Ce rapport contient des explications sur les choix techniques pris lors de la réalisation du TEA, pour une explication sur comment utiliser le programme, se référer au fichier **README.md**.

1. Choix pour la réalisation du projet :

Dans ce TEA (pour temps en ***accompagnement***) le but était de compléter le code fourni lors d'un TP et le compléter afin que le code implémente le multithreading, du pseudo-CGI, ainsi que des échanges chiffrés en utilisant le protocole HTTPS/SSL.

La première décision pour réaliser ce projet a été de ne pas utiliser le code fourni, et de partir d'une feuille blanche, le code fourni ayant été conçu pour une utilisation trop spécifique, exemple :

```
private static String contentType(String fileName) {
    if (fileName.endsWith(".htm") || fileName.endsWith(".html"))
    {
        return "text/html";
    }
    return "";
}
```

Le code suivant ne peut gérer que le type html, l'objectif de notre code est d'aussi pouvoir charger des fichiers ayant un type plus exotique (.css par exemple), si l'on devait utiliser le code précédent, il faudrait créer une chaine de 'if' à la suite pour pouvoir gérer tous les types de fichiers…

La méthode a donc été remplacée par une méthode déjà existante :

```
Files.probeContentType(file.toPath());
```

Elle permet de déterminer le MIME type du fichier à ouvrir de manière bien plus efficace.

Aussi j'ai pris la décision de commenter et de réaliser la documentation de mon code en anglais pour qu'il puisse être lu et comprit par tous.

2. Structure du code :

Dans les explications suivantes, nous nous concentrerons uniquement sur les classes Main.java et ClientHandler.java, le reste du code commenté en anglais est présent dans les sources, une Javadoc complète est aussi présente, elle peut être affichée en mettant en route le serveur et en cliquant sur le lien Read Javadoc.

J'ai décider de créer un serveur, qui aille chercher pour un dossier « www/ » qui contient les fichiers .html et qui cherche pour un fichier index.html afin d'afficher une page de garde, à la manière d'un serveur WAMP.

Par conséquent, au lancement, le 'main' cherche pour un dossier « www/ », si ce dossier n'existe pas, alors le reste du code ne se lancera pas.

Ensuite, afin de pouvoir réaliser des échanges chiffrés entre le serveur et le client, le serveur charge un keystore contenant le certificat à présenter au client (le navigateur web).

Une fois le certificat chargé la logique de gestion d'un client peut être lancée.

Notre code devait donc implémenter Runnable pour pouvoir être Threadé, à la connexion d'un client nous créons donc un Thread() qui portera notre classe ClientHandler qui est Runnable.

```
void start() {
    System.out.println("Accepting client...");
    System.out.println("Creating Thread...");
    new Thread(this, ThreadName).start();
}
```

Le gros de la logique du serveur se situe dans la méthode ‘’.run()’’, celle-ci se charge de lancer la transaction sécurisée entre le client et le serveur, puis analyse la requête envoyée par le client et détermine si il s’agit d’une requête GET ou POST, si il ne s’agit ni de l’une ni de l’autre, alors on retournera une erreur.

```
@Override
public void run() {
    try {
        // Once the thread is created, we start the handshake.
        this.client.startHandshake();

        // Create the Input and Output.
        in = new InputHandler(this.client.getInputStream());
        out = new OutputHandler(this.client.getOutputStream());

        […]
        // If the request's type or protocol isn't equal to the permitted
options, we throw a 400 error.
        if (!type.equalsIgnoreCase("HTTP/1.1")) { INVALIDRequest();}
        else if (request.equalsIgnoreCase("GET")) {GETRequest(path);}
        else if (request.equalsIgnoreCase("POST")) {POSTRequest(path);}
        else { INVALIDRequest();}
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Dans son état actuel, le programme ne gère que les requêtes POST et GET, le pseudo-CGI est une requête GET sur une route spécifique.

Pour implémenter d’autres types de requêtes, il suffira de rajouter un test supplémentaire avec le type de requête à tester puis implémenter la logique du nouveau type dans une méthode.

Si la requête ne résulte pas en une erreur 400 (INVALIDRequest) alors les méthodes GETRequest et POSTRequest sont appelées.

La méthode GETRequest contient la logique de chargement du fichier ciblé par la route appelée par le client, ainsi qu’un test qui vérifie si le client cherche à accéder à une route spécifique, si c’est le cas alors le serveur exécute une ligne de commande et retourne le résultat au client (pseudo-CGI).

Si la route ne correspond pas à notre pseudo-CGI alors la méthode cherche à ouvrir le fichier visé, si l’ouverture est effectuée avec succès alors il est renvoyé au client, sinon une erreur 404 est renvoyée.

La méthode POSTRequest, elle, se charge de lire le contenu de la requête du client pour en extraire le contenu du POST, ensuite elle renvoie le contenu dans un bloc HTML. Il n’y a pas de logique

d’extraction du contenu du POST particulière, cela permet d’avoir un formulaire ayant un nombre de champs variable.

Pour finir la méthode la plus importante est WriteToClient(), celle-ci est en charge de mettre en forme la réponse pour le navigateur du client, du statut de la réponse au contenu du squelette HTML en passant par la longueur du contenu. Une fois que cette méthode a terminé d’écrire sa réponse vers le client, elle termine le lien en fermant le Socket et les Streams.