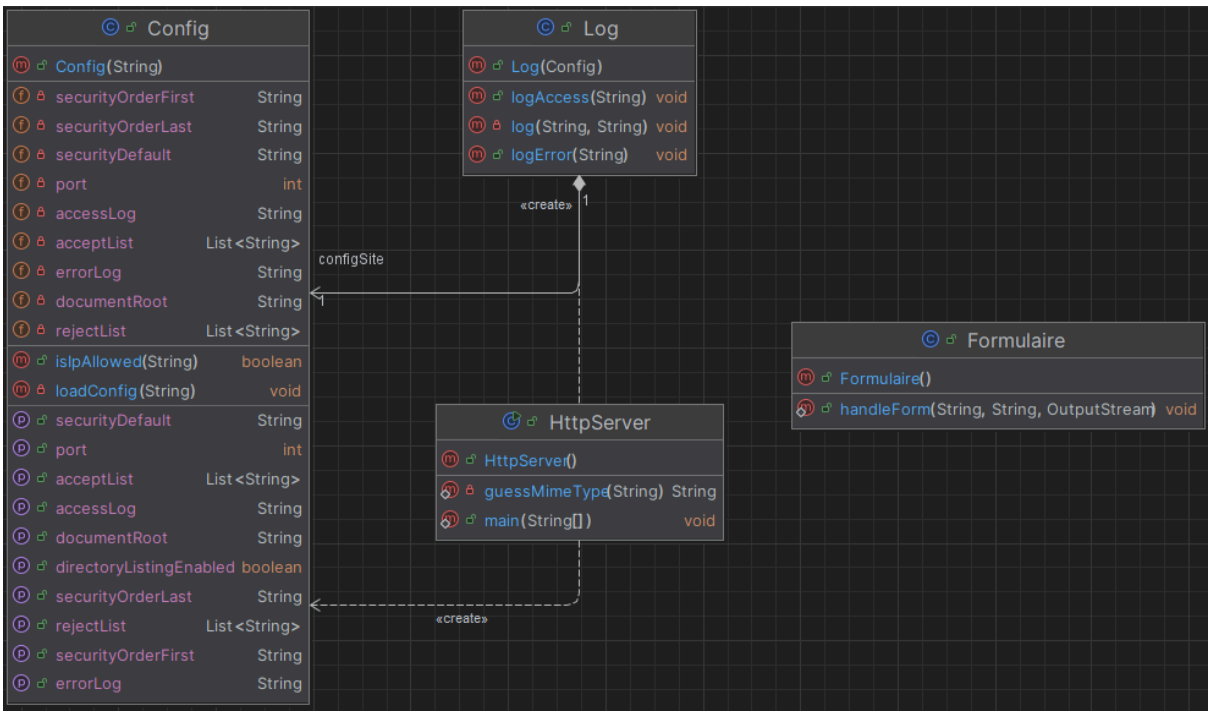
Rapport de la SAE 2.03 - Programmation d’un serveur Web

Logeart Pierre - Adam Tristan

Sommaire

1) Structure du projet

1. Diagramme de classe



2. Fonctionnement du fichier XML et de sa lecture avec la classe Config

Le fichier XML myweb.conf

Ce fichier permet de configurer dynamiquement le comportement du serveur HTTP.
Il contient les paramètres principaux du serveur, regroupés dans la balise <webconf>.

Exemple de structure :

```
<webconf>
    <port>8080</port>
    <DocumentRoot>web/</DocumentRoot>

    <Directory>
        <Options>Indexes</Options>
    </Directory>
    <security>
        <order>
            <first>accept</first>
            <last>reject</last>
        </order>
        <default>accept</default>
```

```
        <accept>192.168.0.0/24</accept>
        <reject>192.168.0.2/24</reject>
    </security>
    <accesslog>documents/access.log</accesslog>
    <errorlog>documents/error.log</errorlog>
</webconf>
```

Principaux paramètres :
<port> : Port d'écoute du serveur (ex. 8080)
<DocumentRoot> : Dossier où se trouvent les fichiers à servir
<Directory><Options> : Active l'affichage du contenu des répertoires (Indexes)
<security> : Règles de filtrage IP (accept, reject, default, order)
<accesslog> : Chemin du fichier journal des accès
<errorlog> : Chemin du fichier journal des erreurs

La classe Config

La classe Config est chargée de :

- Lire le fichier XML une seule fois au lancement du serveur
- Extraire tous les paramètres utiles
- Fournir des méthodes d'accès (getters) pour les autres classes (HttpServer, Log, etc.)

Fonctionnement interne :

- Lecture du fichier avec Java DOM (DocumentBuilder + org.w3c.dom).
- Parcours des balises XML avec getElementsByTagName(...).
- Extraction des textes avec .getTextContent().trim(). Le .trim() sert à retirer les espaces blancs avant et après le texte.
- Stockage des valeurs dans des attributs privés.

Méthodes utiles :

- Les getters qui retournent les éléments voulus.
- isIpAllowed(ip) qui vérifie si une IP est autorisée

Résultat obtenu

- Séparation entre la logique Java et la configuration.
- Le serveur devient entièrement modifiable via un simple fichier XML.

2) Fonctionnalités

Fonctionnalité 1 : Port configurable

Objectif

Permettre de choisir le port d'écoute du serveur HTTP via un fichier de configuration XML.

Fonctionnement attendu

Le port est défini dans le fichier myweb.conf via :

```
<port>8080</port>
```

Le serveur écoute sur ce port au lancement.

Implémentation technique

Lecture dans myweb.conf

```
<port>8080</port>
```

Si cette balise est absente ou incorrecte, la valeur par défaut 8080 est utilisée.

Lecture dans Config.java

```
NodeList portNode = root.getElementsByTagName("port");
if (portNode.getLength() > 0) {
    port =
Integer.parseInt(portNode.item(0).getTextContent().trim());
}
```

Stockage dans :

```
private int port = 8080;
```

Accès via :

```
public int getPort() { return port; }
```

Utilisation dans HttpServer.java

```
Config config = new Config("myweb.conf");
int portNumber = config.getPort();
ServerSocket server = new ServerSocket(portNumber);
```

Le serveur écoute automatiquement sur le port défini par la config.

Résultat obtenu

Configuration	Résultat
<port>8080</port>	Le serveur écoute sur le port 8080
<port>80</port>	Le serveur écoute sur le port 80 (si autorisé)
Balise manquante	Le serveur écoute sur le port 8080 (défaut)

Fonctionnalité 2 : Répertoire racine configurable (DocumentRoot)

Objectif

Permettre de définir le répertoire racine du serveur, c'est-à-dire l'endroit où sont stockés les fichiers à servir, sans modifier le code Java.

Cela permet de :

- Changer de site ou d'environnement sans recompiler
- Séparer le code du serveur et les fichiers web

Fonctionnement attendu

L'utilisateur spécifie un chemin vers un dossier local via la balise XML :
`<DocumentRoot>web/</DocumentRoot>`
Le serveur utilise ce chemin comme point de départ pour rechercher tous les fichiers demandés dans les requêtes.

Implémentation technique

Lecture dans myweb.conf
`<DocumentRoot>web/</DocumentRoot>`
Ce chemin est relatif ou absolu. Il peut pointer vers n'importe quel dossier sur le système.

Lecture dans Config.java

```
NodeList docRootNode = root.getElementsByTagName("DocumentRoot");
if (docRootNode.getLength() > 0) {
    documentRoot = docRootNode.item(0).getTextContent().trim();
}
```

Stockage dans :

```
private String documentRoot = ".";
```

Accès via :

```
public String getDocumentRoot() { return documentRoot; }
```

Utilisation dans HttpServer.java

```
String root = config.getDocumentRoot();
File file = new File(root, fileName);
```

Le chemin du fichier demandé est construit à partir de DocumentRoot + fichier demandé (GET /index.html ⇒ web/index.html).

Résultat obtenu

Cas	Résultat
Fichier existe dans DocumentRoot	Il est servi au client (200 OK)
Fichier n'existe pas	Erreur 404 + log erreur
Dossier mal configuré	Aucun fichier servi (404/403)

Fonctionnalité 3 : Affichage du contenu d'un dossier (Indexes)

Objectif

Permettre au serveur HTTP d'afficher automatiquement la liste des fichiers d'un répertoire (un "index") à condition que cela soit autorisé dans le fichier de configuration.

Fonctionnement attendu

Le serveur détecte qu'un client demande un répertoire.
Si l'option <Options>Indexes</Options> est définie dans le fichier myweb.conf, un listing HTML est généré.
Sinon le serveur renvoie une erreur 403 Forbidden.

Implémentation technique

Lecture dans le fichier XML myweb.conf :

```
<Directory>
    <Options>Indexes</Options>
</Directory>
```

Si la balise <Options> contient le mot Indexes, alors l'option est activée.

Lecture dans Config.java :

```
NodeList dirNodes = root.getElementsByTagName("Directory");
if (dirNodes.getLength() > 0) {
    Element dir = (Element) dirNodes.item(0);
    NodeList opts = dir.getElementsByTagName("Options");
    if (opts.getLength() > 0 &&
opts.item(0).getTextContent().contains("Indexes"))
        enableDirectoryListing = true;
}
```

Stockage du résultat dans un booléen :

```
private boolean enableDirectoryListing = false;
```

Méthode d'accès :

```
public boolean isDirectoryListingEnabled() { return
enableDirectoryListing; }
```

Traitement dans HttpServer.java
Le programme va tester si le fichier est un répertoire :

```
if (file.isDirectory())
```

Ensuite, il va tester si le listing est activé, et si c'est le cas, il va créer le listing en html et va l'envoyer :

```
StringBuffer listing = new StringBuffer("<html><body><ul>");
for (File f : file.listFiles()) {
```

```
      listing.append("<li><a href=\"")
            .append(fileName).append("/").append(f.getName())
            .append("\">").append(f.getName()).append("</a></li>");
}
listing.append("</ul></body></html>");

output.write("HTTP/1.1 200
OK\r\nContent-Type:text/html\r\n\r\n".getBytes());
output.write(listing.toString().getBytes());
```

Et si le listing est désactivé il va envoyer un message d'erreur.

Résultats obtenus

Cas	Résultat
Le répertoire contient index.html	index.html est servi normalement
Pas de index.html + Indexes ON	Listing HTML généré
Pas de index.html + Indexes OFF	Erreur HTTP 403

Fonctionnalité 4 : Filtrage IP (sécurité)

Objectif

Permettre au serveur HTTP de contrôler les accès en fonction de l'adresse IP de l'utilisateur. Cela permet de :

- Restreindre l'accès à certains réseaux/IP,
- Empêcher les connexions non autorisées,
- Contrôler les connexions client.

Fonctionnement attendu

Les règles de sécurité sont définies dans le fichier myweb.conf :

```
<security>
    <order>
        <first>accept</first>
        <last>reject</last>
    </order>
    <default>reject</default>
    <accept>192.168.0.0/24</accept>
    <reject>192.168.0.2/24</reject>
</security>
```

<accept> : IP ou sous-réseau explicitement autorisé
<reject> : IP ou sous-réseau explicitement interdit
<order> : Ordre de vérification (accept avant reject ou inversement)
<default> : Politique par défaut si l'IP ne correspond à aucune règle

Implémentation technique

La lecture est réalisé dans Config.java. La classe lit les balises est réalisé via les balises <accept>, <reject>, <order>, <default> sont lues via getElementsByTagName.

La méthode isIpAllowed(String ip) permet de savoir si une adresse ip est autorisé à accéder au site ou non par rapport à l'ordre définit dans le fichier de configuration.

```
public boolean isIpAllowed(String ip) {
    if (securityOrderFirst.equals("accept")) {
        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 securityDefault.equals("accept");
}
```

La méthode compare l'IP à chaque liste, en respectant l'ordre, et applique la règle par défaut si aucune correspondance.

On effectue aussi un traitement de l'adresse IPv6 (localhost) dans HttpServer :

```
if (clientIp.equals("0:0:0:0:0:0:0:1")) {
    clientIp = "127.0.0.1/32";
}
```

Cela garantit que localhost est reconnu comme 127.0.0.1/32 dans les règles.

Application dans HttpServer.java

```
if (!config.isIpAllowed(clientIp)) {
      String msgErreur = "<h1>403 - Acces refuse </h1>";
      output.write("HTTP/1.1 403 Access Denied\r\n".getBytes());
      output.write("Content-Type: text/html\r\n\r\n".getBytes());
      output.write(msgErreur.getBytes());
}
```

Résultat obtenu

Les connexions sont automatiquement acceptées ou refusées selon le fichier XML.

Fonctionnalité 5 : Journalisation (logs)

Objectif

Permettre au serveur HTTP de suivre son activité en enregistrant :

- Tous les accès clients
- Toutes les erreurs générées (403, 404, etc.)

Tout ceci dans deux fichiers distincts, pour faciliter le suivi et le débogage.

Implémentation technique

1. Configuration XML

Fichier *myweb.conf* :

```
<accesslog>documents/access.log</accesslog>
<errorlog>documents/error.log</errorlog>
```

2. Classe Config

Lecture des chemins depuis les balises XML depuis la classe *Config*:

```
NodeList accessNode = root.getElementsByTagName("accesslog");
if (accessNode.getLength() > 0) {
    accessLog = accessNode.item(0).getTextContent().trim();
}

NodeList errorNode = root.getElementsByTagName("errorlog");
if (errorNode.getLength() > 0) {
    errorLog = errorNode.item(0).getTextContent().trim();
}
```

Avec leurs getters :

```
public String getAccessLog() { return accessLog; }
public String getErrorLog() { return errorLog; }
```

3. Classe Log

La classe Log permet de gérer tout le mécanisme d'enregistrement des journaux.
Tout d'abord dans son constructeur, il va récupérer la configuration du serveur et va créer les deux fichiers *.log* s'ils ne sont pas encore créés.
La création des fichiers se font grâce à ces lignes (ici pour logAccess) :

```
File fileAccess = new File(c.getAccessLog());
if (!fileAccess.exists()) {
     try {
          fileAccess.createNewFile();
     } catch (IOException e) {
          System.out.println("Erreur de création du fichier
logAccess" + e.getMessage());
     }
}
```

Ensuite la méthode *log* va écrire le message passé en paramètre dans le fichier (.log) passé également en paramètre :

```
try {
      FileWriter fw = new FileWriter(fileName, true);
      fw.write("[" + new Date() + "] " + message + "\n");
      fw.close();
} catch (IOException e) {
       System.out.println("Erreur d'écriture du log : " +
e.getMessage());
}
```

Enfin, les deux méthodes *logAccess* et *logError* seront utilisé pour écrire le message passé en paramètre dans le fichier de log correspondant.

3. Utilisation dans HttpServer.java

Les méthodes *log.logAccess(message)* ou *log.logError(message)* selon la fichier où le log sera écrit.

Résultat obtenu

- Fichiers log créés automatiquement (même si les dossiers n'existent pas)
- Chaque événement important est tracé dans son fichier dédié sous la forme :
  `[Wed Jun 12 12:03:24 CEST 2025] Fichier servi à 127.0.0.1 : index.html`

Fonctionnalité 6 : Affichage de l'état de la machine

Objectif :

L'objectif de cette fonctionnalité est de permettre à l'utilisateur d'accéder aux informations système (accessible à l'adresse `http://@serveur/status`), celles-ci sont :

- La mémoire disponible
- L'espace disque disponible
- Le nombre de processus en cours
- Le nombre d'utilisateurs connectés

Implémentation :

Cette fonctionnalité est intégrée au sein du serveur HTTP Java, dans la méthode principale de traitement des requêtes. Lorsque le chemin `/status` est détecté dans la ligne de requête, une méthode dédiée est appelée pour générer dynamiquement une page HTML contenant les données système.

Détection de l'URL "`/status`":

```
if (fileName.equals("status")) {
    handleStatusRequest(output);
}
```

Méthode `handleStatusRequest` (permettant la réalisation de la fonctionnalité):

1. Mémoire disponible

Pour obtenir la mémoire libre, nous utilisons la méthode statique de la classe `Runtime`:

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

Elle retourne la mémoire libre en octets, on convertit ensuite cette valeur en Ko ou Mo pour l'affichage.

2. Espace disque disponible

Pour connaître l'espace disque disponible sur le système de fichiers racine, nous utilisons la classe `java.io.File`:

```
File root = new File("/");
long freeSpace = root.getFreeSpace();
```

La méthode `getFreeSpace()` retourne l'espace disque disponible en octets.

3. Nombre de processus:

Sous Linux, chaque processus possède un dossier dans `/proc` nommé par son PID (un entier). On peut alors compter les dossiers numériques pour estimer le nombre de processus :

```
int nbProcess = new File("/proc").list((dir, name) ->
name.matches("[0-9]+")).length;
```

Cela fonctionne uniquement sur les systèmes Unix/Linux.

4. Nombre d'utilisateurs connectés:

Pour connaître le nombre d'utilisateurs connectés, on utilise la commande Unix who, qui retourne une ligne par utilisateur connecté. On exécute cette commande depuis Java avec :

```
Process process = Runtime.getRuntime().exec("who");
BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
int userCount = 0;
while (reader.readLine() != null) userCount++;
```

Résultat

Le serveur génère une page HTML contenant les informations suivantes :

```
<h1>État de la machine</h1>
<p>Mémoire disponible : 123456 Ko</p>
<p>Espace disque disponible : 10000 Mo</p>
<p>Nombre de processus : 175</p>
<p>Nombre d'utilisateurs connectés : 2</p>
```

Le tout est envoyé avec les en-têtes HTTP:

```
output.write("HTTP/1.1 200 OK\r\n".getBytes());
output.write("Content-Type: text/html\r\n\r\n".getBytes());
```

Méthodes essentielles utilisées:

Fonctionnalité	Méthode utilisée	Source de documentation
Mémoire libre	`Runtime.getRuntime().freeMemory()`	Oracle JavaDoc

Espace disque	`File.getFreeSpace()`	Oracle JavaDoc
Nombre de processus	`new File("/proc").list(...)` + RegExp	Manuels Linux / Java
Utilisateurs connectés	`Runtime.getRuntime().exec("who")` + `BufferedReader`	Stack Overflow / man who

Fonctionnalité 7 : Encodage des fichiers multimédia

Objectif :

Cette fonctionnalité a pour but d'améliorer les performances de transfert du serveur HTTP en compressant les fichiers multimédia (images, sons, vidéos) avant de les envoyer au client.
L'encodage utilisé est GZIP, qui est pris en charge par la majorité des navigateurs modernes.

L'encodage s'applique uniquement aux fichiers ayant une extension spécifique :
`.jpg`, `.jpeg`, `.png`, `.gif`, `.mp3`, `.wav`, `.mp4`, `.webm`

Fonctionnement :

Le serveur, avant d'envoyer un fichier multimédia :

1. Vérifie si l'extension du fichier correspond à un type compressible.
2. Compresse le fichier avec GZIP.
3. Envoie les en-têtes HTTP adaptés (`Content-Encoding`, `Content-Type`).
4. Transmet les données compressées au client.

Si le fichier n'est pas multimédia ou ne peut être compressé, il est envoyé tel quel.

Implémentation :

1. Détection des extensions multimédia :

On utilise une collection Java pour stocker les extensions concernées :

```
Set<String> compressibleExtensions = Set.of("jpg", "jpeg", "png",
"gif", "bmp", "mp3", "wav", "mp4", "webm", "ogg");
```

L'extension du fichier est extraite puis comparée à cette liste pour décider de compresser ou non :

```
String extension = fileName.contains(".") ?
fileName.substring(fileName.lastIndexOf('.') + 1) : "";
boolean compress =
compressibleExtensions.contains(extension.toLowerCase());
```

2. Compression avec GZIP

La compression est effectuée via la classe `GZIPOutputStream` :

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
GZIPOutputStream gzipOut = new GZIPOutputStream(baos);
FileInputStream fis = new FileInputStream(file);
byte[] buffer = new byte[1024];
int bytesRead;
while ((bytesRead = fis.read(buffer)) != -1) {
    gzipOut.write(buffer, 0, bytesRead);
}
fis.close();
gzipOut.close();

byte[] compressedData = baos.toByteArray();
```

Le tableau `compressedData` contient le fichier compressé à envoyer.

3. En-têtes HTTP :

Avant l'envoi, le serveur prépare les en-têtes :

```
output.write("HTTP/1.1 200 OK\r\n".getBytes());
output.write(("Content-Type: " + guessMimeType(fileName) +
"\r\n").getBytes());
output.write("Content-Encoding: gzip\r\n".getBytes());
output.write(("Content-Length: " + compressedData.length +
"\r\n").getBytes());
output.write("\r\n".getBytes());
```

La méthode `guessMimeType(String fileName)` permet d'attribuer le bon type MIME :

```
private static String guessMimeType(String fileName) {
    if (fileName.endsWith(".jpg") || fileName.endsWith(".jpeg"))
return "image/jpeg";
    if (fileName.endsWith(".png")) return "image/png";
    if (fileName.endsWith(".mp3")) return "audio/mpeg";
    if (fileName.endsWith(".mp4")) return "video/mp4";
    // etc.
    return "application/octet-stream";
}
```

Résultat :

Lorsqu'un fichier multimédia est demandé par un navigateur :

- Il est compressé avec GZIP
- Le navigateur reçoit les en-têtes HTTP corrects
- L'affichage/l'écoute reste inchangé côté client
- Les temps de transfert sont optimisés (fichiers plus légers)

Méthodes et classes essentielles utilisées :

Fonctionnalité	Classe / Méthode utilisée	Documentation
Détection du type MIME	`guessMimeType(String)` + `String.endsWith()`	Oracle JavaDoc

Compression GZIP	`GZIPOutputStream`, `ByteArrayOutputStream`	`java.util.zip`
Lecture de fichier	`FileInputStream`, `read()`	`java.io`
Écriture HTTP	`OutputStream.write()`	`java.io`

Fonctionnalité 8 : Gestion des formulaires HTML

Objectif :

Cette fonctionnalité permet au serveur Web de recevoir, traiter et répondre aux données envoyées par un formulaire HTML via la méthode HTTP POST. Cela ouvre la possibilité d'interactions entre l'utilisateur et le serveur (par exemple : inscription, connexion, envoi de messages). Cela permet également de stocker les données dans un .txt (se situant dans le même dossier que les journaux).

Fonctionnement :

Lorsqu'un client soumet un formulaire via POST, le serveur doit :

- Lire le corps de la requête HTTP contenant les données du formulaire
- Extraire et décoder les paires clé/valeur des champs du formulaire.
- Traiter ces données (stockage et validation).
- Construire une réponse HTTP appropriée : un message de confirmation ou une redirection.

Implémentation :

1. **Détection d'une requête POST**
   Le serveur détecte que la méthode HTTP est `POST` en lisant la première ligne de la requête :

```
if (requestLine.startsWith("POST")) {
    // traitement du formulaire
}
```

2. **Lecture des données du formulaire**

- Le serveur récupère la taille du corps à partir de l'en-tête `Content-Length`.

- Il lit ensuite exactement ce nombre d'octets depuis le flux d'entrée (`InputStream`) qui correspondent aux données encodées du formulaire.

3. **Décodage des données**

- Les données sont sous forme de chaîne `clé=valeur&clé2=valeur2...` encodée en URL (ex : espaces encodés `%20`, caractères spéciaux encodés).

- Le serveur utilise la méthode `URLDecoder.decode(String, "UTF-8")` pour décoder chaque paire clé/valeur.

4. **Traitement des données**

- Après décodage, les données peuvent être stockées, affichées, ou utilisées pour une logique métier (ex : ajouter un utilisateur).

- On sauvegarde le nom et l'email transmis par le formulaire.

5. **Construction de la réponse HTTP**

- Le serveur envoie une réponse 200 OK avec un contenu HTML indiquant par exemple que la soumission a réussi.

version du code simplifié avec les lignes essentielles :

```
// Lecture de Content-Length
int contentLength = Integer.parseInt(headers.get("Content-Length"));

// Lecture des données POST
byte[] postData = new byte[contentLength];
inputStream.read(postData);

// Décodage
String formData = new String(postData, StandardCharsets.UTF_8);
Map<String, String> params = parseFormData(formData);

// Exemple d'analyse des paramètres
String name = params.get("name");
String email = params.get("email");

// Traitement (sauvegarde)
```

```
// Envoi réponse
output.write("HTTP/1.1 200 OK\r\n".getBytes());
output.write("Content-Type: text/html\r\n\r\n".getBytes());
output.write(("<html><body>Merci " + name + ", votre email " + email
+ " a été enregistré.</body></html>").getBytes());
```

Résultat :

- Le serveur reçoit et interprète les données des formulaires soumis.
- Les données sont correctement décodées et exploitables.
- Une réponse personnalisée est envoyée au client, confirmant la réception.
- Cette fonctionnalité permet d'interagir dynamiquement avec le serveur via des formulaires HTML.

Méthodes et classes essentielles utilisées :

Fonctionnalité	Classe / Méthode utilisée	Documentation
Lecture de la requête POST	`InputStream.read()`, `BufferedReader`	`java.io`
Décodage URL-encoded	`URLDecoder.decode(String, "UTF-8")`	`java.net`
Analyse des paramètres	Traitement des chaînes (split, etc.)	`java.lang.String`
Écriture HTTP	`OutputStream.write()`	`java.io`