d'hôte peut être un nom de domaine, un nom de machine ou une adresse IP.
- `getServerPort()` : cette méthode retourne le port utilisé par la requête HTTP.
- `getContextPath()` : cette méthode retourne le nom de l'application. Dans un environnement de développement, le nom de l'application correspond au nom du projet.
- `getServletPath()` : cette méthode retourne le chemin d'accès à la servlet. Cette information correspond à un des patterns d'URL définis dans la déclaration de la servlet (dans le fichier `web.xml` ou dans l'annotation `@WebServlet`).
- `getQueryString()` : cette méthode retourne une chaîne de caractères correspondant à tous les paramètres passés à la requête HTTP.

Des méthodes supplémentaires sont disponibles pour obtenir des informations complémentaires sur l'émetteur et le récepteur de la requête HTTP :

- `getRemoteAddr()` : cette méthode retourne sous la forme d'une chaîne de caractères l'adresse IP de l'émetteur de la requête HTTP.
- `getRemoteHost()` : cette méthode retourne sous la forme d'une chaîne de caractères le nom de l'hôte de l'émetteur de la requête HTTP.
- `getLocalAddr()` : cette méthode retourne sous la forme d'une chaîne de caractères l'adresse IP de la carte réseau du serveur par laquelle la requête HTTP a été reçue.
- `getLocalName()` : cette méthode à l'instar de la méthode précédente retourne sous la forme d'une chaîne de caractères le nom d'hôte de la carte réseau du serveur par laquelle la requête HTTP a été reçue.
- `getLocalPort()` : cette méthode retourne sous la forme d'un entier le port sur lequel a été reçue la requête HTTP.

3.7.2. LECTURE DE L'EN-TETE DE LA REQUETE

L'en-tête de la requête est composé d'attributs (que l'on nommera en-têtes par la suite) qui sont des informations envoyées par le navigateur vers le serveur. Elles ne sont pas saisies par l'utilisateur.

Les en-têtes existants sont décrits dans le chapitre Introduction à Java EE dans la section Le protocole HTTP - La requête - Les attributs de requêtes.

L'interface `HttpServletRequest` fournit les méthodes nécessaires à la lecture de ces en-têtes. Certains en-têtes standards possèdent une méthode dédiée, les autres en-têtes sont accessibles avec des méthodes générales prenant en paramètre le nom de l'en-tête recherché.

Voici la liste des méthodes dédiées disponibles :

- `getCharacterEncoding()` : cette méthode retourne l'encodage utilisé pour le corps de la requête ou null si la requête ne définit pas d'encodage.
- `getContentLength()` et `getContentLengthLong()` : ces méthodes retournent la taille en octet des informations situées dans le corps de la requête ou -1 si la taille n'est pas connue ou supérieure à `Integer.MAX_VALUE`. Cela correspond à l'en-tête de requête `Content-Length`.
- `getContentType()` : cette méthode retourne le type de média (MIME) du corps de la requête ou null s'il n'y a pas de corps ou si le type n'est pas connu. Cela correspond à l'en-tête de requête `Content-Type`.
- `getLocale()` : cette méthode retourne un objet de type `java.util.Locale`. Il correspond à la Locale préférée désignée dans l'en-tête de requête `Accept-Language`. Pour obtenir l'ensemble des locales paramétrées, il faut utiliser la méthode `getLocales()` qui retourne une collection de `Locale` sous la forme d'un objet de type `Enumeration<Locale>`. Ces informations sont modifiables par l'utilisateur en paramétrant les langues à utiliser au niveau du navigateur. Certains frameworks, comme Struts 2, peuvent se baser sur cet attribut pour gérer l'internationalisation de l'application.
- `getMethod()` : cette méthode retourne la méthode utilisée par la requête (GET, POST...).

À ces méthodes, s'ajoutent des méthodes générales dont voici la liste. En effet, il existe tellement d'en-têtes, qu'il est plus aisé d'utiliser des méthodes communes.

- `getHeader(String name)` : cette méthode retourne la valeur de l'en-tête passé en paramètre sous la forme d'une chaîne de caractères.
- `getHeaders(String name)` : cette méthode retourne la valeur de l'en-tête passé en paramètre sous la forme d'une collection de chaînes de caractères. Cela permet de lire facilement les attributs qui possèdent plusieurs valeurs séparées par un point-virgule comme `Accept-Language`.
- `getIntHeader(String name)` : cette méthode retourne la valeur de l'en-tête passé en paramètre sous la forme d'un int ou -1 si l'en-tête n'est pas présent. Attention, l'en-tête lu doit être compatible avec le type int sous peine d'avoir une exception de type `NumberFormatException`.
- `getDateHeader(String name)` : cette méthode retourne la valeur de l'en-tête passé en paramètre sous la forme d'un long correspondant au nombre de millisecondes depuis le premier janvier 1970 ou -1 si l'en-tête n'est pas présent. Attention, l'attribut lu doit être compatible avec le type Date sous peine d'avoir une exception de type `IllegalArgumentException`. Cette méthode est utile pour lire un paramètre comme `If-Modified-Since`.

- `getHeaderNames()` : cette méthode retourne une collection de l'ensemble des noms des en-têtes présents sur la requête. Ceci peut être utile puisque les navigateurs n'envoient pas tous les mêmes en-têtes.

3.7.3. LECTURE DES PARAMETRES

La lecture des paramètres d'une requête est l'élément primordial pour traiter la réponse comme il se doit. Lorsqu'un utilisateur saisit des informations dans un formulaire, ces informations sont envoyées sous la forme de paramètres au niveau de la requête.

Dans le cas d'une requête de type GET, les paramètres sont écrits au niveau de l'URL sous la forme suivante :

`URLValide?parametre1=valeur1¶metre2=valeur2...`

Ils sont séparés de l'URL par un point d'interrogation. Chaque paramètre est valorisé avec l'opérateur `=`. Il peut y avoir plusieurs paramètres, ils sont alors séparés par le caractère `&`.

Dans le cas d'une requête de type POST, les paramètres sont structurés de la même manière mais sont écrits dans le corps de la requête.

Il est possible d'avoir deux paramètres de même nom. C'est typiquement le cas lorsqu'il y a une sélection multiple possible dans une liste déroulante ou dans un ensemble de cases à cocher.

Dans le cadre d'un formulaire HTML, le nom du paramètre correspond à l'attribut `name` de la balise HTML permettant la saisie. La valeur correspond à la valeur saisie pour une zone de texte, à l'attribut `value` pour les balises de sélection.

Voici un exemple de formulaire HTML expliquant le mécanisme de base. Le contenu du fichier `formulaireExemple.html` est le suivant :



```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <title>Formulaire exemple</title>
6 </head>
7 <body>
8 <form method="get">
9 <input type="text" name="inputText"/>
10 <br/>
11 <select name="selectSport" multiple="multiple">
12 <option value="1">Badminton</option>
13 <option value="2">Squash</option>
14 <option value="3">Tennis</option>
15 </select>
16 <br/>
17 <input size="50" type="text" value="input sans attribut name. Pas de paramètre créé"/>
18 <br/>
19 <input type="submit" value="Valider"/>
20 </form>
21 </body>
22 </html>

```

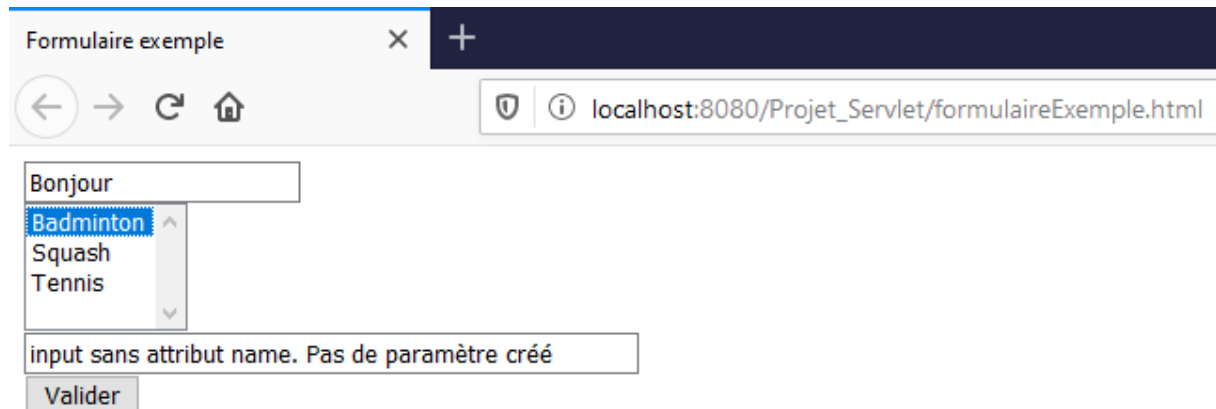
Ce formulaire possède trois zones de saisies :

- Une zone de saisie libre nommée inputText.
- Une liste déroulante nommée selectSport avec une sélection multiple possible.
- Une troisième zone de saisie libre sans nom. Cette zone ne donnera donc jamais naissance à un paramètre.

La soumission du formulaire provoque l'exécution d'une requête de type GET sur la même URL car l'attribut action n'a pas été défini au niveau de la balise <form>.

Voici deux exemples à l'usage du formulaire dont l'accès est possible au travers de l'URL localhost:8080/Projet_Servlet/formulaireExemple.html

Le premier exemple correspond à une saisie classique comme le montre l'écran suivant :



Formulaire exemple

Bonjour

Badminton
Squash
Tennis

input sans attribut name. Pas de paramètre créé

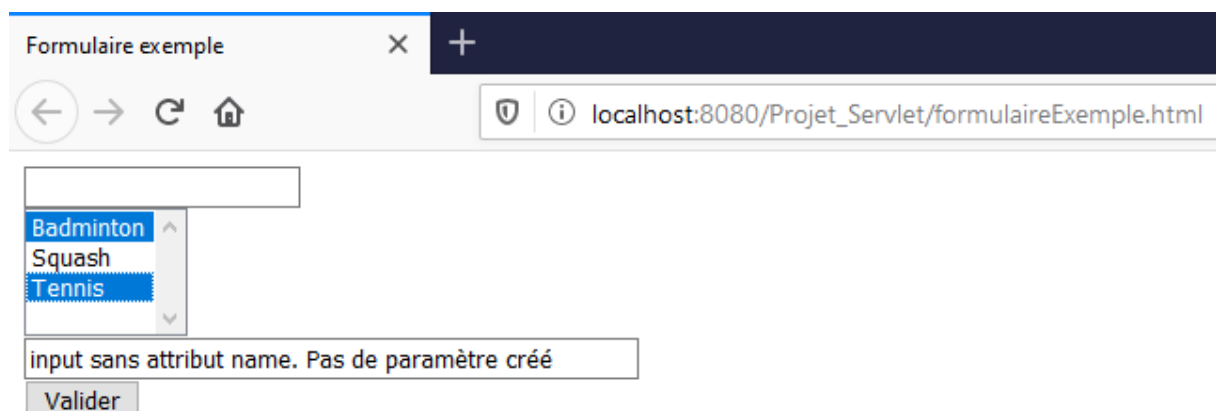
Valider

La soumission de ce formulaire provoque l'émission de la requête suivante :

`http://localhost:8080/Projet_Servlet/formulaireExemple.html?inputText=Bonjour&selectSport=1`

Deux paramètres de même nom que les zones de saisies sont créés. La zone de saisie sans nom ne provoque pas la création de paramètre.

Le second exemple correspond à une saisie multiple au niveau de la liste déroulante comme le montre l'écran suivant :



Formulaire exemple

Badminton
Squash
Tennis

input sans attribut name. Pas de paramètre créé

Valider

La soumission de ce formulaire provoque l'émission de la requête suivante :

`http://localhost:8080/Projet_Servlet/formulaireExemple.html?inputText=&selectSport=1&selectSport=3`

Trois paramètres sont présents. Le premier paramètre correspond à la zone de saisie libre nommée inputText. Il n'est associé à aucune valeur. Le deuxième et le troisième paramètre possèdent le même nom (selectSport) car ils proviennent d'une saisie multiple.

3.7.4. LECTURE DES PARAMETRES DANS LA SERVLET


La lecture des paramètres au niveau de la servlet se fait encore avec l'objet de type `HttpServletRequest`. Cette interface met à disposition les méthodes suivantes :

- `getParameter(String name)` : cette méthode retourne la valeur du paramètre passé en paramètre sous la forme d'une chaîne de caractères. Si le paramètre n'existe pas, cette méthode retourne null. S'il existe plusieurs paramètres avec le même nom, c'est la première valeur qui est retournée.
- `getParameterValues(String name)` : cette méthode est utile pour les paramètres apparaissant plusieurs fois dans la requête. Elle retourne un tableau de chaîne de caractères correspondant aux différentes valeurs.

L'usage de ces deux méthodes nécessite de connaître la structure du formulaire pour connaître les noms des paramètres à utiliser. Pour mettre en place des servlets indépendantes des formulaires sous-jacents, il faut pouvoir découvrir les paramètres disponibles au niveau de la requête. Deux méthodes supplémentaires sont disponibles pour répondre à ce besoin :

- `getParameterNames()` : cette méthode retourne une collection de chaînes de caractères correspondant aux noms des paramètres disponibles dans la requête. Un nom de paramètre n'est présent qu'une seule fois dans cette collection même s'il est présent plusieurs dans la requête. L'usage de la méthode `getParameterValues(...)` est donc adapté ensuite pour connaître la ou les valeurs associées à ce paramètre. Cette méthode retourne une collection vide si la requête ne possède aucun paramètre.
- `getParameterMap()` : cette méthode permet d'obtenir un objet de type `Map<String,String[]>`. La clé correspond au nom du paramètre, la valeur est un tableau de chaîne de caractères correspondant aux valeurs associées à ce paramètre.

Voici un exemple classique de lecture de paramètres. La première tâche à réaliser est le formulaire de saisie. Le propriétaire du complexe sportif souhaite pouvoir ajouter un nouveau terrain et les sports praticables sur celui-ci. Le formulaire `formulaireAjoutTerrain.html` a la structure suivante :



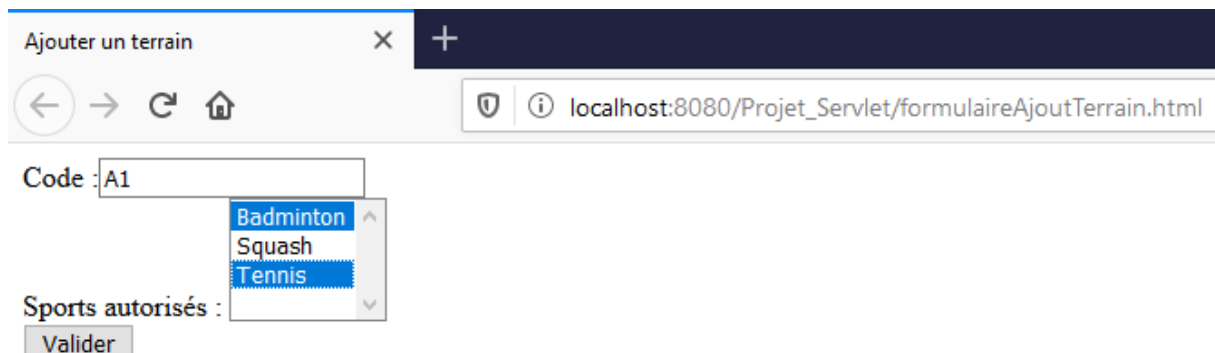
```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>Ajouter un terrain</title>
6  </head>
7  <body>
8  <!-- Consulter le chapitre traitant de JSP pour variabiliser le nom
9  de l'application dans l'URL avec la méthode getContextPath() de l'objet
10 de type HttpServletRequest -->
11 <form method="post" action="/Projet_Servlet/AjouterTerrain">
12     Code :<input type="text" name="codeTerrain"/>
13     <br/>
14     Sports autorisés :
15     <select name="sportsAutorises" multiple="multiple">
16         <option value="1">Badminton</option>
17         <option value="2">Squash</option>
18         <option value="3">Tennis</option>
19     </select>
20     <br/>
21     <input type="submit" value="Valider">
22 </form>
23 </body>
24 </html>
```


Le formulaire pointe vers l'URL /Projet_Servlet/AjouterTerrain.

Cette URL est associée à la servlet AjouterTerrain. Seule la méthode doPost(...) est substituée car la requête créée à la soumission du formulaire est une requête de type POST. Voici le code de cette servlet :

```
MaServlet.java | web.xml | formulaireExemple.html | *formulaireAjoutTerrain.html | *AjouterTerrain.java
1 package fr.julian.javaee.servlets;
2
3 import java.io.IOException;
4
5
6 /**
7  * Servlet implementation class AjouterTerrain
8  */
9 @WebServlet("/AjouterTerrain")
10 public class AjouterTerrain extends HttpServlet
11 {
12     private static final long serialVersionUID = 1L;
13
14     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
15         String codeTerrain = request.getParameter("codeTerrain");
16         String[] sportsAutorises = request.getParameterValues("sportsAutorises");
17
18         System.out.println("Code du terrain : " + codeTerrain);
19         for(String codeSport:sportsAutorises)
20         {
21             System.out.println("Code sport autorisé : " + codeSport);
22         }
23     }
24 }
```

Accédez au formulaire de saisie comme le montre la capture d'écran suivante :



Saisissez un **Code** et des **Sports autorisés** puis validez en cliquant sur le bouton **Valider**. La servlet est alors exécutée et la trace suivante s'affiche dans la console :

```
Markers | Properties | Servers | Data Source Explorer | Snippets | Problems | Console | Progress
Tomcat v8.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (22 avr. 2020 10:31:20)
INFO: Starting ProtocolHandler ["ajp-nio-8009"]
avr. 22, 2020 10:31:21 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 239 ms
Code du terrain :A1
Code sport autorisé :1
Code sport autorisé :3
```

La réponse restituée au client est une simple page blanche comme le montre la capture suivante.

Cette capture affiche les informations sur la requête émise et la réponse apportée.

Observez les en-têtes envoyés et reçus par le navigateur.

La requête est de type POST. Le code d'état est positionné à **200** pour indiquer que tout s'est bien passé. L'en-tête de requête Content-Length vaut 50. Cela correspond à la taille du corps de la requête qui contient les valeurs saisies.

L'en-tête de réponse de même nom vaut 0 car une page blanche est retournée.

The screenshot shows a web browser window with the address bar displaying 'localhost:8080/Projet_Servlet/AjouterTerrain'. Below the browser window, the 'Réseau' (Network) tab is selected in the developer tools. The network log shows a POST request to 'AjouterTerrain' with a status of 200 OK. The response headers indicate a Content-Length of 0, confirming a blank page was returned.

É...	Mé	Doma...	Fichier	Source	T...	Transf...	T...
200	P...	loc...	AjouterTerrain	docu...	plai	713 o	0...
404	GET	loc...	favicon.ico	img	htm	mis e...	0...

En-têtes

URL de la requête : http://localhost:8080/Projet_Servlet/AjouterTerrain
Méthode de la requête : POST
Adresse distante : [::1]:8080
Code d'état : 200 OK
Version : HTTP/1.1
Politique de référent : no-referrer-when-downgrade

En-têtes de la réponse (102 o)

- Content-Length: 0
- Date: Wed, 22 Apr 2020 08:35:21 GMT
- Server: Apache-Coyote/1.1

En-têtes de la requête (561 o)

- Accept: text/html,application/xhtml+xml;q=0.9,image/webp,*/*;q=0.8
- Accept-Encoding: gzip, deflate
- Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
- Connection: keep-alive
- Content-Length: 50
- Content-Type: application/x-www-form-urlencoded
- Host: localhost:8080
- Origin: http://localhost:8080
- Referer: http://localhost:8080/Projet_S...et/formulaireAjoutTerrain.html
- Upgrade-Insecure-Requests: 1
- User-Agent: Mozilla/5.0 (Windows NT 10.0; ...) Gecko/20100101 Firefox/75.0

3.8. LA CREATION DE LA REPONSE

La création de la réponse se découpe en deux étapes. La première étape est l'écriture de l'en-tête de la réponse. La seconde étape est l'écriture du corps de la réponse. Tout cela se fait au travers de l'objet de type `HttpServletResponse` proposant les méthodes nécessaires. L'écriture de l'en-tête doit se faire avant l'écriture du corps.

3.8.1. ECRITURE DE L'EN TETE DE LA REPONSE

L'interface `HttpServletResponse` met à disposition les méthodes suivantes pour définir les en-têtes de la réponse.

La première méthode est la méthode définissant le code de statut de la réponse. Elle s'appelle `setStatus(int sc)`. Elle prend en paramètre un entier correspondant au statut souhaité. Pour faciliter son usage, l'interface `HttpServletResponse` propose un ensemble de constantes de type `int` correspondant aux différents statuts.

Voici quelques exemples de constantes disponibles :

- `HttpServletResponse.SC_SWITCHING_PROTOCOLS` : cette constante correspond à la valeur 101. Cela permet d'indiquer que la communication va changer de protocole. C'est une étape utilisée pour communiquer par `WebSocket`.
- `HttpServletResponse.SC_OK` : cette constante correspond à la valeur 200. Cela permet d'indiquer que tout s'est bien passé dans le traitement de la requête.
- `HttpServletResponse.SC_MOVED_PERMANENTLY` : cette constante correspond à la valeur 301. Cela permet d'indiquer que la ressource demandée n'est plus accessible à cette URL mais qu'il faut maintenant utiliser une autre URL. C'est ce que l'on appelle une redirection. L'en-tête de requête `Location` doit être défini en parallèle pour indiquer au client quelle requête réaliser pour obtenir la ressource attendue. Cela peut se faire avec la méthode `setHeader(...)` décrite un peu plus loin. Le risque est d'oublier d'appliquer cet en-tête. Une méthode dédiée à la redirection est disponible.
- `HttpServletResponse.SC_FORBIDDEN` : cette constante correspond à la valeur 403. Cela permet d'indiquer que le serveur a refusé de traiter la requête. C'est le cas lorsque l'on souhaite accéder à une ressource sécurisée demandant une authentification préalable.
- `HttpServletResponse.SC_INTERNAL_SERVER_ERROR` : cette constante correspond à la valeur 500. Cela permet d'indiquer qu'une erreur de traitement a eu lieu côté serveur. Cette erreur ne devrait pas arriver en production.

Il existe d'autres méthodes disponibles pour l'écriture des en-têtes à l'image des méthodes de lecture disponibles sur l'interface `HttpServletRequest` :

- `setCharacterEncoding(String charset)` : cette méthode permet de définir l'encodage utilisé pour l'écriture du corps de la réponse. La valeur attendue est une chaîne de caractères correspondant à un encodage (Character Sets) défini par l'**IANA**. La liste est disponible à l'adresse suivante : <http://www.iana.org/assignments/character-sets/character-sets.xhtml>. Les valeurs les plus connues sont **UTF-8** et **ISO-8859-1**.
- Cette méthode écrase la valeur potentiellement définie à l'aide des méthodes `setContentType(...)` et `setLocale(...)`.

- `setContentLength(...)` et `setContentLengthLong(...)` : ces méthodes permettent de définir la taille du corps de la réponse. L'information sera écrite au niveau de l'en-tête `Content-Length`.
- `setContentType(...)` : cette méthode permet de définir le type de média (type MIME) et éventuellement l'encodage. Il est ainsi possible d'écrire :
- `setLocale(...)` : cette méthode permet de définir la Locale liée à la réponse rendue. Elle permet aussi de définir l'encodage (approprié à la Locale) si celui-ci n'a pas déjà été défini par la méthode `setCharacterEncoding(...)` ou `setContentType(...)`.

Pour finir, il existe des méthodes générales pour l'écriture des en-têtes à l'image des méthodes de lecture disponibles sur l'interface `HttpServletRequest` :

- `setHeader(String name, String value)` : cette méthode permet d'appliquer une valeur (value) à un en-tête donné (name).
- `setDateHeader(String name, long date)` : cette méthode est spécialisée dans l'écriture des en-têtes de type `Date`.
- `setIntHeader(String name, int value)` : cette méthode est spécialisée dans l'écriture des en-têtes de type `int`.

Il existe trois méthodes supplémentaires `addHeader(...)`, `addDateHeader(...)` et `addIntHeader(...)` pour permettre d'ajouter plusieurs valeurs à un en-tête. En effet, l'usage d'une méthode `setXXX(...)` provoque l'écrasement de la dernière valeur définie pour un en-tête.

3.8.2. ECRITURE DU CORPS DE LA REPONSE

L'écriture du corps de la réponse est une étape très importante car elle permet de définir le contenu qui sera affiché par le client. Pour écrire le corps, il existe deux possibilités :

- L'utilisation d'un objet de type `java.io.PrintWriter` obtenu par l'appel de la méthode `getWriter()` de l'interface `HttpServletResponse`. Cet objet est à utiliser pour écrire un contenu textuel.
- L'utilisation d'un objet de type `javax.servlet.ServletOutputStream` obtenu par l'appel de la méthode `getOutputStream()` de l'interface `HttpServletResponse`. Cet objet est à utiliser pour écrire un contenu binaire.

Une exception de type `IllegalStateException` est levée si ces deux méthodes sont appelées pour la même réponse.

Les deux objets fonctionnent globalement de la même manière. L'objet le plus couramment utilisé est du type `PrintWriter`. Pour écrire la réponse, cette classe offre les méthodes classiques suivantes :

- `append(...)`, `print(...)`, `println(...)`, `printf(...)`, `write(...)` : ces méthodes permettent d'ajouter le contenu passé en paramètre.
- `flush()` : cette méthode permet de valider le contenu ajouté dans l'objet jusqu'au moment de l'appel de cette méthode. Lorsque les méthodes d'écriture sont utilisées, le contenu est placé dans un tampon en attente qu'il soit rempli pour le pousser vers le destinataire. La méthode `flush()` permet de déclencher programmatiquement cette action.
- `close()` : cette méthode permet de fermer le flux. Il n'est alors plus possible d'ajouter du contenu.

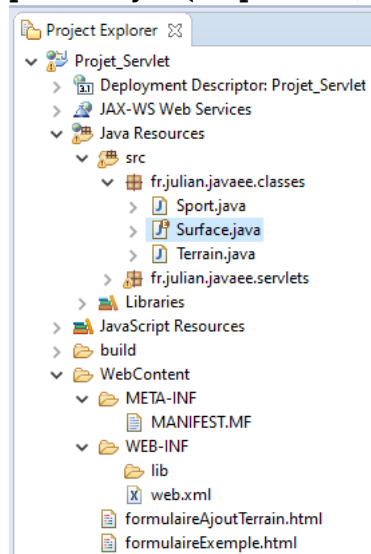
3.8.3. LA GESTION DE L'ENVOI DE LA REPONSE

Tout ce qui est écrit avec les méthodes présentées précédemment (pour l'écriture de l'en-tête et l'écriture corps de la réponse) n'est pas directement envoyé vers le client. Les informations sont tout d'abord entreposées dans un tampon avant d'être envoyées. Cela permet une gestion plus efficace de l'envoi des informations. Les caractéristiques de ce tampon sont modifiables via l'utilisation de méthodes présentes sur l'interface `HttpServletResponse`.

- `setBufferSize(int size)` : cette méthode permet de définir la taille souhaitée pour le tampon contenant le corps de la réponse. Un grand tampon permet d'écrire plus de contenu avant que quoi que ce soit ne soit envoyé au client. Cela laisse donc plus de temps pour définir les en-têtes de réponses. Un petit tampon diminue la consommation mémoire côté serveur et permet au client de recevoir plus rapidement le début de la réponse. Cette méthode doit être appelée avant le début de l'écriture du corps de la réponse. Si ce n'est pas le cas, une exception de type `IllegalStateException` est levée.
- `getBufferSize()` : cette méthode permet d'obtenir la taille actuelle du tampon.
- `reset()` : cette méthode efface tout ce qui a pu être écrit au niveau de la réponse : en-têtes et corps. Si la réponse a déjà commencé à être retournée au client, alors une exception de type `IllegalStateException` est levée.
- `resetBuffer()` : cette méthode efface le contenu du corps de la réponse qui a pu être écrit. Si la réponse a déjà commencé à être retournée au client, alors une exception de type `IllegalStateException` est levée.
- `isCommitted()` : cette méthode permet de savoir si la réponse a déjà commencé à être retournée au client.
- `flushBuffer()` : cette méthode permet de forcer l'envoi au client de ce qui a déjà pu être écrit au niveau de la réponse.

3.8.4. PREMIER EXEMPLE DE MISE EN ŒUVRE

Pour réaliser cette servlet, nous aurons besoin de deux classes Java Sport et Terrain et d'un enum Surface. Créez ces classes en fonction des informations disponibles dans le pdf 0.Projet. (Proprement, dans un nouveau package)



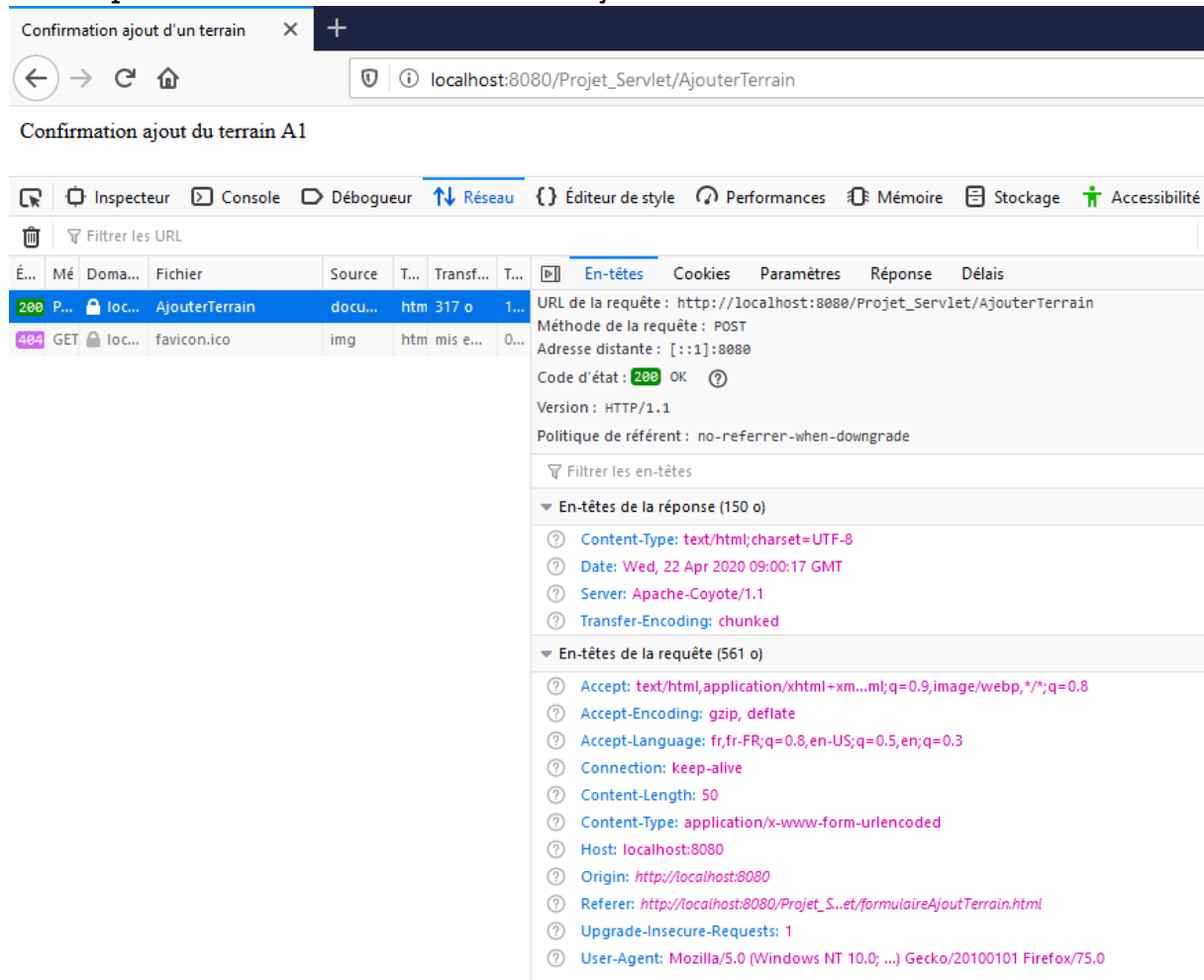
Enrichissons la servlet AjouterTerrain pour ne plus avoir une réponse blanche. Voici le nouveau code de la méthode doPost(...) de cette servlet :

```
19 @WebServlet("/AjouterTerrain")
20 public class AjouterTerrain extends HttpServlet {
21     private static final long serialVersionUID = 1L;
22
23     @Override
24     protected void doPost(HttpServletRequest request, HttpServletResponse response)
25         throws ServletException, IOException {
26         // Lecture des informations de la requête
27         String codeTerrain = request.getParameter("codeTerrain");
28         String[] sportsAutorises = request.getParameterValues("sportsAutorises");
29         // Réalisation du traitement métier (Normalement déporté dans les
30         // classes du projet)
31         Terrain terrainAjoute = new Terrain(codeTerrain, Surface.BETON);
32         // System.out.println("Code du terrain : " + codeTerrain);
33         for (String codeSport : sportsAutorises) {
34             // System.out.println("Code sport autorisé : " + codeSport);
35             Sport sport = new Sport();
36             sport.setIdentifiant(Integer.parseInt(codeSport));
37             terrainAjoute.getSportsAutorises().add(sport);
38             // ...
39         }
40         // Écriture de la réponse
41         // Les en-têtes
42         response.setStatus(HttpServletResponse.SC_OK);
43         response.setContentType("text/html");
44         response.setCharacterEncoding("UTF-8");
45         // Le corps
46         PrintWriter pw = response.getWriter();
47         pw.println("<!DOCTYPE html>");
48         pw.println("<html>");
49         pw.println("<head>");
50         pw.println("<meta charset=\"UTF-8\">");
51         pw.println("<title>Confirmation ajout d'un terrain</title>");
52         pw.println("</head>");
53         pw.println("<body>");
54         pw.println("Confirmation ajout du terrain " + terrainAjoute.getCode());
55         pw.println("</body>");
56         pw.flush();
57         pw.close();
58     }
59 }
```

Quatre étapes sont facilement identifiables :

- La lecture de la requête.
- La réalisation du traitement métier. Normalement, ce traitement n'est pas fait dans la servlet mais dans les classes Java du projet.
- L'écriture des en-têtes de la réponse.
- L'écriture du corps de la réponse. Cette partie est fastidieuse à écrire dans une servlet. L'usage des JSP est plus adapté.

L'exécution et la validation du formulaire `formulaireAjoutTerrain.html` donnent le résultat suivant pour la saisie d'un nouveau terrain ayant le code **A1** :



Observez le **Code d'état** positionné à **200** et la valeur de l'en-tête **Content-Type** dont la valeur est **text/html; charset=UTF-8**.

La servlet `AjouterTerrain`, dans sa version actuelle, n'est pas satisfaisante. En effet, elle réalise toutes les tâches nécessaires au traitement de la requête. Il est important de séparer les rôles clairement.

La servlet `AjouterTerrain` a pour rôle de réceptionner la demande d'ajout. Pour cela, elle décortique les paramètres reçus pour ensuite demander aux classes Java du projet l'ajout effectif du terrain.

Lorsque cette tâche est terminée et que la servlet reçoit le résultat du traitement, elle doit alors déléguer la tâche de fournir une réponse à une autre ressource. Cette ressource peut être une servlet (mais c'est souvent une JSP).

Ajout d'attributs à la requête

Pour que cette délégation soit efficace, il faut pouvoir transférer des informations complémentaires calculées ou obtenues lors du traitement métier vers cette seconde ressource.

Pour cela, il y a deux solutions :

- La première solution est de placer ces informations en session pour qu'elles soient accessibles depuis n'importe quelle requête provenant du même client. Cependant, ce n'est pas l'option la plus intéressante pour la gestion d'informations dont la durée de vie est limitée à la requête. Elle peut avoir son intérêt si l'architecture physique en production implique l'utilisation de plusieurs serveurs pour répondre aux requêtes de l'ensemble des clients.
- La seconde solution est d'enrichir la requête de nouvelles informations avant de déléguer la suite du traitement à une nouvelle servlet (ou JSP) si ces informations ont une durée de vie limitée à la requête. Cette solution consiste à ajouter des attributs.

L'ajout d'un attribut se fait grâce à la méthode `setAttribute(String name, Object o)` de l'interface `HttpServletRequest`. Cette méthode attend deux paramètres à savoir le nom de l'attribut (`name`) et la valeur (`o`) dont le type est indéterminé.

Dans l'exemple, il est ainsi facile de passer en paramètre un objet de type `Terrain` correspondant au terrain nouvellement ajouté.

```
Terrain terrainAjoute = new Terrain(codeTerrain, Surface.BETON);  
//Ajout d'informations complémentaires pour la ressource s'occupant de l'affichage  
request.setAttribute("terrainAjoute", terrainAjoute);
```

La lecture d'un attribut se fait simplement grâce à la méthode `getAttribute(String name)`. Cette méthode retourne la valeur correspondant à l'attribut dont le nom a été passé en paramètre. Attention la valeur est retournée sous la forme d'un `Object`. Un transtypage est donc souvent nécessaire.

```
Object objetAttribut = request.getAttribute("terrainAjoute");  
if(objetAttribut instanceof Terrain)  
{  
    Terrain terrainAjoute = (Terrain)objetAttribut;  
    //...  
}
```

Deux méthodes supplémentaires sont disponibles pour compléter les fonctionnalités disponibles :

- `getAttributeNames()` : cette méthode retourne une collection de chaînes de caractères correspondant à l'ensemble des noms des attributs présents sur la requête.
- `removeAttribute(String name)` : cette méthode permet de supprimer un attribut de la requête.

Utilisation de l'interface RequestDispatcher

Une servlet (comme une JSP) est un élément géré par le conteneur. Pour l'utiliser à partir d'une autre servlet il est interdit de l'instancier nous-mêmes, il faut passer par le conteneur. Cela est possible en utilisant un objet de type RequestDispatcher. Cette interface propose les méthodes nécessaires pour manipuler des ressources disponibles sur le serveur (servlets, JSP, fichiers HTML).

- La méthode forward(...) prend en paramètre les objets de type HttpServletRequest et HttpServletResponse de la méthode doXXX(...). Les attributs positionnés sur la requête sont donc bien transmis. La réponse en cours d'élaboration est bien transmise aussi. Cela permet d'écrire la réponse progressivement. Cette méthode permet de transférer la requête vers une autre ressource pour finaliser le traitement. Normalement, après l'appel de cette méthode, il ne faut plus faire de traitement dans la servlet à l'origine de la délégation.
L'utilisation de cette méthode provoque une exception IllegalStateException si des informations ont déjà été retournées au client. Si des informations sont présentes dans le tampon, celles-ci sont automatiquement effacées avant le transfert. L'objet représentant la requête (HttpServletRequest) est mis à jour pour que les méthodes retournant des informations sur l'URL correspondent à la nouvelle ressource.
- Si des traitements complémentaires sont nécessaires par la suite, il est préférable d'utiliser l'inclusion en utilisant la méthode include(...). Cette méthode attend les mêmes paramètres que la méthode forward(...). Cette méthode est adaptée pour inclure des éléments dans le traitement de la servlet à l'origine de l'inclusion.

Lorsque la méthode forward(...) ou include(...) est exécutée, alors le traitement classique d'une requête est effectué. C'est comme si une nouvelle requête était envoyée par le client mais tout en restant à l'intérieur du serveur. Le type de la requête reste celui de la requête initiale. La méthode doXXX(...) appropriée est appelée.

Avant de pouvoir utiliser ces méthodes, il faut obtenir un objet de type RequestDispatcher. Deux méthodes sont disponibles :

- La méthode getRequestDispatcher(String path) de l'interface HttpServletRequest. Cette méthode attend un chemin d'accès qui peut être :
 - Relatif au chemin d'accès à la servlet. Attention, une servlet peut être accessible par différentes URL. Ce mécanisme est à utiliser avec précaution.
 - Absolu. Le chemin d'accès doit donc commencer par un slash (/) pour indiquer qu'il débute à la racine de l'application.

```
RequestDispatcher rd = request.getRequestDispatcher("/URL");  
rd.forward(request, response);
```

- La méthode getNamedDispatcher(String name) de l'interface ServletContext prend en paramètre le nom de la ressource recherchée telle que définie dans le paramètre name de l'annotation @WebServlet ou dans la balise <servlet-name> du fichier web.xml.

```
RequestDispatcher rd = this.getServletContext().getNamedDispatcher("nom");  
rd.forward(request, response);
```

Ces deux méthodes retournent null si aucune ressource n'est trouvée.

3.8.5. SECOND EXEMPLE DE MISE EN ŒUVRE

Modifions la servlet AjouterTerrain pour séparer la partie traitement métier de la partie affichage.

La première étape consiste à créer une servlet spécialisée dans l'affichage. Voici le code de la servlet ConfirmationAjoutTerrain :

```
@WebServlet(name = "ConfirmationAjoutTerrain", value = "/RecapitulatifNouveauTerrain")
public class ConfirmationAjoutTerrain extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Lecture des attributs de la requête
        Terrain terrainAjoute = (Terrain) request.getAttribute("terrainAjoute");
        // Écriture de la réponse
        // Les en-têtes
        response.setStatus(HttpServletResponse.SC_OK);
        response.setContentType("text/html");
        response.setCharacterEncoding("UTF-8");
        // Le corps
        PrintWriter pw = response.getWriter();
        pw.println("<!DOCTYPE html>");
        pw.println("<html>");
        pw.println("<head>");
        pw.println("<meta charset=\"UTF-8\">");
        pw.println("<title>Confirmation ajout d'un terrain</title>");
        pw.println("</head>");
        pw.println("<body>");
        pw.println("Affichage après délégation<br/>");
        pw.println("Confirmation ajout du terrain " + terrainAjoute.getCode());
        pw.println("</body>");
        pw.flush();
        pw.close();
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

La servlet est nommée ConfirmationAjoutTerrain et est accessible au travers de l'URL /RecapitulatifNouveauTerrain.

Le code de la méthode doGet(...) commence par la lecture d'un attribut de requête nommé terrainAjoute.

Elle continue ensuite par la création de la réponse. Notez bien que cette servlet ne fait que de la lecture d'informations et que l'écriture de la réponse.

La seconde étape est la modification (simplification) de la servlet AjouterTerrain :

```
@WebServlet("/AjouterTerrain")
public class AjouterTerrain extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // Lecture des informations de la requête
        String codeTerrain = request.getParameter("codeTerrain");
        String[] sportsAutorises = request.getParameterValues("sportsAutorises");
        // Réalisation du traitement métier
        Terrain terrainAjoute = new Terrain(codeTerrain, Surface.BETON);
        for (String codeSport : sportsAutorises) {
            // System.out.println("Code sport autorisé : " + codeSport);
            Sport sport = new Sport();
            sport.setIdentifiant(Integer.parseInt(codeSport));
            terrainAjoute.getSportsAutorises().add(sport);
            // ...
        }
        // Ajout d'informations complémentaires pour la ressource s'occupant de
        // l'affichage
        request.setAttribute("terrainAjoute", terrainAjoute);

        // Délégation du traitement à une autre ressource (Servlet/JSP)
        // RequestDispatcher rd =
        // request.getRequestDispatcher("/RecapitulatifNouveauTerrain");
        // rd.forward(request, response);
        // ou
        RequestDispatcher rd = this.getServletContext().getNamedDispatcher("ConfirmationAjoutTerrain");
        rd.forward(request, response);
    }
}
```

Lorsque le traitement métier est terminé, cette servlet positionne le nouveau terrain en attribut de requête avec le nom terrainAjoute. Ensuite, la servlet recherche la ressource pour gérer l'affichage avant de lui transférer le traitement avec la méthode forward(...). Notez, dans le code, les deux possibilités pour rechercher la ressource avec le nom ou l'URL de celle-ci.

Il est possible d'améliorer la servlet ConfirmationAjoutTerrain en répartissant la gestion de l'affichage à différentes ressources (servlets, JSP, pages HTML). En effet, dans une application web, une charte graphique fait en sorte que toutes les pages aient la même structure. Typiquement, l'en-tête de la page, le pied de la page ou encore le menu sont des éléments qui peuvent se retrouver sur un grand nombre de pages. L'utilisation de la méthode include(...) de l'interface RequestDispatcher est adaptée.

Prenons l'exemple d'un pied de page matérialisé par le fichier piedDePage.html. Ce n'est pas un fichier HTML valide mais un fragment de fichier. Voici son contenu :

```
1 <footer>
2     JAVA EE
3 </footer>
```

Voici la modification nécessaire au niveau de la servlet ConfirmationAjoutTerrain pour inclure cette ressource :

```
35     pw.println("<meta charset=\"UTF-8\">");
36     pw.println("<title>Confirmation ajout d'un terrain</title>");
37     pw.println("</head>");
38     pw.println("<body>");
39     pw.println("Affichage après délégation<br/>");
40     pw.println("Confirmation ajout du terrain " + terrainAjoute.getCode());
41     //Inclusion du pied de page
42     RequestDispatcher rd = request.getRequestDispatcher("/piedDePage.html");
43     rd.include(request, response);
44     pw.println("</body>");
45     pw.flush();
46     pw.close();
```

3.8.6. LA REDIRECTION

La redirection est différente de la délégation décrite précédemment. Lorsque le mécanisme de redirection est mis en œuvre, le serveur demande au navigateur d'effectuer une nouvelle requête vers l'URL indiquée par l'en-tête de réponse Location. Le code de statut de la réponse doit également avoir une valeur parmi les suivantes :

- HttpServletResponse.SC_MOVED_PERMANENTLY ou 301 pour une ressource qui a définitivement changée d'URL.
- HttpServletResponse.SC_MOVED_TEMPORARILY ou 302 pour une ressource qui a momentanément changée d'URL.

Voici un exemple de redirection écrit dans la servlet RedirectionVersFormulaireAjoutTerrain :

```
@WebServlet("/Redirection")
public class RedirectionVersFormulaireAjoutTerrain extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
        response.setHeader("Location", request.getContextPath() + "/formulaireAjoutTerrain.html");
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Cette servlet accessible au travers de l'URL /Redirection demande au navigateur d'effectuer une nouvelle requête vers le formulaire d'ajout de terrain.

L'URL définie au niveau de l'en-tête Location peut être une URL absolue (http://www.servlet_projet.fr) ou une URL relative.

Pour une URL relative, il faut y ajouter le nom de l'application. Cela peut se faire avec la méthode `getContextPath()`.

Une méthode plus rapide est disponible pour faire une redirection temporaire. Il s'agit de la méthode `sendRedirect(String location)`.

Voici un exemple de redirection avec cette méthode :

```
response.sendRedirect(request.getContextPath()+"/formulaireAjoutTerrain.html");
```

Si une réponse a déjà commencé à être retournée au client, alors une exception de type `IllegalStateException` est levée.

3.8.7. LA GESTION DES ERREURS

La méthode `sendError(...)` de l'interface `HttpServletResponse` permet de renvoyer une réponse d'erreur au client et d'effacer le contenu du tampon. Une exception de type `IllegalStateException` est levée si une réponse a déjà commencé à être retournée au client.

Il existe deux surcharges de cette méthode. La première surcharge prend uniquement en paramètre le code de statut de la réponse retournée. La seconde surcharge prend en plus le message à afficher.

La servlet `GestionDesErreurs` montre ces deux possibilités :

```
@WebServlet("/GestionDesErreurs")
public class GestionDesErreurs extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR, "Erreur grave");
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

À l'exécution, voici le résultat obtenu :



État HTTP 500 – Internal Server Error

Type Rapport d'état

message Erreur grave

description Le serveur a rencontré une erreur interne qui l'a empêché de satisfaire la requête.

Apache Tomcat/8.0.53

Pour avoir un affichage personnalisé, il suffit de modifier le fichier web.xml pour indiquer le fichier à utiliser en cas d'erreur. Il est possible de paramétrer un fichier différent en fonction du code de statut ou du type de l'exception levée.

La balise <error-page> est responsable de ce paramétrage. Elle attend les sous-balises suivantes :

- La balise <error-code> ou la balise <exception-type> pour définir le code de statut ou le type d'exception avec le nom de la classe Java pleinement qualifié.
- La balise <location> pour définir la page à afficher.

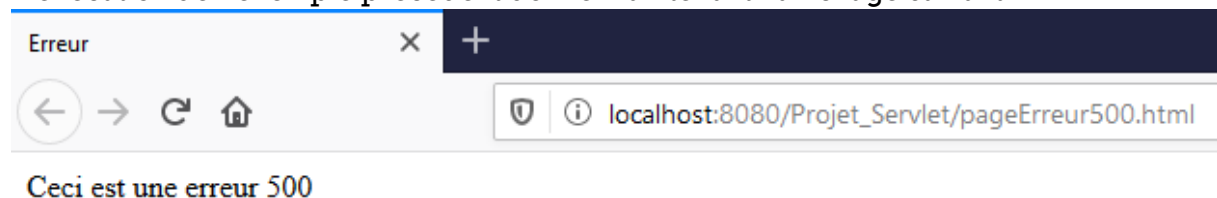
Voici un exemple de mise en œuvre avec le paramétrage du fichier web.xml :

```
<error-page>
  <error-code>500</error-code>
  <location>/pageErreur500.html</location>
</error-page>
```

Et le contenu du fichier pageErreur500.html :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Erreur</title>
</head>
<body>
  Ceci est une erreur 500
</body>
</html>
```

L'exécution de l'exemple précédent donne maintenant l'affichage suivant :



3.8.8. LES SERVLETS ASYNCHRONES

Le fonctionnement des servlets présenté jusque-là implique l'utilisation d'un thread par requête client. Lorsque le nombre de requêtes simultanées tend à grossir, il peut y avoir des problèmes d'adaptation pour répondre à toutes les requêtes. Plusieurs solutions peuvent améliorer la situation : des solutions matérielles pour pouvoir augmenter le nombre de threads disponibles mais aussi des solutions logicielles pour diminuer le temps de traitement de chacun de ces threads. Dans les solutions logicielles, il faut faire en sorte que tous les threads utilisés pour répondre aux requêtes des clients ne soient pas en attente à un moment ou à un autre. Ce temps d'attente est du temps perdu pour traiter les requêtes pas encore prises en charge.

Il existe deux cas standards dans lesquels le thread peut se trouver en attente :

- Attente d'une ressource nécessaire pour construire la réponse. C'est typiquement le cas lorsqu'un accès à une base de données est nécessaire.
- Attente d'un événement pour construire la réponse. C'est typiquement le cas lorsqu'une information d'un autre client est nécessaire.

Ces tâches sont bloquantes et devraient être traitées par des threads enfants pour libérer les threads gérant les requêtes en entrée.

Depuis la version 3.0 de la spécification des servlets, il est possible de mettre en place un mécanisme d'asynchronisme pour les servlets.

La première étape consiste à indiquer qu'une servlet utilise le mécanisme d'asynchronisme en ajoutant le paramètre `asyncSupported` avec la valeur `true` sur l'annotation `@WebServlet`.

```
@WebServlet(value = "/MaServletAsynchrone", asyncSupported = true)
public class MaServletAsynchrone extends HttpServlet {
```

La seconde étape est l'utilisation de la classe `javax.servlet.AsyncContext`. Cette classe met à disposition toutes les fonctionnalités nécessaires au bon fonctionnement des traitements asynchrones.

Pour créer un objet de type `AsyncContext`, il faut utiliser la méthode `startAsync()` de l'interface `HttpServletRequest` :

```
final AsyncContext contexteAsynchrone = request.startAsync();
```

Le traitement asynchrone démarre à ce moment-là. Aucune réponse ne sera renvoyée au client une fois l'exécution des méthodes `doXXX(...)` et `service(...)` terminés. La réponse ne sera générée qu'à la fin du traitement asynchrone.

La classe `AsyncContext` met à disposition des méthodes permettant de faire le traitement asynchrone dans les mêmes conditions qu'un traitement synchrone :

- `start(Runnable run)` : cette méthode est le point d'entrée du traitement asynchrone. Elle prend en paramètre un objet qui implémente l'interface `Runnable`. C'est typiquement une classe anonyme dans laquelle la méthode `run()` contient le code à exécuter en asynchrone.
- `getRequest()` : cette méthode donne accès à un objet de type `ServletRequest` qui correspond dans le cadre d'une requête HTTP à un objet de type `HttpServletRequest`. Cet objet est celui présent en paramètre de la méthode `doXXX(...)`.

- `getResponse()` : cette méthode donne accès à un objet de type `ServletResponse` qui correspond dans le cadre d'une requête http à un objet de type `HttpServletResponse`. Cet objet est celui présent en paramètre de la méthode `doXXX(...)`.
- `complete()` : cette méthode indique que le traitement asynchrone est terminé. Le contenu de la réponse est alors envoyé vers le client.
- `disptach(String path)` : cette méthode permet de déléguer la génération de la réponse à une autre servlet.

Voici un exemple complet d'utilisation décrit dans la servlet `MaServletAsynchrone`.

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("Le traitement qui suit va se faire de façon asynchrone");
    // Obtention d'un contexte asynchrone afin de faire des traitements en
    // tâche de fond
    final AsyncContext contexteAsynchrone = request.startAsync();
    // Lancement du traitement asynchrone avec la méthode start
    contexteAsynchrone.start(new Runnable() {
        @Override
        public void run() {
            // Cette méthode doit contenir le traitement à réaliser de manière asynchrone
            HttpServletRequest request = (HttpServletRequest) contexteAsynchrone.getRequest();
            HttpServletResponse response = (HttpServletResponse) contexteAsynchrone.getResponse();
            System.out.println("Début de la réalisation du traitement asynchrone qui dure un certain temps");
            try {
                Thread.sleep(3000);
                response.setStatus(HttpServletResponse.SC_OK);
                response.setContentType("text/plain");
                PrintWriter pw = response.getWriter();
                pw.println("Traitement asynchrone");
                pw.close();
            } catch (InterruptedException | IOException e) {
                e.printStackTrace();
            }
            System.out.println("Fin du traitement asynchrone");
            // Notification de la fin du traitement en appelant la méthode complete
            // La réponse est alors renvoyée au client
            contexteAsynchrone.complete();
        }
    });
    System.out.println("Fin du traitement de la requête, un traitement asynchrone a été lancé");
    System.out.println("Le thread est déjà prêt à traiter une autre requête client");
}
```