PFE 2018/2019 - P06

Rapport intermédiaire

Duquenoy Antoine – Feutrier Simon

19 Décembre 2018

Table des matières

1 Contexte

Notre projet s'inscrit dans le cadre d'un projet de recherche à l'IRCICA sur un réseau de capteurs à large échelle (100 noeuds). Face à un réseau de cette ampleur, il est intéressant de développer une solution de maintenance à distance de ce réseau. Il sera bien plus aisé pour les chercheurs de tester leurs hypothèses en déployant rapidement leurs logicielles sur tout ou une partie des noeuds disponibles grâce à des fonctionnalités telles que la sélection d'une partie du réseau ou encore la mise à jour d'un ensemble de capteurs de même type / architecture.

2 Cahier des charges

L'objectif de notre travail consiste à développer simultanément le *front-office* et le *back-office*. La partie front sera réalisée en utilisant les technologies web classiques. La partie *back-end* sera constituée d'un ensemble de *webservices* en différents langages permettant la communication réseau et la gestion d'une base de données de noeuds et capteurs. Il faudra y développer des services classiques comme la gestion des utilisateurs et des services spécifiques (compilation du code source, téléchargement à distance du logiciel sur les noeuds). La partie *front-end* aura vocation à être utilisée par des utilisateurs externes au projet et devra donc être ergonomique et facile de compréhension. Elle devra également être fonctionnel et modifiable facilement.

Notre cahier des charges n'a pas été précisé dès le début mais a plutôt évolué au fur et à mesure de notre projet et donc de notre progression.

3 Travail effectué

3.1 Mise en place

Pour commencer, nous avons tout d'abord choisi les différentes technologies que nous allions utiliser. Premièrement, notre tuteur nous a demandé de réaliser le coeur du projet en Java EE (abrégé en JEE dans la suite du rapport). Nous en avons profité pour utiliser le *framework* **Spring** qui permet de s'affranchir de la gestion des *servlets* ou encore de la gestion direct des bases de données (couche DAO, requêtes SQL brutes) en utilisant un *framework* de persistance des objets, **Hibernate** dans notre cas. Nous avons également choisi d'utiliser **Spring Boot** qui permet de s'affranchir des problèmes de dépendances et d'automatiser la création des .jar.

Nous utiliserons donc également Hibernate pour récupérer et stocker les données stockées dans une base de données PostgreSQL.

Pour gérer la sécurité de notre site internet et les types d'utilisateurs, nous utiliserons **Spring Security**, un autre framework.

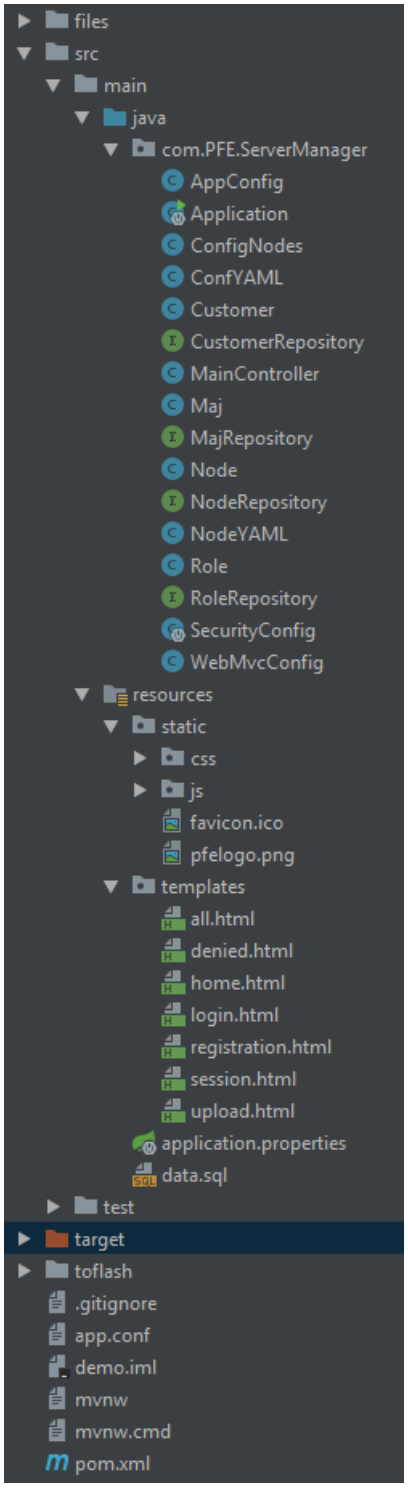
Puis pour gérer la communication entre le front et le *back*, nous avons choisi le moteur de *template* **Thymeleaf** possédant une syntaxe plus souple que les jsp traditionnelles. Celui-ci va permettre d'afficher à l'utilisateur des informations venant de la partie back-office, comme des valeurs persistées en base de données par exemple.

Ensuite, pour la partie front du site, c'est **Bootstrap** que nous utiliserons comme en association avec HTML 5, CSS 3 et JavaScript. Bootstrap est un *framework* qui permet de faciliter la mise en page de notre site et d'uniformiser le rendu. Son principale intérêt réside dans son système de grille particulièrement modulaire et *responsive*, c'est à dire s'adaptant à tous les types d'appareils. Il fournit également un ensemble de composants restylés pour faire plaisir aux yeux.

Enfin, pour la programmation nous avons décidé d'utiliser l'IDE **IntelliJ** que nous avions déjà utilisé en cours et qui va nous permettre de ne pas avoir à nous soucier du conteneur web **Apache Tomcat** qui va accueillir notre site. Il offre également tout un ensemble d'outils tels que l'autocomplétion ou la structuration du projet.

Pour finir, nous tenons à jour un projet GIT sur le site de Polytech Lille : *archives.plil.fr* pour mener à bien notre projet en binôme.

3.2 Structure du projet



Racine du projet

- **Dossier files :** C'est dans ce dossier que sont enregistrés les fichiers téléversés sur le serveur.
- **Dossier toflash :** Ici se trouve les fichiers YAML correspondant à des mises à jour programmées.
- **app.conf :** Fichier de configuration de l'application, il contient pour le moment l'email et le mot de passe de l'utilisateur par défaut. Son contenu est sujet à évolution.
- **pom.xml :** C'est le fichier utilisé par Maven dans lequel se trouve principalement les dépendances du projet.

Sources

- **AppConfig :** Classe contenant une méthode d'initialisation de l'application. Pour le moment, seul le fichier *app.conf* est traité.
- **Application :** Il s'agit de la classe contenant le *main* démarrant l'application Spring Boot.
- **ConfigNodes :** Classe utilisée pour représenter sous forme d'objet le fichier YAML de description des noeuds.
- **ConfYAML :** Idem mais pour le fichier *app.conf*.
- **Customer :** Classe entité qui caractérise un utilisateur du site.
- **CustomerRepository :** Le *repository* de la classe correspondante.

— **MainController :** C'est le fichier le plus important. Il vient faire correspondre toutes les requêtes de l'utilisateur à des méthodes.
— **Maj :** Classe entité qui caractérise une mise à jour.
— **MajRepository :** Le *repository* de la classe correspondante.
— **Node** Classe entité qui caractérise un noeud.
— **NodeRepository :** Le *repository* de la classe correspondante.
— **NodeYAML** Classe utilisée pour décrire l'objet noeud du fichier de configuration YAML du réseau. Cet objet est utilisé par *ConfigNodes* pour représenter la liste des noeuds du fichier.
— **Role :** Classe entité qui caractérise un rôle utilisateur.
— **RoleRepository :** Le *repository* de la classe correspondante.
— **SecurityConfig :** Configuration utilisée par le *framework* Spring Security.
— **WebMvcConfig :** Un autre fichier de configuration pour Spring Security.

Dossier resources

— On retrouve dans le dossier *static* tous les fichiers qui ne sont pas concernés par du traitement opéré par un *framework* comme Thymeleaf par exemple.
— Le dossier *templates* est celui utilisé par Thymeleaf pour générer les fichiers HTML à envoyer au client après le traitement de la *template* concernée.
— **application.properties :** Ce fichier contient un certain nombre de paires clé/valeur utilisées par exemple pour renseigner les informations de connexion à la base données.
— **data.sql :** Nous n'utilisons plus ce fichier mais il permet d'exécuter des instructions PSQL au démarrage de l'application.

3.3 Base de données

Comme précisé par notre tuteur Thomas Vantroys, nous utiliserons **PostgreSQL** pour ce projet même si en réalité, toutes les interactions avec celle-ci seront dissimulées par l'environnement Spring.

En effet, il est possible de personnaliser la création d'une base de données (BDD) grâce à Hibernate, une spécification de l'interface JPA. Ce *framework* va également nous permettre d'utiliser des classes comme table de données et nous proposer un ensemble de méthodes pour nous permettre de nous affranchir des requêtes SQL de sélection de base. Il reste évidemment possible d'écrire des requêtes plus complexes si cela est nécessaire.

Nous avons donc deux classes : *Customer.java* et *Role.java* avec une jointure (multiple, si l'on veut donner plusieurs rôles à un seul utilisateur).

Remarque : nous utilisons une classe *Customer.java* et non pas *User.java* car le mot-clef user est réservé en PostgreSQL.

Pour faire la passerelle entre classe et table, on utilise des classes *Repository* qui héritent d'une classe de JPA indiquant que cette classe doit être utilisée comme table. Grâce à cet héritage, nous pourrons également utiliser de nombreuses méthodes qui représentent les commandes SQL comme par exemple *findByEmail()* qui permet de trouver un utilisateur grâce à son e-mail.Ces méthodes doivent être déclarées au sein de cette classe mais ne doivent pas être implémentées tant qu'il s'agit de requêtes d'extraction simples qui seraient l'équivalent de *SELECT * FROM table WHERE id = 1* par exemple.

Ainsi que détaillé dans la partie *2. Structure du projet*, la classe *MainController.java* va principalement se charger de rediriger l'utilisateur à travers le site. Cette classe va de plus permettre de faire le lien entre les différents fichiers du projet. Dans cette classe, on crée une instance de *RoleRepository.java* et de *UserRepository.java*, ce qui permet concrètement de créer deux services pour interagir avec les tables *role* et *customer* créées par Hibernate.

Ensuite, pour ce qui concerne la gestion des utilisateurs, on utilise *findByEmail()* couplé à une méthode de Spring Security qui détermine l'email utilisé pour la session afin de savoir quel est l'utilisateur actuel et donc quel est son rôle. Avec *customerRepository.save()*, on sauvegarde de nouveaux utilisateurs dans la base de données. On peut également bien sûr créer de nouveaux utilisateurs ou des rôles directement depuis le *backend* et leur appliquer les *set()* et *get()* habituelles pour ajouter des champs manuellement.

Par la suite nous avons implémenté la possibilité de sauvegarder des mises à jour créées par les utilisateurs. Nous avons donc ajouter une classe *Maj.java* et son dépôt associé *MajRepository.java*. La table *customer* possède une jointure supplémentaire pour associer chaque utilisateur à ses mises à jour personnelles. L'utilisateur se servira du site internet et plus précisément la page */session* pour créer une nouvelle mise à jour. Une fois le formulaire rempli et envoyé, il est récupéré et traité dans *MainController.java*. On crée une nouvelle instance de *Maj.java* et on la peuple grâce aux données récupérées puis on fait en sorte qu'elle soit associée à l'utilisateur.

3.4 Connexion et Spring Security

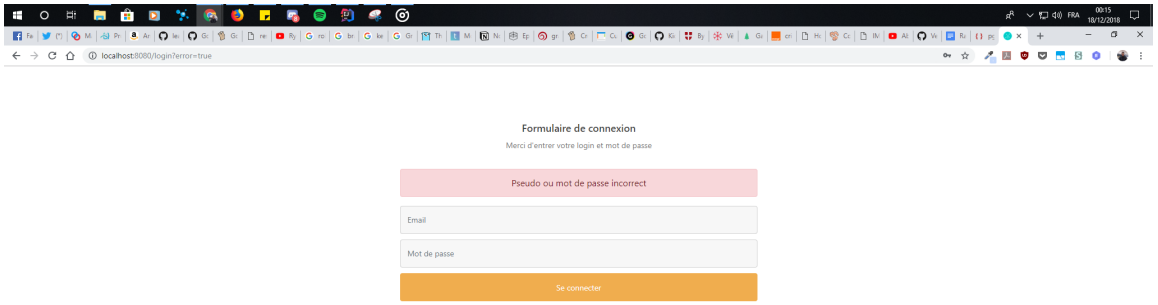
Grâce aux tables des utilisateurs et des rôles créés, nous allons pouvoir paramétrer Spring Security pour qu'il s'occupe de gérer l'accès aux différentes pages.

Premièrement, nous créons une classe SecurityConfig.java qui sera exclusivement consacrée à Spring Security. Dans cette page, nous utilisons deux méthodes. La première indique à Spring Security comment est faite l'authentification : ici on utilisera l'e-mail de l'utilisateur. On spécifie également quelle table est à utiliser comme rôle. Enfin, on précise également que l'on veut que le mot de passe soit haché en spécifiant l'algorithme, en l'occurrence *BCryptPasswordEncoder*.

La deuxième méthode va permettre d'indiquer quel rôle à accès à quelle page. En effet, les utilisateurs avec le rôle USER auront un accès restreint puisqu'ils ne peuvent pas ajouter d'autres utilisateurs ni regarder la liste complète des utilisateurs. On précise également quelle page va servir de page de connexion et ou mènera une déconnexion ou une erreur.

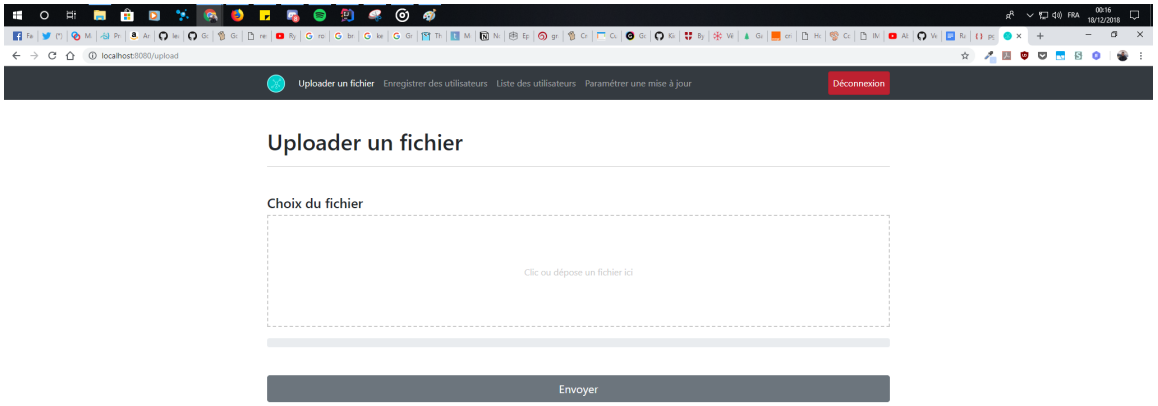
Le *framework* est en réalité très haut niveau puisqu'il n'y a rien d'autre à paramétrer pour faire fonctionner la sécurité d'accès sur nos pages. Par exemple, une fois choisi, la page de connexion est automatiquement utilisée par Spring Security par la suite et il n'y a pas nécessité de récupérer les valeurs renvoyés via le formulaire car le *framework* s'en charge seul.

Au niveau de la partie *front*, elle se compose d'un formulaire doté de deux champs, un de type *email* et l'autre de type *password* avec un bouton pour valider. On retrouve également une alerte qui s'affiche si la combinaison entrée n'est pas valide.



3.5 Téléversement d'un fichier

Le gros du travail pour cette fonction se trouve dans la partie **JavaScript** (JS). Ce script intègre la possibilité d'envoyer un fichier via une requête POST dans un dossier du serveur. Le dépôt de ce fichier peut se faire par sélection depuis l'explorateur de fichiers ou en effectuant un glisser-déposer. Après avoir pris connaissance des événements écoutables au niveau de l'action de *drag and drop*, nous avons fait en sorte d'afficher le nom du fichier dans le cadre une fois celui-ci déposé. Dans une optique d'interactivité, nous avons fait en sorte que la bordure de la zone s'épaississe quand un fichier survole la zone.



Pour stocker les données qui seront envoyées, nous avons utilisé l'objet *FormData* qui permet de construire un ensemble de paires clé/valeur représentant les données que nous enverrons. Il se trouve qu'au final nous n'avons qu'une paire mais initialement d'autres informations étaient transmises avec le fichier. L'avantage étant que cet objet peut être envoyé via la méthode *send()* de l'objet *XMLHttpRequest*. Ainsi au moment de l'événement "*drop*", nous stockons l'objet "*File*" qui sera converti en binaire à l'envoi. Lorsque l'utilisateur dépose plus d'un fichier dans l'espace prévu, un *pop-up* d'avertissement apparaît pour indiquer qu'un seul fichier ne peut être transféré simultanément.

Nous avons ensuite ajouté la possibilité de cliquer sur cette zone pour choisir le fichier dans l'explorateur. Pour ce faire, nous avons rendu invisible via le CSS un input de type file. En JS, nous observons l'événement "*click*" sur la zone de dépôt puis nous simulons un clic sur le input invisible faisait apparaître la fenêtre de l'explorateur. Un autre événement "*change*" sur l'élément input permet de recueillir le fichier et ses informations.

Il a fallu également s'attaquer aux quelques cas d'usage particuliers comme lorsque que l'utilisateur dépose ou choisi un nouveau fichier. Pour ce cas, nous avons fait le choix de remplacer le fichier existant par le nouveau.

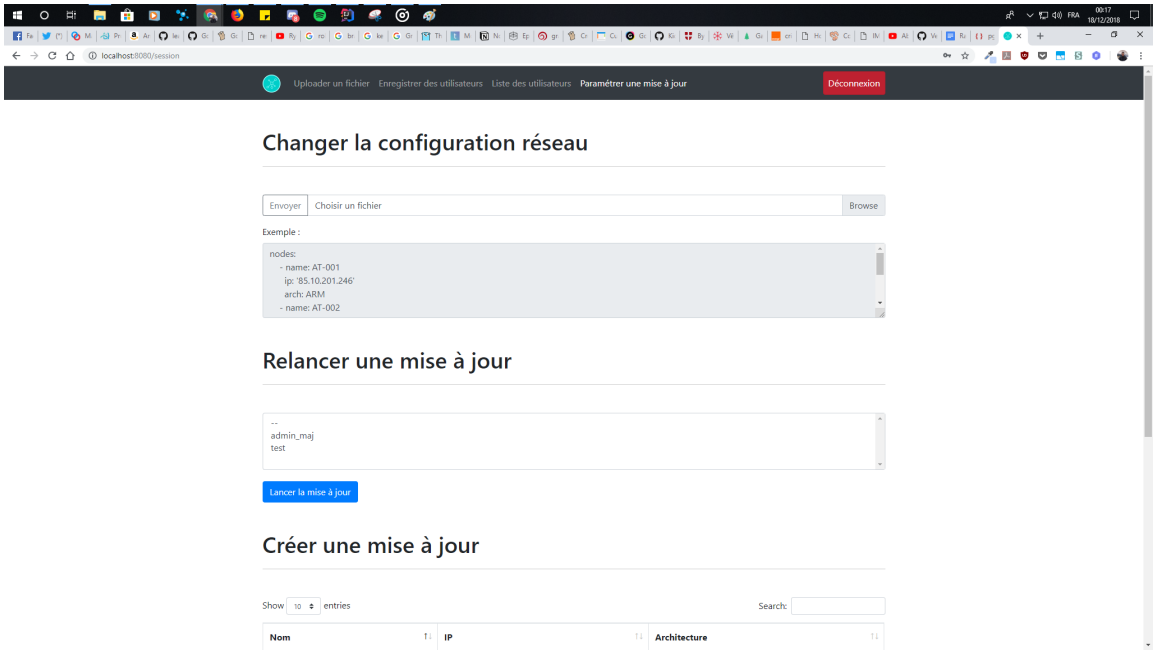
Enfin, pour procéder à l'envoi, il suffit de cliquer sur le bouton en bas de page. Si aucun fichier n'a été fourni, un *pop-up* apparaîtra pour stipuler l'erreur. En revanche si tout se passe bien, la requête POST peut être envoyée. Dernière petite

fonctionnalité, *XMLHttpRequest* nous permet de récupérer le nombre d'octets total à transmettre ainsi que le nombre déjà transmis ce qui est parfait pour intégrer une barre de progression. Une fois le transfert opéré, un message apparaît et la page est rechargée.

Si on s'intéresse maintenant à la partie Java, nous avons eu besoin d'une nouvelle route, */file*, qui accepte seulement des requêtes de type POST. Le fichier transféré est stocké dans un répertoire nommé *files* et qui contient d'autres répertoires sous la forme *pseudo_ timestamp* dans lesquels un fichier binaire se trouve.

3.6 Création d'un mise à jour

Pour satisfaire les besoins des chercheurs, notre tuteur nous a demandé d'ajouter une interface de création de mises à jour en ligne. L'idée est de pouvoir choisir les noeuds (et donc les composants) sur lesquels seront déployés la mise à jour mais également de pouvoir lui donner un nom et lui attribuer une date de lancement.



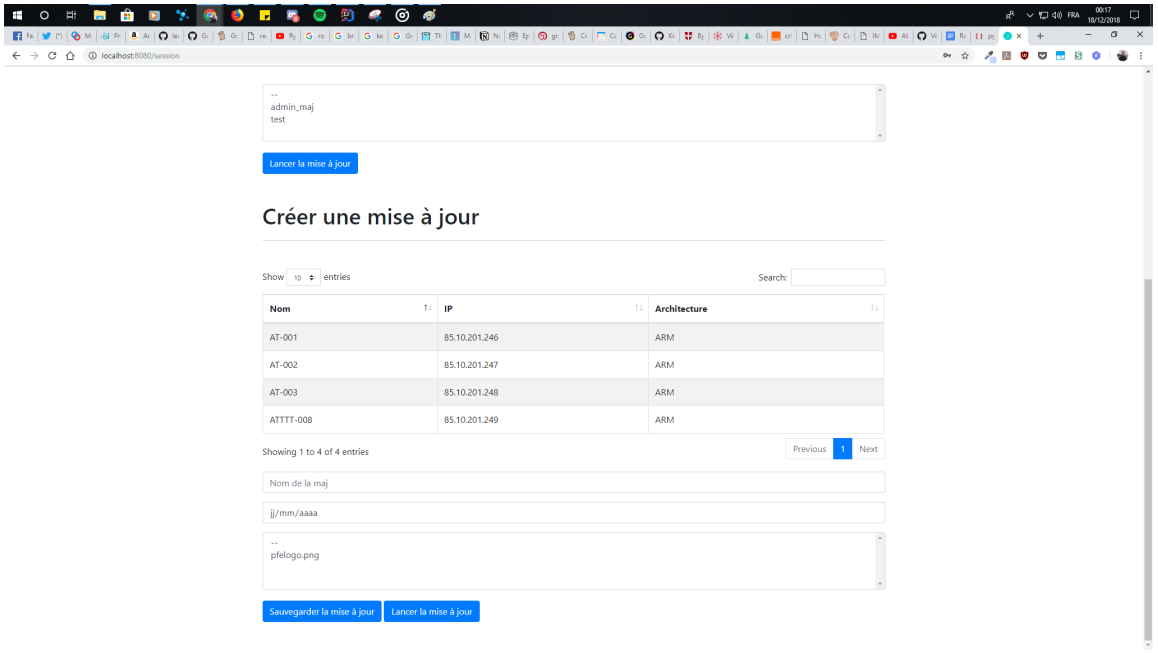
Pour pouvoir mener à bien les mises à jour des capteurs sur le réseau, l'utilisateur doit fournir un fichier depuis une autre page qui sera téléversé sur les capteurs sélectionnés. Pour l'instant le type de fichier n'est pas décidé même si le choix le plus judicieux serait un format binaire pouvant être transféré directement sur les capteurs.

Pour faire les différents choix, nous avons ajouté une page session.html qui permet de remplir un formulaire contenant tous les champs cités plus haut. On utilise notamment un tableau avec des filtres pour pouvoir sélectionner les noeuds. Deux boutons sont disponibles : le premier sert à lancer la mise à jour immédiatement (ou à la date prévu) tandis que le second permet de sauvegarder la mise à jour paramétrée. Celle-ci n'est donc pas exécutée dans l'immédiat mais elles se retrouve dans une liste plus haut sur la page, prêt à être lancée..

Comme précisé dans la partie *3. Base de données*, nous avons créé de nouvelles classes pour qu'un utilisateur puisse enregistrer les mises à jour qu'il paramètre. Pour l'instant, chaque utilisateur n'a accès qu'à ses mises à jour mais cette solution viendra à évoluer dans la suite du projet tel que détaillé dans la partie *4. Travail restant.*

Passons maintenant à la partie JavaScript. Celle-ci peut se découper en deux parties indépendantes. La première partie concerne le lancement d'une mise à jour qui a été enregistrée au préalable. Dans ce cas, une requête POST contenant le nom de la mise à jour est envoyée lors du clique sur le bouton concerné. Si aucun nom n'est sélectionné dans la liste, un avertissement visuel est transmis à l'utilisateur.

Le seconde partie concerne la création d'une mise à jour. Pour ce faire l'utilisateur dispose d'un tableau avec la liste des noeuds. Une bibliothèque JS permet à l'utilisateur d'interagir avec le tableau avec des fonctionnalités telles que le tri, la sélection ou encore la recherche. Il est possible également d'écouter des événements pour savoir lorsqu'une ligne est sélectionnée ou désélectionnée. Nous utilisons un Set pour stocker les noms des noeuds et pour pouvoir les supprimer aisément. Lors de l'envoi, il sera transformé en chaîne de caractères séparée par des points-virgules. Une fois la sélection terminée, l'utilisateur doit entrer le nom de la mise à jour, la date à laquelle il veut qu'elle s'effectue (il faudra d'ailleurs discuter de l'intérêt d'ajouter les heures/minutes) et sélectionner le fichier à transmettre dans li des fichiers déjà téléversés. Deux boutons sont ensuite proposés. L'un permet uniquement d'enregistrer cette configuration pour pouvoir lancer la mise à jour ultérieurement et l'autre lance la mise à jour de manière anonyme, sans l'enregistrer. Comme pour la page */upload*, nous avons doté ce script de sécurité. En effet, l'envoi ne se fera que si tous les champs sont renseignés.



3.7 Fichier de description du réseau de capteurs

Étant donné que notre projet s'inscrit dans un contexte de recherche, le réseau de capteur est amené à évoluer en fonction des expérimentations menées. Il était donc nécessaire de donner la possibilité à l'utilisateur d'indiquer au site la configuration actuelle des noeuds.

Pour permettre cela, on demande à l'utilisateur de fournir un fichier au format YAML. Ce format permet simplement une représentation de données par sérialisation ce qui permet à l'utilisateur de rédiger plus simplement le fichier.

Le fichier est donc ajouté via la page *session.html* et est ensuite traité côté *back-end*. Pour cela on utilise la bibliothèque **SnakeYAML** qui va simplement nous permettre de parser le fichier pour récupérer les valeurs entrées par l'utilisateur.

Il s'agit pour le moment d'une fonctionnalité réservée aux administrateurs du site. Le fichier qu'il faut renseigner possède l'allure suivante :

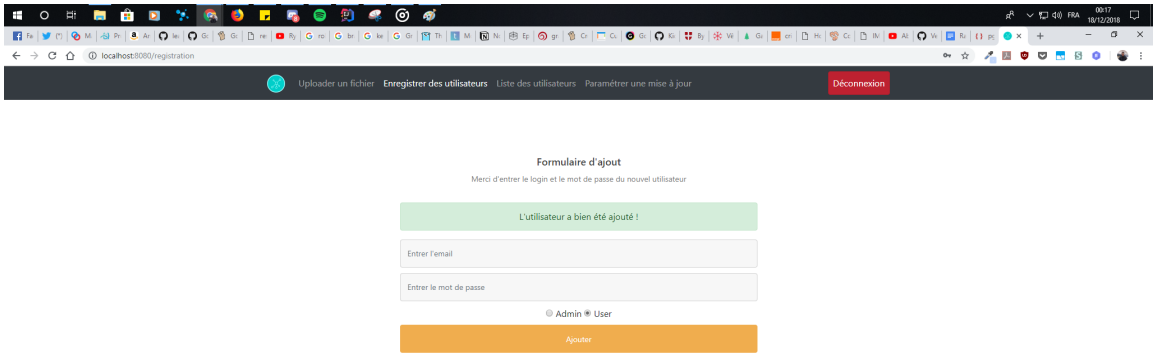
```
nodes:
  - name: AT-001
    ip: '85.10.201.246'
    arch: ARM
  - name: AT-002
    ip: '85.10.201.247'
```

```
    arch: ARM
- name: AT-003
    ip: '85.10.201.248'
    arch: ARM
- name: ATTTT-004
    ip: '85.10.201.249'
    arch: ARM
```

Une fois le bouton "Envoyer" pressé, la modification du réseau est visible directement dans le tableau des noeuds situé plus bas.

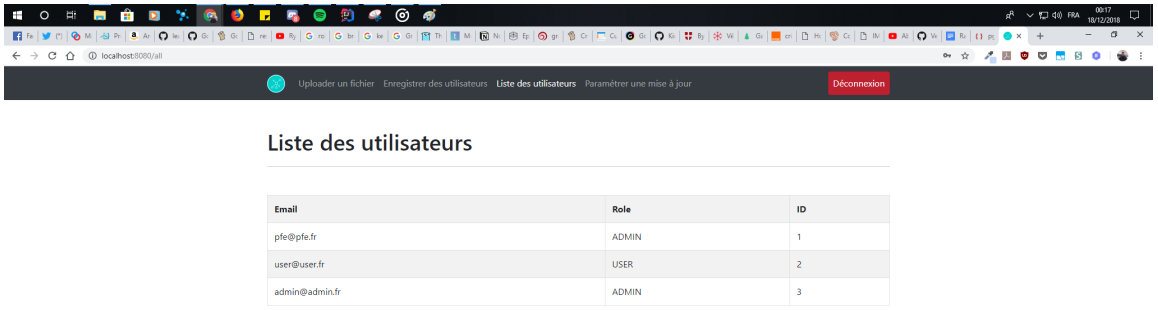
3.8 Diverses fonctionnalités

L'utilisateur avec un rang ADMIN sur le site a la possibilité de créer de nouveaux utilisateurs en renseignant leur email, mot de passe et en choisissant leur rôle. Cette page est très similaire visuellement à la page de connexion. Du point de vue *back-end*, il nous faut dans un premier temps vérifier que l'email renseigné n'est pas déjà utilisé par un autre membre du site. Si tel est le cas, une erreur est renvoyée sur la vue. Dans le cas contraire, nous créons un objet *Customer* avec les informations entrées dans le formulaire avant de l'enregistrer dans la BDD.

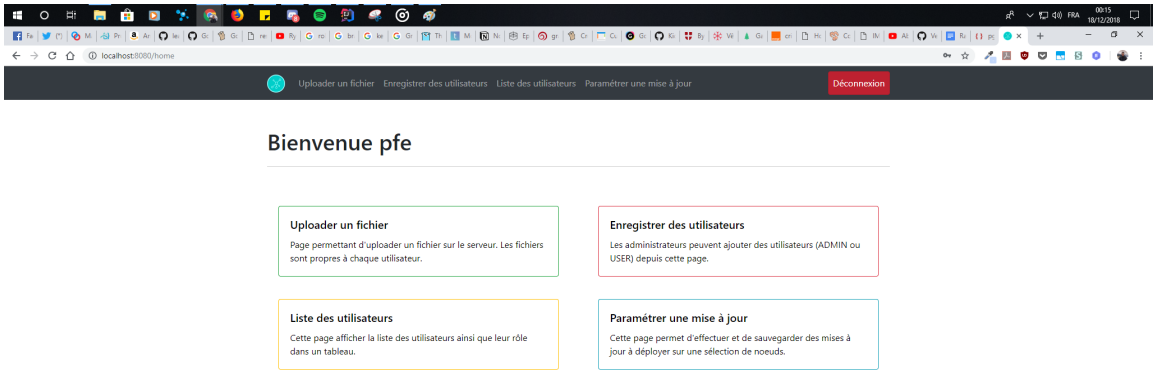


Une autre page consacrée aux administrateurs est celle qui permet de visualiser la

liste complète des utilisateurs avec leur rôle. Il s'agit d'un tableau pour lequel les lignes sont générées au moyen d'une boucle dans la *template* Thymeleaf.



Enfin, nous avons souhaité ajouter une page d'accueil qui propose un rapide descriptif des fonctionnalités offertes en accord avec le rang de l'utilisateur connecté.



3.9 Déploiement du JAR sur la machine virtuelle

Nous avons eu à notre disposition en cours de semestre une machine virtuelle pour héberger notre application. Nous n'avons pas de suite travaillé dessus à sa réception, nous avons préféré y mettre une version du projet plus aboutie directement. Avant toute chose, il nous a fallu générer un JAR de notre application avec les outils proposés par IntelliJ. Nous avons préféré ce format à défaut d'un WAR car il a l'avantage d'intégrer le conteneur web Tomcat et donc de se lancer facilement.

Ensuite, nous avons dû porter le projet sur notre machine virtuelle avec toutes les dépendances qu'il nécessite. La première idée était de créer une image Docker contenant les différentes briques logiciels nécessaires mais cette solution n'a finalement pas été retenue. En effet, il y a seulement deux éléments à installer pour faire fonctionner le JAR de notre projet : Java JRE et PostgreSQL. De plus, s'agissant déjà d'une VM destinée uniquement à notre PFE, nous ne voyons pas forcément la nécessité de venir isoler davantage ces applications. Nous avons donc finalement choisi d'installer directement les paquets sur notre machine.

Pour que notre programme puisse fonctionner, nous avons donc dû installer le paquet *default-jre* correspondant à la version 8 de Java ainsi que le paquet *postgresql-10* pour la base de données. Il s'agit, comme précisé plus haut, des seuls dépendances externes du projet. Enfin, il nous a fallu créer une base de données en concordance avec le nom utilisé dans les sources et avec également le même mot de passe. Nous

envisageons par la suite de créer un script pour automatiser tout ce processus et de sortir hors des sources les informations de connexions relatives à la BDD comme son nom, l'identifiant et le mot de passe.

De plus, nous prévoyons de créer un *daemon* grâce à un script init dans ce bon vieux répertoire */etc/init.d/* de Devuan pour faciliter le démarrage et l'arrêt de notre service.

4 Travail restant

4.1 Création de groupes pour les mises à jour

La première chose que nous comptons ajouter pour ce projet est la gestion de groupes de travail. En effet, dans un laboratoire plusieurs équipes travaillent sur des projets différents. Cependant, dans une même équipe, chaque personne doit pouvoir avoir accès à une mise à jour créée par son collègue.

L'idée est donc de créer une nouvelle classe *Group.java* et donc une nouvelle table de données. Celle-ci sera lié en jointure avec l'utilisateur comme pour le rôle par exemple. Ainsi le premier groupe de travail ajouté aura l'identifiant 1 et cette information sera également présente dans la table de l'utilisateur.

La table de données des mises à jour ne sera par contre plus jointe avec l'utilisateur mais avec son groupe. Il y aura donc une jointure entre *maj* et *group*. Grâce à cette nouvelle implémentation, toutes les personnes appartenant à un même groupe pourront voir les mises à jour créées par leur collègue.

4.2 Ajout d'un didacticiel sur la création du fichier YAML

Étant donné que les personnes utilisant le site ne sont pas forcément dans le domaine informatique ou ne connaissent simplement pas le format YAML, il est nécessaire d'ajouter une explication de la création du fichier de noeuds.

L'idée est simplement d'avoir une documentation succincte pour que n'importe qui puisse actualiser l'état du réseau tel qu'il est.

Pour cela une simple page web, ou la page d'importation du fichier, contiendra un didacticiel court permettant de former ce fichier pour représenter le réseau.

4.3 Communication avec les noeuds et réception de données

Pour cette partie du projet il est encore nécessaire de discuter avec notre tuteur pour définir les caractéristiques de la communication et notamment celle qui se feront entre les Raspberry Pi (noeuds) et les capteurs qui lui sont connectés.

Premièrement nous devons choisir si le serveur sera capable de détecter lui-même les noeuds du réseau et les capteurs qui s'y trouvent. L'idée actuellement est que les utilisateurs indiquent au serveur les noeuds et leur IP. La détection du réseau n'est donc pas dynamique. De plus, nous devons savoir si les Raspberry enverrons des données telles que la liste des capteurs et leurs caractéristiques.

Nous avons l'idée de créer un programme annexe en Java agissant en arrière plan. Son but serait de scruter le dossier *toflash* présenté plus haut et de détecter les mises à jour qui doivent être effectuées. C'est donc ce démon qui aura la tâche finale d'acheminer le binaire sur le noeud en question. Une fois ce travail effectué, il supprimera la tâche dans le dossier *toflash*. On peut même imaginer la création de tâches récurrentes si le besoin est présent ou même de tâches s'effectuant un certain nombre de fois. Dans ce cas, il faudrait ajouter quelques paramètres dans le fichier YAML.

4.4 Documenter le projet

Afin que notre tuteur notamment puisse modifier le projet ultérieurement, il est nécessaire de fournir une documentation plus synthétique sur le code et sur le fonctionnement de notre programme en complément des rapports.

Pour cela il va falloir fournir un document qui contiendra les bibliothèques utilisées ainsi que les différentes dépendances externes de notre projet. Il sera également utile de préciser des éléments essentiels tel que l'accès au site, le redémarrage du site ou encore son arrêt.

Ensuite, nous fournirons en plus des commentaires dans les sources, une documentation expliquant notre programme pour que les personnes en charge du projet puisse modifier le fonctionnement de la base de données, l'agencement de l'interface utilisateur ou tout autre mécanisme permettant de faire évoluer l'application.

Finalement, nous avons utilisé pour ce projet le logiciel de gestion de versions décentralisé GIT qui permet d'avoir l'historique de toutes les étapes "stables" du projet et qui nous a permis de travailler efficacement en binôme.

4.5 Calendrier prévisionnel (GANTT)

Nous sommes relativement confiant quant à l'avancée du projet. Nous espérions en début de semestre terminer la majeur partie de l'interface utilisateur ainsi que la partie *back-end* pour pouvoir nous concentrer pendant les vacances et au semestre prochain sur le déploiement réel des fichiers sur les noeuds. Cet objectif est globalement rempli bien que certaines fonctionnalités sont toujours sujettes au changement. Pour cette seconde partie du projet, nous avons élaboré un diagramme de GANTT afin de gérer plus efficacement notre temps et pour ne pas se laisser déborder par une fin d'année qui arrive très vite.

	S1	S2	S3	S4	S5	S6
Tâches	07/01 - 13/01	14/01 - 20/01	21/01 - 27/01	28/01 - 03/02	04/02 - 10/02	11/02 - 17/02
Création de groupes d'utilisateurs						
Création d'un programme externe pour téléverser						
Ajout des nouvelles fonctionnalités utilisateurs						
Réunions sur la communication avec les noeuds						
Réunion sur de nouvelles fonctionnalités						
Ajout d'un didacticiel YAML						
Documentation du projet						
Rédaction du rapport final						