Serveur web dynamique en Java : les bases

bernard

Table des matières

1 Pourquoi un serveur web en Java

1.1 Pourquoi un serveur

Dans le cadre d'un système d'informations, on veut donner accès *simultané* à des informations à un nombre indéterminé d'utilisateurs.

1.2 Pourquoi un serveur web

Un serveur web est en fait :
— un serveur internet ou intranet, c'est-à-dire utilisant les protocoles de transport TCP/IP
— utilisant le protocole de communication HTTP.

L'intérêt d'utiliser les protocoles TCP/IP est évidemment de tirer partie de toute l'infrastructure réseau existante. L'intérêt d'utiliser HTTP est de tirer parti de tous les logiciels/bibliothèques existantes, notamment les navigateurs web.

1.3 Pourquoi un serveur web dynamique

Les informations gérées par le S.I. (Système d'Informations) peuvent être modifiées par les utilisateurs. En conséquence, il faut que le serveur puisse générer dynamiquement (en cours d'utilisation) les informations à envoyer. À contrario, un site web statique ne pourrait que servir des informations figées.

1.4 Pourquoi en Java

Java est un langage populaire pour la réalisation de sites web dynamiques pour plusieurs raisons :

concurrence/parallélisme Il est possible de gérer les connections de plusieurs utilisateurs en parallèle, tirant ainsi parti des architectures multi-cœurs des ordinateurs.

performance En plus de tirer parti de tous les cœurs d'un ordinateur, chacun de ceux-ci est utilisé efficacement grâce à la performance de la JVM.

portabilité Grâce à la JVM, un serveur programmé en Java peut facilement être déployé sur n'importe quelle architecture disposant d'une JVM. Cela permet notamment d'externaliser l'hébergement du site (cf. *cloud*, IaaS voire PaaS).

fiabilité Le fait que Java soit un langage compilé à typage statique aide à la réalisation de programmes fiables, même si les erreurs de compilation **ne remplacent pas** les tests !

disponiblitié de bibliothèques/frameworks Le fait que les autres qualités aient fait reconnaître l'intérêt d'utiliser Java pour la programmation de serveurs web a provoqué le développement de nombreuses bibliothèques et frameworks qui facilitent la programmation de serveurs.

2 Principes de la programmations de serveurs dynamiques Web en Java

2.1 TCP/IP et HTTP

TCP/IP permet de désigner une ressource à l'aide d'une URL. Cette URL indique :

— le protocole (HTTP/HTTPS)
— le nom du serveur (qui permet d'obtenir l'adresse IP grâce au DNS)
— le port (par défaut 80 pour un serveur web en HTTP et 443 pour le HTTPS)

Pour le port, on doit tenir compte du fait que les ports "de notoriété publique" (*well known ports*), et notamment ceux qui nous intéressent, ne peuvent être utilisés par un simple utilisateur et requiert donc les privilèges administrateur. En pratique, on développera en utilisant un autre port disponible pour les simples utilisateurs et libre. On rendra paramétrable au déploiement le port utilisé par notre serveur. En effet, il ne peut y avoir qu'un seul programme en charge d'un port donné sur un machine donnée. Pour cette raison, on devra s'assurer de terminer le programme en court d'exécution avant de le relancer lors du cycle de développement.

2.2 Servlet, containers et serveurs embarqués

En Java, le composant de base de la programmation d'un serveur est une *Servlet*. Il y a deux façons de déployer :

- Sous la forme d'un WAR à déployer sur un container de servlets lui-même déjà déployé (par exemple un Tomcat)
- Sous la forme d'un programme java autonome (JAR, le plus souvent "fat JAR" incluant toutes les dépendances), qui contient lui-même le serveur (par exemple Jetty).

De plus en plus, on préfère des livrables les plus autonomes possibles (par exemple des machine virtuelles embarquant jusqu'au système d'exploitation !) et par soucis de simplicité (pour ne pas avoir à installer de serveur Tomcat), on commencera par utiliser un serveur Jetty embarqué. Ensuite, on utilisera un framework qui permettra (notamment à l'aide de maven), de générer aussi bien un fat JAR autonome qu'un WAR destiné à un container de Servlets.

3 Embarquer un serveur Jetty

Ci-dessous, les étapes nécessaires pour la réalisation d'un serveur web minimal consultable sur un port donné (i.e. 9092)

3.1 Configuration

On utilise Maven pour gérer les dépendances de nos projets. Pour réaliser un serveur web embarquant Jetty, il suffit d'ajouter les dépendances `jetty-servlet` et `jetty-server` de `org.eclipse.jetty` dans le fichier `pom.xml`. Par exemple :

```
1   <dependency>
2     <groupId>org.eclipse.jetty</groupId>
3     <artifactId>jetty-server</artifactId>
4     <version>9.4.6.v20170531</version>
5   </dependency>
6   <dependency>
7     <groupId>org.eclipse.jetty</groupId>
8     <artifactId>jetty-servlet</artifactId>
9     <version>9.4.6.v20170531</version>
10  </dependency>
```

3.2 Code de serveur Jetty élémentaire

On peut créer un objet de classe `org.eclipse.jetty.server.Server` associé à un port disponible quelconque (par exemple `9092`). Ensuite, on utilise un objet de classe `org.eclipse.jetty.servlet.ServletHandler` pour gérer les servlets qui implémenteront notre serveur. Ensuite, on peut associer, dans ce `handler`, une classe implémentant l'interface Servlet . Ensuite, on lance le serveur dans un nouveau flux d'exécution concurrent et l'on attend la fin de celui-ci :

```
1  package co.simplon.poleEmploi.server;
2
3
4  import org.eclipse.jetty.server.*;
5  import org.eclipse.jetty.servlet.*;
6
7  public class HelloServer {
8
9     public static void main(String args[]) throws Exception{
10           Server server = new Server(9092);
11           ServletHandler handler = new ServletHandler();
12           server.setHandler(handler);
13           handler.addServletWithMapping(HelloGenericServlet.class, "/*");
14           server.start();
15           server.join();
16    }
17 }
```

3.3 Code de Servlet HTTP élémentaire

Les servlets permettent d'implémenter tout types de serveurs, mais pour un serveur HTTP, on utilisera plus précisément la classe javax.servlet.GenericServlet . Il suffit alors de définir la méthode service. Les arguments sont la requête et la réponse à construire. Pour cette dernière, on utilise le fait que l'argument passé est une référence : les modifications effectuées sur l'objet de classe javax.servlet.ServletResponse seront donc disponibles pour le code appelant.

```
1  package co.simplon.poleEmploi.server;
2
3  import java.io.IOException;
4
5  import javax.servlet.GenericServlet;
6  import javax.servlet.ServletException;
7  import javax.servlet.ServletRequest;
8  import javax.servlet.ServletResponse;
9
10 public class HelloGenericServlet extends GenericServlet {
11   private static final long serialVersionUID = 1L;
```

```
12    @Override
13    public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
14          response.getWriter().println("Hello from HelloGenericServlet !\n"+"got:\n"+request);
15
16    }
17
18 }
```

Il suffit ensuite de se rendre à l'aide d'un navigateur sur l'adresse `http://localhost:9092/` . On peut aussi utiliser n'importe quel autre client :

```
1  wget --quiet http://localhost:9092/ -O /dev/stdout
```

ou

```
1  curl http://localhost:9092
```

3.4 Notion de cycle de vie (lifecycle)

On remarque que l'argument de l'appel de la méthode addServletWithMapping n'est pas une instance de notre classe `HelloGenericServlet` mais une instance d'une classe générique Class (!) qui représente une classe en Java, ici l'objet représente notre classe `HelloGenericServlet`. Mais nous n'avons pas instancié d'objet de cette classe avec un appel à `new`. C'est le serveur qui aura le contrôle, entre autres, de l'instanciation et de l'initialisation de l'objet servlet.

4 Serveur : inversion de contrôle, multithread

4.1 Inversion de contrôle

Au niveau de l'architecture du code, il y a un changement fondamental entre les programmes "classiques" en ligne de commande que nous avons implémentés jusqu'à présent et les serveurs. En effet, lorsque les programmes "classiques" en ligne de commande sont lancés, ils effectuent une tâche en demandant éventuellement à l'environnement (utilisateur, disque, réseau, bases de données,...), des informations avant de produire un résultat et de se terminer. Pour un serveur, comme pour certaines applications avec une interface graphique, le programme n'a pas l'initiative : il passe son temps à attendre des requêtes/évènements pour y répondre. Le code qui implémente la réponse n'est pas appelé explicitement par le programme principal, mais enregistré pour être appelé lorsqu'une requête/un évènement survient. Parmi les

conséquences de ce mécanisme, il y a le fait que les signatures sont souvent prédéterminées. On pourrait être tenté de contourner cette contrainte en utilisant des attributs en lecture et en écriture pour contourner cette contrainte. Cependant, pour le cas des serveur, le fait que plusieurs requêtes puissent arriver en même temps rend les modifications par *effets de bord* périlleuses, à cause des accès concurrent de la programmation multithread.

4.2 Multithread, accès concurrents

Pour des raisons de qualité de service, on voudra évidemment pouvoir répondre à plusieurs requêtes en même temps. Cela pose un problème particulier si l'implémentation modifie un état partagé, en raison d'accès concurrents. En Java, on utilise le multithread pour implémenter de façon performante les serveurs, mais il faut alors soit éviter de modifier un état global, soit synchroniser ces modifications comme on va le constater.

4.3 Implémentation d'une servlet avec effet de bord

On va implémenter une servlet qui modifie des compteurs `counterA` et `counterB`, transférant le contenu d'un compteur à l'autre. À chaque requête, un compteur est décrémenté et l'autre est incrémenté, donc leur somme doit rester constante. Si ce n'est pas le cas, on est dans une situation "impossible" et l'on compte le nombre d'appels où le serveur était dans une telle situation :

```
1  package co.simplon.poleEmploi.server;
2
3  import java.io.IOException;
4
5  import javax.servlet.GenericServlet;
6  import javax.servlet.ServletException;
7  import javax.servlet.ServletRequest;
8  import javax.servlet.ServletResponse;
9
10 public class StatefulServlet extends GenericServlet {
11   private static final long serialVersionUID = 1L;
12   public static final long SUM=10000;
13   private long counterA= SUM;
14   private long counterB=0;
15   private boolean fromAToB= true;
16   private long impossibleCounter=0;
17   private long slowTransfert(final long v, final long delta) {
18     System.err.println("slowTransfert called");
19     return        v+delta;
20   }
21     @Override
22     public void service( ServletRequest request,
23                          ServletResponse response ) throws ServletException,
```

```
                                                        IOException {
        if (fromAToB) {
            if(counterA > 0) {
              counterB=slowTransfert(counterB, 1);
              counterA=slowTransfert(counterA, -1);
            }else {
              fromAToB=false;
            }
          }else {
            if(counterB > 0) {
              counterA=slowTransfert(counterA, 1);
              counterB=slowTransfert(counterB, -1);
            }else {
              fromAToB= true;
            }
          }
        if((counterA+counterB) != SUM) {
            ++impossibleCounter;
          }
          response.getWriter().println("a= "+counterA+" b= "+counterB+" sum = "+
          (counterA+counterB)+"\n going from "+
          (fromAToB ? "A to B":"B to A")+"\n impossible count="+impossibleCounter+"\n");
      }
}
```

Si l'on se rend à l'adresse `http://localhost:9092/`, autant de fois que l'on veut, le serveur semble fonctionner correctement. Mais si l'on décide de faire un grand nombre de requêtes *en parallèle*, les comptes ne sont plus justes et le serveur est dans un état "impossible" :

```
seq 100 |parallel -j 100 curl localhost:9092  &>/dev/null
```

— Comment expliquer cela ?
— Que se passe-t'il si l'on enlève l'appel à `System.err.println("slowTransfert called");` ?
— Que se passe-t'il si à la place, on execute une requête vers une base de données ?

4.4 Résolutions de problèmes d'accès concurrents

Une première solution pourrait être de *synchroniser* la méthode `service`. On peut pour cela utiliser le mot clé synchronized, mais quelles seraient les conséquences ?

Quelle solution doit-on utiliser pour avoir un serveur acceptable ?

5 HTTP

Lorsque l'on implémente un serveur HTTP, on peut utiliser (dériver de-) la classe javax.servlet.http.HttpServlet plutôt que la classe javax.servlet.GenericServlet afin d'avoir des méthodes plus spécifiques correspondant aux différents *verbes* ou *méthodes* du protocole HTTP, ainsi qu'à des requête et réponse spécifique prenant en compte diverses *méta-données*.

5.1 Principales méthodes HTTP

On s'intéressera tout d'abord seulement aux méthodes HTTP utilisées pour l'implémentation de sites/services web, en insistant sur celles qui sont utilisables directement à partir de pages HTML : `GET` et `POST`.

5.1.1 GET

La méthode `GET` est celle qui est utilisée par un navigateur web lorsque l'on visite une page web en indiquant une URL ou en cliquant sur un lien. Cela correspond à la lecture des données associées à l'URL, sans modification des informations stockées côté serveur.

5.1.2 POST

La méthode `POST` est celle qui est utilisée par un navigateur web lorsque l'on envoie le contenu d'un formulaire. Cela correspond au cas général de l'envoi d'informations qui vont avoir un effet côté serveur.

5.1.3 PUT

La méthode `PUT` concerne elle l'envoi des données qui correspondent à l'URL, en réciproque de `GET`. Cette méthode doit être *idempotente*, c'est-à-dire qu'un même `PUT` doit pouvoir être répété plusieurs fois sans que le résultat soit différent côté serveur que si le `PUT` n'était fait qu'une seule fois. Pour illustrer avec du code, l'instruction `x= 4;` est idempotente alors que `x= x + 4;` ne l'est pas.

5.1.4 DELETE

La méthode `DELETE` concerne elle la suppression des données qui sont à l'URL concernée.

5.2 Principales méta-données HTTP

5.2.1 Code de status

La première information qu'on peut considérer comme méta-donnée est le code de statut qui accompagne la réponse du serveur. Tout le monde connaît sans doute le fameux code 404 qui indique qu'il n'y a rien à l'URL indiquée, et il y a tout une liste de codes pour différents cas de figure. Lorsque l'on programme en Java, on n'écrira bien sûr pas le code numérique mais l'on utilisera les constantes nommées de la classe javax.servlet.http.HttpServletResponse, comme HttpServletResponse.SC_OK ou HttpServletResponse.SC_INTERNAL_SERVER_ERROR.

5.2.2 Entêtes (*headers*)

Aussi bien la requête que la réponse peuvent contenir des informations sous la forme de entêtes. La liste des entêtes est longue d'autant qu'il est possible d'ajouter des entêtes en plus de ceux qui sont standardisés. On s'intéressera principalement aux suivants :

1. Content-Type Cet entête indique le type de contenu qui est renvoyé. Par exemple du HTML, qui est une sous-catégorie du texte, et l'encodage, par exemple l'utf-8 : `Content-Type: text/html; charset=utf-8`.
2. Set-Cookie Cet entête permet d'enregistrer un cookie sur le client. Pour ajouter un cookie à une réponse, on n'utilisera pas de méthode générique de type addHeader sur la réponse, mais la méthode spécifique addCookie. Il est possible d'ajouter plusieurs cookies à une même réponse HTTP.

6 Contenu statique et contenu dynamique

Le plus souvent, le contenu à envoyer en réponse à une requête n'est pas purement dynamique. Évidemment, on voudra mettre le moins possible de données dans le code, donc on voudra pouvoir aussi renvoyer le contenu de fichiers : il s'agit de contenu statique.

6.1 DefaultServlet

On peut envoyer le contenu de fichiers (ou de répertoires) correspondant à une URL donnée en utilisant la class DefaultServlet fournie par Jetty. Il est possible de permettre ou non de renvoyer le contenu des répertoires lorsque l'URL correspond à un répertoire et non un fichier.

6.2 WelcomeFiles

Plutôt que de renvoyer le contenu d'un répertoire lorsqu'aucun nom de fichier n'est indiqué dans l'URL, on veut renvoyer le contenu d'un fichier. On parle alors de Welcome Files. On peut spécifier, par programmation (setWelcomeFiles) ou par configuration (dans web.xml), la liste ordonnée des noms de fichiers à essayer pour trouver le contenu à renvoyer lorsqu'aucun fichier n'est indiqué dans l'URL. On utilise pour cela le plus souvent le fichier `index.html` :

```
String [] welcomeFiles = {"index.html"};
context.setWelcomeFiles(welcomeFiles);
context.setResourceBase("./src/main/resources/");
```

Comme on le voit, il ne faut pas oublier de spécifier le répertoire à partir duquel rechercher les ressources, avec setResourceBase ou en utilisant setInitParameter sur le `ServletHolder` : `setInitParameter("dirAllowed","true")` ou `setInitParameter("dirAllowed","false")`.

7 Utilisation d'une page HTML pour interagir avec le site

7.1 Liens et formulaires

7.1.1 Requête GET avec balise <a>

Il suffit d'utiliser une balise `a` avec un lien `href` vers l'URL pour laquelle on a associé une servlet :

```
<a href="./dynamic">lien vers Servlet</a>
```

7.1.2 Requête POST avec un formulaire

On peut utiliser un formulaire avec le paramètre `method` à `"post"` et le paramètre `action` indiquant l'URL pour laquelle on a associé une servlet :

```
<form action="./dynamic" method="post">
    <div>
        <label for="nom">Nom :</label>
        <input type="text" id="name" name="name" />
    </div>
<div>
    Password: <input type="password" name="password"/> <br/>
```

```
</div>
    <div>
        <label for="courriel">Courriel :</label>
        <input type="email" id="email" name="email"/>
    </div>
    <div>
        <label for="message">Message :</label>
        <textarea id="message" name="message"></textarea>
    </div>
    <div class="button">
        <button type="submit">Envoyer votre message</button>
    </div>
</form>
```

Ensuite, on peut récupérer les paramètres côté serveur à partir de l'objet de type `HttpServletRequest`, soit en récupérant le corps de la requête (*body*) avec la méthode getReader(), ou (exclusif !) en utilisant les méthodes getParameterMap() ou getParameterNames() et getParameterValues(String paramName).

7.2 Javascript et méthodes HTTP

Si l'on veut utiliser d'autres méthodes (`PUT`, `DELETE`), il faut utiliser du code javascript. Avec la bibliothèque `jQuery`, on peut par exemple écrire :

```
<script>
  $(document).ready(function(){
    $('#DeleteButton').click(function(){
      $.ajax({
          url: '/dynamic/data/123',
          method: 'DELETE'
        })
          .done(function( data ) {
            console.log(data);
          });
    });
  });
</script>

<input type="button" value="deleteValue" id="DeleteButton" >
```

après avoir inclus la bibliothèque jQuery :

```
<script
        src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
        integrity="sha256-k2WSCIexGzOj3Euiig+TlR8gA0EmPjuc79OEeY5L45g="
        crossorigin="anonymous"></script>
```

Côté serveur, il faut bien sûr implémenter la méthode doDelete.

Pour voir le résultat, il faut afficher la console du navigateur.
On peut faire la même chose avec une méthode `PUT`.

8 Documents dynamique

On a vu qu'on pouvait retourner une chaîne de caractère construite dynamiquement par une servlet, ou que l'on pouvait retourner le contenu d'un fichier. Parfois, on voudra pouvoir générer dynamiquement des fragment d'un document dont la mise en page sera, elle, statique. On utilise généralement pour cela des mécanismes de *templating*, par exemple avec de JSP (*Java Server Pages*).

9 Du site web dynamique à la webapp

Si l'on désire implémenter un client *CRUD* (*Create*, *Read*, *Update*, *Delete*) :

— quelles méthodes HTTP semblent pertinentes ?
— quelles URLs semblent pertinentes (cf RESTful) ?
— quel format de données semble pertinent ?