h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Smarthepia

Thèse de Bachelor présentée par

Monsieur Etienne GUIGNARD

Pour l'obtention du titre de

Bachelor of Science HES-SO en Ingénierie des technologies de l'information avec orientation en Communication, multimédia et réseaux

Septembre 2018

Professeur responsable :
M. Eric JENNY

Table des matières

1. Avant-propos

1.1 Présentation

Ce mémoire a été réalisé en vue de l'obtention d'un Bachelor en Ingénierie des Technologies de l'Information, avec spécialisation en communication, multimédia et réseau. Il a été effectué au sein de l'hepia, durant une période de onze semaines, et à plein temps.

1.2 Conventions typographiques

Afin d'en simplifier la lecture, on a rédigé ce travail en suivant les règles typographiques suivantes :

- L'*italique* est employé pour tous les termes en anglais ainsi que les légendes des figures et des tables.
- Toute référence à des noms de fichiers, à des chemins d'accès, ou à des utilisations de variables et de commandes est faite avec la police d'écriture `Courier New`.
- Enfin, tous les extraits de code sont insérés dans une zone de texte encadrée en noir avec un fond gris.

1.3 Remerciements

Je tiens à remercier M. Eric Jenny pour son suivi et ses conseils ainsi que pour m'avoir donné la possibilité de réaliser ce travail de Bachelor dans un domaine qui m'est cher.

Ma reconnaissance va également à M. Mohamed Nizar Bbouchedakh qui m'a aidé et conseillé durant mes recherches et la réalisation de ce projet.

Finalement, je remercie ma compagne pour la relecture de ce document ainsi que ma famille et mes proches pour leur soutien apporté toute au long de ce travail.

2. Introduction

Étant donné les possibilités qu'elle offre en matière d'automatisation et de contrôle de systèmes au sein d'un bâtiment ainsi que le développement et la production toujours plus importante d'objets connectés, la domotique est aujourd'hui une technologie qui suscite l'intérêt des entreprises en bâtiment mais également de l'ingénierie informatique.

D'ailleurs, l'hepia de Genève partage cet intérêt car depuis quelques années, elle souhaite procéder à l'automatisation de ses salles. Toutefois, à ce jour, malgré la mise en place d'éléments prévus à cet effet tels que des multi-senseurs, aucune application de gestion n'a encore été développée. C'est donc dans ce but que ce projet a été conçu et réalisé. À plus large spectre, il vise également à créer une émulation générale sur la question de l'automation à l'hepia et susciter d'autres éventuels projets de ce genre.

D'autre part, un tel projet pourrait aussi participer de manière significative – par l'élaboration de différents scénarii – à effectuer des économies d'énergie et d'argent dans les divers bâtiments de l'hepia, notamment en matière de chauffage.

Enfin, j'apporte un soin tout particulier à ce travail de Bachelor car la domotique est un domaine que j'affectionne beaucoup. Le principe d'automatisation en soi est en effet un principe que je trouve des plus passionnants. J'ai du reste déjà eu l'occasion de travailler sur cette thématique lors de mon travail de technicien ES qui portait précisément sur la réalisation d'un système de domotique à partir de cartes de développement. De plus, j'utilise moi-même au quotidien nombres d'objets connectés tels que des prises permettant de visualiser la consommation d'électricité, des stores électriques ou encore des lampes connectées.

Par ce projet, j'espère donc contribuer un peu plus au développement de la domotique.

2.1 Objectifs

L'objectif premier de projet est de mettre en place une infrastructure permettant à des utilisateurs de gérer un réseau constitué d'un ensemble de multi-senseurs ayant pour fonction de mesurer la température, l'humidité, la luminosité et la présence mais également de régler des vannes de radiateurs ainsi que des stores. Cette infrastructure sera ensuite administrée par deux applications : l'une permettant la gestion globale de l'infrastructure et l'autre, l'automatisation des divers éléments du réseau.

On a également souhaité que ces deux applications soient le plus facile d'utilisation pour ceux qui seront amenés à l'employer. On va donc porter une attention toute particulière à l'ergonomie et le design de leurs interfaces.

2.2 Cahier des charges

Le cahier des charges est le suivant :

1. Étudier et décrire la structure actuelle ainsi que les éléments constituants le réseau *Smarthepia*
2. Analyser les différentes solutions envisageables pour réaliser un système de gestion basé sur les thématiques suivantes : *Fault*, *Configuration*, *Accounting*, *Performance and security management*.
3. Concevoir et réaliser une application permettant à des utilisateurs de gérer et de visualiser l'état des divers éléments constituant le réseau *Smarthepia*. En d'autres termes, celle-ci doit permettre la gestion d'utilisateur ainsi que leurs droits associés ; la gestion et le quittancement des alarmes.
4. Élaborer une application de gestion afin de pouvoir contrôler et automatiser de manière individuelle les élément constituant le réseau *Smarthepia*.

3. Planning

Avant de commencer ce projet de Bachelor, il a été demandé d'établir un planning prévisionnel où les grandes étapes doivent être définies puis divisées à la demi-semaine afin de pouvoir déterminer si le projet était réalisable dans le temps imparti.

3.1 Prévisionnel

Voici les tâches qui ont été agencées à la demi-semaine.

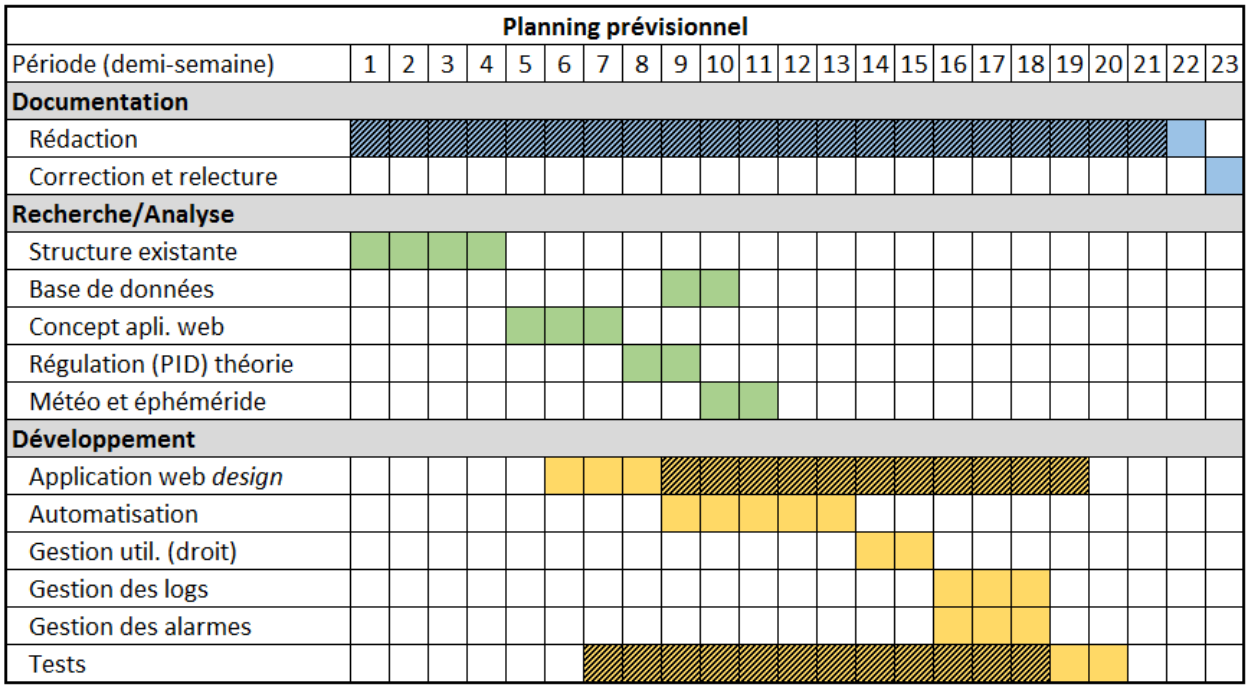


Figure 1 : Planning prévisionnel

3.2 Actuel

Voici maintenant concrètement comment les tâches ont été exécutées. Le journal de bord a en effet permis de savoir à quel moment celles-ci ont été commencées et terminées.

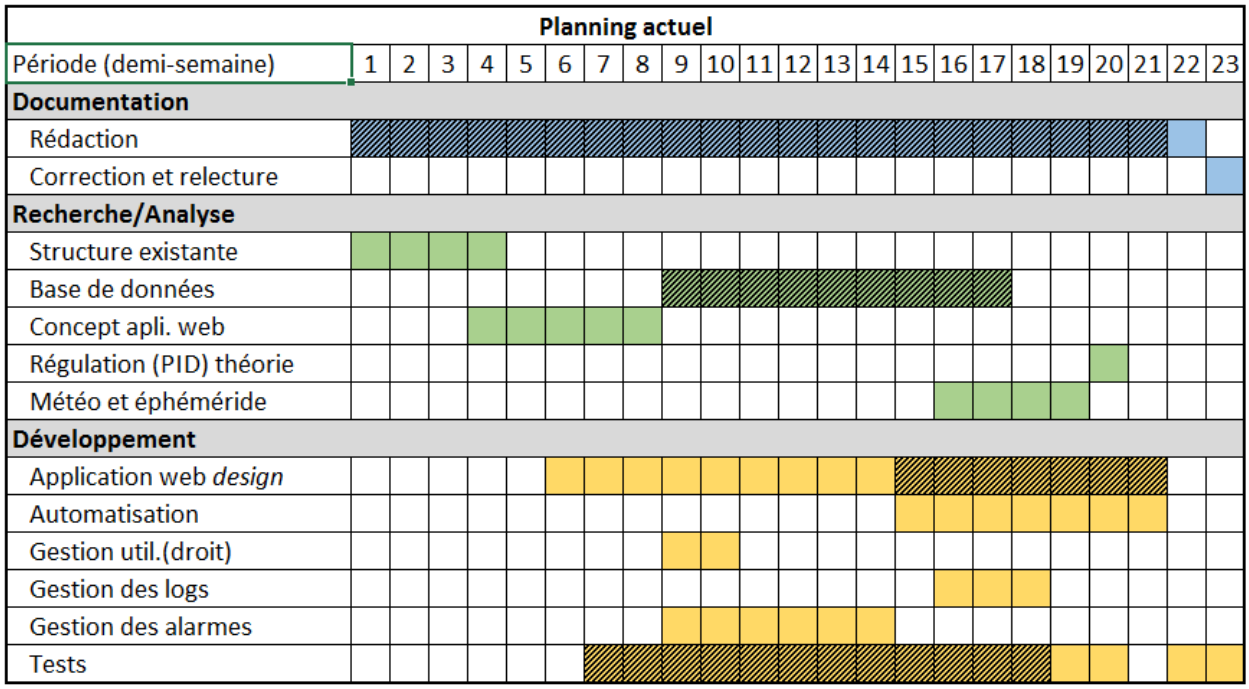


Figure 2 : Planning actuel

4. Analyse de l'existant

4.1 *Smarthepia*

Entre 2014 et 2018, le projet de recherche iNUIT (1), fondé par la HES-SO, a pour objectif de concevoir et mettre en œuvre une infrastructure fiable, sécurisée et évolutive de collecte de données provenant de différents types de capteurs. L'hepia de Genève (rue de la Prairie) s'est alors jointe à cette initiative avec pour but de développer une plateforme IoT nommée *Smarthepia* (2). Cette dernière permettrait ainsi de collecter plusieurs types de données – comme par exemple, la température, la présence, l'humidité et la consommation d'énergie – afin de les utiliser pour contrôler les stores, les lumières et les radiateurs du bâtiment.

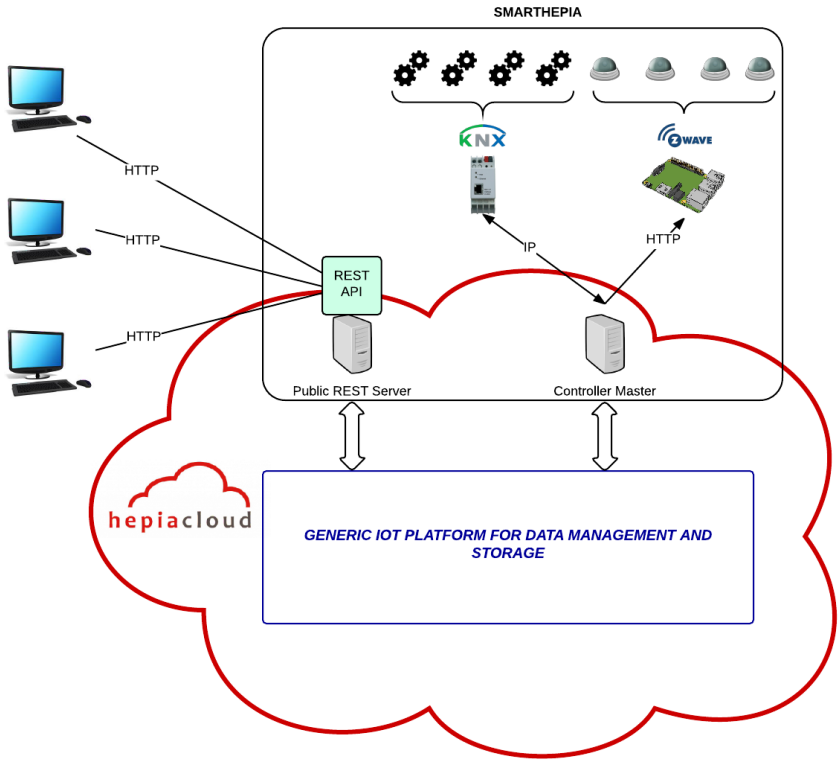


Figure 3 : Projet Smarthepia

À ce jour, *Smarthepia* est un réseau de multi-senseurs et d'actionneurs – stores et vannes de radiateur – qui a été mis en place uniquement dans certaine salle du quatrième et cinquième étage du bâtiment A. Toutefois, les données collectées par les multi-senseurs ne sont pas utilisées, c'est-à-dire aucune automatisation ou régulation n'a à ce jour été mise en place sur les actionneurs. C'est pourquoi l'infrastructure est uniquement utilisée dans un cadre éducatif pour par exemple réaliser des travaux personnels de Bachelor ou de Master.

Voici les composants qui sont à disposition :

5. Multi-senseur Z-Wave
6. Raspberry PI 2B+ avec *dongle* USB Z-Wave
7. Actionneur de vannes de radiateur
8. Actionneur de stores
9. Gateway KNX

4.2 Composants

Dans cette section, pour bien comprendre comment interagissent les différents composants qui vont suivre, voici une brève description des deux modes de communication utilisés :

10. Multi-senseur : technologie Z-Wave
11. Actionneur : technologie KNX

4.2.1 Z-Wave

Z-Wave (3) est un protocole radio conçu spécifiquement pour les applications de domotique ou d'habitat communicant. Il communique en utilisant une technologie radio de faible puissance dans une bande de fréquence allant de 800-900 [MHz], ce qui est idéal pour les éléments qui fonctionnent sous batterie. D'autre part, la technologie Z-Wave est sécurisé car le chiffrement (4) des échanges est assuré par l'algorithme de chiffrement AES-OFB disposant d'une clé symétrique de 128 [bits]. Dans réseau Z-Wave, il deux types de périphériques :

12. Contrôleur : celui qui permet d'ajouter un nœud sur le réseau et de relayer son information
13. Nœud : celui qui donne des informations comme par exemple la température (thermomètre)

Dans ce réseau, il est possible d'ajouter un maximum de 232 nœuds (5) sur un contrôleur. Chaque contrôleur dispose d'un identifiant unique (4 bytes) qui lui est attribué afin d'éviter l'échange d'informations entre un réseau et un autre par les nœuds (isolation entre les réseaux). À l'ajout d'un nœud, le contrôleur lui attribue ainsi un identifiant unique (1 byte) de réseau ce qui permettra de l'identifier.

Contrairement à un réseau sans fil traditionnel, lors d'un éloignement trop important ou d'une perturbation qui empêcherait alors d'attendre le contrôleur, le nœud perdra automatiquement la connexion et il ne pourra par conséquent plus envoyer d'informations.

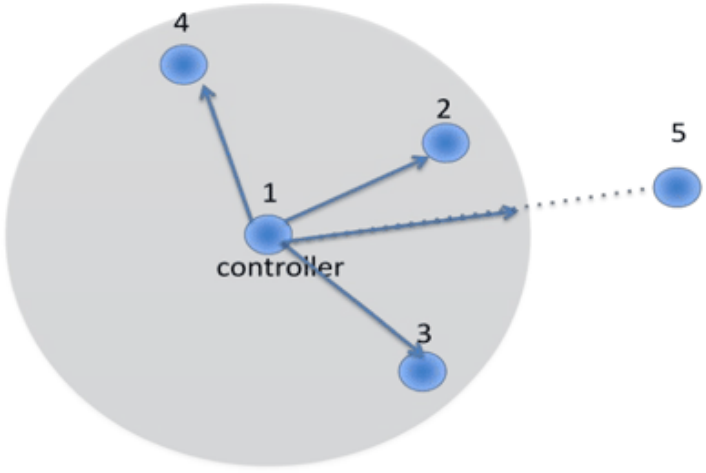


Figure 4 : Réseau traditionnel

Pour remédier à cette problématique (Figure 4), la technologie Z-Wave offre un mécanisme qui permet de relayer (Figure 5) l'information d'un nœud à un autre sans avoir de connexion directe avec le contrôleur. Cela augmente considérablement le rayon d'action des nœuds dans le cas où ceux-ci sont positionnés idéalement.

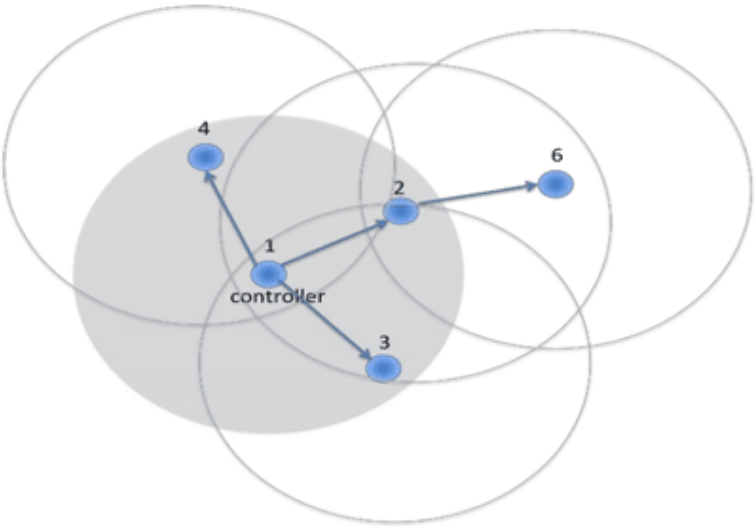


Figure 5 : Z-Wave relais de l'information

4.2.2 KNX

KNX (6) est à la fois un bus de terrain et un protocole d'automatisme pour bâtiment. Il a été mis en place à la suite à un besoin de standardisation permettant l'interopérabilité entre plusieurs constructeurs. Le bus KNX étant un bus de commandes, il ne délivre donc pas l'énergie pour l'alimentation des éléments. Le transport de l'information du protocole KNX peut s'effectuer de plusieurs manières :

- Ethernet
- Courant porteur
- Radio
- Paire torsadée

Dans le cas du *Smarthepia*, le contrôleur communique par une paire de cuivre torsadé qui permettent des débits allant de 4800 [bit/s] à 9600 [bit/s].

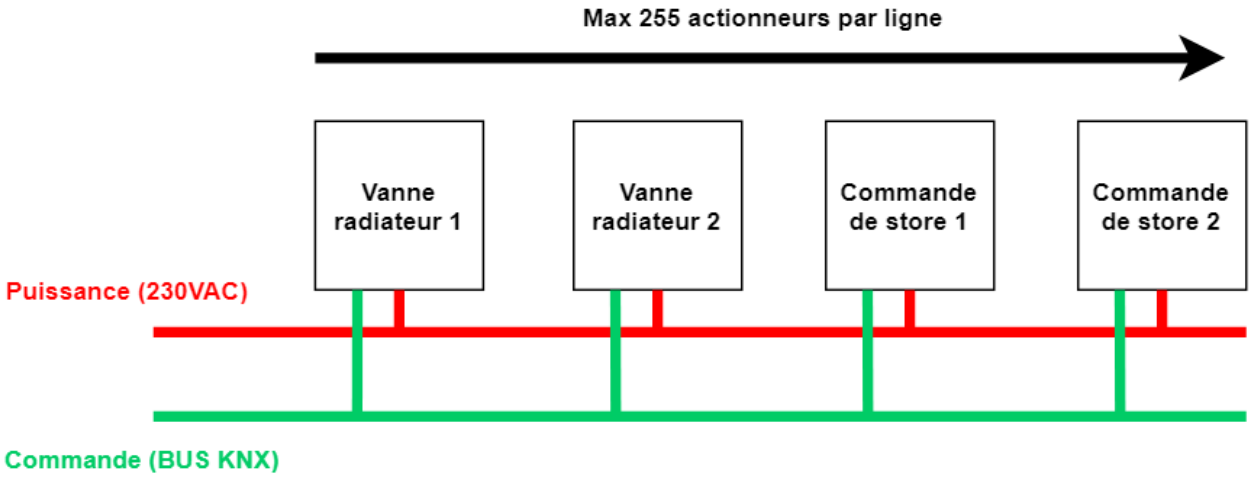


Figure 6 : KNX ligne exemple

Une ligne (Figure 6) permet de connecter en série jusqu'à 256 éléments, qu'ils soient de type actionneur ou multi-senseur. Chaque élément d'une ligne (bus KNX) dispose d'une adresse unique qu'on appelle « adresse physique ». Cette adresse physique doit être unique car lors de l'envoi d'une commande tous les éléments connectés sur la ligne reçoivent l'information mais seulement celui ayant la bonne adresse récupère la commande et la traite.

Une adresse physique (Table 1) est composée de trois parties ce qui permet d'étendre considérablement le nombre d'éléments connectés au bus KNX.

Area (4 bits)	Ligne (4 bits)	Élément sur le bus (1 byte)

Table 1 : KNX adresses physiques

Il est donc possible de créer 57600 éléments sur le bus KNX :

- 256 éléments par ligne
- 16 lignes moins la ligne principale qui relie les autres lignes pour chaque area
- 16 areas moins la ligne qui relie toutes les areas (*backbone*)

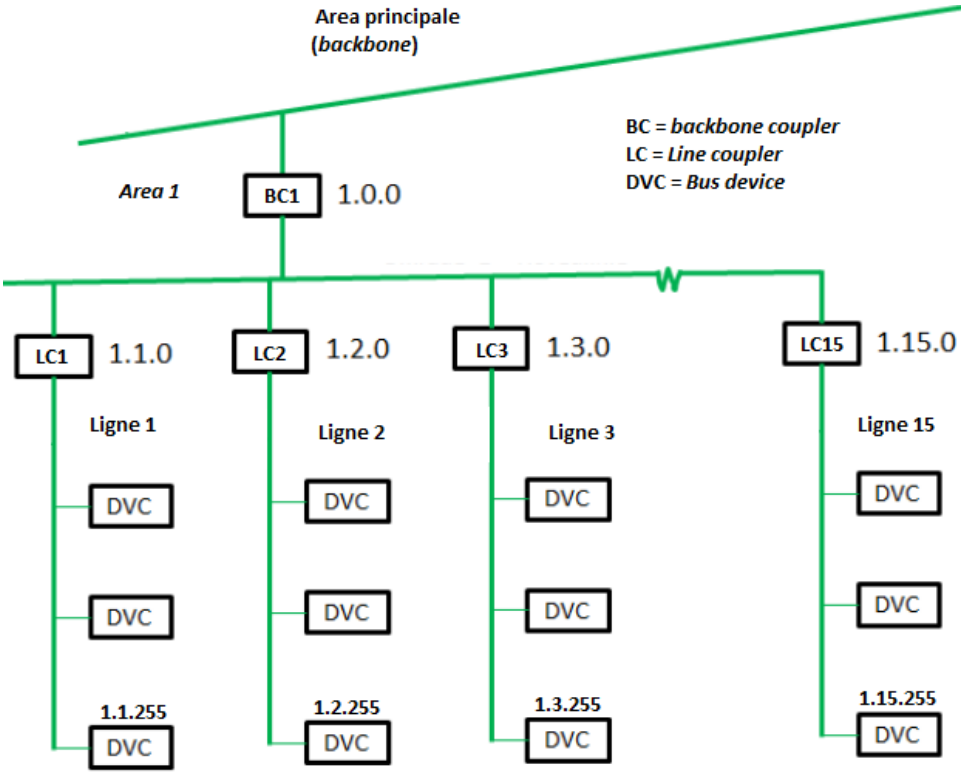


Figure 7 : Exemple de répartition des adresses physiques sur le bus KNX

En plus de l'adresse physique, il existe un autre type d'adresse prénommée « adresse de groupe » ou « adresse logique ». Cette adresse logique permet de grouper des éléments sur le bus KNX par leurs adresses physiques puis, de leur attribuer une action à effectuer. Cela permet par exemple d'allumer tous les radiateurs dans une salle à l'aide d'une commande. Les adresses logiques (Table 2) sont définies par l'installateur en fonction des éléments qu'il a disposition.

Groupe principale (5 bits)	Groupe secondaire (11 bits)	
Groupe principale (5 bits)	Groupe intermédiaire (3 bits)	Groupe secondaire (8 bits)

Table 2 : KNX adresse logique

Pour ajouter et gérer des éléments sur un bus KNX, il faut utiliser le logiciel ETS5 qui est à disposition sur le site officiel de KNX (7). Il propose en effet une version gratuite pour gérer cinq éléments sur le bus KNX. Au-delà, il faut acheter une licence.

4.2.3 Multi-senseurs

Le réseau de multi-senseurs est pour l'instant seulement disponible sur deux étages et uniquement dans le bâtiment A de l'hepia. Le quatrième étage (Figure 8) est équipé de 19 multi-senseurs dont trois sont dans le couloir, le reste étant ajouté par paire dans les salles de cours pour avoir une redondance.

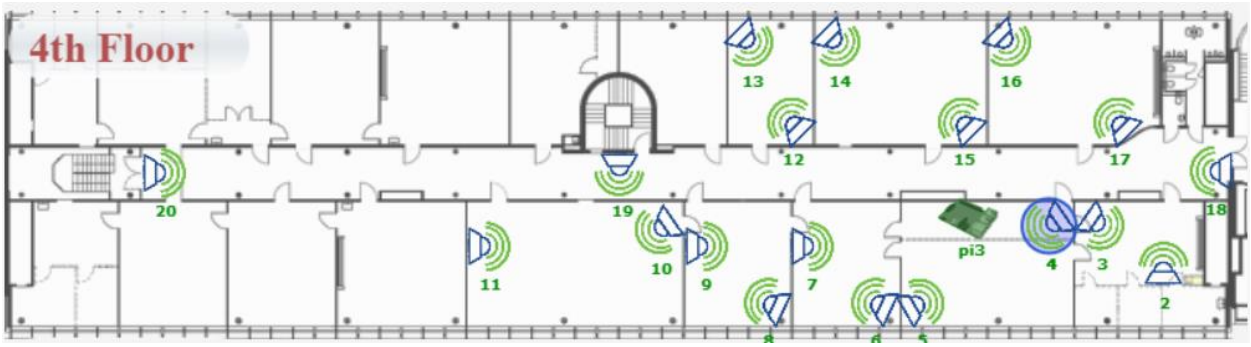


Figure 8 : disposition multi-senseurs quatrième étage

Le cinquième étage (Figure 9) est quant à lui équipé de 23 multi-senseurs dont quatre sont dans le couloir, le reste étant ajouté par paire dans les salles de cours.

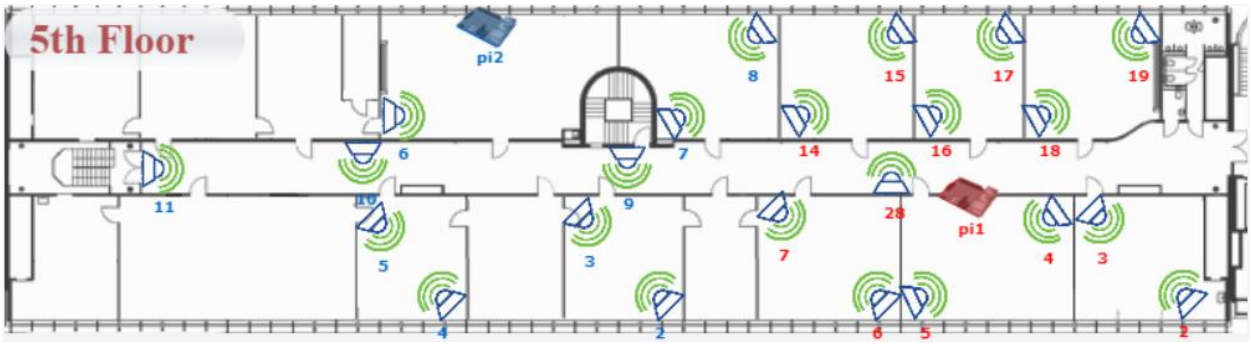


Figure 9 : disposition multi-senseurs cinquième étage

Voici les mesures qui sont disponibles pour chaque multi-senseur :

- Température
- Humidité
- Présence
- Luminosité
- Niveau de batterie

Le niveau de batterie est indicatif car les trois quarts des multi-senseurs sont reliés directement à une prise de courant. L'intervalle de rafraîchissement des données des multi-senseur est de quatre minutes pour éviter que le réseau Z-Wave ne soit surchargé de requêtes par les utilisateurs.

On remarque que sur les deux images représentant les salles du quatrième (Figure 8) et cinquième étage (Figure 9), chaque multi-senseurs est associé à une RPI. En effet, les multi-senseurs fonctionnant avec la technologie Z-Wave, il a fallu les interfacer pour qu'un utilisateur puisse disposer facilement des mesures. Pour cela, trois RPI ont été réparties stratégiquement en fonction de la position des multi-senseurs afin

de garantir une couverture idéale. Chaque RPI dispose d'un *dongle* USB qui fait office de contrôleur Z-Wave et qui effectue ainsi le lien entre les multi-senseurs qui lui sont associé. La RPI dispose d'un serveur de type REST qui va permettre à l'utilisateur de récupérer les informations d'un multi-senseur en particulier par le bais de requêtes HTTP de type GET.

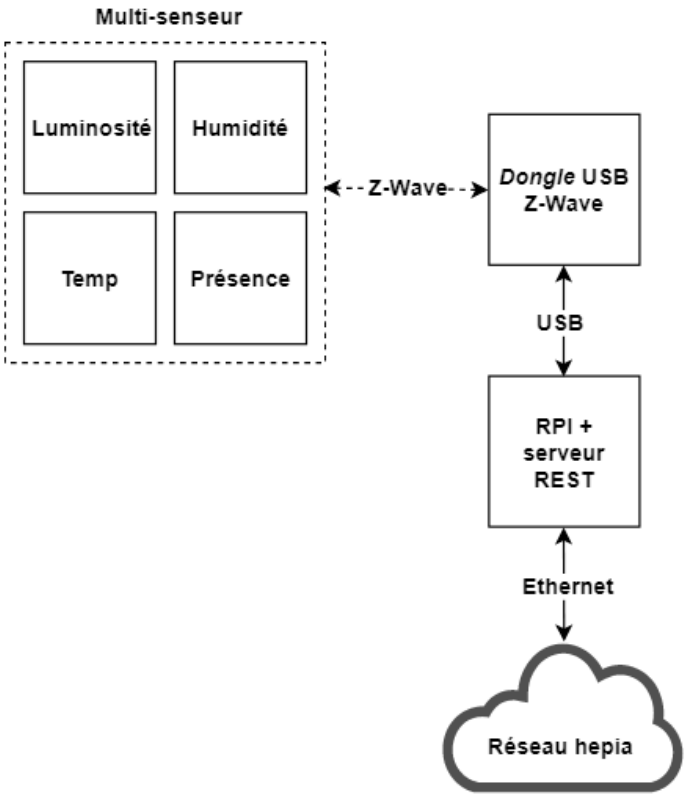


Figure 10 : Réseau multi-senseur schéma bloc

Les trois RPI sont disponibles aux adresses suivantes sur le port 5000 :

- 129.194.184.124 (RPI1 4èmes étages)
- 129.194.184.125 (RPI2 5èmes étages)
- 129.194.185.199 (RPI3 5èmes étages)

Chaque multi-senseur dispose d'un identifiant unique par réseau. Pour déterminer quel est cet identifiant, il faut effectuer une requête HTTP de type GET sur le serveur REST de la RPI qui possède le multi-senseur :

```
wget http://<IP RPI>:5000/sensors/get_sensors_list

Réponse au format JSON (HTTP 200 OK):
{  2: "Multi Sensor",
   3: "Multi Sensor",
   28: "Multi Sensor" }
```

Table 3 : Commande HTTP GET pour obtenir la liste des multi-senseurs

Le numéro à gauche représente l'identifiant d'un multi-senseur. Par conséquent, il est possible de récupérer les mesures d'un de ces multi-senseur. Voici comment procéder :

```
wget http://{IP RPI}:5000/sensors/{ms id}/get_all_measures

Réponse au format JSON (HTTP 200 OK):
{  battery: 66,
   controller: "Pi 1",
   humidity: 44,
   location: "A501",
   luminance: 4,
   motion: false,
   sensor: 2,
   temperature: 26.7,
   updateTime: 1530826071 }
```

Table 4 : Commande HTTP GET pour obtenir les mesures d'une multi-senseur

Il est aussi possible de récupérer les mesures individuellement en utilisant les requêtes suivantes :

```
wget http://{IP RPI}:5000/sensors/{ms id}/get_temperature
wget http://{IP RPI}:5000/sensors/{ms id}/get_humidity
wget http://{IP RPI}:5000/sensors/{ms id}/get_luminance
wget http://{IP RPI}:5000/sensors/{ms id}/get_motion
```

Table 5 : Liste des commande HTTP GET pour obtenir un seul type de mesure pour un multi-senseur

4.2.4 Actionneurs

Le réseau d'actionneurs n'est pour l'instant que disponible dans certaines salles du quatrième et cinquième étage. On distingue deux types d'actionneurs :

- Les actionneurs de stores
- Les actionneurs de vannes de radiateur

Pour pouvoir contrôler ces deux types d'actionneurs, voici un schéma d'interconnexion entre les différents éléments mis en place :

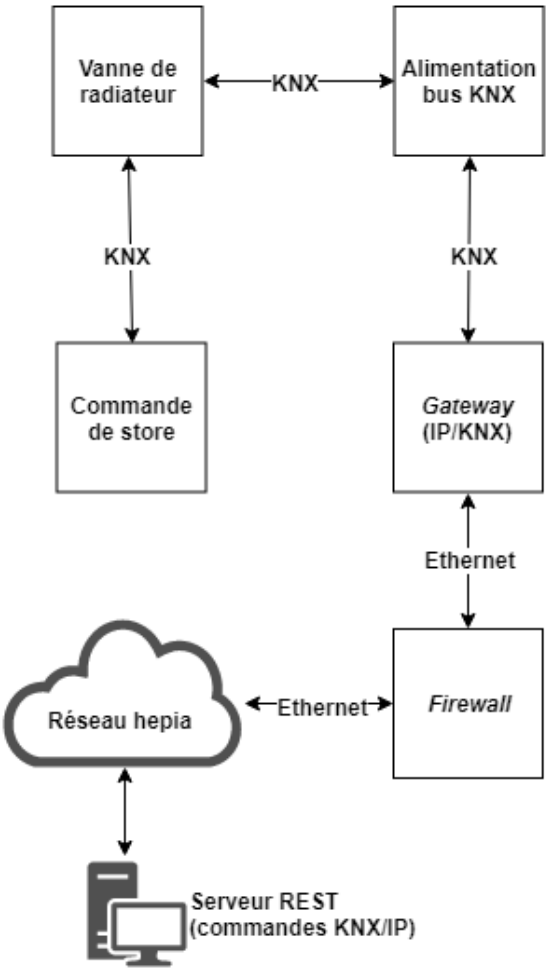


Figure 11 : Réseau KNX schéma bloc

Les actionneurs de stores et de radiateurs utilisent un bus de données de type KNX matérialisé par une paire de cuivre torsadé. Pour contrôler un store ou un radiateur, on dispose d'une machine physique avec un serveur REST qui va permettre à un utilisateur d'envoyer des requêtes HTTP de type GET pour récupérer la valeur courante ou écrire une valeur spécifique sur un élément. Le message envoyé par l'utilisateur est alors traité par le serveur REST, retranscrit en type KNX puis, envoyé à la *gateway*. La *gateway* va désencapsuler le paquet IP puis l'envoyer le message sur le bus KNX. Vu que tous les

actionneurs sont connectés en série, ils recevront tous ce message mais grâce à un champ de type adresse, seul l'actionneur ayant l'adresse correspondante actionnera sont mécanisme.

Pour des raisons de sécurité, la *gateway* a été placée derrière un *firewall* et seuls les clients autorisés peuvent envoyer des commandes sur le bus KNX.

L'adresse logique pour chaque actionneur est formée par la combinaison de trois éléments – x/y/z – :

- X : action (lecture/écriture) sur un actionneur
- Y : numéro de l'étage
- Z : numéro du bloc qui comporte une vanne de radiateur et un store (identifiant)

Pour interroger un actionneur, il est nécessaire d'obtenir la liste des adresses logiques correspondantes. Les adresses logiques pour ces deux types sont construites de la manière suivante (ex : élément n°1 au cinquième étage) :

- Adresse pour les vannes de radiateurs
  - Lecture : 0/5/1
  - Écriture : 0/5/1
- Adresse pour les stores
  - Lecture : 3/5/1
  - Écriture : 4/5/1

Les valeurs qui sont lues ou que l'on écrit sur un actionneur correspondent à un intervalle de 0 à 255. Pour une vanne de radiateur, 0 indique que la vanne est complétement fermée et 255 qu'elle est complétement ouverte. À l'inverse, pour un store, 0 indique qu'il est complètement ouvert et 255 qu'il est complétement fermé.

Pour récupérer la valeur d'un actionneur de type vanne de radiateur ou store, il faut effectuer une requête HTTP de type GET sur le serveur REST de la manière suivante :

```
wget http://{IP}:5000/v0/{type}/read/{numéro étage}/{actionneur id}

Réponse au format JSON (HTTP 200 OK:
{ response:185,
  status: 0}
```

Table 6 : Commande HTTP GET pour obtenir la valeur d'un actionneur

Pour écrire une valeur sur un actionneur de type vanne de radiateur ou store, il faut effectuer une requête HTTP de type GET sur le serveur REST de la manière suivante :

```
wget http://{IP}:5000/v0/{type}/write/{numéro étage}/{actionneur
id}/{valeur}

Réponse au format JSON (HTTP 200 OK:
{ response: "Command sent",
  status: 0}
```

Table 7 : Commande HTTP GET pour écrire une valeur sur un actionneur

La *gateway* n'étant pas accessible pour tous les clients, un simulateur a été créé pour reproduire le comportement des actionneurs lors de l'envoi de commandes sur le bus KNX.

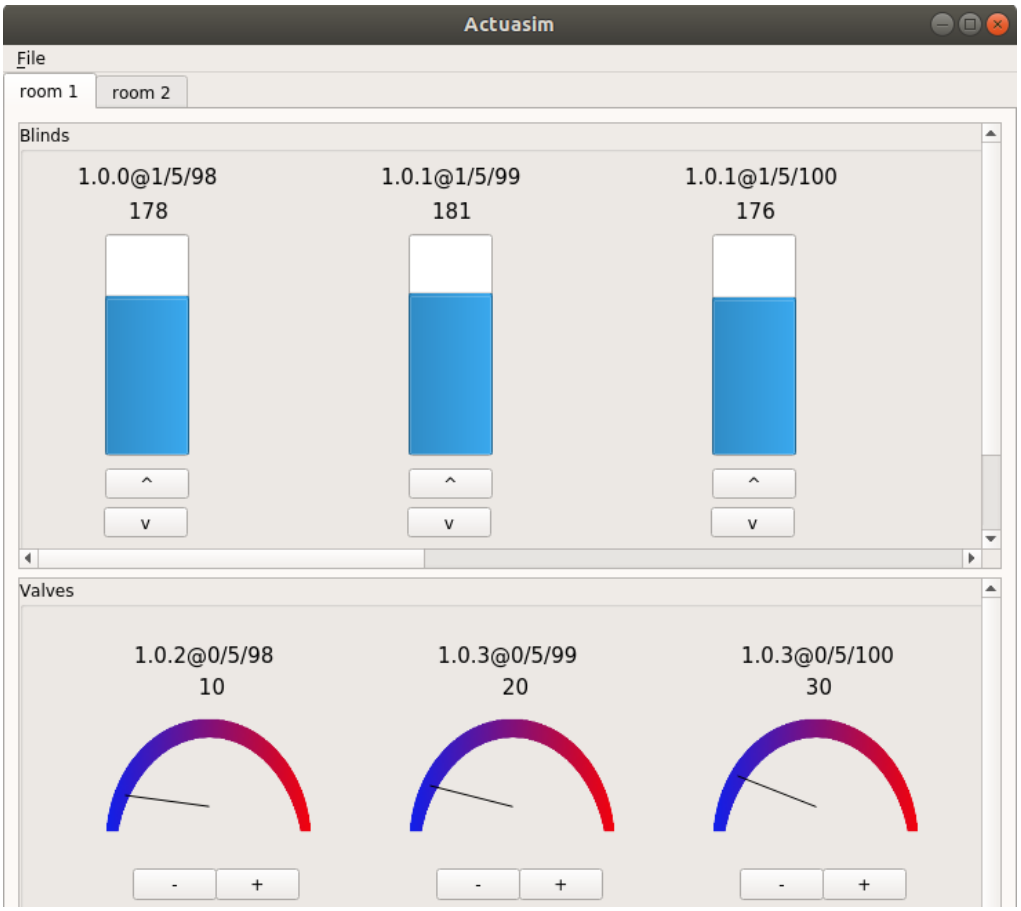


Figure 12 : Simulateur élément KNX

Le simulateur permet en outre d'ajouter des salles, des vannes de radiateur ainsi que des stores. L'ajout de ces éléments se fait alors par le biais d'un fichier JSON. Les adresses (physique@logique) qui se situent en haut de chaque élément correspondent à l'adresse physique suivie d'un arobase puis, de l'adresse logique.

5. Analyse préliminaire

Pour la réalisation de ce projet *Smarthepia*, il est nécessaire de définir préalablement les éléments importants qui le composeront :

- Une application web pour gérer le *Smarthepia*
- Une application pour contrôler et automatiser les actionneurs ainsi que les multi-senseurs
- Une base de données pour centraliser les informations entre les deux applications

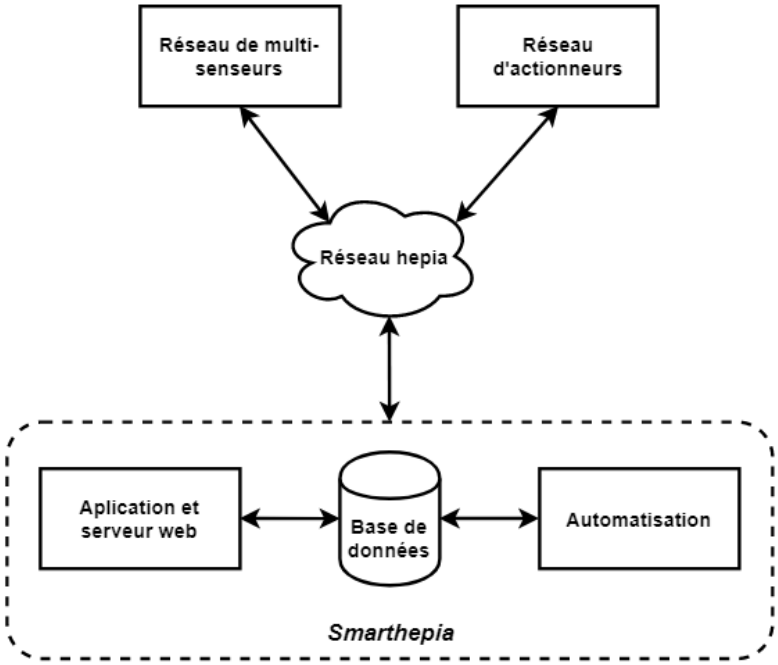


Figure 13 : Smarthepia schéma initial

D'autre part, pour mettre en œuvre ces trois éléments, il a fallu faire des choix. Pour cela, on a fait des recherches pour définir ce qui serait le plus judicieux d'utiliser.

5.1 Application web

Un application web peut être conçue de nombreuses manières mais trois éléments sont essentiels : le *front-end* correspondant à ce que l'utilisateur peut visualiser de l'application ; le *back-end*, c'est-à-dire le traitement réalisé lors d'une requête d'un utilisateur ; et finalement le serveur web qui va traiter les requête HTTP de l'utilisateur.

Pour ce qui est de la partie *front-end*, on a l'habitude d'utiliser le trio HTML5, CSS3 et Javascript – les plus employés par les applications web actuelles – car ces trois éléments sont complémentaires quand il s'agit de présenter à l'utilisateur une page dans son navigateur. En effet, l'HTML permet de structurer tout ce qui concerne le texte et la structure de la page. Le CSS3 est quant à lui utilisé pour la partie présentation, c'est-à-dire le design. Enfin, JavaScript permet de rendre dynamique les éléments proposés à l'utilisateur, soit de créer une interaction entre l'utilisateur et la page web. Avec ce langage de programmation, on peut également effectuer des tâches sur le client et non sur le serveur, ce qui peut s'avérer pratique dans certains cas.

Toutefois, on a trouvé intéressant d'étudier également de nouvelles options : comme *Angular* – un Framework proposer par Google –, ou *Vue*, un Framework sous licence MIT proposé par Evan You. Mais, après avoir effectué quelques tests sur ces deux solutions, on a finalement décidé de les écarter car cela nécessitait d'acquérir des notions qui n'auraient pas pu l'être dans le temps imparti pour ce projet. Elles restent néanmoins des solutions très intéressantes qui pourront éventuellement être étudiées à la suite de ce projet.
Par conséquent, après avoir fait la synthèse des différentes options envisageables, on a décidé de garder le trio HTML5, CSS3 et Javascript.

Le serveur web va permette à l'utilisateur d'accéder aux données que le serveur met à disposition. En guise de serveur web, on a ainsi décidé d'utiliser Nodejs. Il s'agit d'une plateforme orientée application réseau mais dont l'un des avantages est de pouvoir installer une multitude de modules dont un module HTTP servant précisément à la récupération de requêtes envoyées par l'utilisateur.
Pour installer un de ces modules, il faut au préalable avoir installé *npm*.

```
npm install express --save
```

Table 8 : Installation module avec npm

Pour créer une première application avec Nodejs, il est nécessaire de créer un fichier qui va contenir la base de toute l'application web. Pour cela, on créer un fichier *server.js* qui sera exécuté par Nodejs.

```
Fichier server.js

const http = require('http')
const port = 3000
const requestHandler = (req, res) => {
  response.end('Hello Node.js Server!')}
const server = http.createServer(requestHandler)
server.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err)
  }
  console.log(`server is listening on ${port}`)})
```

Table 9 : Démarrage d'une application web avec Nodejs

Pour l'exécuter, il suffit de faire un appel à Nodejs.

```
node server.js
```

Table 10 : Création d'une application web avec Nodejs

Ayant toujours utilisé Apach ou Nginx avec PHP et MySql pour le web, il a en effet semblé intéressant d'étudier une autre option. Nodejs peut être exécuté sur la plupart des plateformes comme Windows, Unix et Mac OSX en installant seulement un seul paquet. Il permet en outre d'exécuter un grand nombre de tâches de manière asynchrone, ce qui est très utile mais qui peut créer des complications pour le *debug* de l'application.

Pour le *back-end*, deux modules spécifiques sont souvent utilisés quand une requête sur le serveur est envoyée par un client. Le premier se nomme *express* et fait office de *middleware*, c'est-à-dire que le module accède à la requête de l'utilisateur et interagit sur la réponse qui lui sera envoyé. Ce module est aussi essentiel pour la création des routes qui permettent à l'utilisateur de naviguer sur l'application web. Voici comment est définie la route qui permet d'accéder à la racine du serveur web.

```
var express = require('express');
var app = express();
app.get('/', function(req, res){
   // middleware
});
```

Table 11 : Exemple d'utilisation du module expresse

De plus, *express* intègre aussi un moteur de rendu qui permet d'utiliser des *templates* et ainsi afficher dynamiquement des pages HTML en donnant des arguments au *template*. Le deuxième module, *ejs* est quant à lui un langage de *template* qui permet de générer des page HTML. Il utilise des balises (<% %>) spécifiques qui seront interprétées par *express* à la suite d'une requête d'un client. Voici à quoi ressemble un *template* :

```
Fichier index.ejs

<html>
  <head><title><%= title %></title></head>
  <body>
    welcome <%= user%>;
  </body>
</html>
```

Table 12 : Template de ficher ejs

Pour utiliser ce *template*, il faut tout d'abord ajouter *ejs* comme moteur de rendu puis, appeler le *template* en lui donnant des variables (*title* et *user*).

```
app.set('view engine', 'ejs');
app.get('/', function(req, res){
     res.render('index',{user: "Great User",title:"homepage"});
});
```

Table 13 : Exemple de passage d'argument dans me middleware

Ejs permet d'effectuer plusieurs opérations sur un *template*, telles que des boucles, des conditions, des inclusions d'autres fichiers *ejs*. Il possède également un grand nombre optimisations supplémentaires pouvant notamment permettre d'afficher dynamiquement des informations à l'utilisateur.

Pour l'application web, plusieurs autres modules ont été ajoutés à Nodejs. Cependant, il serait trop long de tous les détailler ici mais ils le seront dans la suite de ce rapport, dans la partie consacrée à la description des différents points de développement de l'application web.

5.2 Automatisation :

Concernant l'application de gestion et d'automatisation des actionneurs et des multi-senseurs, on a dû faire un choix quant au type de langage à privilégier. Le choix s'est logiquement tourné vers Python, car ce dernier offre un grand nombre de librairies et permet de programmer rapidement des fonctionnalités, ce qui serait moins évident à faire dans un autre langage comme le C. De plus, étant pas dans la filière « communication et réseau », la programmation n'est pas l'option la plus pratiquée. Donc, Python semblait le meilleur candidat pour optimiser la productivité tout en offrant des outils très performants et une syntaxe simple d'utilisation.

5.3 Base de données

Après avoir fait le choix de Nodejs comme serveur web, utiliser MongoDB comme base de données apparait comme la solution la plus logique. En effet, MongoDB enregistre ses données en format BSON (Binary JSON), lui-même basé sur le format de JSON (JavaScript Object Notation). Ainsi, il peut supporter différents types de données JavaScript, c'est pourquoi il est le meilleur choix pour les plateformes Nodejs. D'autre part, la particularité de ce type de base de données est qu'elle est de type NoSQL. Or, contrairement ce qu'on pourrait penser, NoSQL signifie « Not Only SQL » et non « No SQL ». Il ne s'agit donc pas d'un nouveau langage de requête mais d'une nouvelle approche du stockage de données. Concrètement, ce genre de base stocke des objets, appelés communément des « documents ». Ces documents sont alors regroupés à l'intérieur de collections, qui sont l'équivalent des tables dans le modèle relationnel. Contrairement à une base de données relationnelle, qui stockent ces données par ligne et par colonne de taille fixe, une base de données telle que MongoDB propose un stockage dans un document sans contrainte, autorisant ainsi un contenu et une arborescence très modulaire. Un autre point avantage non négligeable de la base de données MongoDB, est qu'il est possible d'effectuer des réplications et ainsi diminuer le risque d'inconsistance des données.

Dans ce projet, la base de données à un rôle central. Il est donc important que les ajouts et les recherches dans la base de données se fassent rapidement. C'est pour cela qu'on a décidé de faire un test – consistant à ajouter un grand nombre de données dans une collection – puis, d'effectuer une requête sur un des éléments présents dans cette base de données. Voici, les résultats du test pour deux millions d'entrées :

```
Résultat d'un find: 0.0060002803802490234 secondes
```

Table 14: Résultat d'un find avec MongoDB

Au vu des performance et des possibilités de MongoDB, on a décidé de l'utiliser comme base de données pour l'ensemble du projet.

6. Conception et réalisation

Dans cette section, on va détailler de manière chronologique toutes les étapes de ce projet, telles que la réalisation et mise en œuvre de la base de données, de l'application web et finalement, de l'application d'automatisation ainsi que des modifications apportées au serveur REST KNX. Toutefois, comme dans certains cas l'application web interagit directement avec la base de données ainsi que l'application d'automatisation, cet ordre ne sera pas toujours respecté. Mais avant cela, il a paru important d'expliquer tout d'abord dans quelles conditions ce projet a été réalisé.

6.1 Environnement de développement

Pour le développement des deux applications – web et automatisation –, on a travaillé avec le même environnement tout au long du projet afin d'éviter de rencontrer des problèmes de librairie ou d'incompatibilité avec les systèmes d'exploitation.

- Pour le programme d'automatisation :
    - Système d'exploitation : Windows 10 Pro 64x
    - Environnement de développement : Pycharm Community
    - Langage : Python 3.6

- Pour l'application web :
    - Système d'exploitation : Windows 10 Pro 64x
    - Environnement de développement : Webstorm
    - Langage : JavaScript, CSS3, HTML5

Le *versioning* du code des deux applications est réalisé sur un même *repository* Github (8) privé car dans le code, certaines informations ne doivent pas être diffusées, comme par exemple les adresses IP – publique et privée – ou certains logins.

6.2 Base de données

La base de données est un ensemble de collections. Or, comme mentionné dans le chapitre précèdent (cf. 5.3), MongoDB n'est pas une base de données de type relationnel. Cela implique qu'il n'y a pas de relation directe entre les collections et par conséquent, qu'il n'existe aucune clé primaire ou étrangère pour faire ce lien. Pour remédier à cela, on va lister les différentes collections en indiquant dans quels programmes (application web ou automatisation) elles sont utilisées et quelle est leur utilité. Pour simplifier, on utilisera APPW pour l'application web et APPA pour l'application d'automatisation.

Nom de la collection	**Programmes**		**Utilité**
	APPW	**APPA**	
Alarm	X	X	Contient les alarmes qui doivent être transmises aux deux applications.
Apicurrents	X	X	Cumul des données météorologiques fournies toutes les deux heures par une API.
Apiforecast		X	Cumul des données météorologiques fournies toutes les deux heures sur 3 jours par une API.
Automations	X	X	Données utilisées pour ajuster les options du programme d'automatisation.
Dependencies	X	X	Données qui correspondent aux dépendances entre les multi-senseurs, les actionneurs et le programme d'automatisation.
Devices	X	X	Données qui regroupent la hiérarchie des emplacements des multi-senseurs et des actionneurs.
Identitycounters	X		Sert à établir la relation entre les multi-senseurs, les actionneurs et leurs emplacements.
Rules	X	X	Règles qui seront appliquées sur les éléments du système *Smarthepia*.
Sessions	X		Permet de sauvegarder les sessions des utilisateurs lors de leur authentification sur l'application web.
Stats	X	X	Cumul des mesures des multi-senseurs collectées toutes les cinq minutes.
Status	X	X	Sert à mémorise l'état de disponibilité des éléments du *Smarthepia*
Users	X		Données personnelles de chaque utilisateur du *Smarthepia* concernant l'authentification et la gestion des permissions.

Table 15 : Listes des collections de la base de données MongoDB

6.3 Concept de l'application web

L'objectif de cette application web est de créer un système de gestion pour qu'un ou plusieurs utilisateurs puissent gérer le réseau *Smarthepia*. Plusieurs fonctionnalités ont été ainsi développées :

- Gestion des utilisateurs et des profils
- Gestion des alarmes et des logs
- Gestion des éléments du réseau
    - Multi-senseurs et actionneurs
    - Dépendance (les éléments auxquels les multi-senseurs et actionneurs sont reliés)
    - Emplacements où sont positionnés les multi-senseurs et actionneurs
- Gestion des règles à appliquer sur chaque élément du réseau
- Gestion des options pour le programme d'automatisation
- Affichage de l'état du réseau ainsi que de son statut
- Les statistiques des différents éléments sur le réseau

6.3.1 Template et croquis

Avant de commencer le développement de l'application web, il a fallu déterminer de quelle manière aborder la partie « interface utilisateur ». Cette partie est essentielles car l'utilisateur final ne sera en mesure d'apprécier le travail réalisé que par le biais de l'ergonomie et de la partie visible de l'application web. Pour cela, on a commencé à faire un ensemble de croquis des fonctionnalités afin de définir une « ligne de graphique » ou structure optimale. Comme le design web n'est pas enseigné durant les cursus proposés par l'hepia et que le développement d'une application web demande de telles connaissances, on a effectué des recherches de *templates* déjà existants avec les éléments constitutifs de l'application définis au préalable dans ces croquis :

- Page principale
- Gestion
- Option
- Statistique
- Barre de navigation

D'autre part, il faut préciser que ces croquis ne sont pas forcément représentatifs de ce qui a été réellement implémenté, autant pour les fonctionnalités que pour la mise en page. Il semble toutefois important de faire figurer dans ce rapport ces croquis car ils représentent une étape importante dans la réflexion relative au développement de cette application.

Enfin, en ce qui concerne la langue utilisée dans la partie visible de l'application, il s'agit de l'anglais uniquement. En effet, on a décidé de ne pas faire un site multilingue car on a jugé qu'à ce jour, les connaissances en anglais des personnes dans le milieu technique sont suffisantes à son utilisation.

6.3.1.1 Page principale

La page principale est celle où l'utilisateur arrive après s'être authentifié sur l'application web. Ensuite, grâce à la couleur, l'utilisateur doit tout de suite pouvoir identifier les éléments qui sont disponibles et indisponibles sur le réseau. Enfin, s'il clique sur l'un des éléments disponibles, il peut apercevoir son état actuel dans le réseau.

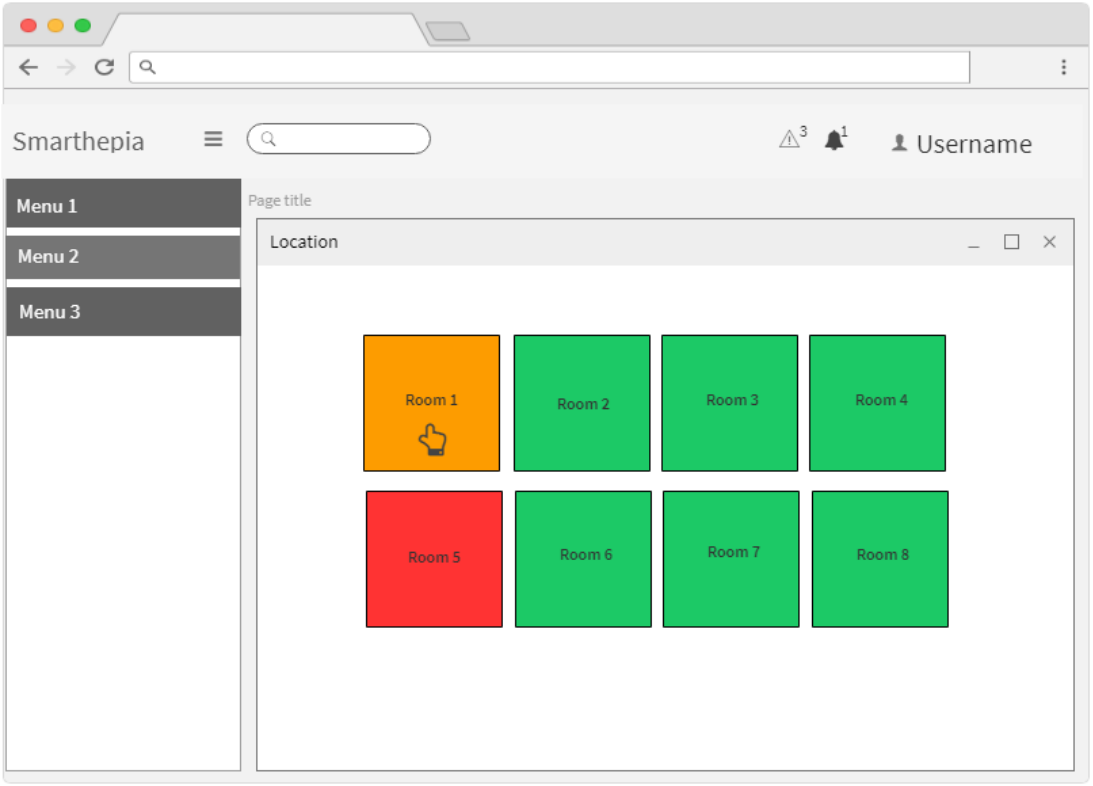


Table 16 : Croquis page principale

6.3.1.2 Page gestion

Une application web de gestion implique de pouvoir ajouter, modifier et supprimer des éléments. Dans le cas de ce projet, il s'agit d'alarmes, de lieux, de dépendances ou encore d'éléments tels que des multi-senseurs et des actionneurs. L'ensemble de tous les éléments crées sont listés dans un tableau. Ces éléments peuvent être modifiés ou supprimés au moyen de deux icônes situées sur la colonne de droite du tableau. D'autre part, l'utilisateur a aussi la possibilité d'ajouter un nouvel élément. Pour ce faire, au bas de la page se trouve des champs et des sélecteurs. Ces champs seront disposés de manière différente en fonction de quel élément on veut ajouter (alarmes, lieux etc.).

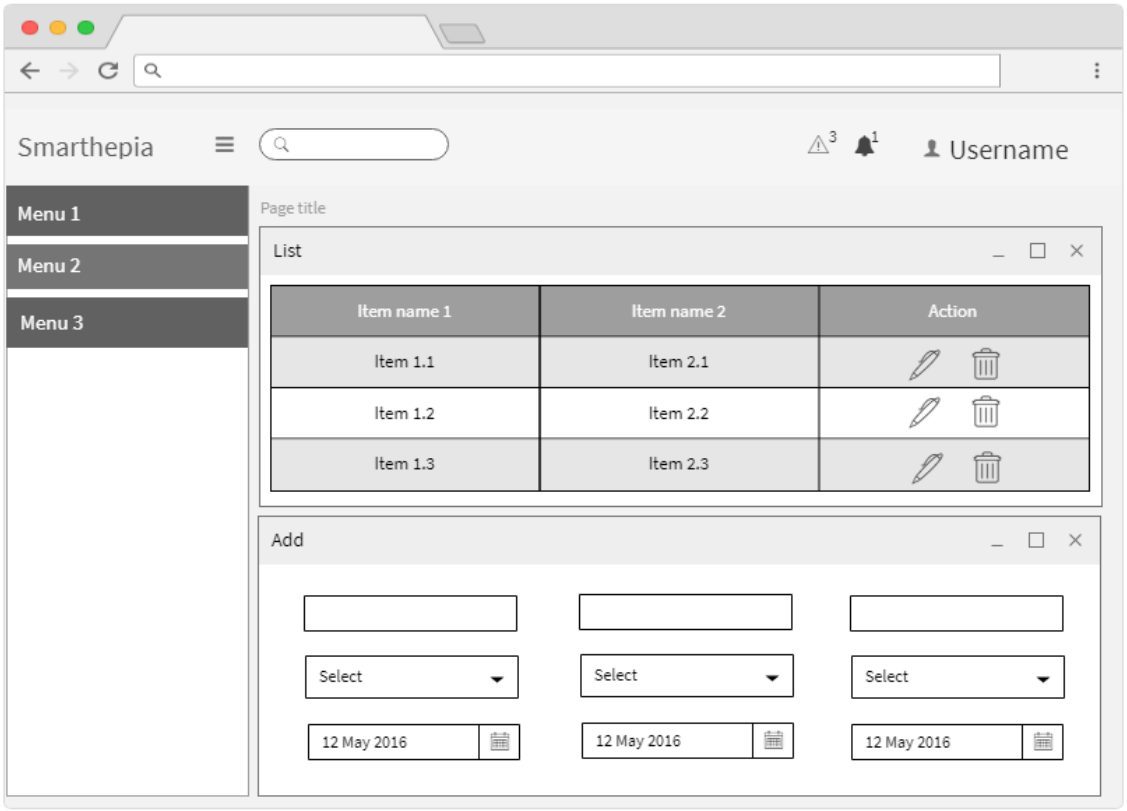


Table 17 : Croquis page gestion

6.3.1.3 Page options

La page « option » est utilisée pour entrer des paramètres qui seront utilisés par l'application d'automatisation du *Smarthepia*. Cela évite à l'utilisateur de devoir modifier des fichiers de configuration manuellement et ainsi risquer d'ajouter d'éventuelles erreurs.

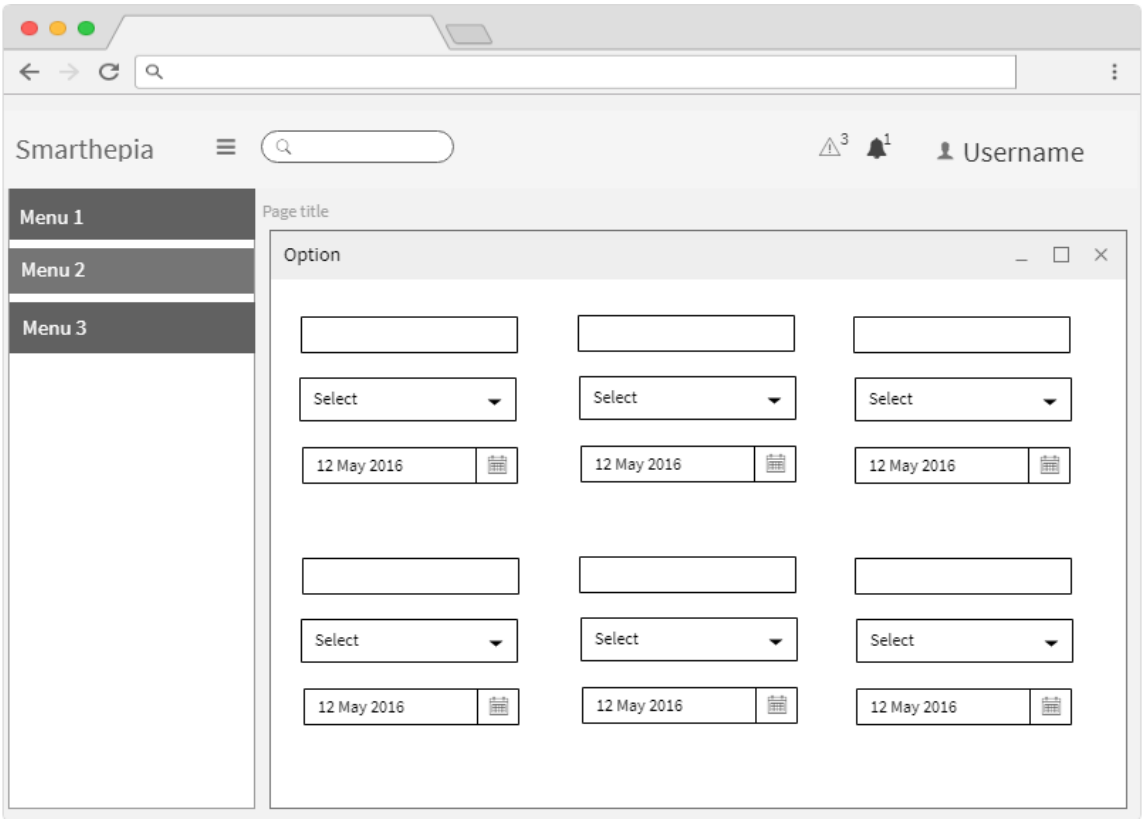


Table 18 : Croquis page options

6.3.1.4 Page statistiques

La page « statistique » permet d'afficher différentes mesures qui ont été collectées par les multi-senseurs ou les actionneurs, mais aussi d'autres éléments du *Smarthepia* que l'on retrouve notamment dans le programme d'automatisation.

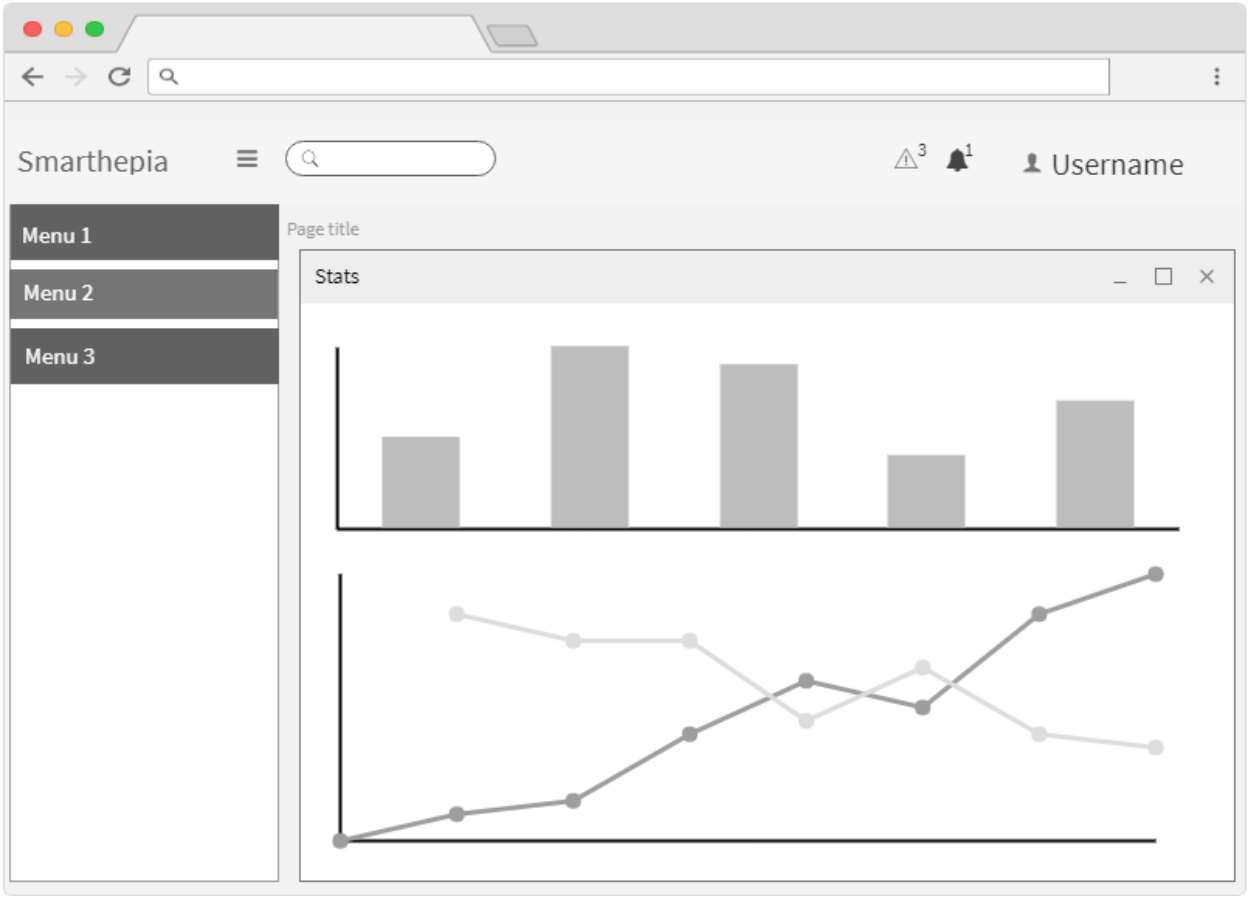


Table 19 : Croquis page statistiques

6.3.1.5 Barre de navigation

La barre de navigation a un rôle bien particulier. En effet, elle comporte deux icônes qui indiquent en temps réel, à l'utilisateur, si une alarme s'est déclenchée. Cela évite ainsi à l'utilisateur de devoir accéder à la page alarme et de constater qu'une nouvelle alarme est apparue.

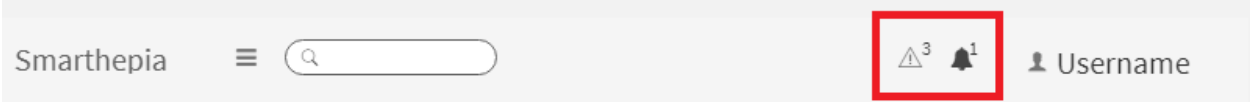


Figure 14 : Croquis barre de navigation

6.3.1.6 Choix du template

Après avoir fait plusieurs recherches et testé des *templates* gratuits et disponibles sur internet, aucun ne convenait, tant au niveau des possibilités offertes qu'au design. Par conséquent, des recherches supplémentaires ont été nécessaire, mais cette fois-ci aussi avec des *templates* payants. Rapidement le *template* « Stack » (9), disponible sur le site themforest.com, s'est avéré le meilleur compromis autant au niveau du prix, du design ou encore des fonctionnalités. Comme il s'agit d'un élément important de ce travail, on a décidé de l'acheter et de l'utiliser durant tout le développement de l'application web. Néanmoins, si l'utilisation de ce *template* peut se faire sur plusieurs postes puisqu'il a été acheté, la mise en ligne d'un seul *template* est autorisée.

6.4 Réalisation de l'application web

6.4.1 Structure de l'application

Pour une application de ce type, il est important d'établir dès le départ une structure claire et précise pour ne pas rencontrer, par la suite, des problèmes dans son développement. Pour ce faire, on a décidé de se baser sur le modèle MVC consistant à séparer l'application en trois parties :

- Modèle : contient les schémas de structure permettant de récupérer les données dans la base de données.
- Vue : représente la partie HML5, CSS3 et (JavaScript)
- Contrôleur : la logique qui contient les actions que l'utilisateur peut effectuer ainsi que les routes correspondantes.

En réalité, pour plus de clarté, on a décidé de diviser en plusieurs dossiers les éléments qui font partie du contrôleur et de la vue.

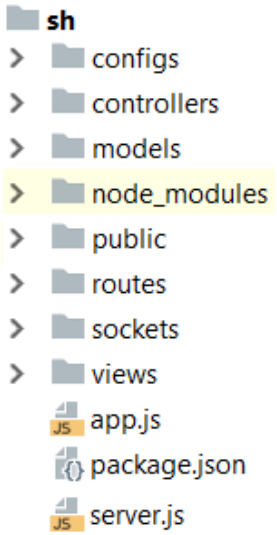


Figure 15 : Arborescence du projet de l'application web Smarthepia

Dans l'arborescence de l'application web, à la racine, on retrouve ainsi le répertoire sh. Puis, pour démarrer l'application web, on a besoin de quatre éléments : le fichier *server.js* contenant les déclarations qui permettent l'initialisation du serveur web ; le fichier *app.js* qui comporte les définitions de l'application web c'est-à-dire, les routes ainsi qui seront utilisées ; le dossier *node_module* regroupant

tous les modules (librairie) installés pour Nodejs ; et enfin, le fichier *package.json* qui contient la liste des modules ajoutés en plus de ceux déjà présents à l'installation de Nodejs.

Comparaison de cette arborescence avec le modèle MVC :

- Pour la partie modèle, on retrouve le dossier modèle qui contient la structure des modèles utilisés pour récupérer les informations de la base de données.
- Pour la partie vue, il s'agit du dossier *views* regroupant les *templates* qui seront visibles à l'utilisateur ainsi que le répertoire *public* contenant tous les fichiers CSS3 et JavaScript.
- Quant à la partie contrôleur, elle contient trois dossiers. Le premier se nomme « *socket* » et contient l'initialisation de même que les fonctions des *websockets*. Le second dossier – « *routes* » – contient les routes appelées lors d'une requête de l'utilisateur. Ces routes contiennent d'autre part les fonctions présentes dans les fichiers du dossier *controlleur*, comme par exemple l'authentification d'un utilisateur ou la gestion d'erreur.

6.4.2 Permissions

Chacune des pages du dossier *routes* (Figure 16) – dont on donnera une description plus détaillée par la suite (cf. 6.4.6) – correspond à une ou plusieurs fonctionnalités pour la gestion du *Smarthepia*. Il est donc important que ces pages soient correctement protégées afin que pas n'importe quel utilisateur puisse y accéder.

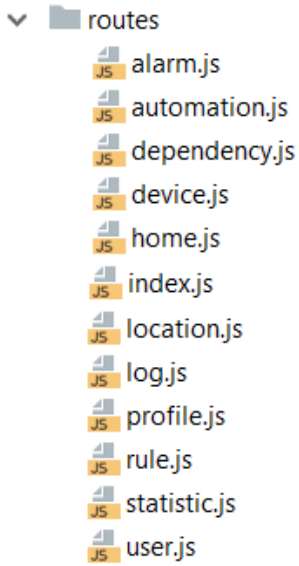


Figure 16 : Pages application web

Pour remédier à ce problème, on a mis en place un système de permissions en définissant trois utilisateurs types :

- Utilisateur basique (BU) : permission de niveau 1
- Gestionnaire ayant accès aux éléments du *Smarthepia* (G) : permission de niveau 2
- Administrateur ayant accès à tout le système (A) : permission de niveau 3

Répartition des permissions par pages (Figure 16) :

Page	BU	G	A
Alarm		X	X
Automation		X	X
Dependency		X	X
Device		X	X
Home	X	X	X
Index	X	X	X
Location		X	X
Log		X	X
Profile	X	X	X
Rule		X	X
Statistic	X	X	X
User		X	X

Table 20 : Permission par page

6.4.3 Authentification (Passport), sessions et sécurité

Pour authentifier les utilisateurs, on a utilisé un module qui s'appelle « Passeport ». Ce type de module est implémenté pour utiliser des stratégies existantes et ainsi, simplifier et sécuriser l'authentification d'un utilisateur sur un serveur. Dans le cas de *Smarthepia*, on utilise la stratégie *Local* qui consiste à récupérer les identifiants que l'utilisateur a posté sur la page de login lors de son authentification. C'est une stratégie basique mais néanmoins efficace. D'ailleurs, elle reste une méthode encore fréquemment utilisée par les sites internet actuels.

D'autre part, le module *Passport* permet aussi d'autres types de stratégies d'authentification – non locales – dont celle de Google (OpenID/OAuth 2.0). Toutefois, on a décidé de ne pas mettre en place ce type de stratégies car le *Smarthepia* est destiné à être utilisé « localement », c'est-à-dire que seules les personnes faisant partie de l'hepia auront accès à cette application web.

Une fois que l'utilisateur s'est correctement authentifié, *Passport* génère donc un identifiant qui permet par la suite de déterminer si celui-ci a le droit d'accéder à l'application web. Cette identifiant est rattaché à la session de l'utilisateur de cette manière :

```
req.session.passport.user = {id:'xyz'}
```

Figure 17 : Identifiant généré par Passport qui sera attaché à la variable de session user

Les variables de session, telles que l'identifiant généré par le module *Passport*, sont ensuite stockées dans la base de données MongoDB, dans la collection « sessions ». Cela évite de garder les sessions en RAM dans le cas d'un nombre trop important d'utilisateurs connectés. En outre, cela permet également de maintenir les sessions « persistante » dans le cas de problèmes sur le serveur.

6.4.3.1 Certificat

Pour des raisons évidentes de sécurité, nous avons décidé de chiffrer la connexion entre le client et le serveur grâce à un certificat utilisé par SSL permettant de chiffrer les données transmises. Cependant, l'application web n'étant pas disponible sur internet mais uniquement au sein du réseau de l'hepia, il n'est pas possible de créer un certificat conventionnel validé par une autorité de certification reconnue.

C'est pour cela qu'on a créé un certificat auto-signé. Toutefois, ce dernier n'étant pas reconnu par une autorité certifiée, l'utilisateur sera obligé de valider une « erreur » à chaque première connexion via un nouveau navigateur.

Si cela n'est guère pratique pour l'utilisateur, cela permet néanmoins de garantir la sécurité des données, chose très importante car il s'agit d'un système qui gère des composants physiques (actionneurs) ne devant pas être accessibles à n'importe qui.

À partir du 08.07.2018, le certificat généré devra être renouvelé tous les ans sur le serveur :

```
$private = /etc/ssl/private/nginx-selfsigned.key
$public = /etc/ssl/certs/nginx-selfsigned.crt
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout private
-out $public
```

Table 21 : Génération certificat auto-signé

6.4.3.2 Bycrypt

Pour sécuriser les mots de passe de la base de données de *Smarthepia*, on a décidé d'employer un module utilisant des fonctions de « hashage », c'est-à-dire une méthode permettant de créer une empreinte d'un texte en clair. Il s'agit de *Bycrypt.* Cette fonction de hashage se base sur l'algorithme de chiffrement *Blowfish* et diffère quelque peu des fonctions de hashage traditionnelles. Voici en effet un mot de passe hashé par *Bycrypt* (Figure 18).

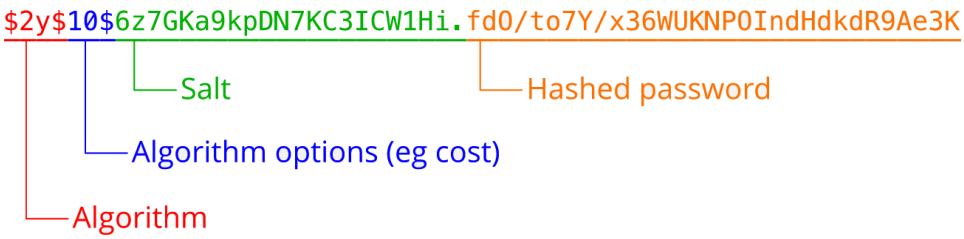


Figure 18 : Bycrypt décomposition du hash

La première suite de caractères – en rouge – correspond au type d'algorithme utilisé. En effet, si on le souhaitait, il serait possible de spécifier un autre algorithme de hashage comme MD5 ou autres. La troisième suite de caractères indique le *salt* utilisé. Celui-ci sera ajouté au mot de passe pour le rendre encore plus compliqué à déchiffrer lors d'une éventuelle attaque. En orange, on retrouve le mot de passe hashé. Finalement, la deuxième suite de caractère – inscrite en en bleu – est la partie la plus intéressante car elle assure à *Bycrypt* un haut niveau de sécurité. En effet, contrairement à une fonction de hashage traditionnelle, telle que MD5 ou SHA1 réputés pour leur rapidité de calcul, *Bycrypt* préfère privilégier la complexité du hash à la rapidité d'exécution.

Ainsi, dans le cas où la base de données serait compromise, Bycrypt, en augmentant le temps nécessaire pour générer une empreinte, multiplie également le temps pour retrouver le mot de passe en clair, lors d'une attaque par dictionnaire. Des tests (Figure 19) ont alors été effectués sur un processeur cadencé à 2Ghz pour déterminer combien de temps en moyenne, il faudrait pour calculer un hash.

```
rounds=8 : ~40 hashes/sec
rounds=9 : ~20 hashes/sec
rounds=10: ~10 hashes/sec
rounds=11: ~5  hashes/sec
rounds=12: 2-3 hashes/sec
rounds=13: ~1 sec/hash
rounds=14: ~1.5 sec/hash
rounds=15: ~3 sec/hash
rounds=25: ~1 hour/hash
rounds=31: 2-3 days/hash
```

Figure 19 : Temps pour générer un hash avec Bycrypt en fonction du nombre de rounds

Afin d'éviter un temps d'attente trop long et donc gênant pour l'utilisateur, *Bycrypt* propose également une autre fonction permettant de calculer le nombre optimal de *rounds* pour la machine sur laquelle est exécuté l'algorithme. Dans le cas de *Smarthepia*, la fonction retourne 13 *rounds*.

6.4.4 Websocket

Certaines fonctionnalités de l'application nécessitent une connexion bidirectionnelle entre le client et le serveur. C'est pourquoi on a décidé d'utiliser les *websockets* qui offrent cette possibilité tout en garantissant la compatibilité avec les navigateurs actuels. En effet, pour établir cette connexion le client envoie une requête HTTP de type UPGRADE au serveur pour notifier qu'il souhaite ouvrir une connexion *websocket* avec lui. Si le serveur l'accepte, alors une connexion est établie. Cette connexion est persistante et basée sur le protocole TCP jusqu'à qu'un des deux acteurs décide de la fermer.

Le client et le serveur utilisent la librairie *socker.io* pour communiquer. Elle offre effectivement plusieurs fonctions qui permettent de simplifier l'envoi de messages :

1. La fonction *on.connection* permet de savoir à quel moment une connexion est établie entre le client et le serveur et on.*disconnect* quand cette connexion est rompue.
2. La fonction *emit* permet d'envoyer un message entre le client et le serveur
3. La fonction *broadcast.emit* permet d'envoyer un message à tous les clients qui sont connectés au serveur.
4. La fonction *on* suivie d'un identifiant de message permet de savoir qu'un message qui correspond à cet identifiant a été envoyé.

6.4.5 Requête Ajax

Pour rendre l'application web dynamique, c'est-à-dire lorsque l'utilisateur valide, met à jour, supprime ou sélectionne un objet, il ne faut pas qu'il soit obligé d'actualiser la page manuellement pour voir les changements. Pour remédier à ce problème, il existe en Javascript une librairie nommée « Jquery » qui implémente la possibilité d'effectuer des requête HTTP asynchrones. Cette fonctionnalité est connue sous le nom d' « Ajax » et permet, du point de vue de l'utilisateur, de rendre « transparente » l'interaction entre le client et le serveur. Ainsi, pour l'application web, dès qu'on devra effectuer une requête sur le serveur web, cette fonctionnalité sera employée. Cependant, lors de la description des fonctionnalités de l'application web, on utilisera l'appellation requête ou requête HTTP sans préciser qu'il s'agit d'une requête Ajax asynchrone.

6.4.6 Fonctionnalités

Dans cette section, il sera question de décrire les fonctionnalités de l'application web. Cette description se fera dans l'ordre où ces fonctionnalités ont été développées, tout en indiquant à quelles pages de l'application (Figure 16) elles correspondent :

- Gestion des profils utilisateur (*profile*)
- Gestion des utilisateurs (*user*)
- Gestion des dépendances (*dependency*)
- Gestion des emplacements (*location*)
- Gestion des multi-senseurs, actionneurs (*device*)
- Affichage de l'état du réseau (*home*)
- Gestion des règles (*rule*)
- Option d'automatisation (*automation*)
- Gestion des alarmes (*alarm*)
- Gestion des logs (*log*)
- Affichage des statistiques (*stat*)

La partie « gestion du profil de l'utilisateur » permet à ce dernier de modifier ses informations personnelles. Elle comporte deux options : premièrement, la possibilité de changer d'adresse email, de nom et de prénom ; deuxièmement, celle de modifier le mot de passe. Cependant, pour des raisons de sécurité, quand l'utilisateur souhaite changer son mot de passe, il lui faut introduire son mot de passe actuel afin d'éviter qu'une tierce personne puisse le changer, notamment dans le cas où il aurait oublié de se déconnecter.

La gestion des utilisateurs, disponible seulement après permission de l'administrateur, se divise en deux parties : la première est représentée par l'affichage – dans un tableau – des utilisateurs existants, ceux-ci pouvant être supprimés ou modifiés. La deuxième est utilisée pour l'ajout d'utilisateurs. Lors de l'ajout d'un utilisateur, il est possible de l'ajouter en mode « désactivé » de sorte qu'il ne puisse pas s'authentifier directement après l'ajout. C'est aussi dans cette partie que l'on doit définir les permissions d'un utilisateur. Pour faire économiser du temps à l'administrateur au moment de l'ajout des utilisateurs, l'adresse email servira de « nom d'utilisateur » lors de toute authentification sur l'application web.

Les dépendances représentent les éléments qui font le lien entre les multi-senseurs ou actionneurs et le *Smarthepia*.

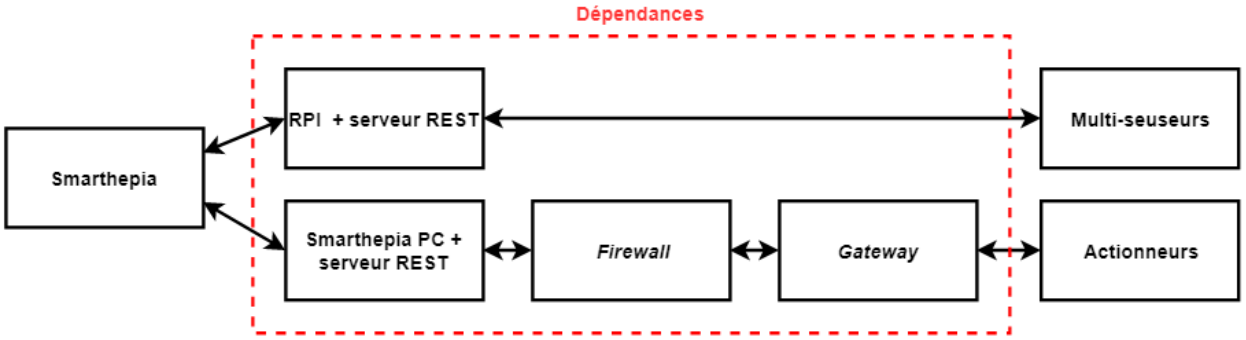


Figure 20 : Dépendances des multi-senseur et actionneurs

Il est important de savoir que, si l'un des composants d'une de ces dépendances venait à ne plus fonctionner, il est impossible d'actionner ou de récupérer les mesures des multi-senseurs. Pour contrôler qu'un composant dans la dépendance fonctionne correctement, il faut vérifier deux choses. La première : s'il s'agit d'un composant physique ou d'un serveur REST/HTTP. La deuxième : quelle est son port et son adresse IPV6 ou IPV4. S'il ne fonctionne pas, une alarme sera alors générée.

La gestion des emplacements des multi-senseurs ainsi que des actionneurs a été mise en place pour créer une hiérarchie cohérente (bâtiment, étage, salle et élément). En d'autres termes, pour pouvoir ajouter un multi-senseur ou un actionneur, il faudra auparavant ajouter et spécifier où il se situe dans le bâtiment, dans l'étage et finalement dans la salle. Cette hiérarchisation a également pour objectif de rendre la gestion d'un élément dynamique. C'est-à-dire qu'à la création d'un multi-senseur ou d'un actionneur, il est possible de le positionner où on le souhaite dans le bâtiment.

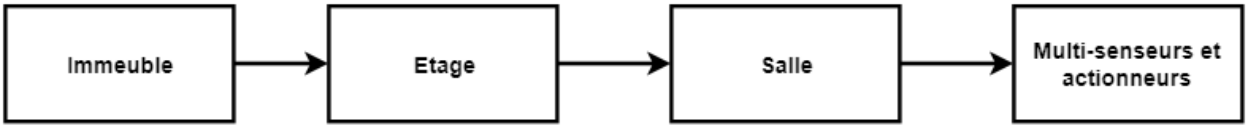


Table 22 : Emplacement schéma

Enfin, cette hiérarchisation a été créé pour afficher de l'état du réseau. En effet, lorsqu'un utilisateur s'authentifie sur l'application, il est redirigé sur la page d'accueil. Sur cette page, seule la vue des bâtiments – représentés par des rectangles – est visible dans un premier temps car c'est le premier composant dans la hiérarchie. Puis, pour naviguer dans la hiérarchie, il lui faut cliquer sur les rectangles et ainsi, passer au niveau inférieur.

Si on constate un problème sur le réseau alors, le rectangle change automatiquement de couleur (Figure 24) :

- Vert : il n'y a pas d'erreur
- Orange : il y a une erreur mais le système est encore en fonction
- Rouge : une ou plusieurs fonctionnalités n'assurent plus le bon fonctionnement du réseau *Smarthepia*.

En plus de la couleur indiquant l'élément dysfonctionnant – multi-senseur, actionneur ou dépendance –, un encadré en gris vient spécifier le nom du ou des éléments qui en sont la cause. Cela permet ainsi de voir rapidement d'où vient le problème sans avoir à naviguer dans toute la hiérarchie.

Figure 21 : Hiérarchie du réseau (bâtiment)

Quand un composant rencontre un problème, comme dans ce cas (Figure 22) un multi-senseur, on peut constater que la couleur – rouge – est transmise jusqu'en haut de la hiérarchie (bâtiment). Suivant cette logique et en prenant en compte les couleurs qui définissent l'état du réseau, si deux éléments (un orange et un rouge) de la même salle ou du même étage tombent en panne simultanément, seule la couleur de l'élément dont la gravité est la plus élevée apparaitra au niveau supérieur de la hiérarchie.

Figure 22 : Hiérarchie du réseau (salle)

Les règles et les options d'automatisation sont des parties importantes de ce projet car la première indique de quelle manière les actionneurs sont utilisés tandis que la seconde permet de voir comment le programme d'automatisation se comporte. Pour des raisons de logique et de compréhension, ces deux thématiques seront reprises et décrites par la suite dans la partie « automatisation » (cf. 6.6.7).

La partie « gestion des alarmes » est un autre point important du projet. En effet, une alarme permet d'avertir l'administrateur ou le manager de l'état d'un élément du *Smarthepia*. Pour cela, trois couleurs ont été utilisé :

- Bleu : information sur un élément sans gravité
- Orange : détection d'une erreur mais le système est encore en fonction
- Rouge : une ou plusieurs fonctionnalités n'assurent plus le bon fonctionnement du réseau *Smarthepia*

Pour notifier à l'utilisateur qu'une nouvelle alarme est apparue, on a décidé d'utiliser un composant qu'on appellera « badge ». Celui-ci se situe dans la barre de navigation de l'application web, à droite.

Figure 23 : Barre de navigation

Ce badge permet ainsi d'indiquer le nombre d'alarmes en cours ainsi que leur gravité grâce à l'échelle couleur décrite ci-dessus. Il se retrouve dans toutes les pages de l'application web ce qui évite à l'utilisateur de devoir aller exprès dans la rubrique « alarme » pour consulter ses alarmes.

Pour actualiser ce badge, il a fallu trouver une méthode pour que le client ne soit pas obligé de faire du *polling*, c'est-à-dire interroger la base de données à intervalle régulière pour savoir si une nouvelle alarme est apparue. Le problème du *polling* est le suivant : si cet intervalle est trop important, l'alarme n'arrivera pas en temps réel mais en différé. Par conséquent, pour effectuer cette opération de manière optimale, voici les étapes que l'on a suivies :

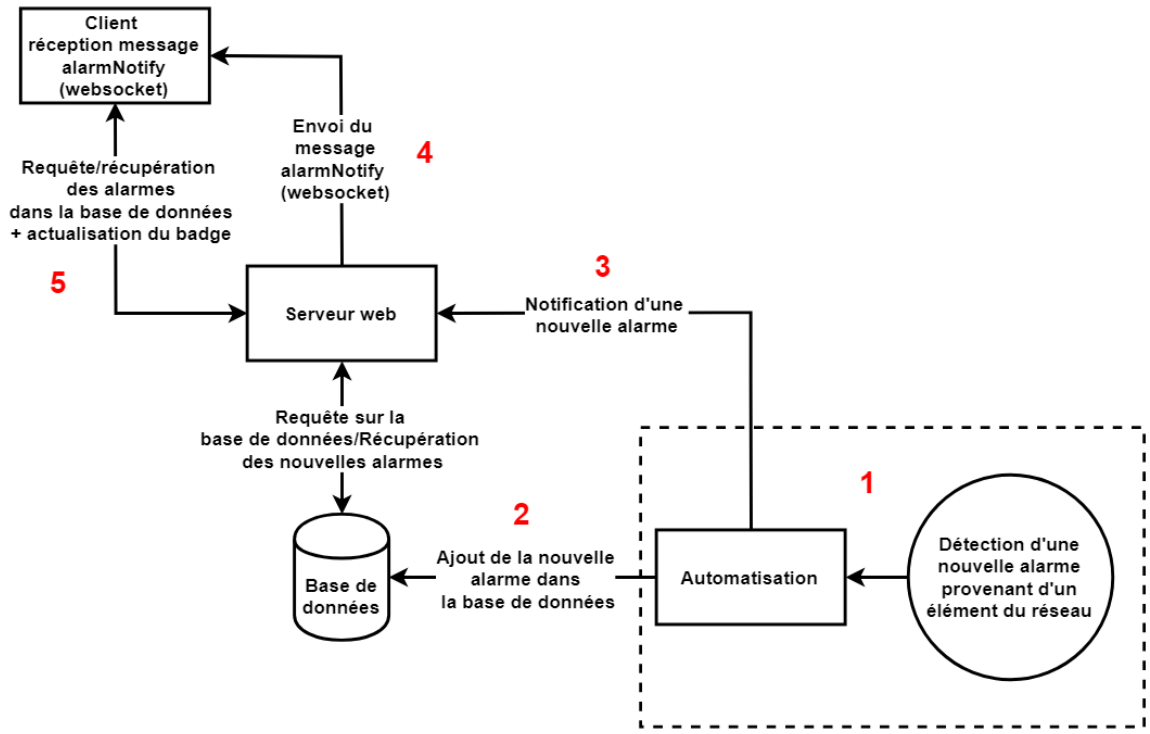


Figure 24 : Actualisation des alarmes

1. Réception et traitement d'une nouvelle alarme – venant par exemple d'un multi-senseurs – par le programme d'automatisation
2. Envoi de cette alarme dans la base de données. Cependant, si une alarme du même type et concernant le même élément est déjà apparue, alors on fait une mise à jour afin de pouvoir incrémenter le nouveau nombre d'alarme ainsi que la dernière date enregistrée. Cela évite en effet qu'un nombre trop important d'alarmes du même type se retrouvent dans la base de données. Voici comment la mise à jour d'une alarme déjà existante est affichée pour l'utilisateur dans l'application web :

Figure 25 : Visualisation des alarmes en fonction de leur apparition

3. On fait ensuite une requête de type HTTP GET sur le serveur web pour lui indiquer qu'une nouvelle alarme est apparue. Pour effectuer cette opération, on a créé un utilisateur caché (notify@gmail.com) qui va permettre de s'authentifier sur l'application et faire la requête.
4. Le serveur web reçoit alors la requête qui notifie la nouvelle alarme puis, il envoie un message (*alarmNotify*) à tous les clients connectés à l'application web, grâce à un *websocket,* qu'une nouvelle alarme est apparue.
5. Enfin, le client reçoit un message (*alarmNotify*) et va récupérer la liste des toutes les nouvelles alarmes.

La partie « gestion des alarmes » rassemble encore deux fonctionnalités supplémentaires : assigner une alarme à un utilisateur spécifique et « acquitter » une alarme. On a regroupé ces deux fonctionnalités sur le même schéma car, à part le type de notification, elles utilisent le même principe (Figure 26).

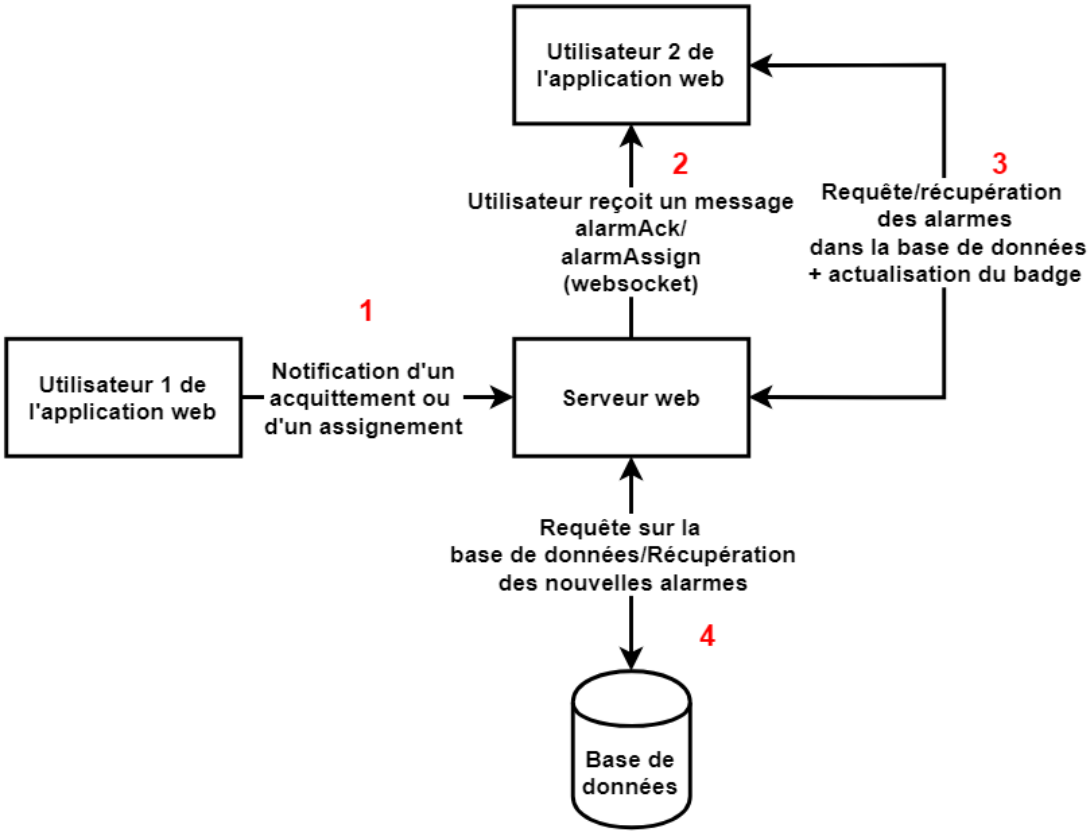


Figure 26 : Assignation et acquittement d'une alarme schéma

Lorsqu'une nouvelle alarme arrive sur l'interface utilisateur, tous les administrateurs et gestionnaires reçoivent cette alarme. Si un des utilisateurs souhaite assigner une de ces alarmes – par exemple, un niveau faible de batterie d'un multi-senseur – à un autre utilisateur en particulier, il peut le faire grâce à la fonctionnalité que l'on a implémentée. Ainsi, l'alarme assignée sera supprimée de la liste des autres utilisateurs.

1. Lors d'un assignement d'une alarme par un utilisateur, une requête est envoyée au serveur web avec l'identifiant de l'alarme. Cette identifiant va permettre de trouver l'alarme en question dans la collection *alarm* de la base de données et ainsi, mettre à jour un champ *assign* correspondant à *anyone*, dans le cas où elle est attribuée à tout le monde, ou à une adresse email d'un utilisateur en particulier.
2. À la suite de ça, tous les utilisateurs connectés au server par un *websocket* vont être avertis par un message (*alarmAssign*) qu'une nouvelle alarme a été assignée.
3. À la suite de cette notification, chaque client effectue une requête sur le serveur web pour demander la nouvelle liste des alarmes.
4. Le serveur web retourne alors cette nouvelle liste au client qui met à jour le badge ainsi que la table listant toutes les alarmes.

Concernant l'acquittement d'une alarme, voici maintenant comment on a procédé. Lorsqu'une nouvelle alarme survient, l'utilisateur prend connaissance de cette alarme mais souhaite la faire disparaître car le problème est désormais résolu. Pour ce faire, on a décidé d'implémenter la possibilité pour un utilisateur de pouvoir acquitter une ou plusieurs alarmes.

1. Lors d'un acquittement d'une alarme par l'utilisateur, une requête est envoyée au serveur web avec l'identifiant de l'alarme. Cette identifiant va permettre de trouver l'alarme en question dans la collection *alarm* de la base de données et ainsi mettre à jour un champ *ack* correspondant à 1 pour « acquitté » et 0 pour « non acquitté ».
2. À la suite de ça, tous les utilisateurs connectés au server par un *websocket* vont être avertis par un message (*alarmAck*) qu'une nouvelle alarme a été acquittée.
3. À la suite de cette notification, chaque client effectue une requête sur le serveur web pour demander la nouvelle liste des alarmes.
4. Le serveur web retourne alors cette nouvelle liste au client qui met à jour le badge ainsi que la table listant toutes les alarmes.

Quand une alarme est acquittée, elle est alors définie comme « log ». On peut alors retrouver le nom d'utilisateur de la personne qui l'a acquitté ; la date et l'heure de la première fois où elle est apparue ; et, le moment où elle a été acquittée. Ces champs pourraient permettre par la suite de pouvoir identifier différents problèmes qui survendraient de manière régulière.

L'option « statistique » permet à un utilisateur de visualiser plusieurs types de statistiques : les mesures effectuées toutes les cinq minutes pour chaque multi-senseur ; les données métrologiques relevées toutes les deux heures ; finalement, les commandes effectuées sur les actionneurs comme le positionnement d'une vanne de radiateur ou celle d'un store. Toutes ces mesures sont en outre disponibles par salles en fonction d'une date et d'une heure de début et de fin.

6.4.7 Dépendance de suppression

Toutes les fonctionnalités de gestion de l'application web, hormis la gestion des utilisateurs et des alarmes, sont dépendantes. C'est-à-dire que ces fonctionnalités de gestion permettent de créer ou supprimer un élément mais en fonction du niveau de dépendance de l'élément, ce dernier sera plus ou moins relié à un ou plusieurs autres éléments. Par exemple, lorsqu'on créé un multi-senseur ou un actionneur, celui-ci est lié à une dépendance. Or, si on souhaite supprimer cette dépendance, il faudra alors supprimer tous les éléments qui lui sont associés. Ce choix été fait dans le but de protéger la suppression automatique en cascade des éléments. Voici un schéma de principe mettant en évidence les dépendances des fonctionnalités de l'application web, de la plus forte (située en haut) à la plus faible (située en bas) :

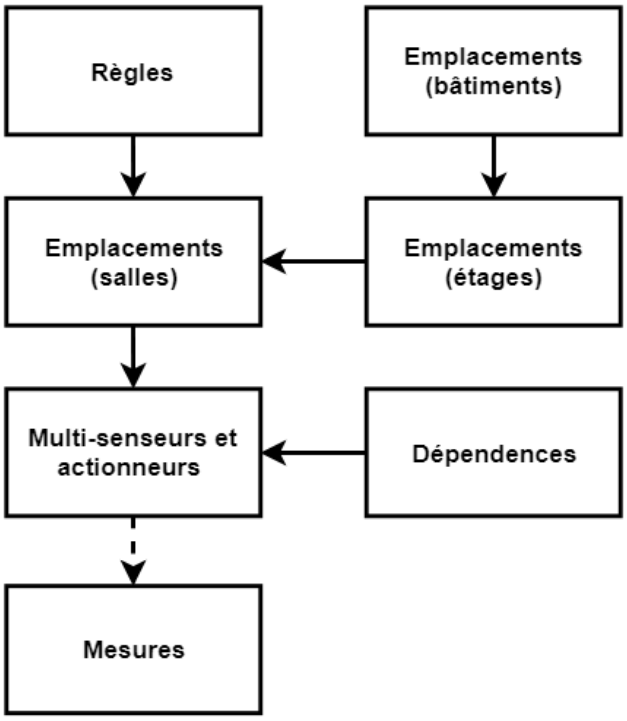


Figure 27 : Schéma dépendance de suppression

Les seuls éléments supprimés en cascade en même temps que les multi-senseurs et actionneurs, sont les mesures.

6.5 Conception de l'application d'automatisation

L'application d'automatisation a été développée dans le but gérer les différents composant de *Smarthepia* puis, de pouvoir automatiser ses actionneurs. Cependant, avant de parler concrètement de la réalisation de cette application, on a jugé important de donner quelques précisions quant à ce processus d'automatisation et aux recherches qui ont été nécessaires à son développement.

6.5.1.1 API météo

Pour les besoins de cette application, il est premièrement nécessaire de récupérer les conditions météorologiques courantes ainsi que les prévisions du jour suivant, au moins. Pour ce faire, on a trouvé une API gratuite (600 appels/heure) permettant de fournir ces données (10). Pour pouvoir accéder à ces données, il faut toutefois s'être auparavant enregistré et avoir généré une clé d'API. Ces données sont ensuite sauvegardées, au format JSON, dans deux collections de la base de données afin qu'elles puissent être réutilisées :

- Collection (*apicurrents*) : conditions météorologiques courantes
- Collection (*apiforecast*) : prévisions sur 3 et 5 jours

Dans la version gratuite, les données sont actualisées toutes les deux heures. Par conséquent, dans le cas où survient un changement brutal de température ou de temps, on ne pourra pas être averti rapidement de ces modifications.

Voici un exemple de récupération des conditions météorologiques actuelles selon les coordonnées longitude et latitude de l'hepia :

```
latitude = 46.20949
longitude = 6.135212
https://samples.openweathermap.org/data/2.5/weather?lat={latitude}
&lon={longitude}&appid={clé d'API}
```

Table 23 : Requête HTTP GET pour récupérer les conditions météorologiques actuel

6.5.1.2 Module PID

Pour réguler la température dans les salles de classe, on a décidé d'utiliser un module PID (11) permettant d'améliorer les performances d'un asservissement en corrigeant les écarts trop importants autour de la température idéale (consigne). En d'autres termes, ce de dernier permet de calculer un signal de commande à partir de la différence entre la consigne et la mesure. En effet, l'utilisation d'une autre méthode – telle que celle appelée « tout-ou-rien » – engendrerait un phénomène d'oscillation autour de la température de consigne pouvant être dommageable pour le matériel.

Pour calculer un signal de commande puis le corriger, le module PID prend en compte trois paramètres :

- *Kp* : proportionnel
- *Ki* : intégrale
- *Kd* : dérivé

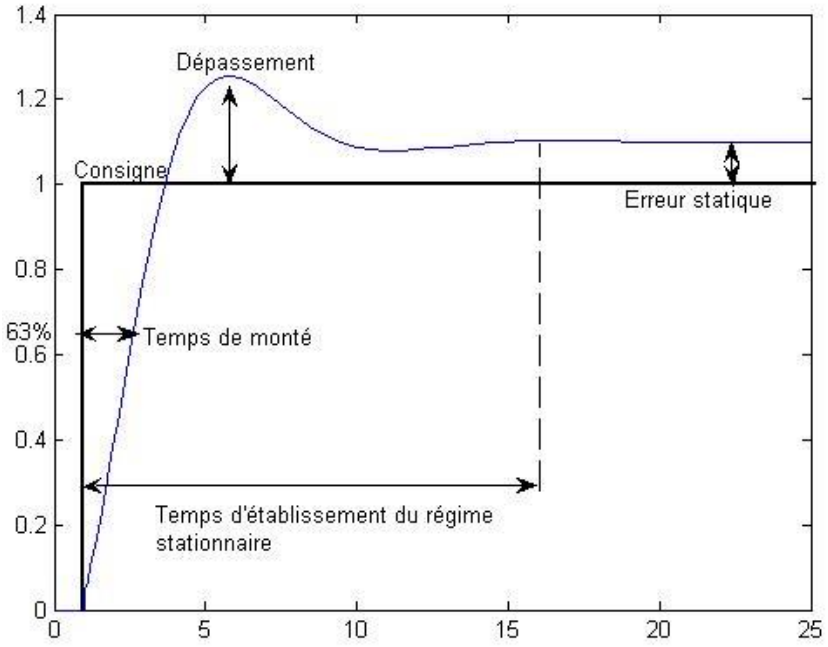


Table 24 : Régulation PID influence

Ces trois paramètres influencent en effet le temps d'établissement d'un régime stationnaire – qu'on souhaite obtenir rapidement –, l'erreur statistique, le temps de montée et le dépassement.

Pour une consigne fixée à 1, voici comment les paramètres *Kp*, *Ki* et *Kd* influent sur le signal de commande en sortie du module PID :

- Lorsqu'on augmente *Kp*, le temps de montée est plus court et l'erreur statique est réduite mais il provoque alors un dépassement plus important.

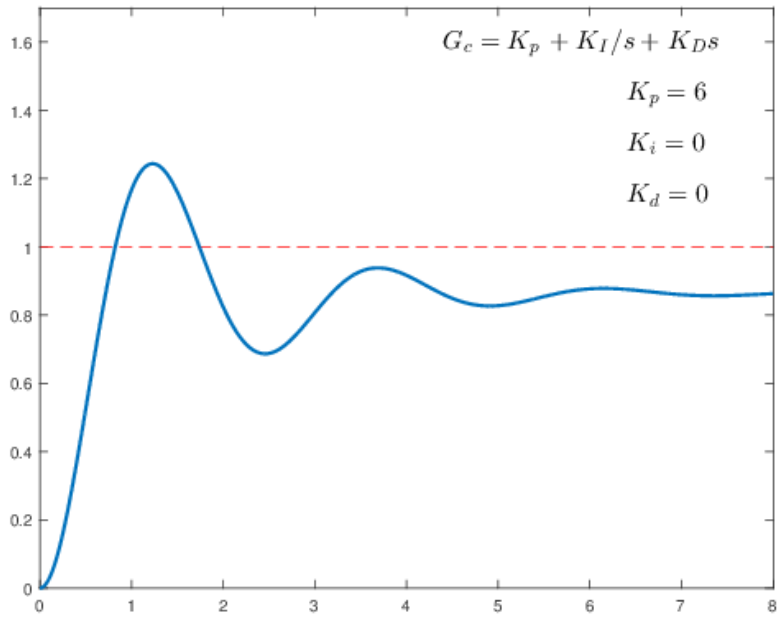


Figure 28 : Augmentation de la valeur du paramètre Kp d'un module PID

- Lorsqu'on augmente *Ki*, la valeur de consigne est rapidement atteinte mais le temps d'établissement en régime stationnaire s'allonge.

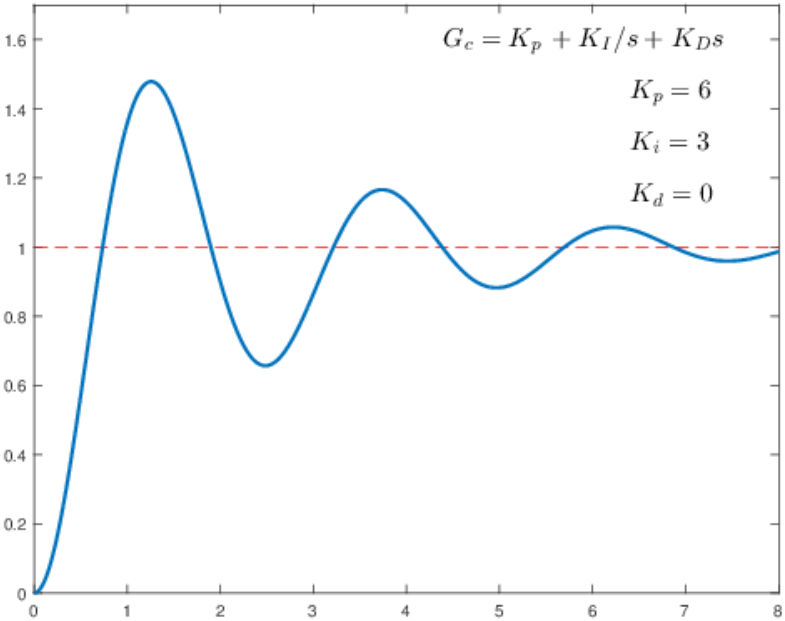


Figure 29 : Augmentation de la valeur du paramètre Ki d'un module PID

- Lorsqu'on augmente *Kd*, le temps de montée diminue – à condition qu'une action proportionnelle soit associée – et le dépassement diminue tout en ne n'ayant aucune influence sur l'erreur statistique.

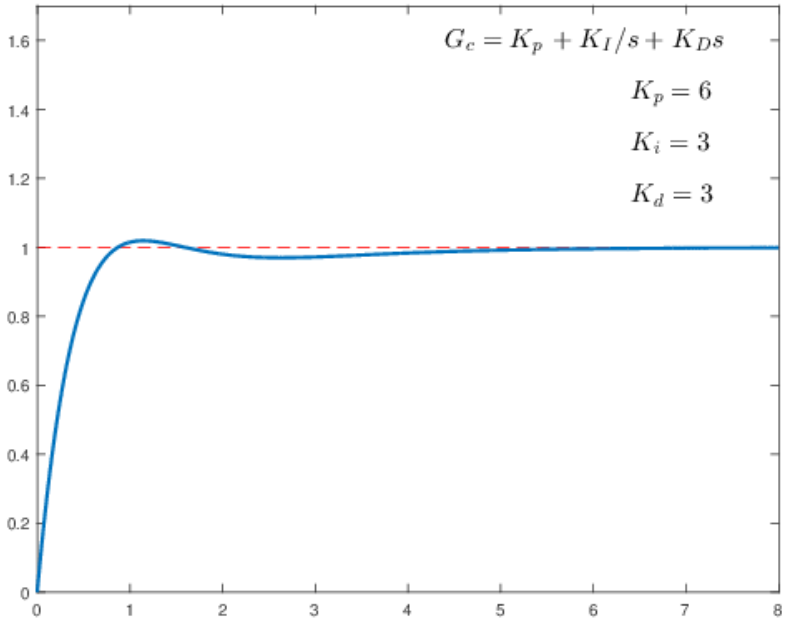


Figure 30 : Augmentation de la valeur du paramètre Kd d'un module PID

Voici un tableau récapitulatif qui met en avant l'influence de l'augmentation respective de ces trois paramètres.

	Précision	**Stabilité**	**Rapidité**
Kp	↗	↘	↗
Ki	↗	↘	↘
Kd	↘	↗	↗

Table 25 : Récapitulatif de l'influence de l'action des paramètres Kp, Ki et Kd

Ce module PID permet de définir la valeur minimum et maximum du signal de commande pour que ces correspondent à celles des actionneurs de vannes de radiateurs (0-255). De plus, il est possible aussi de modifier pendant l'exécution les paramètre *Kp*, *Ki*, *Kd* ainsi que la consigne.

Pour résumer, comme on n'a pas de données de comparaison pour le bâtiment de l'hepia, on a décidé de mettre les valeurs suivantes pour la régulation. Ces valeurs pourront être modifiées par la suite, après avoir étudié les courbes de température et de commande définies par la consigne :

- *Kp* : 6
- *Ki* : 3
- *Kd* : 3

6.5.1.3 Pénétration du soleil dans une salle

Pour déterminer si le soleil pénètre dans une salle, on a utilisé la librairie « Pysolar » (12). Cette librairie implémente deux fonctions :

- L'azimute qui équivaut à un angle entre 0 et 360 degrés par rapport à l'axe vertical, sachant que 0/360 correspond au nord.
- L'élévation qui représente un angle entre 90 et -90 degrés par rapport à l'axe horizontal

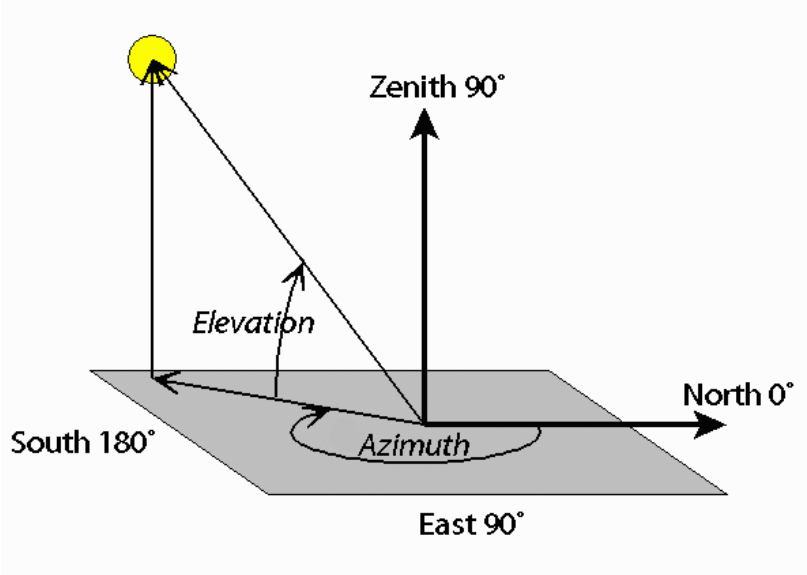


Figure 31 : Azimute et élévation par rapport à un point

Pour calculer l'azimute ou l'élévation, on a besoin de connaître plusieurs éléments :

- Une coordonné longitudinale
- Une coordonné latitudinale
- La date et l'heure

Par conséquent, pour déterminer si le soleil pénètre dans une salle, il faut se placer en face des fenêtres de la classe et calculer l'angle avec une boussole posée à l'horizontal. Puis, en fonction de cet angle, on définit une ouverture horizontale et verticale minimum et maximum.

Concernant l'ouverture verticale, on s'est basé sur deux choses, l'élévation maximum du soleil – soit le 21 juin correspondant au solstice d'été – et son élévation minimum définie par rapport aux montagnes.

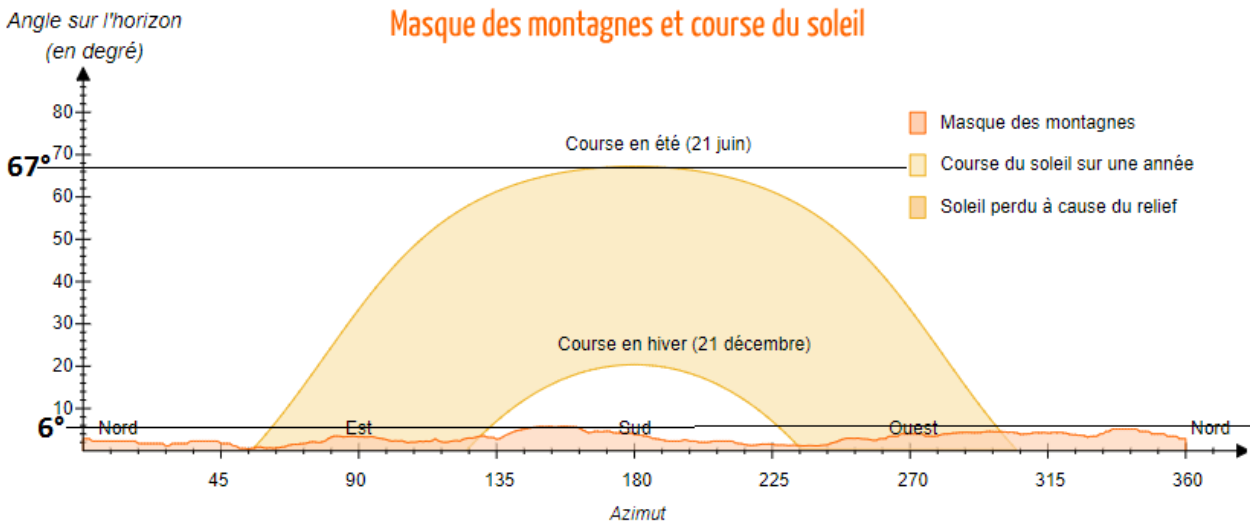


Figure 32 : Angle élévation maximum et minimum

Quant à l'ouverture horizontale, on a simplement calculé, avec un rapporteur, l'angle critique au-delà duquel, le soleil pénètre de manière trop importante dans la pièce.

Grace à la combinaison des deux ouvertures et de la position exacte du soleil (azimute et l'élévation), on peut alors déterminer si le soleil entre dans la salle ou non.

D'autre part, le calcul des obstacles autres que les montages n'a pas été pris en compte dans le calcul de pénétration du soleil dans la salle car si on change de bâtiment, il est probable que les obstacles aux alentours changent également et ainsi, que le calcul soit inexact.

6.5.1.4 Vérification de la température des multi-senseurs

Pour s'assurer que la température des multi-senseur corresponde bien à la température réelle de la salle, on a dû effectuer plusieurs tests. Le premier consiste à vérifier si cette température est comprise dans la plage minimum et maximum (*threshold*) définie par l'utilisateur. Le deuxième vise à récupérer tous les multi-senseurs disponibles dans la salle et à corréler leurs températures. Pour cela, on calcule la médiane de toutes les températures. Cette opération a pour objectif de ramener une température trop élève ou trop basse (dans les extrêmes) à celle qui semble être la plus correcte. Finalement, on vérifie si la dernière mise à jour de ces mesures n'est pas antérieure à deux heures, ce qui indiquerait en effet un mauvais fonctionnement du multi-senseur ou du réseau auquel cas, on ne prendrait pas en compte ces mesures pour vérifier l'état de du système.

6.5.1.5 Vérification de la présence des multi-senseurs

La présence est un facteur déterminant de l'automatisation des actionneurs car, dans le cas où il y aurait quelqu'un dans une salle alors, on n'effectue pas de commande sur les actionneurs (cf 6.6.7). Ce sont les multi-senseurs qui, toutes les quatre minutes, permettent de détecter une éventuelle présence.

Pour s'assurer que la présence soit correctement détectée, on prend les quatre dernières valeurs qui se trouvent dans la collection (*statistics*) base de données. Elles correspondent aux quatre dernières valeurs – soit environ vingt minute – des mesures récupérées par le processus (*measure*). À la suite de ça, on récupère l'heure et la date à laquelle la mesure a été ajoutée. Si cette valeur est plus petite que l'heure actuelle moins vingt-cinq minutes, on en déduit que le multi-senseur n'est pas à jour et par conséquent,

on ne le prend pas pour vérifier l'état du système. Si en revanche il est à jour, on récupère cette fois-ci les mesure de présence et on les compare entre elles. Si l'une de ces mesures indique une présence détectée alors, on le notifie.

6.6 Réalisation de l'application d'automatisation

L'objectif de cette application d'automatisation est de créer un système qui va automatiser des tâches à intervalle régulier afin de rendre le système *Smarthepia* autonome. Pour réaliser cette application, plusieurs fonctionnalités ont été ainsi développées :

- Récupération des mesures des multi-senseurs (*measure*)
- Vérification de la disponibilité des multi-senseurs, des actionneurs, des dépendances et des traitements des alarmes (*alarm*)
- Vérification du statut des éléments du *Smarthepia (status_notifier)*
- Automatisation des multi-senseurs et des actionneurs (*automation*)

Pour mettre en place ces quatre fonctionnalités, on a décidé de les séparer de manière bien distincte, en utilisant quatre processus différents. L'avantage de lancer des processus différents est que, s'il y a un problème avec l'un ou plusieurs des processus, ceux restants ne sont pas affectés. Les quatre processus sont lancés depuis la fonction principale qui est elle-même lancée seulement une fois par l'utilisateur.

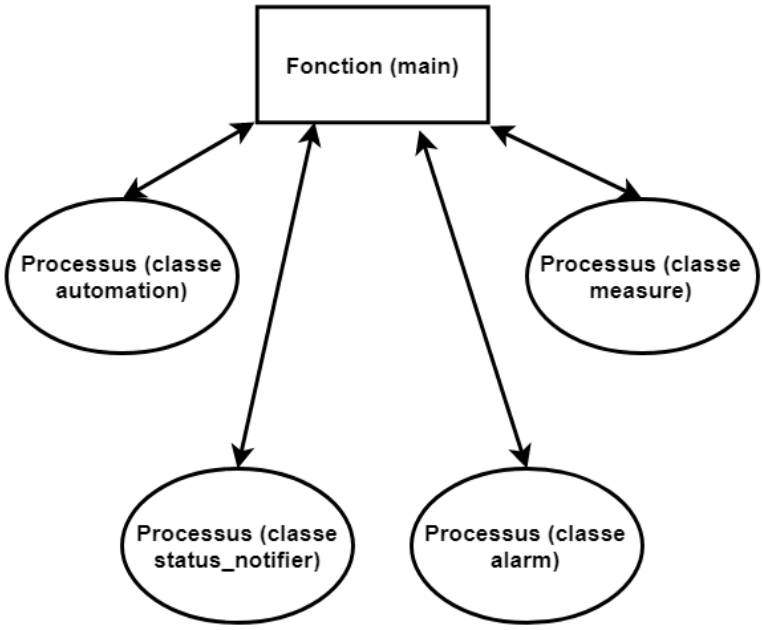


Figure 33 : Processus programme d'automatisation

6.6.1 Structure

Pour une application de ce type, il est important d'établir dès le départ une structure cohérente pour ne pas rencontrer, par la suite, des problèmes dans son développement. Pour ce faire, l'application a été divisée en fichiers de classe pour les grandes fonctionnalités – comme les processus mentionnés dans la section précèdent (cf. 6.6), les logs, la régulation PID – et pour le reste des fonctions, elles ont été séparées dans des fichiers nommés en fonction de leur utilité.

Voici comment a été structurée l'application par classes en fonction des noms de fichiers :

- Classe (*alarm*)
- Classe (*measure*)
- Classe (*status_notifier*)
    - Fichier de fonctions (*database*) : contenu d'un email en cas d'erreur de la base de données
    - Fichier de fonctions (web_*server*) : contenu d'un email en cas d'erreur du serveur web
- Classe (*automation*)
    - Fichier de classe (*heater*) : régulation grâce au module PID
    - Fichier de fonctions (*sun*) : éléments en rapport avec le soleil
    - Fichier de fonctions (*rule*) : règles sur les actionneurs
    - Fichier de fonctions (*weather*) : éléments en rapport avec la météo
    - Fichier de fonctions (*blind*) : actions sur les stores
    - Fichier de fonctions (*valve*) : actions sur les valves de radiateurs
    - Fichier de fonctions (*multisensor*) : vérification des valeurs que retournent les multi-senseurs

Ces quatre classes principales ainsi que leurs fichiers de fonctions utilisent également les fonctions et les classes suivantes :

- Fichier de classe (*logger*) : ajout d'éventuelles informations ou erreurs dans un fichier de log
- Fichier de fonctions (*utils*) : fonctions de téléchargement, d'ouverture de fichiers et autres
- Fichier de fonctions (*const*) : constantes
- Fichier de fonctions (*conf*) : appel des variables de configuration du programme
- Fichier de classes (*datastruct*) : structures d'objets

6.6.2 Fichier de configuration

Certaines variables de configuration nécessitent d'être stockées dans un autre endroit que programme d'automatisation afin que, lors d'une modification volontaire de l'utilisateur, ce dernier ne doive pas être arrêté. Pour cela, on a créé un fichier de configuration au format JSON qui contient plusieurs éléments :

- L'adresse email par défaut, dans le cas où la base de données venait à ne plus fonctionner
- Le répertoire où les logs de l'application sont enregistrés
- La taille maximum du nombre de fichiers de log à conserver

Ce fichier de configuration se nomment *conf.json* et se situent à la racine, là où le programme d'automatisation est exécuté.

6.6.3 Fichiers de log

Chacune des classes mentionnées dans la section (cf. 6.6) utilisent la classe (*logger*) permettant de sauvegarder dans un fichier indépendant toutes les erreurs ou informations qui ont été relevées lors de l'exécution. Les fichiers de logs sont constitués selon la structure suivante :

```
jour_mois_année.nom_de_la_class.smarthepialog
```

Table 26 : Structure du nom du fichier log

- Date du jour complète
- Nom du processus (classe) où le fichier de log est créé
- Extension (*smarthepialog*)

Chaque nouveau log est écrit sur une seule ligne, de la manière suivante :

```
heure:minute:seconde jour-mois-année;type;message
```

Table 27 : Structure ficher de log

Trois « types » sont existants :

- *Error*
- *Warning*
- *Info*

La structure d'un log est séparée par un point-virgule afin de pouvoir par la suite effectuer un post-traitement sur le ficher et ainsi, séparer facilement les différents éléments.

Pour ne pas encombrer le disque dur du PC qui lancera ce programme, on a défini que les fichiers de log possèdent une durée de conservation de 90 jours. C'est-à-dire qu'après avoir généré 91 fichiers, le premier ficher crée sera supprimé pour assurer une gestion des logs sur 90 jours. Néanmoins, si on souhaite modifier cette quantité de fichiers de log, il faut alors changer la valeur par défaut située dans le fichier de configuration de l'application, soit en l'augmentant, soit en la diminuant (cf. 6.6.3).

6.6.4 Processus (*alarm*)

Ce processus permet de détecter, toutes les cinq minutes, si un élément du réseau ne fonctionne pas correctement :

- Dépendances
- Multi-senseurs
- Actionneurs

Si une anomalie est décelée alors, elle est classée selon deux critères : son niveau de gravité allant de 0 à 3 – 3 étant le plus sévère – et son type, au nombre de trois :

- *Error*
- *Warning*
- *Info*

Ainsi, toutes les cinq minutes, on va premièrement récupérer, dans la base de données, tous les multi-senseurs et les actionneurs en fonction de leurs éléments de dépendance. Puis, chaque élément de la dépendance est vérifié de la manière suivante :

1. S'il s'agit d'un élément physique, comme par exemple une RPI, un PC ou une *gateway*, on effectue un *ping* sur l'élément. S'il répond, alors on détermine qu'il est fonctionnel. Si ce n'est pas le cas, c'est qu'il y a une erreur et que par conséquent, l'élément est non disponible.
2. S'il s'agit d'un élément logiciel, comme par exemple un serveur REST ou un serveur web, on effectue une requête HTTP de type GET sur l'élément. S'il répond et qu'il n'y a pas de *timeout*, alors on détermine qu'il est fonctionnel. Dans le cas contraire, cela signifie qu'il s'est produit une erreur et donc, que l'élément n'est pas disponible.

Après avoir vérifié tous les éléments la dépendance, on détermine si tous ces éléments sont disponibles. Si l'un d'eux est indisponible, alors on passe à l'éventuelle deuxième dépendance car dans ce cas, on n'est pas en mesure de vérifier les multi-senseurs ou actionneurs. Dans le cas contraire, s'ils sont tous disponibles, alors on peut commencer à vérifier les multi-senseurs ou actionneurs.

Dans un deuxième temps, on vérifie les multi-senseurs. Pour ce faire, un récupère l'adresse IP et le port de l'élément de dépendance qui correspond au serveur REST de la RPI correspondante. Puis, on effectue une requête HTTP de type GET en utilisant la route pour accéder aux mesures du multi-senseur (Table 4). S'il ne répond pas ou qu'il se produit un *timeout*, alors on détermine qu'il y a une erreur et donc, que l'élément est non disponible. Dans le cas contraire, on va récupérer ces mesures afin de déterminer l'état de la batterie ainsi que sa dernière mise à jour.

Troisièmement, on contrôle les actionneurs. Pour cela, on récupère l'adresse IP et le port de l'élément de dépendance qui correspond au serveur REST KNX. Puis, on effectue une requête HTTP de type GET en utilisant la route pour lire la valeur de l'actionneur (Table 6). S'il ne répond pas ou qu'il se produit un *timeout*, alors on détermine qu'il y a eu une erreur et que par conséquent, l'élément n'est pas disponible.

Quand tous les éléments ont été vérifiés, tous les résultats – *error, warning, info* – sont récupérés puis traités. Voici la liste des résultats possibles pour chaque élément :

Nom	Type	Gravité	Description	Message
Dépendance	*Error*	3	Un ou plusieurs éléments de la dépendance ne répondent pas	*No response*
Multi-senseur	*Error*	3	-	*Sensor not exists*
Multi-senseur	*Error*	3	-	*Wrong id*
Multi-senseur	*Error*	3	-	*Measures are not up-to-date*
Multi-senseur	*Warning*	2	-	*Battery less than 10%*
Multi-senseur	*Info*	2	-	*Battery less than 20 %*
Actionneur	*Error*	3	S'il ne s'agit pas d'un actionneur défini (cf. 4.2.4)	*Wrong actuator type*
Actionneur	*Error*	3	L'identifiant de l'actionneur n'est pas conforme (Table 1).	*Id malformed*

Table 28 : Résultat de la vérification des éléments sur le réseau Smarthepia

Cette liste représente les résultats possibles à cet instant du projet mais elle peut bien entendu être modifiée en ajoutant simplement un nouvel élément dans la liste (*self.alarm*).

Après avoir été traité, les alarmes sont envoyées à la base de données dans la collection (*alarm*) puis, le graphique correspondant à l'état actuel du réseau est modifié dans la collection (*device*). Enfin, on avertit le serveur web qu'il y a une ou plusieurs nouvelles alarmes (Figure 24).

6.6.5 Processus (*measure*)

Toujours suivant l'intervalle de cinq minutes, ce processus permet de collecter les mesures que retournent tous les multi-senseurs disponibles du *Smarthepia*. Au préalable, il faut récupérer, dans la base de données, tous les multi-senseur en fonction de leurs dépendances. Puis, pour chaque dépendance et chaque multi-senseur, on récupère leurs mesures. Si les mesures recueillies sont à jour et qu'elles n'existent pas déjà dans la base de données, alors on les ajoute. On effectue cette vérification pour ne pas surcharger la base de données de mesures inutiles.

6.6.6 Processus (*status_notifier*)

Ce processus vise à détecter – toutes les cinq minutes – si tous les éléments du *Smarthepia* fonctionnent correctement. Voici les éléments qui sont contrôlés :

- Base de données
- Serveur web
- Processus (*automation, alarm, measure*)
- Serveur REST KNX

Pour la base de données, on procède à une tentative de connexion. Si celle-ci échoue alors, on envoie un email à l'adresse de contact par défaut en notifiant l'erreur (cf. 6.6.2). Quant au serveur web, on effectue un requête HTTP de type GET. S'il ne répond pas, alors on envoie un email à tous les administrateurs du *Smarthepia* en notifiant l'erreur.

D'autre part, en ce qui concerne les processus et le serveur REST KNX, on récupère la liste des processus existants du PC sur lequel est exécuté le *Smarthepia* puis, on vérifie qu'ils soient bien présents. On contrôle également qu'ils ne soient pas arrêtés ou déclarés comme processus « zombis », ce qui indiquerait en effet qu'ils ne fonctionnent pas correctement. Suite à cette vérification, on met à jour la collection (*status*) dans la base de données en indiquant les processus disponibles ou non, ainsi que l'heure et la date à laquelle la dernière vérification a été faite. Finalement, on effectue une requête sur le serveur web qui enverra, grâce à un *websocket*, un message avec un identifiant (*status*) à tous les clients connectés. Ce message actualisera alors le statut des éléments dans la page (*home*) où est affiché ce statut. Sur cette même page, on exécute une action par le biais d'un *timer* consistant à contrôler si la dernière vérification a été faite. Si ce n'est pas le cas alors, on indique « *not up-to-date* », ce qui indique que le processus (*status_notifier*) ne fonctionne pas correctement.

6.6.7 Processus (*automation*)

Ce processus permet, suivant certaines règles, d'effectuer toutes les vingt minutes l'automatisation des actionneurs.

Afin de bien comprendre le fonctionnement de ce processus et de ces règles, il est nécessaire de décrire au préalable certaines consignes que l'on a établies. Ces consignes sont de deux types : celles qui se rapportent aux salles et celles qui s'appliquent au système d'automatisation en lui-même.

Celles qui concernent les salles permettent de spécifier les choses suivantes :

- Les cycles (périodes) :
    - Journée
    - Nuit
- La température idéale
- Humidité idéale
- Activation ou désactivation de l'automatisation de la salle
- Les règles d'automatisation des actionneurs :
    - Vanne de radiateur pendant la journée et la nuit
    - Store pendant la journée
    - Store pendant la nuit

Celles qui s'appliquent au système d'automatisation sont plus complexes car elles nécessitent de comprendre correctement certains points importants du programme d'automatisation :

- La période durant laquelle les salles sont chauffées (période de chauffe)
- Les valeurs – minima et maxima (*threshold*) – de température pour déterminer si un multi-senseur indique des mesures erronées hors ou pendant la période de chauffe
- Les valeurs minima de température, à la fois intérieure et extérieure, indiquant s'il faut ouvrir les vannes de radiateur hors de la période de chauffe
- La température maximum à laquelle on ferme les stores lorsque le soleil illumine la pièce, qu'aucune présence n'est pas détectée dans la salle, et qu'on soit pendant l'été ou le printemps
- Les paramètres – proportionnel, intégrale et dérivé – qui permettent de régler le module PID

Après avoir décrit ces consignes, on peut maintenant revenir au fonctionnement propre du processus d'automatisation.

Avant toute chose, il faut récupérer dans la base de données toutes les salles qui possèdent des multi-senseur et des actionneurs. Puis, pour chaque salle, on traite tout d'abord les actionneurs de stores et ce, en fonction des règles suivantes :

1. Jour
    a. Règles désactivées
    b. Fermeture des stores quand personne n'est présent dans la salle et qu'il y a du soleil
    c. Fermeture des stores quand personne n'est présent dans la salle et qu'il pleut
    d. Combinaison des règles b et c
2. Nuit
    a. Règles désactivées
    b. Fermeture des stores

Deuxièmement, pour chaque salle, on traite les actionneurs de vannes de radiateur en fonction des règles suivante :

3. Jour et nuit
    a. Les règles sont désactivées
    b. La régulation des vannes de radiateur

Ces règles ont été volontairement implémentées dans le but de ne pas gêner les personnes qui se trouvent dans une salle. Pour cette raison, on a décidé qu'elles déclencheraient un actionneur uniquement dans le cas où personne n'est présent dans la salle. En effet, on a pu constater que dans certains bâtiments conçus suivant des règles plus « intrusives » les stores se fermaient ou s'ouvraient alors que des personnes étaient à l'intérieur de la salle.

6.6.7.1 Règle 1a

Cette règle a été définie dans le cas où l'utilisateur souhaiterait que les stores ne s'actionnent pas pendant la journée.

6.6.7.2 Règle 1b

Cette règle a été spécifiée afin d'éviter que le soleil pénètre dans une salle et fasse augmenter la température de manière trop importante. Pour cela, on a défini plusieurs critères qui ont pour objectif de fermer tous les stores d'une salle si toutes les conditions suivantes sont réunies :

1. Vérification de la période : jour
2. Vérification de la présence dans la salle : pas de présence
3. Vérification de la quantité de nuages dans le ciel : présence de nuages dont le facteur couvrant est inférieur à 60%
4. Vérification de la pénétration du soleil dans la salle : oui
5. Vérification de la saison : printemps ou été
6. Vérification de la température extérieure maximum à laquelle on détermine qu'une pénétration supplémentaire du soleil provoquerait une augmentation trop importante de la température intérieure de la salle.

6.6.7.3 Règle 1c

Cette règle a été définie pour empêcher l'infiltration d'eau de pluie lorsque que, par inadvertance, la dernière personne sortant de la salle oublie de fermer les fenêtres. En effet, il y a quelques mois, une salle avec du matériel informatique a été partiellement inondée ce qui a nécessité le remplacement de plusieurs machines. À cet effet, on a déterminé plusieurs critères qui entraîneront la fermeture de tous les stores d'une salle si toutes les conditions suivantes sont réunies :

1. Vérification de la présence dans la salle : pas de présence
2. Vérification de la présence de précipitations pendant la journée : oui

6.6.7.4 Règle 1d

Cette règle a été édifiée dans le but de combiner les règles se rapportant aux précipitations ou à la pénétration du soleil lorsqu'aucune présence dans la salle n'est détectée.

6.6.7.5 Règle 2a

Cette règle a été spécifiée dans le cas où l'utilisateur souhaiterait que les stores ne s'actionnent pas pendant la nuit.

6.6.7.6 Règle 2b

Pour des raisons de sécurité et de conservation de la température dans la salle pendant la période de nuit, on ferme les stores.

6.6.7.7 Règle 3a

Cette règle a été définie dans le cas où l'utilisateur souhaiterait que les vannes de radiateur ne s'actionnent pas, ni durant le jour, ni durant la nuit. Au moment où cette règle est appliquée par le processus d'automatisation, la régulation n'alors pas effectuée et les vannes de radiateur restent à la même position.

6.6.7.8 Règle 3b

Cette règle a été édifiées afin d'effectuer une régulation de la température dans une salle au moyen d'un module PID et ce, autant pendant la période jour que celle de nuit.

6.7 Serveur REST KNX

Tout au long de la réalisation de ce projet, on n'avait pas la possibilité d'accéder aux actionneurs de stores et aux vannes de radiateur afin d'éviter que cela provoque d'éventuels problèmes sur l'infrastructure existante. De ce fait, on a dû travailler sur un simulateur couplé d'un serveur de type REST pour modifier l'état des actionneurs. Ce serveur REST fonctionne de la même manière si on l'utilise avec le simulateur ou sur le réseau réel d'actionneurs. La seule chose qui diffère, c'est l'adresse IP ainsi que le port auquel les messages sont envoyés :

- Simulateur
    - IP : 0.0.0.0
    - Port : 3672
- Réseau réel d'actionneurs :
    - IP : {IP du routeur/firewall}
    - Port : 3671

Pour plusieurs raisons, on a dû changer et ajouter des fonctions dans le serveur REST :

1. La structure des routes a été modifiée pour écrire et lire sur les actionneurs car à l'origine, il n'était pas possible d'interroger des actionneurs d'un autre étage.
2. La méthode permettant de changer les identifiants des actionneurs a également été modifiée. En effet, dans le code, il existe une liste d'identifiants statiques qui permet de définir si lors de l'envoi d'un message, l'identifiant de l'actionneur est bien correct. Un problème se pose alors : depuis l'application web, il est possible d'ajouter et de supprimer des actionneurs et par conséquent, modifier faut pouvoir modifier cette liste statique. Ainsi, on a créé une route (fonction *update*) qui sera appelée lors d'un ajout d'actionneur par l'application web et qui permettra de faire un appel à la base de données pour récupérer la liste des nouveaux identifiants tout en actualisant cette liste statique.
3. Une route (fonction *root*), permettant de faire une requête HPPT GET sans avoir à interroger un actionneur pour déterminer si le serveur REST est toujours disponible, a été ajoutée.

6.8 Environnement de déploiement

Après avoir développé l'application web et celle d'automatisation, on a dû rassembler tous les éléments de *Smarthepia* et les faire passer sur une seule machine dont voici les spécifications :

- Système d'exploitation : Ubuntu 18.04 LTS 64x
- PC : Dell Optiplex 9020 (CPU : 4*3.2GHz, RAM : 16Go)

D'autre part, pour des raisons de sécurité et de fiabilité, on a effectué plusieurs manipulations :

- Installation de Nodejs 8.10 et *npm* 3.5.2 : rendre disponible l'application web
- Installation d'Openssh-server : permettre une connexion SHH sur le serveur
- Installation du serveur web proxy Nginx 1.14: répondre aux requêtes de l'utilisateur sans directement interroger le serveur qui fonctionne avec Nodejs. (Ngnix 80 -> Nodejs 3000)
- Mise en place du certificat auto-signé sur le serveur Nginx
- Installation de python 3.6 : lancer le programme d'automatisation ainsi que le serveur REST KNX
- Installation de MongoDB 3.6.3
- Installation de PM2 : permettre de faire tourner en tache de fond l'application web et de la redémarrer automatiquement si elle venait à ne plus répondre
- Installation des toutes les librairie et dépendances

Une fois ces opérations effectuées, on a lancé l'application web, le serveur REST KNX, le simulateur et le programme d'automatisation pour vérifier la quantité de ressources nécessaires au bon fonctionnement de ces applications. Voici les résultats :

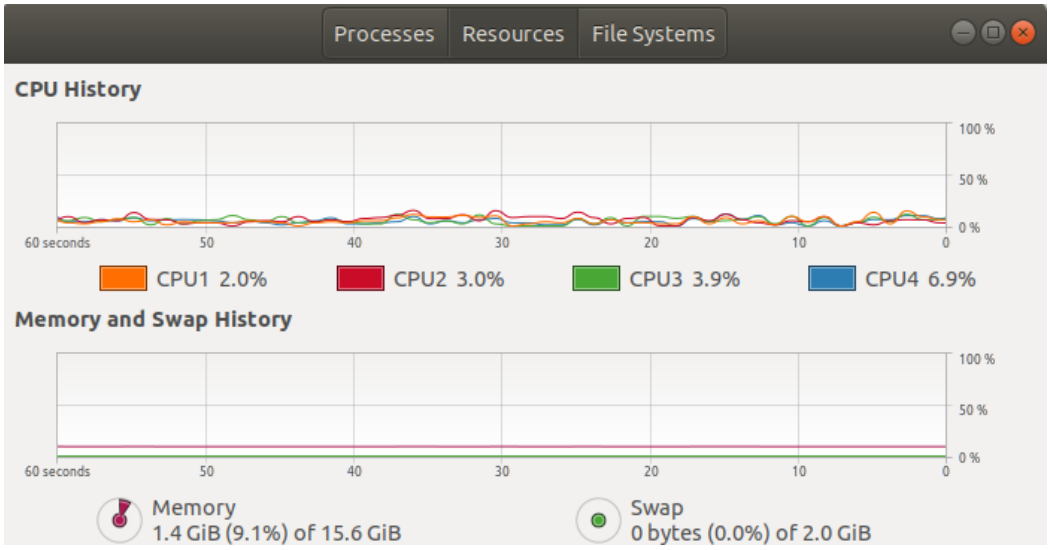


Figure 34 : Charge du PC Dell Optiplex avant l'exécution de Smarthepia

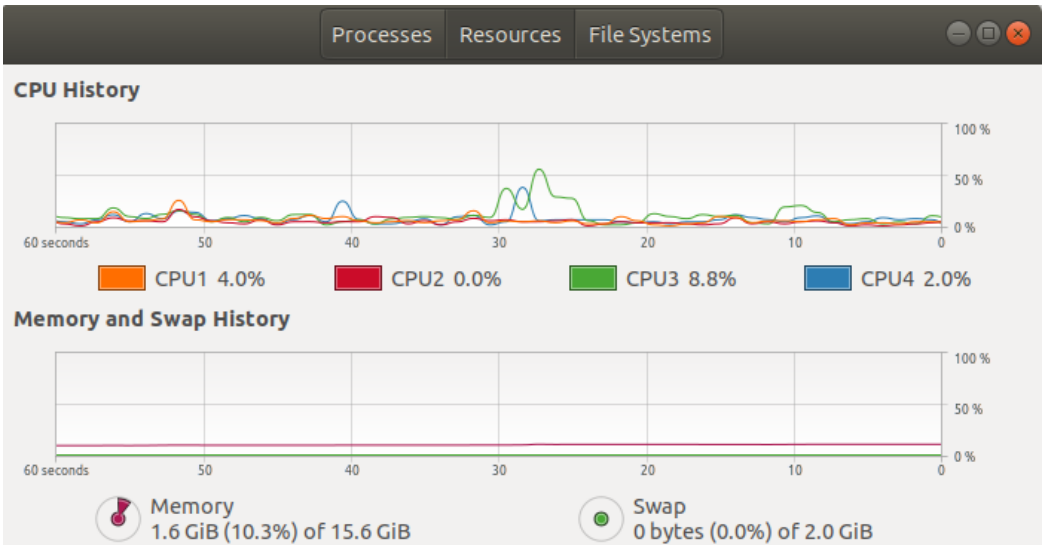
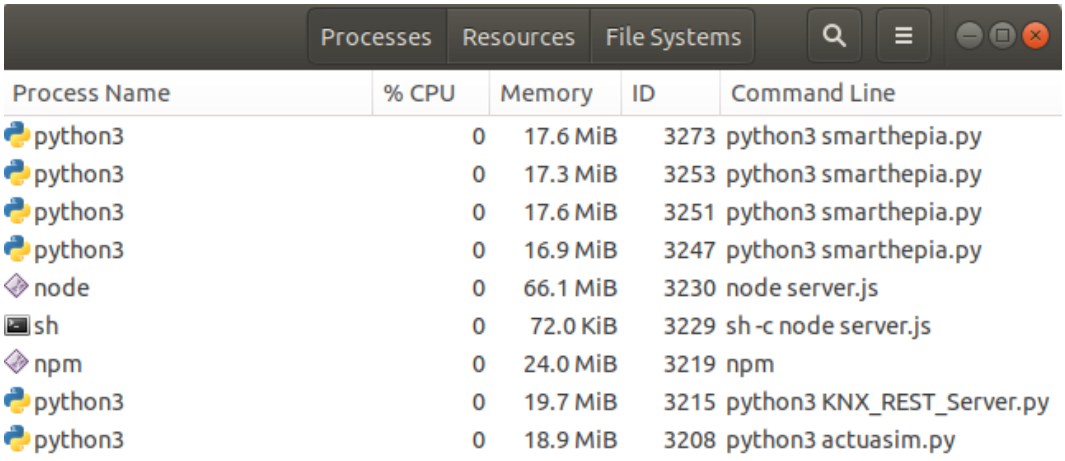


Figure 35 : Charge du PC Dell Optiplex après l'exécution de Smarthepia



Process Name	% CPU	Memory	ID	Command Line
python3	0	17.6 MiB	3273	python3 smarthepia.py
python3	0	17.3 MiB	3253	python3 smarthepia.py
python3	0	17.6 MiB	3251	python3 smarthepia.py
python3	0	16.9 MiB	3247	python3 smarthepia.py
node	0	66.1 MiB	3230	node server.js
sh	0	72.0 KiB	3229	sh -c node server.js
npm	0	24.0 MiB	3219	npm
python3	0	19.7 MiB	3215	python3 KNX_REST_Server.py
python3	0	18.9 MiB	3208	python3 actuasim.py

Figure 36 : Détaille de la charge du PC Dell Optiplex après l'exécution de Smarthepia

On remarque que le *Smarthepia* ne consomme pas beaucoup de ressources. En effet, entre avant (Figure 34) et après (Figure 35) l'exécution des éléments du *Smarthepia*, on constate seulement deux légères augmentations :

- RAM : 0.2 Go
- CPU : non quantifiable car trop faible

7. Conclusion

Comme expliqué au début de ce travail, l'objectif initial de ce projet était de mettre à dispositions une plateforme – *Smarthepia* – permettant, grâce à différentes mesures collectées par des multi-senseurs, d'automatiser le contrôle des stores et des vannes de radiateurs au sein d'une salle de cours. Pour gérer cette plateforme, deux applications ont été développées : une application web et une application d'automatisation.

Au terme de ce rapport, on peut dire que le cahier des charges a été rempli. En effet, tous les éléments de ce projet ont pu être réalisés et les deux applications citées ci-dessous fonctionnent parfaitement sur un simulateur. Compte tenu d'un accès trop tardif aux actionneurs réels – soit ceux se trouvant dans les salles de l'hepia – on n'a effectivement pas eu le temps de réaliser les tests souhaités.

Si, comme on vient de le dire, ce projet s'est bien déroulé, on a quand même éprouvé quelques difficultés. Tout d'abord, comme on n'a pas utilisé le duo « node/express », on ne savait pas comment traiter les tâches asynchrones lors d'une requête de l'utilisateur, ce qui a provoqué quelques *bugs* que l'on a pu résoudre par la suite.

Par ailleurs, on s'est aperçu que toutes les personnes non authentifiées à l'application web pouvaient accéder au *websocket*. Il a donc fallu que l'on partage les variables de session entre les *websockets* et les requêtes faites par les utilisateurs sur le serveur web.

Enfin, avec Nodejs, on utilise un module permettant de construire des schémas qui vont ensuite interroger une base de données, ici MongoDB. Cependant, comme il est aussi possible de les créer à la main, des problèmes de cohérence et des erreurs quelque peu surprenantes ont été retournés dans certain cas.

Pour conclure et mettre en évidence l'intérêt et le potentiel d'un tel projet, on ne saurait ne pas mentionner les multiples améliorations que l'on pourrait encore y apporter. Premièrement, si on possédait pour les années précédentes une quantité suffisante d'informations labelisées sur les multi-senseur et les actionneurs, il serait possible de faire du « *machine learning* », soit connaître les habitudes et les préférences – stores ouverts ou fermés – des personnes se trouvant dans les salles. Cela permettrait notamment de perfectionner les règles d'automatisation. Deuxièmement, il serait intéressant de procéder à une automatisation davantage complète de l'hepia prenant en compte l'ensemble des bâtiments, des étages et des salles ainsi qu'en automatisant d'autres éléments que les stores ou les vannes de radiateurs, à l'exemple des lumières.

8. Références

1. **Projet iNUIT. [En ligne] http://lsds.hesge.ch/cloud-computing-for-internet-of-things-infrastructure/.**

2. Smarthepia. [En ligne] http://lsds.hesge.ch/smarthepia/.

3. Z-Wave siteweb spec. [En ligne] http://www.z-wave.com/learn.

4. Z-Wave chiffrement. [En ligne] https://www.orange-business.com/fr/blogs/securite/nouvelles-technologies/protocole-sans-fil-z-wave-focus-sur-le-chiffrement-des-donnees.

5. Nombr de controleurs. [En ligne] https://www.vesternet.com/resources/technology-indepth/understanding-z-wave-networks/.

6. KNX wikipedia. [En ligne] https://fr.wikipedia.org/wiki/KNX_(standard).

7. ETS5 téléchargement. [En ligne] https://www2.knx.org/no/software/eitt/downloads/index.php.

8. Github. [En ligne] https://github.com/.

9. Template Stack. [En ligne] https://themeforest.net/item/stack-responsive-bootstrap-4-admin-template/20039431.

10. API openweathermap. [En ligne] https://openweathermap.org/city.

11. Module PID. [En ligne] https://pypi.org/project/simple-pid/.

12. Librairie Pysolar. [En ligne] http://pysolar.readthedocs.io/en/latest/.

13. Gateway KNX à IP. [En ligne] https://www.eibmarkt.com/cgi-bin/eibmarkt.storefront/5ae2f24e00db8d7c2748ac1e0402063b/Product/View/N000401.

14. Adressage KNX. [En ligne] http://knxer.net/?p=49.

15. Adresse KNX. [En ligne] http://elnoter.dk/ibi-intelligente-bygningsinstallationer/knx/.

16. Régulation PID. [En ligne] https://fr.wikipedia.org/wiki/R%C3%A9gulateur_PID.

9. Table des acronymes

HTTP	*Hypert Text Protocol*
REST	*Representational State Transfer*
RPI	*Raspberry PI*
NPM	*Hypertext Markup Language*
HTML	*Hypertext Markup Language*
CSS	*Cascading Style Sheets*
AJAX	*Asynchronous JavaScript And XML*
PID	*Proportional Integral Derivative*

10. Table des illustrations

11. Table des tableaux