Réalisé en monôme par Killian PEREZ.

Semestre 5 – Licence Informatique

Projet TOO/CAI :

Application WEB « WHOIS »

Table des matières

Présentation du projet 2
Description du projet demandé 2
Fonctionnement du projet proposé 2
Explications et choix faits 2
Création du projet 2
Serveur 2
Technologie Websocket 2
Pourquoi ne pas utiliser une connexion websocket sécurisée 3
Technologie JNDI 3
Json 3
Interface web 4
Détails sur le code 4
Partie Java 4
Différentes classes 4
Gestion des données 4
Fonctions utilitaires 4
Partie Javascript 5
Evènement window.onload 5
Envoi des ordres de requêtes 5
Réception et affichage des résultats 5
Fonctions utilitaires 6
Interface web 6

Présentation du projet

Description du projet demandé

Le but de ce projet était de créer une application Web permettant de rechercher des informations sur un nom de domaine donné, et cela en utilisant l'interface « Java Naming and Directory Interface » (JNDI).

Les échanges d'informations doivent se faire via une connexion websocket et les informations doivent être au format JSON.

Fonctionnement du projet proposé

Une fois le projet compilé et exécuté, un serveur local démarre et une page web est chargé.

Sur cette interface web , l'utilisateur peut rentrer le nom de domaine ainsi que les différents enregistrements DNS à interroger.

Une fois la recherche faite les résultats s'affichent en bas de la page.

Explications et choix faits

Création du projet

Ce projet a été créé en utilisant l'outil Maven (site officiel), il s'agit d'un outil facilitant la création de projet Java. Je l'ai utilisé pour faciliter la gestion des dépendances.

Ce projet utilise Java 8 pour plusieurs raisons :

- Java 8 est très bien documenté et est encore supporté par ses créateurs (version LTS).
- JNDI fonctionne mieux avec Java 8.
- Nous n'avons pas besoin des ajouts des versions supplémentaires dans ce projet.

Serveur

Une application Web a forcément besoin d'un serveur pour effectuer les échanges d'informations, ici étant donné que c'est un petit projet, un serveur local est suffisant.
J'ai donc opté pour Tyrus (documentation officiel) qui est proposé ici et qui est simple d'utilisation.

Technologie Websocket

Afin de communiquer les données, nous utilisons la technologie websocket. Il s'agit d'un protocole réseau basé sur le TCP, et permet une communication bidirectionnelle entre deux points finaux de communication (« Endpoint »).

Du côté Javascript, j'ai utilisé l'API WebSocket.

Du côté Java, j'ai utilisé le package Websocket de l'API javax.

Pourquoi ne pas utiliser une connexion websocket sécurisée

L'API Websocket permet d'établir une connexion sécurisée via le protocole « WSS ».

Pour rester simple et ne pas faire un exposé sur les connexions sécurisées :

- Afin d'obtenir une connexion sécurisée il faut obtenir un certificat de sécurité SSL.
- Pour obtenir un certificat de sécurité il faut en faire la demande à un service tiers.
- Aucun service tiers n'est autorisé à fournir un certificat pour localhost qui représente donc un serveur local.
- Il est possible de créer son propre certificat mais il faut alors l'installer sur la machine.
- Si on ne l'installe pas sur sa machine les navigateurs web récents sont fait pour rejeter la connexion.

De plus, outre ce problème de certificat, il est beaucoup plus long et complexe de configurer une connexion sécurisée. Cela demande de nombreuses heures de recherche.

Conclusion : il faut soit payer un nom de domaine externe et obtenir un certificat (gratuit sur une courte période) ou installer le certificat sur chaque machine où l'on veut faire tourner le projet. Par conséquent on laisse tomber cette option.

Technologie JNDI

L'API JNDI fournit différents services de nommages ou d'annuaires et définie une interface pour permettre l'accès à ces services. Ici nous utilisons le service DNS de l'API JNDI.

Pour utiliser cette API il faut définir un contexte racine (c'est un objet qui assure le dialogue avec le service utilisé).

Il y a plusieurs choix :

1. javax.naming.directory.DirContext.java
2. sun.jndi.dns.DnsContext.java

Ces options reviennent au même car ils sont créés à partir du même DnsContextFactory, j'ai donc opté pour l'objet DirContext de javax car il est plus simple à implémenter.

Json

Les données sont échangées au format Json.

Côté Javascript : j'ai utilisé l'API JSON pour traiter ce format.

Côté Java : j'ai utilisé l'API GSON créer par Google.

Interface web

Afin de créer une interface web plus rapidement et simplement, je me suis servi du framework Bootstrap 4 pour la partie HTML/CSS.
J'ai également utilisé l'API jQuery pour faciliter le code javascript.

Détails sur le code

Partie Java

Différentes classes

J'ai choisi de divisé le côté Java du projet en 3 classes donc 3 fichiers:

1. Main.java : contient la classe Main qui est le départ d'exécution du projet et la création du serveur Tyrus ainsi que les fonctions liées à la gestion du formatage JSON.
2. JNDI_DNS.java : contient tout ce qui sert à une recherche DNS.
3. Websockets.java : contient le serveur endpoint côté Java.

Gestion des données

Lorsque l'interface web envoie son ordre de requête via la connexion websocket, cette donnée arrive dans l'évènement @javax.websocket.OnMessage.

Cet ordre reçu est un tableau contenant un nom de domaine valide ainsi qu'un tableau d'enregistrements DNS au format JSON, on le stock donc dans une variable de type ArrayList, après l'avoir transformé en objet Java.

Une fois l'ordre parse, on créer une instance de la classe JNDI_DNS. Cet objet va donc remplir un tableau contenant les résultats de la requête DNS sous forme de liste de tableau de string.

Finalement on utilise une fonction accesseur pour récupérer ces résultats et les envoyer au format JSON dans la connexion websocket.

Fonctions utilitaires

Le code étant assez simple il y a donc peu de fonctions utilitaires :

1. Main.parseJson(string Json) : permet de transformer un string au format JSON en objet Java ArrayList.
2. Main.createJson(Object o) : permet de transformer un objet Java en string au format JSON.
3. JNDI_DNS.getRes() : fonction accesseur pour récupérer l'attribut resultats de la classe JNDI_DNS.

Partie Javascript

Evènement window.onload

Cet événement javascript se déclenche lorsque la page a fini de charger dans le navigateur, puis exécute le code qui s'y trouve, voilà ce que j'ai placé dedans :

- le endpoint websocket
- un code qui coche tous les enregistrements DNS lorsque la checkbox « Tout sélectionner » est cochée.
- un code qui permet d'enclencher le bouton « Rechercher » lorsqu'on appuie sur entrée depuis l'input text.
- un code qui affiche la validé du nom de domaine saisi en temps réel (vérifié à chaque event keyup depuis le champ de saisi)

Ces morceaux de code **doivent** se situer dans cet évènement onload car ce sont des écouteurs d'évènements, ils sont donc obligés d'être chargé avant toute action de l'utilisateur sinon ils ne fonctionnent pas.

Envoi des ordres de requêtes

Une fois que l'utilisateur a rempli un nom de domaine et les enregistrements DNS qu'il souhaite rechercher via l'interface, il clique sur le bouton rechercher qui va exécuter la fonction send() de mon code js.

La console est nettoyée pour effacer les traces d'éventuelles précédentes recherches. Le nom de domaine est donc récupéré et vérifié puis si tout est en ordre, une variable de type tableau est créée puis remplie de la sorte :

- la première cellule de ce tableau est remplie par le nom de domaine valide sous forme string
- la seconde cellule est un tableau de string contenant les noms des enregistrements DNS à interroger.

J'ai choisi cette façon de stocker les ordres de recherche car elle est flexible; si jamais je voulais ajouter des options à mon interface web, je n'aurais qu'à remplir une autre cellule de ce tableau.

Enfin, ce tableau est transformé au format JSON puis envoyé sur la connexion websocket.

Réception et affichage des résultats

Une fois que la partie Java a envoyé ses résultats via le websocket, l'évènement javascript Websocket.onmessage récupère ce message, le parse afin de transformer le JSON en objet javascript.

Puis la fonction print() est appelée afin d'afficher les résultats dans le HTML :

Une balise <section> jusqu'alors cachée est affichée, vidée(pour effacer les précédentes recherches) puis remplie de manière structurée. L'esthétique se fait avec le framework Bootstrap 4.

Fonctions utilitaires

Puisque ce code javascript se situe dans un seul fichier, j'ai créé plusieurs fonctions utilitaires afin de faciliter la lecture du code ainsi que sa compréhension :

- createRecords() : créer un tableau contenant les enregistrements cochés par l'utilisateur
- clearHTML(element) : vide le texte d'un élément HTML donné.
- clearConsole() : vide la console et réaffiche la connexion websocket.
- correctURL(url) : renvoi le nom de domaine entré en enlevant « http:// », « https:// » ou « www. »
- isValidDomain(domain) : booléen qui vérifié la validité d'un nom de domaine à l'aide d'une expression régulière.
- showHTML(elementID) : permet d'afficher un element HTML grâce à son ID.

Interface web

J'ai fait le choix d'utiliser Bootstrap 4 afin de réaliser quelque chose de joli et propre en peu de temps car ce n'est pas le plus important dans ce projet.

J'ai voulu une interface très simple et claire, il aurait été possible de rajouter plus d'options pour la recherche JNDI (comme le choix du serveur DNS pour effectuer les recherches par exemple) mais je n'ai pas jugé ça pertinent.

La seule option pour l'utilisateur est donc le choix des enregistrements DNS à interrogé car c'est le cœur même de cette application web.

J'ai également ajouté une petite barre en haut de l'écran avec pour seule intérêt de posséder un bouton qui fait apparaitre un bouton qui explique le projet avec notamment un lien pour en apprendre davantage sur les enregistrements DNS.