SAE 2.03

Date de création	@13 juin 2025 16:36
Étiquettes	

Lecture du fichier `configuration.xml`

Il s'agit ici de **parser** les élements, pour cela on a décidé de créer une classe `Config` prenant chaque champ du fichier de configuration comme un attribut.

```
public class Config {
    private int port = 80;
    private String documentRoot = ".";
    private String accessLog;
    private String errorLog;
  ...
```

Pour parser, on utilise la librairie `javax.xml.parsers`

```
import javax.xml.parsers
```

On pourra ensuite faire appel à ces attributs grâce aux `getter` créés comme suit :

```
socket_serv = new ServerSocket(config.getPort());
```

Fonctionnalité 1 : Le port d'écoute sur le réseau

L'objectif de cette fonctionnalité est de rendre le port d'écoute du serveur configurable, sans nécessiter de recompilation. Ce port est défini dans le fichier XML de configuration à l'aide de la balise `<port>`. Le serveur lit

cette valeur au lancement. Dans la classe `Config`, on extrait cette valeur comme cela:

```
this.port = Integer.parseInt(root.getElementsByTagName("port").item(0).getTextContent());
```

La méthode `getPort()` permet ensuite de récupérer ce port dans la classe `HttpServer`. Lors de la création du `ServerSocket`, on utilise ce port pour démarrer le serveur sur le port configuré :

```
socket_serv = new ServerSocket(config.getPort());
System.out.println("Serveur démarré sur le port " + port);
```

Cela permet à l'utilisateur de modifier le port dans le fichier XML sans avoir à toucher au code Java.

Fonctionnalité 2 : La spécification d'un répertoire servant de base au site web

Cette fonctionnalité permet de définir dynamiquement le dossier à partir duquel les fichiers HTML sont servis. Le chemin est défini dans le fichier XML à l'aide de la balise `<DocumentRoot>`. Dans la classe `Config`, le document root est récupéré comme suit :

```
this.documentRoot = root.getElementsByTagName("DocumentRoot").item(0).getTextContent();
```

Dans `HttpServer`, ce chemin est utilisé pour construire le chemin absolu du fichier demandé par le client. Cela se fait dans la méthode `envoyerPageHtml` :

```
File fichierHtml = new File(config.getDocumentRoot() + cheminPage);
```

Cette ligne permet de combiner le document root avec le chemin relatif demandé pour localiser le fichier à servir. Ainsi, en changeant simplement le document root dans le fichier XML, on peut déployer le serveur sur un autre site web sans recompilation.

Fonctionnalité 3 : La spécification d'une propriété de lecture des répertoires

Cette fonctionnalité consiste à permettre au serveur d'afficher automatiquement le contenu d'un répertoire lorsqu'aucun fichier `index.html` n'est présent dans celui-ci. Cela permet de naviguer facilement à travers les fichiers et sous-dossiers hébergés.

Dans la méthode `envoyerPageHtml`, le serveur vérifie si le fichier demandé existe. Si ce n'est pas le cas, et que le chemin demandé se termine par `/index.html`, il considère qu'il s'agit d'un dossier et appelle la méthode `envoyerListingRepertoire` :

```
if (!fichierHtml.exists()) {
    if (cheminPage.endsWith("/index.html")) {
        String cheminRepertoire = cheminPage.substring(0, cheminPage.lastIndexOf("/index.html"));
        File repertoire = new File(config.getDocumentRoot() + cheminRepertoire);
        envoyerListingRepertoire(client, repertoire, cheminRepertoire);
    } else {
        envoyerErreur404(client);
    }
    return;
}
```

Dans `envoyerListingRepertoire`, le serveur lit tous les fichiers présents dans le répertoire et ***construit dynamiquement*** une page HTML :

```
File[] fichiers = repertoire.listFiles();
for (File f : fichiers) {
    String nom = f.getName();
    String lienComplet;
    if (f.isDirectory()) {
        lienComplet = cheminRelatif + "/" + nom + "/index.html";
    } else {
        lienComplet = cheminRelatif + "/" + nom;
    }
    String affichage = f.isDirectory() ? nom + "/" : nom;
    html.append("<li><a href=\"").append(lienComplet).append("\">").append(affichage).append("</a></li>");
}
```

Cette méthode génère donc une page HTML contenant tous les liens vers les fichiers et dossiers présents, ce qui offre une interface de navigation automatique à travers l'arborescence.

Fonctionnalité 4 : La spécification d'adresses IP qui seront acceptées ou refusées par le serveur

Cette fonctionnalité permet de filtrer les connexions entrantes en fonction de leur adresse IP. Le fichier XML permet de spécifier les listes d'adresses acceptées et rejetées, l'ordre de traitement (`accept` avant `reject`, ou l'inverse), ainsi qu'une politique par défaut (`accept` ou `reject` pour les IP non listées). Dans le XML, cela ressemble à ceci :

```
<security>
    <order>
        <first>accept</first>
        <last>reject</last>
    </order>
    <default>reject</default>
    <accept>127.0.0.1</accept>
    <reject>192.168.1.1</reject>
</security>
```

Dans la classe `Config`, les IP acceptées et refusées sont stockées dans des `HashSet` (permettant de stocker un ensemble d'éléments uniques sans ordre particulier, ce qui faisait sens dans ce contexte. Un ami nous a conseillé cette structure car plus rapide, mais on aurait clairement pu utiliser des ArrayList) :

```
private Set<String> acceptList = new HashSet<>();
private Set<String> rejectList = new HashSet<>();
```

La méthode `estAutorise` détermine si une adresse IP est autorisée à se connecter :

```
public boolean estAutorise(String ip) {
    ip = ip.trim();
    if (acceptFirst) {
        if (acceptList.contains(ip)) return true;
        if (rejectList.contains(ip)) return false;
    } else {
```

```
        if (rejectList.contains(ip)) return false;
        if (acceptList.contains(ip)) return true;
    }
    return defaultPolicy.equals("accept");
}
```

Dans la méthode `main` de `HttpServer`, cette vérification est faite immédiatement après l'acceptation du socket.

On a d'ailleurs pris un certain temps avant de comprendre que le localhost donnait l'adresse en IpV6. Comme cela ne concerne apparemment que le localhost, nous avons décidé de remplacer par l'adresse connue plutôt que de créer un algorithme de conversion.

```
String ipClient = socket_client.getInetAddress().getHostAddress();
if (ipClient.equals("0:0:0:0:0:0:0:1")) {
    ipClient = "127.0.0.1";
}
if (!config.estAutorise(ipClient)) {
    Logger.logErreur("Connexion refusée : " + ipClient);
    socket_client.close();
    continue;
}
Logger.logAcces("Connexion autorisée depuis : " + ipClient);
```

Si l'IP est refusée, la connexion est immédiatement interrompue et une ligne est ajoutée au fichier de log des erreurs.

Fonctionnalité 5 : Enregistrement des accès et des erreurs

Cette fonctionnalité permet de conserver une trace des événements du serveur, à savoir les connexions autorisées et refusées, ainsi que les erreurs comme les fichiers non trouvés. Le fichier XML permet de spécifier les chemins des fichiers de log à l'aide des balises `<accesslog>` et `<errorlog>`.

Dans la classe `Logger`, les chemins sont enregistrés et utilisés pour écrire les messages dans les fichiers correspondants. Les fichiers sont créés automatiquement s'ils n'existent pas, à l'aide de la méthode suivante :

```
private static void ensureFileExists(String path) {
    try {
        File file = new File(path);
        File parent = file.getParentFile();
        if (parent != null && !parent.exists()) {
            parent.mkdirs();
        }
        if (!file.exists()) {
            file.createNewFile();
        }
    } catch (IOException e) {
        System.err.println("Impossible de créer le fichier de log : " + path);
    }
}
```

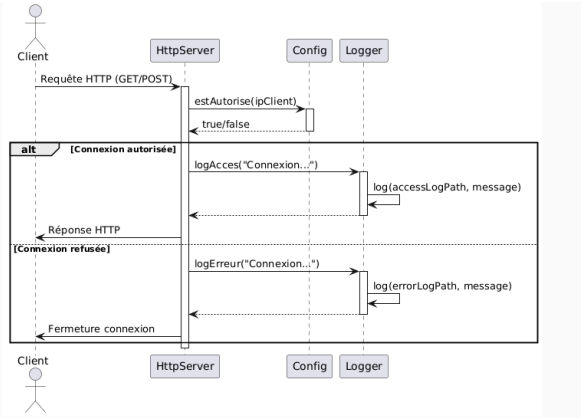
L'écriture d'un message dans les logs se fait avec :

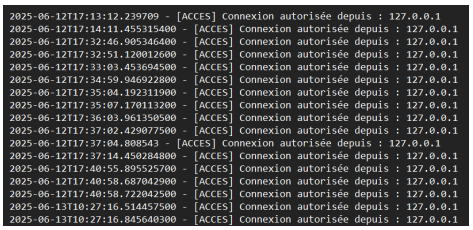
```
try (FileWriter fw = new FileWriter(file, true)) {
    fw.write(LocalDateTime.now() + " - " + message + "\n");
    fw.flush();
}
```

Ces logs sont utilisés dans `HttpServer` pour enregistrer chaque connexion ou refus, par exemple :

```
Logger.logErreur("Connexion refusée : " + ipClient);
Logger.logAcces("Connexion autorisée depuis : " + ipClient);
```

Cela permet de suivre l'activité du serveur et de détecter les tentatives de connexion non autorisées.





fichier de log

Fonctionnalité 6 : La possibilité d'afficher l'état de la machine

Cette fonctionnalité consiste à fournir une page spéciale accessible via l'URL `/status`, qui affiche des informations comme la mémoire libre, l'espace disque disponible, le nombre de processus et d'utilisateurs connectés.

1. Appel dans `envoyerPageHtml()`

Lorsqu'un client demande `/status`, le serveur intercepte la requête et appelle la méthode dédiée :

```
if (cheminPage.equals("/status")) {
    envoyerStatus(client);
    return;
}
```

---

2. Récupération des Métriques Système

La méthode `envoyerStatus()` utilise des API Java (**`Runtime`**, **`FileSystems`**, **`ProcessHandle`**) pour obtenir :

- **Mémoire disponible** (`Runtime.freeMemory()`)
- **Espace disque** (`FileStore.getUsableSpace()`)
- **Nombre de processus** (`ProcessHandle.allProcesses().count()`)
- **Nombre d'utilisateurs** (variable statique `nbUtilisateurs`)

```
Runtime runtime = Runtime.getRuntime();
long memLibre = runtime.freeMemory();
```

```
long storageLibre = 0;
for (var store : java.nio.file.FileSystems.getDefault().getFileStores()) {
   storageLibre += store.getUsableSpace();
}

// Nombre de processus en cours
long nbProc = ProcessHandle.allProcesses().count();
```

3. Génération de la Page HTML

Les données sont formatées en HTML pour être affichées dans le navigateur :

```
String html = "<html><body>" +
   "<h1>Statut du serveur</h1>" +
   "<ul>" +
   "<li>Mémoire disponible : " + (memLibre / (1024 * 1024)) + " Mo</li>" +
   "<li>Espace disque : " + (storageLibre / (1024 * 1024 * 1024)) + " Go</li>" +
   "<li>Processus : " + nbProc + "</li>" +
   "<li>Utilisateurs : " + nbUtilisateurs + "</li>" +
   "</ul></body></html>";
```

4. Envoi de la Réponse HTTP

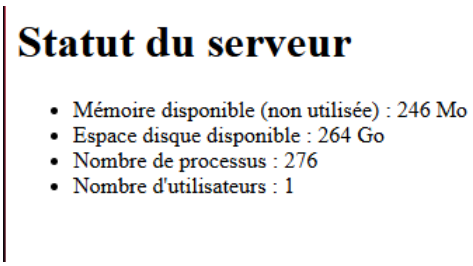
Le serveur envoie la réponse avec le bon en-tête (`Content-Type: text/html`), ce qui évite toute gestion de variables à intégrer dans des pages:

```
String enteteHttp = "HTTP/1.1 200 OK\\r\\n" +
   "Content-Type: text/html; charset=UTF-8\\r\\n" +
   "Content-Length: " + html.getBytes(StandardCharsets.UTF_8).length + "\\r\\n\\r\\n";

client.getOutputStream().write(enteteHttp.getBytes());
client.getOutputStream().write(html.getBytes(StandardCharsets.UTF_8));
```

Exemple de Sortie

Un client accédant à `http://localhost:80/status` verra :



Fonctionnalité 7 : Encoder images, sons ou vidéos dans un format accepté par le navigateur

Cette fonctionnalité a pour but d'optimiser les performances du serveur en compressant certains fichiers avant de les transmettre. Typiquement, cela concerne les fichiers binaires (images, sons, vidéos), qui peuvent être compressés avec `gzip`. Cela nécessite de vérifier le type MIME du fichier, et d'envoyer l'en-tête `Content-Encoding: gzip` si applicable.

1. Vérification du type de média

La méthode `estFichierMedia()` détermine si un fichier est un média (image, audio, vidéo) en fonction de son extension.

```
static boolean estFichierMedia(String nomFichier) {
    String extension = nomFichier.substring(nomFichier.lastIndexOf('.') + 1).toLowerCase();
    return extension.matches("jpg|jpeg|png|gif|bmp|webp|svg|ico");
}
```

2. Compression GZIP pour les médias

Si le fichier est un média, il est compressé avec **GZIP** avant envoi pour réduire la taille des données transmises.

```
if (estFichierMedia(fichier.getName())) {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    try (GZIPOutputStream gzipOut = new GZIPOutputStream(baos)) {
        gzipOut.write(contenuFichier);
        gzipOut.finish();
    }
    byte[] contenuCompresse = baos.toByteArray();

    String enTeteHttp = "HTTP/1.1 200 OK\r\n" +
            "Content-Type: " + typeMedia + "\r\n" +
            "Content-Encoding: gzip\r\n" +
            "Content-Length: " + contenuCompresse.length + "\r\n" +
            "\r\n";

    sortie.write(enTeteHttp.getBytes("UTF-8"));
    sortie.write(contenuCompresse);
}
```

Comme on envoi un en-tête, il faut déterminer la bonne forme, d'où la classe `getTypeMedia(String nomFichier)`

```
static String getTypeMedia(String nomFichier) {
    String extension = nomFichier.substring(nomFichier.lastIndexOf('.') + 1).toLowerCase();
    switch (extension) {
        case "jpg":
        case "jpeg": return "image/jpeg";
        case "png":  return "image/png";
        case "gif":  return "image/gif";
```

Malheureusement, dans un soucis de temps et de compréhension, nous n'avons pas trouvé de moyen de vérifier le bon fonctionnement de la compression, et de mesurer son efficacité de manière précise. Nous avons toutefois pu nous aider de `System.Out.Print()` ainsi que de tutoriels et de forum pour nous assurer d'être sur la bonne piste.

Fonctionnalité 8 : Gérer les formulaires HTML

Cette fonctionnalité concerne la réception de données envoyées depuis des formulaires HTML via les méthodes `GET` ou `POST`.
Elle nécessite d'analyser les paramètres de la requête et de les transmettre à un programme externe pour traitement.

Elle a été la partie la plus fastidieuse car nous avions initialement songé à utiliser php, vu en terminale, ce qui n'a pas fonctionné.

En nous penchant sur l'énoncé, nous avons réalisé qu'il serait possible d'intercepter les éléments depuis notre programme et de gérer la réception des informations du formulaire en java, notamment grâce aux flux de

données.

1. Structure du Code Principal

La méthode `traiterFormulaire()` dans `HttpServer.java` gère le traitement des requêtes POST.

```
static void traiterFormulaire(Socket client, BufferedReader reader) throws IOException {
    // 1. Lecture des en-têtes pour obtenir la taille des données
    int longueurContenu = 0;
    String line;
    while ((line = reader.readLine()) != null && !line.isEmpty()) {
        if (line.startsWith("Content-Length:")) {
            longueurContenu = Integer.parseInt(line.substring("Content-Length:".length()).trim());
        }
    }

    // 2. Lecture du corps de la requête
    char[] buffer = new char[longueurContenu];
    reader.read(buffer, 0, longueurContenu);
    String donnees = new String(buffer);

    // 3. Parsing des données (format: user_name=nom&user_mail=email)
    String[] paires = donnees.split("&");
    String nom = "", email = "";
    boolean possedeNom = false, possedeEmail = false;

    for (String paire : paires) {
        String[] kv = paire.split("=");
        if (kv.length == 2) {
            if (kv[0].equals("user_name")) {
                nom = java.net.URLDecoder.decode(kv[1], "UTF-8");
                possedeNom = true;
            } else if (kv[0].equals("user_mail")) {
                email = java.net.URLDecoder.decode(kv[1], "UTF-8");
                possedeEmail = true;
            }
        }
    }
    // ...
}
```

Étape 1 : Détection de la Requête POST

- Le serveur identifie une requête POST via l'analyse de la première ligne HTTP :

  ```
  POST /programme HTTP/1.1
  ```

- Il appelle alors `traiterFormulaire()`.

Étape 2 : Lecture des Données

1. **Récupération de la taille des données** :
   - Lecture de l'en-tête `Content-Length` pour connaître la taille du corps de la requête.
2. **Extraction des données brutes** :
   - Les données sont lues depuis le flux d'entrée (`BufferedReader`).

Étape 3 : Parsing des Données

- Les données sont au format `user_name=John&user_mail=john@example.com`.
- Découpage avec `split("&")` et `split("=")` pour isoler chaque paire clé-valeur.

```
// 3. Parsing des données (format: user_name=nom&user_mail=email)
  String[] paires = donnees.split("&");
  String nom = "", email = "";
  boolean possedeNom = false, possedeEmail = false;
```

- **Décodage URL** :
  - Les caractères spéciaux (ex: `%20` pour un espace) sont convertis avec `URLDecoder.decode()`

Étape 4 : Traitement des Données

Deux scénarios possibles:

1. **Soumission de formulaire** (nom + email) :
   - Stockage dans un fichier `userlist.txt` au format `nom;email`.
2. **Recherche d'utilisateur** (nom seul) :
   - Consultation du fichier pour retrouver l'email associé.

Extrait : Enregistrement en Fichier

```
File fichierUsers = new File(config.getDocumentRoot() + "/userlist.txt");
try (FileWriter fw = new FileWriter(fichierUsers, true)) {
   fw.write(nom + ";" + email + "\\n"); // Format: "John;john@example.com"
}
```

---

3. Réponse au Client

Le serveur génère une page HTML de confirmation/recherche avec :

- Un message de succès/erreur.
- Un bouton de retour vers le formulaire.

Exemple de Réponse :

```
String reponse = "HTTP/1.1 200 OK\\r\\nContent-Type: text/html\\r\\n\\r\\n" +
      "<html><body>Merci pour votre soumission!<br>Nom: " + nom + "<br>Email: " + email +
      "<a href=\\"index.html\\"><button>Retour</button></a></body></html>";
```
