Conception d'un système d'intelligence ambiante à l'aide de cartes Arduino

Ivan Dal Pio Luogo

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL ASSOCIE DE RHONE-ALPES

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : INFORMATIQUE

OPTION : RESEAUX, SYSTEMES et MULTIMEDIA (IRSM)

par

Ivan DAL PIO LUOGO

Conception d'un système d'intelligence ambiante

à l'aide de cartes Arduino

Soutenu le 17 Avril 2015

JURY

PRESIDENT :	**M. Christophe PICOULEAU**	***Professeur CNAM Paris***
MEMBRES :	**M. Bertrand DAVID**	***Professeur des Universités EC Lyon***
	M. Claude GENIER	***Directeur CNAM Léman***
	M. René CHALON	***Maitre de conférences EC Lyon***
	M. Chen WANG	***Doctorant EC Lyon***

Remerciements

Je tiens tout d'abord à remercier Bertrand David qui m'a offert la possibilité de réaliser mon mémoire d'ingénieur au sein du laboratoire LIRIS et de travailler dans un environnement de recherche sur des sujets innovants et très intéressants. Je remercie aussi René Chalon qui m'a grandement aidé pour réaliser mon projet en m'aiguillant sur les différentes techniques et méthodes à utiliser. Enfin je remercie tous les collègues doctorants et CNAMiens croisés au LIRIS avec qui j'ai pu travailler dans une bonne ambiance et dans le partage (Chen, Bingxue, Huilang, Nathalie, Bertin, Gaël et Loïc).

Je souhaite remercier toutes les personnes du CNAM que j'ai côtoyées durant ces années et qui m'ont bien renseigné pour réaliser mon cursus. Je remercie particulièrement Eléonore Gondeau qui m'a apporté une aide précieuse et sa gentillesse pour m'aider dans la fin du cursus d'ingénieur.

Je remercie ma famille pour toute l'aide apportée durant ces années notamment mon père pour son aide sur différentes matières et les présentations d'entrainement et ma mère pour tous les dossiers qu'elle a corrigé. Je remercie aussi tous mes amis pour leurs encouragements. Pour finir je remercie particulièrement ma copine qui m'a grandement soutenu et encouragé tout au long de mon cursus et qui m'a aussi beaucoup aidé pour répéter mes soutenances.

Liste des abréviations

AmI : **Am**biant **I**ntelligent

TCP/IP : **T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocole

HTTP : **H**yper**T**ext **T**ransfer **P**rotocole

IDE : **I**ntegrated **D**evelopment **E**nviroment

URI : **U**niform **R**essource **I**dentifier

PWM : **P**ulse **W**idth **M**odulation

API : **A**pplication **P**rogramming **I**nterface

REST : **RE**presentational **S**tate **T**ransfer

JSON : **J**ava**S**cript **O**bjet **N**otation

RFID : **R**adio **F**requency **ID**entification

Glossaire

Proof of concept : en français preuve de concept ou encore démonstration de faisabilité. Ce terme définit la réalisation de méthodes ou d'idées pour démontrer leurs faisabilités. C'est une étape précédant la réalisation d'un prototype pleinement fonctionnel.

Physical Computing : système physique interactif au monde réel grâce à l'utilisation de logiciels, capteurs et actionneurs.

Ethernet : standard (nome IEEE 802.3) de transmission de données par câble pour réseau local.

TCP/IP : ensemble de protocoles utilisés pour le transfert de données sur Internet. TCP permet de découper les flux de données en plusieurs paquets pour les transporter sur le réseau et IP encapsule ces paquets pour apporter la notion d'adressage afin d'envoyer les données au destinataire.

API : application programming interface ou interface de programmation. Ensemble de méthodes ou de fonctions normalisées servant de façade à l'utilisation de services d'autres logiciels.

End User Programming : outils et techniques permettant aux utilisateurs de créer un programme sans connaissance en langage de programmation.

URI : uniform resource identifier ou identifiant uniforme de ressource est une chaine de caractère normalisée (RFC 3986) identifiant une ressource sur un réseau. Cette ressource peut être physique ou abstraite.

Table des matières

Introduction

L'Intelligence Ambiante est un concept informatique qui vise à définir un environnement constitué de plusieurs appareils électroniques se fondant dans le quotidien, qui répondent à la présence de personnes ou d'objets divers. C'est derrière ce concept que l'on parle d'objets dit « intelligents » qui sont reliés entre eux en réseau et qui peuvent communiquer avec l'utilisateur afin d'offrir une augmentation de l'environnement en terme de possibilités d'interactions et d'accès à l'information. Cette communication peut être passive pour l'utilisateur où sa seule présence déclenche plusieurs actions sans même qu'il s'en rende compte.

Ces objets intelligents se retrouvent dans plusieurs domaines d'application tel que les réseaux de capteurs déployés dans les villes pour suivre le trafic et ainsi informer en temps réel de l'état de la circulation (exemple : le site Onlymoov pour le trafic lyonnais). Ou encore dans le domaine de la domotique qui permet de faciliter et d'automatiser l'utilisation de tous les appareils électroniques de sa maison (éclairage, climatisation, alarme, chauffage, télévision connectée, vidéo surveillance) et d'établir des règles pour, par exemple, mieux gérer notre consommation énergétique. Ceci peut être fait à distance à travers plusieurs terminaux : tablette utilisée comme tableau de bord, ordinateur ou encore smartphone.

La mise en place de ces solutions permettant d'augmenter un environnement en apportant une intelligence ambiante passe par l'utilisation de capteurs ou d'actionneurs fonctionnant et communiquant de manières différentes suivant plusieurs protocoles. Au sein du laboratoire LIRIS, mon projet est d'étudier la création d'un système, sous forme de proof of concept, pouvant gérer plusieurs capteurs et actionneurs hétérogènes à travers un réseau pour constituer une intelligence ambiante applicable à plusieurs contextes. Le but est de rendre ces capteurs et actionneurs - que nous appellerons objets - communicants ou intelligents, et exploitables sans connaissance en électronique. Pour cela j'utilise des cartes Arduino qui s'occupent de la communication bas-niveau avec les objets et je conçois une application sous forme d'API fournissant les outils pour dialoguer avec ces cartes et par conséquent avec des objets intelligents à travers le Web.

Dans le 1er chapitre, nous décrivons le contexte de recherche dans lequel se situe le projet en présentant le laboratoire LIRIS.

Le 2ème chapitre est consacré au contexte informatique, avec les différents concepts derrière la notion d'intelligence ambiante. Nous définissons les technologies et les techniques mises en œuvre pour l'élaboration du proof of concept.

Le 3ème chapitre traite des étapes de conception du système avec l'élaboration des spécifications de l'application et de son API modélisée suivant le style d'architecture REST. Ce style d'architecture nous permet d'introduire la notion d'objets communicants à travers le Web appelée Web des objets.

Le 4ème chapitre décrit la phase de réalisation du proof of concept avec les choix d'architectures matérielles et logicielles pour arriver à mettre en place les fonctionnalités de notre système.

Le 5ème chapitre présente la validation du système par un cas d'étude en relation avec les travaux du laboratoire sur les « smart cities » (villes intelligentes) : l'allocation dynamique des voieries. A partir de cette mise en pratique, nous dissertons sur le système mis en place pour en faire ressortir des perspectives d'améliorations.

Chapitre 1 : Cadre du projet

1.1 LIRIS

Le LIRIS (Larboratoire d'InfoRmatique en Image et Systèmes d'information) est une unité de recherche regroupant 320 membres dont les tutelles sont le CNRS, L'INSA de Lyon, l'Université Claude Bernard Lyon 1, l'Université Lumière Lyon 2 et l'Ecole Centrale de Lyon. Le champ de recherche est de manière générale les Sciences et Technologies de l'Information. Le LIRIS est divisé en plusieurs équipes de recherche suivant des pôles de compétences.

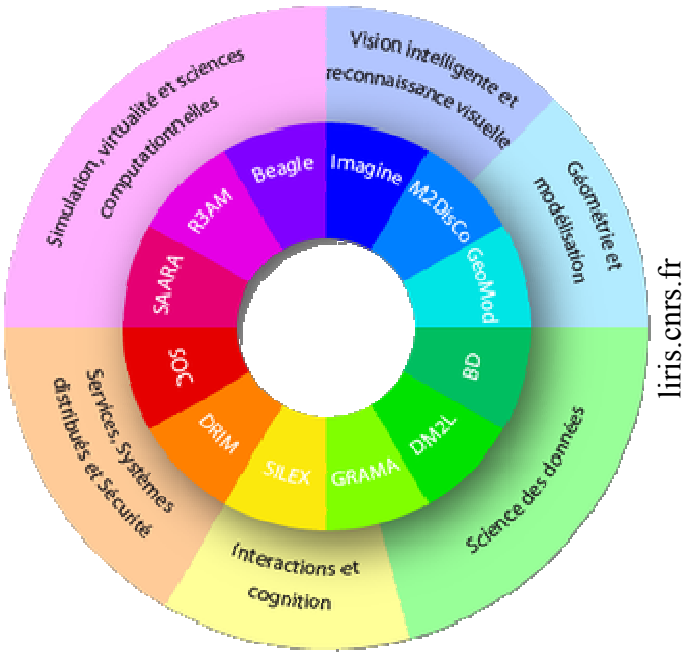


Figure 1 - Pôles de recherches LIRIS

1.2 L'équipe SILEX

Mon travail d'ingénieur s'effectuait au sein de l'équipe SILEX (Supporting Interaction and Learning by Experience) du LIRIS. Elle fait partie du pole interactions et cognition dont l'approche est de considérer le couple utilisateur-machine comme un seul système apprenant et co-évoluant en fonction d'activités menées. L'objectif étant d'apporter une réflexion sur des questions concernant la construction de connaissances, l'assistance à l'utilisateur, l'adaptation du système à l'utilisateur ou encore l'analyse des usages du système par les utilisateurs [Equipe SILEX – 2014].

Au sein de ce pôle de recherche, l'équipe SILEX divise ces travaux en plusieurs thèmes. Mon projet se situe dans celui nommé « Systèmes interactifs adaptatifs ». Ce thème s'intéresse principalement aux interactions entre l'humain et la machine pour la conception de systèmes interactifs, adaptables et adaptatifs. Une partie des recherches se porte sur les interactions « innovantes » dans les nouveaux concepts de l'informatique regroupés sous la thématique de l'informatique ambiante. Par exemple, des travaux de doctorat sont en cours sur les « smart cities » dont l'une des applications concerne le domaine des transports avec la gestion dynamique des voies de circulation. Il est question de concevoir une approche pour aider à la gestion du trafic avec la mise en place d'une intelligence ambiante basée sur des services de localisation des véhicules. Ceci fait intervenir la notion d'internet des objets et demande d'étudier de nouvelles interactions avec ces objets.

C'est dans ce cadre de recherche dans le domaine de l'intelligence ambiante que le projet de concevoir un système pouvant intégrer un réseau de capteurs et d'actionneurs permettant d'augmenter la réalité est né. Ce système doit permettre d'amener une intelligence ambiante dans des scénarii variés en faisant preuve d'adaptabilité. Une application est prévue dans le cadre des travaux sur la gestion dynamique des voieries. Le projet est un « Proof of Concept » s'appuyant sur les cartes Arduino grandement utilisées dans le monde universitaire et dans le prototypage amateur.

Chapitre 2 : Contexte informatique

2.1 L'informatique du 21ème siècle

2.1.1 Informatique Ubiquitaire

L'informatique ubiquitaire est un concept introduit en 1991 par Mark Weiser, chercheur au Xerox PARC, lorsqu'il décrit dans plusieurs articles sa vision de l'ordinateur du 21ème siècle. Il prévoit que l'informatique sera présente partout dans la vie de tous les jours, quelques soit les endroits. C'est pour cela qu'il utilise le terme « ubiquité » qui signifie la capacité d'être présent en tout lieu [Ubiquité – 2014]. Portant un regard sur plusieurs projets au sein du Xerox PARC sur de nouveaux ordinateurs plus petits et portables (palm tops ou des PDA), il voit en ces technologies le début de l'informatique ubiquitaire où celle-ci sera de plus en plus miniaturisée et facilement accessible dans la vie quotidienne des utilisateurs quels que soient les lieux où ils se trouvent.

Pour que cette évolution sur l'utilisation de l'informatique puisse s'effectuer, il faut que les technologies évoluent sur plusieurs aspects. Les ordinateurs et les batteries doivent être miniaturisés pour une utilisation portable et la consommation énergétique doit être plus faible, toujours pour favoriser cette utilisation. Enfin il faut surtout que les réseaux puissent permettre de relier tous les appareils à très grande échelle [Weiser - 1991].

Cette vision a été confirmée car les technologies ont grandement évolué et répondent aux problématiques relevées par Mark Weiser. Les appareils informatiques sont à présent de plus en plus petits et de plus en plus puissants tout en ayant une durée de fonctionnement assez longue pour être utilisés de manière nomade. De plus ils sont proposés à faible coût. L'exemple le plus marquant est l'avènement du smartphone qui représente bien le concept d'informatique

ubiquitaire avec l'accès à l'informatique partout, de manière nomade et utilisé par un très large public. En parallèle, les technologies réseaux ont elles aussi grandement évolué notamment sur la partie mobile avec par exemples les normes : GSM, 3G/4G. L'utilisateur est connecté à un réseau ne nécessitant pas de branchement, quel que soit le lieu. Enfin l'avènement du Web avec le protocole HTTP (HyperText Transfer Protocol) a permis d'avoir l'accès à une multitude de services sur un très grand réseau.

2.1.2 Intelligence Ambiante

L'intelligence Ambiante ou AmI, fait référence à un environnement informatique et électronique qui est sensible et répond à la présence de personnes [Ambient Intelligence – 2014]. Le concept d'intelligence ambiante peut être considéré comme la suite de l'évolution de l'informatique ubiquitaire. En effet, l'évolution des technologies en termes de miniaturisation, de réseaux à grande échelle couplée à des matériels aux coûts de plus en plus bas, constitue principalement les fondements de cet informatique ubiquitaire qui a évolué vers l'intégration des facultés de calcul et de communication dans l'environnement.

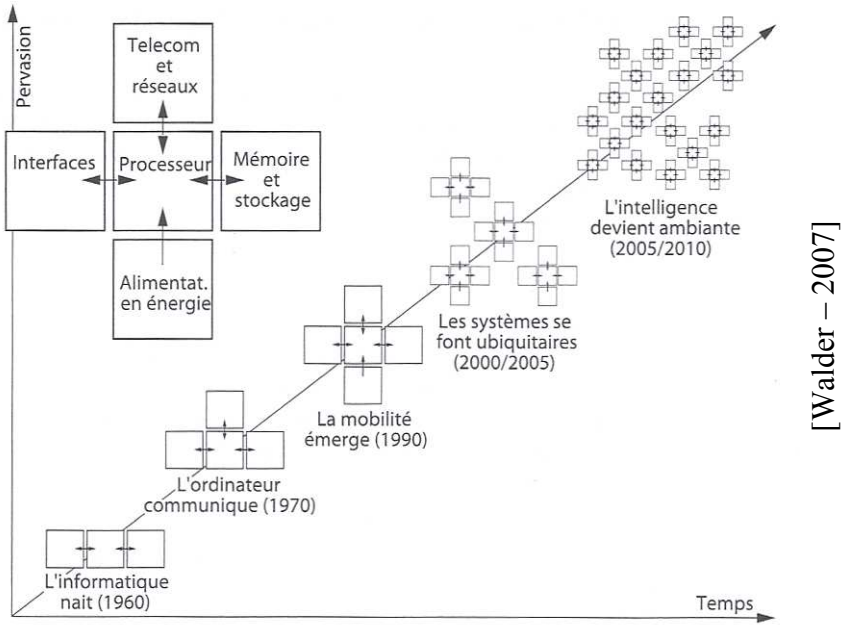


Figure 2 - Evolution de l'informatique vers l'AmI

On parle d'AmI pour un environnement qui est dit intelligent dans lequel les ordinateurs et le réseau sont intégrés dans l'environnement quotidien sans même qu'on les remarque. L'être humain est alors entouré d'interfaces intelligentes et distribuées dans des objets quotidiens

permettant un accès à des services via des interactions simples et intuitives. Ces interactions étant parfois réalisées sans que l'utilisateur s'en rende compte, de manière inconsciente. Ces objets du quotidien sont ainsi « augmentés », c'est-à-dire qu'ils ont des capacités de traitement de l'information et d'interactions avec cette information, on parle alors de « réalité augmentée » [Waldner – 2007].

L'AmI, avec cette informatique embarquée au sein d'une multitude d'objets introduit la notion d'internet des objets. Ce ne sont plus seulement les ordinateurs qui sont capables de communiquer mais tout un tas d'objets munis de capteurs qui peuvent récupérer des informations sur un contexte donné et permettre une analyse de l'environnement. Ces objets interagissent avec un utilisateur par le biais d'actionneurs. Le choix du réseau est très important dans le cadre d'une AmI car l'ensemble d'objets intelligents qui la constituent doit communiquer quelles que soient les distances qui les séparent. C'est pour cela que le choix du Web est approprié avec l'utilisation des principaux protocoles tels que TCP/IP et HTTP.

Un exemple représentatif de cette intelligence ambiante est la mise en avant des objets connectés pour la maison : c'est le domaine de la domotique. Grâce à l'informatique embarquée, il est possible d'intégrer des réseaux de capteurs au sein de sa maison qui peuvent collecter de l'information qui sera traitée de manière intelligente. Ainsi la gestion de la consommation d'énergie peut être maitrisée avec notamment l'éclairage qui se met en marche dès que la présence de personnes est détectée ou encore la gestion de l'intensité de cet éclairage suivant la luminosité en prévenance de l'extérieur. Il en est de même pour la collecte d'information sur la température ambiante de la maison pour gérer intelligemment et automatiquement la climatisation ou le chauffage. Les appareils électroménagers sont eux aussi augmentés, on les appelle des appareils électrodomestiques. Ils ont la particularité d'être connectés au réseau et d'envoyer des informations sur leur fonctionnement et d'être pilotables à distance. Tout ceci s'accompagne d'IHM (Interfaces Homme-Machine) simples et intuitives pour être utilisées par un large panel de consommateurs. De plus ces IHM sont accessibles via internet et le Web, et prévues pour fonctionner sur ordinateur ou smartphone pour apporter une gestion de sa maison à distance.

2.2 Apport de l'Arduino

2.2.1 Présentation

Arduino est une plateforme de développement de projets électroniques se déclinant sous plusieurs types cartes de la taille d'une carte de crédit. Les cartes Arduino possèdent des caractéristiques différentes mais ont un grand nombre d'interfaces en commun. Ces interfaces sont généralement un ensemble de broches utilisables en entrée et en sortie, géré par un microcontrôleur programmable faisant fonctionner un large panel de composants électroniques. Arduino désigne aussi l'environnement de développement ou IDE (Integrated Development Environment) ainsi que le langage de développement créé pour faciliter l'utilisation du microcontrôleur. Le tout a été pensé dans un esprit open-source, aussi bien l'IDE que l'ensemble des cartes dont les schémas sont disponibles. Ainsi tout le monde peut participer à l'évolution d'Arduino aussi bien au niveau logiciel que matériel.

La force de ces cartes est d'ouvrir l'accès au « physical computing » de manière plus simple que ce qui était proposé avec la programmation classique d'un microcontrôleur. On appelle physical computing la création de systèmes physiquement interactifs par le biais de capteurs et d'actionneurs à l'aide de logiciels. Le logiciel de programmation Arduino fonctionne sur les principaux environnements (Windows, Macintosh et Linux). Le langage se base sur le C et le C++ et fournit plusieurs fonctions simples d'utilisation pour gérer les broches d'entrées et de sorties. Le fichier de programmation Arduino est envoyé directement dans la mémoire du microcontrôleur depuis le PC vers l'Arduino à travers l'USB, la carte faisant office de programmateur de microcontrôleur ce qui facile grandement le prototypage.

Figure 3 - Arduino Uno

Arduino est devenu très populaire dans le monde du prototypage grand public et possède une très grande communauté. Il existe de nombreuses extensions officielles ou provenant de vendeurs tiers permettant d'accroitre les fonctionnalités d'interactions avec le monde physique.

Dans le cadre de notre projet, les cartes Arduino nous apportent l'informatique embarquée de notre système d'intelligence ambiante.

2.2.2 Capteurs et actionneurs

Une carte Arduino est capable de fonctionner avec un très grand nombre de capteurs et d'actionneurs du moment qu'ils respectent certaines conditions liées au courant qu'ils acceptent et celui délivré par la carte, mais aussi suivant leurs protocoles de communication.

Le *tableau* 1 et le *tableau 2* montrent quelques exemples de capteurs et d'actionneurs intéressant dans le cadre de l'élaboration d'environnements augmentés.

Actionneur	Descriptif	Actionneur	Descriptif
 sparkfun.com **Led**	S'illumine plus ou moins fort suivant l'intensité de courant envoyée. Existe en différentes couleurs mais aussi en infrarouge ou en ultraviolet	 sparkfun.com **Ecran LCD**	Ecran à cristaux liquides permettant d'afficher des informations
 sparkfun.com **Servomoteur**	Moteur faisant tourner un axe jusqu'à atteindre une position (souvent un angle).	 sparkfun.com **Haut-parleur**	Haut-parleur permettant de générer du son.

Tableau 1 - Exemple d'actionneurs

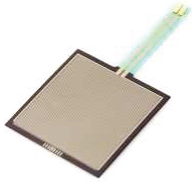
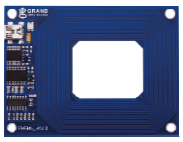
Capteur	Descriptif	Capteur	Descriptif
 sparkfun.com **Capteur de force**	Détecte une pression exercée sur l'ensemble de sa surface. La valeur du voltage en sortie du capteur varie en fonction de la pression exercée	 sparkfun.com **Photorésistance**	Détecte la lumière sur la surface au bout des tiges. La valeur du voltage en sortie du capteur varie en fonction de l'intensité de la lumière
 sparkfun.com **Capteur Ping**	Détecte les objets à proximité du capteur. Fonctionne par envoi d'ultrasons. Suivant le temps de réponse, on peut déterminer la distance de l'objet par rapport au capteur	 wikimedia.org **Capteur PIR**	Détecte les mouvements. Envoie 5v sur une sortie digitale à chaque détection.
 sparkfun.com **Capteur température**	Indique la température en sortie du capteur suivant les variations de voltage grâce à une table de correspondance.	 sparkfun.com **Lecteur RFID**	RFID (Radio Frequency Identification). Permet de récupérer l'identifiant d'un transpondeur ou tag à une distance de plusieurs centimètres.

Tableau 2 - Exemple de capteurs

2.2.3 Interfaces

Pour connecter tous ces composants, l'Arduino propose plusieurs types d'interfaces sous forme de broches. Ces interfaces sont gérées par le microcontrôleur qui s'occupe de traiter l'information en provenance des composants connectés à celles-ci.

Voici la description des interfaces « standard » que l'on retrouve dans une grande partie des cartes de la famille Arduino. Ce sont aussi les principales interfaces utilisées par une majorité des composants :

- **Analog Input** : ce sont des entrées analogiques auxquelles sont reliés principalement des capteurs dont le fonctionnement fait varier la tension entre 0v et 5v. Chaque documentation livrée avec ce type de capteur indique comment faire correspondre cette variation de tension avec la valeur physique en rapport avec la fonctionnalité du capteur. Par exemple, le constructeur d'un capteur de température indique la correspondance entre cette tension et les degrés Celcius suivant une courbe.

- **Digital I/O** : ce sont des interfaces qui peuvent fonctionner soit en entrée digitale soit en sortie digitale. Le microcontrôleur récupère un état binaire 0 ou 1 qui s'exprime par une tension de 0v ou 5v. Ces interfaces sont utilisées pour des composants ayant deux états. Par exemple, dans le cas d'une sortie digitale, cela permet d'utiliser un actionneur LED (Light-Emmitting Diode ou diode électroluminescente) et de l'allumer ou de l'éteindre. Dans le cas d'un bouton poussoir, sur une entrée digitale, on peut savoir si le bouton est pressé ou non.
- **PWM** : ces interfaces sont des « digital I/O » mais ont aussi la possibilité de simuler une sortie analogique par la technique PWM (pulse with modulation). Il est ainsi possible de faire varier la tension entre 0v et 5v sur ces interfaces. Cela permet par exemple de piloter des actionneurs LED pour faire varier l'intensité de la lumière avec la variation de tension mais aussi de piloter des servomoteurs, où cette variation permet de choisir l'angle de rotation ou encore la vitesse de rotation.
- **Serial** : cette interface permet une liaison série avec le protocole UART (Universal Asynchronous Receiver Transmitter). Certains composants ayant besoin d'envoyer des données utilisent ce protocole et cette interface pour communiquer. C'est le cas de certains lecteurs RFID qui utilisent cette interface pour envoyer les identifiants lus sur les tags. Cette interface permet la communication avec tout type d'appareil utilisant ce protocole comme un ordinateur ou encore un autre Arduino.
- **SPI** : cette interface signifie Serial Peripheral Interface et permet d'envoyer des données au même titre que l'interface Serial. Cette liaison a été inventée par Motorola pour établir un standard dans la communication entre un microcontrôleur et plusieurs composants ou alors avec d'autres microcontrôleurs. Elle s'utilise donc avec des composants utilisant ce protocole de communication comme par exemple certains modèles de gyroscopes.
- **I²C** : cette interface signifie Inter-Integrated Circuit et a été inventée par Philips. Au même titre que SPI, elle est conçue pour établir une communication entre plusieurs microcontrôleurs ou entre un microcontrôleur et des composants dialoguant avec ce protocole. Certains capteurs de température ou de pression atmosphérique fonctionnent avec ce protocole

2.2.4 Cartes d'extension « shield »

Les cartes Arduino ont été conçues pour accepter des extensions permettant d'accroitre leurs capacités suivant les besoins de chaque projet. Ces extensions sont sous forme de cartes qui s'enfichent sur le dessus de la carte Arduino et s'appellent « shield ». De par l'aspect open source d'Arduino il existe un grand nombre de shield officiel et non officiel apportant des fonctionnalités très diverses.

Dans le cadre de notre projet nous allons nous intéresser aux cartes apportant une dimension « internet des objets » aux capteurs et actionneurs pilotés par le microcontrôleur Arduino. Par défaut, la carte Arduino standard la plus utilisée (le modèle « UNO ») ne peut communiquer avec d'autres ordinateurs ou cartes Arduino que par liaison série.

Le *tableau 3* présente des modules permettant de communiquer sur un réseau internet TCP/IP.

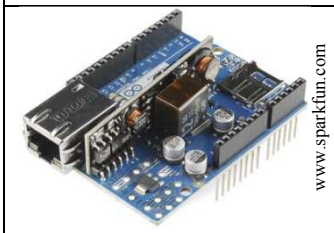
Shield	Descriptif
 **Shield Ethernet**	Cette extension permet à l'Arduino de communiquer sur un réseau Ethernet TCP/IP avec un débit théorique maximum de 100 mbps et ainsi d'être accessible à travers l'internet. De plus le shield Ethernet possède, en option, un module PoE (Power over Ethernet) permettant d'alimenter l'Arduino à travers le câble Ethernet s'il est relié à un appareil le permettant (ex : un switch).
 **Shield Wifi**	Cette extension permet à l'Arduino de communiquer sur un réseau TCP/IP sans fil wifi avec un débit théorique maximum de 56 mbps et ainsi d'être accessible à travers l'internet.
 **Shield GSM**	Cette extension permet à l'Arduino de communiquer sur un réseau mobile GSM (Global System for Mobile communications) pour envoyer et recevoir des appels ou des sms. Elle supporte aussi TCP/IP et permet d'avoir accès à internet avec un débit maximum de 85.6kbps.

Tableau 3 - Shields Arduino

Le « monde » Arduino possède tout ce dont nous avons besoin pour élaborer un système pouvant apporter une AmI. En effet, ces cartes prennent en compte un grand nombre de capteurs et d'actionneurs dialoguant avec les principaux standards de l'électronique. De plus par le biais des extensions elles peuvent être implantées dans un réseau filaire ou mobile.

Les cartes Arduino représentent l'élément central de notre projet autour duquel nous allons, avec plusieurs techniques, établir un proof of concept. Ce proof of concept vise à démontrer la faisabilité de créer un système générique permettant d'élaborer une AmI suivant plusieurs degrés d'abstraction s'adressant à plusieurs types d'utilisateurs.

2.3 Techniques et fonctionnalités retenues pour le proof of concept

2.3.1 Nœud de capteurs et d'actionneurs

L'Arduino est l'informatique embarquée qui par sa petite taille se fond dans l'environnement pour apporter une intelligence ambiante. Nous le transformons en un nœud de capteurs et d'actionneurs capable de récupérer les données en provenance d'un large panel de composants et de les mettre à disposition. Le nœud est interrogeable pour obtenir les données brutes en provenance de nos composants. Nous pouvons à partir de la même carte faire fonctionner une LED, récupérer les variations d'une tension à partir d'un capteur de pression ou de lumière ou encore récupérer l'identifiant d'un transpondeur RFID.

Ce nœud apporte le premier niveau d'abstraction dans la communication avec les objets de notre intelligence ambiante grâce à son microcontrôleur. Nous avons une première interprétation des données physiques provenant des objets.

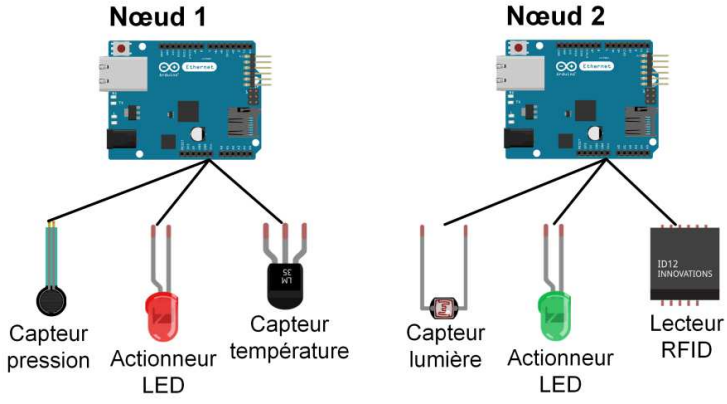


Figure 4 - Nœud de capteurs et d'actionneurs

2.3.2 Serveur centralisé

Le système comprend un moyen de récupérer les données de chaque nœud et de les traiter pour qu'elles soient interprétables par l'utilisateur. Pour cela nous utilisons un serveur centralisé qui agit comme une passerelle entre l'utilisateur et les nœuds de capteurs et d'actionneurs, avec une partie applicative apportant la communication entre le serveur et les Arduino ainsi que l'interprétation des résultats. Par exemple, les données en provenance d'un capteur de température sont interprétées par le serveur pour être délivrées dans plusieurs unités de valeurs telles Celcius ou Fahrenheit. De même pour un actionneur de type LED, le serveur renseigne l'utilisateur sur l'état on ou off de ce composant et apporte le moyen de changer cet état. Toutes les données des objets connectés aux Arduino qui sont traitées par ce serveur sont aussi journalisées de manière à garder un historique.

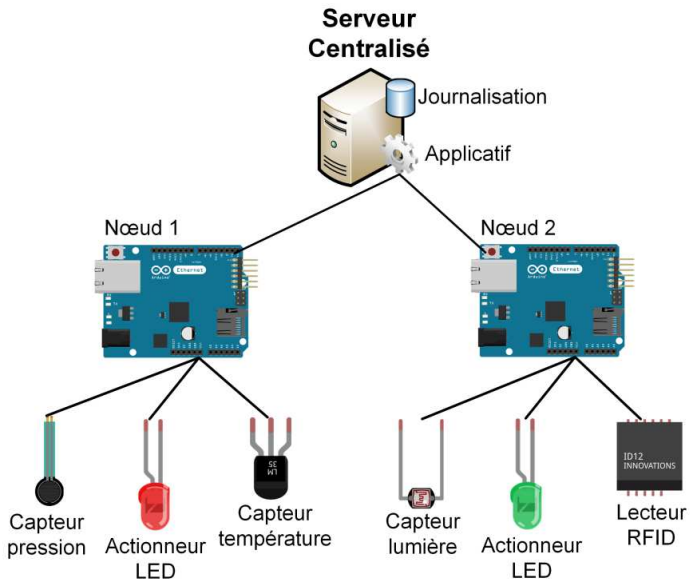


Figure 5 - Serveur centralisé

2.3.3 Multi-utilisateurs

Le système prend en charge plusieurs utilisateurs afin de permettre une utilisation des nœuds Arduino que ne soit pas exclusive. Le terme utilisateur décrit un compte utilisateur qui est utilisé pour une mise en place d'une intelligence ambiante dans le contexte d'un projet. Les informations mises à disposition par le serveur centralisé en provenance de différents objets connectés aux Arduino peuvent être utilisées par plusieurs comptes en même temps. On peut imaginer qu'un capteur de température est utilisé sur différentes applications en même temps,

pour par exemple informer de la température d'une pièce pour l'une, prévoir un déclenchement d'action en fonction d'un certain seuil de température pour une autre.

Le serveur centralisé a un rôle de gestionnaire d'accès aux nœuds par les utilisateurs et prend en charge l'authentification de ceux-ci. Ainsi, une fois authentifié, chaque utilisateur a alors accès à tous les nœuds qui lui sont attribués ainsi qu'à l'historique de toutes les données récupérées de chaque objet avec lequel il aura interagi.

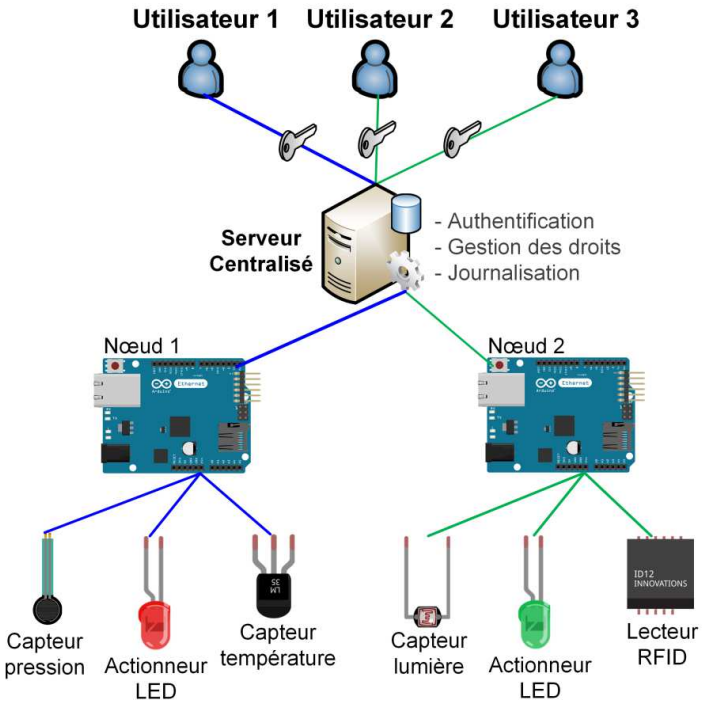


Figure 6 - Multi-utilisateurs

2.3.4 Réseau TCP/IP

Un des points les plus importants pour la mise en place d'une intelligence ambiante est la communication. En effet pour que nos objets soient communicants, ils doivent utiliser un réseau qui les relie entre eux et avec les utilisateurs. L'utilisation du Web permet d'obtenir cette information diffuse à très grande échelle car il est utilisé partout dans le monde par un large panel de type d'utilisateur. Le choix des protocoles utilisés pour le transfert de données afin de communiquer sur le Web est TCP/IP. Notre système est donc conçu pour fonctionner sur ces protocoles. Les utilisateurs peuvent accéder au serveur centralisé par un réseau TCP/IP privé, avec la possibilité d'y accéder à travers le Web si le serveur centralisé y est relié. Les

échanges entre le serveur et les nœuds Arduino se font aussi à travers un réseau TCP/IP. Cela permet de facilement disséminer nos informatiques embarquées dans plusieurs endroits qui peuvent être très éloignés en profitant de l'accès, soit par connexion sans fil soit filaire, à ce type de réseau très démocratisé. Il est quand même préférable d'utiliser un réseau privé entre le serveur centralisé et les cartes Arduino pour éviter que tout le monde puisse accéder aux nœuds Arduino.

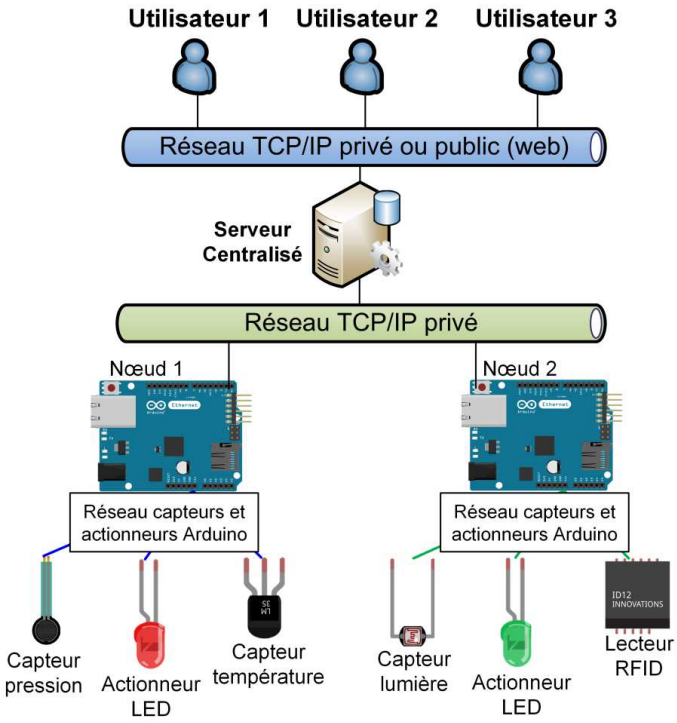


Figure 7 - Communication du système

2.3.5 API

Pour que les utilisateurs puissent utiliser la partie applicative du serveur centralisé afin de dialogueur avec les composants proposées par les nœuds Arduino à travers le Web, une API (Application Programming Interface ou interface de programmation) est mise en place. Elle met à disposition tous les outils nécessaires pour interagir avec les capteurs et actionneurs et ainsi élaborer une intelligence ambiante. Chaque composant de type capteur ou actionneur devient un objet intelligent pouvant communiquer par le biais de l'API à travers le Web.

Un appel de l'API engendre un appel des méthodes de l'applicatif. Cela permet de rendre plus facile l'utilisation de ces méthodes, sans avoir de connaissance du langage de développement

avec lequel elles sont construites. Surtout, cela donne à l'utilisateur des outils pour développer lui-même une application autour des objets communicants du système et ainsi intégrer la notion d'intelligence ambiante dans son projet.

Les fonctionnalités de l'API sont prévues, en plus de l'interaction avec les objets connectés, pour gérer la partie administrative du système. Par exemple, des outils sont mis à disposition pour gérer l'intégration des nœuds Arduino dans le système, mais aussi pour gérer les utilisateurs avec leurs créations, suppressions et leurs droits d'accès aux nœuds. Cette API apporte le deuxième niveau d'abstraction du système pour interagir avec les objets communiquant proposés par les nœuds Arduino.

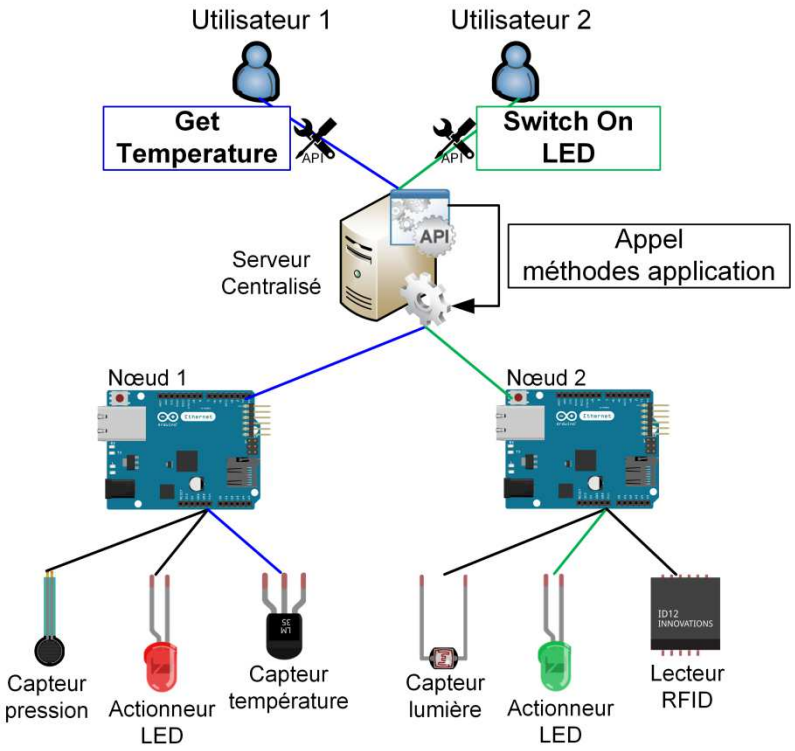


Figure 8 - API

2.3.6 Programmation Arduino assistée

Pour simplifier la préparation d'un nœud qui nécessite des connaissances en langage Arduino ainsi que des notions en électronique, un fichier « modèle » de code est mis à disposition. Ce fichier est générique et sert de base à la configuration des interfaces acceptant les différents objets à rendre communicants ainsi que pour la configuration de l'interface réseau. Il est

accompagné d'une documentation présentant les différents capteurs et actionneurs pris en charge par le système et la manière de les connecter. Ensuite il ne reste qu'à envoyer le code vers la carte Arduino pour finaliser la mise en place du nœud.

Cet aspect s'adresse à la personne en charge de l'administration du système car il faut définir l'adressage IP de chaque nœud pour qu'il s'intègre dans le système d'intelligence ambiante mis en place.

2.3.7 End-user Programming

Le système apporte un troisième niveau d'abstraction de l'utilisation des capteurs et actionneurs par le biais d'une interface « end-user programing ». Cette interface sous forme d'une programmation visuelle utilisant l'API rend la création d'un environnement intelligent à base d'objets connectés simple et ne nécessitant pas de connaissances en programmation. Cette interface proposée par le serveur centralisé est créée au format Web pour rester dans la logique d'un système fonctionnant dans ce grand réseau distribué.

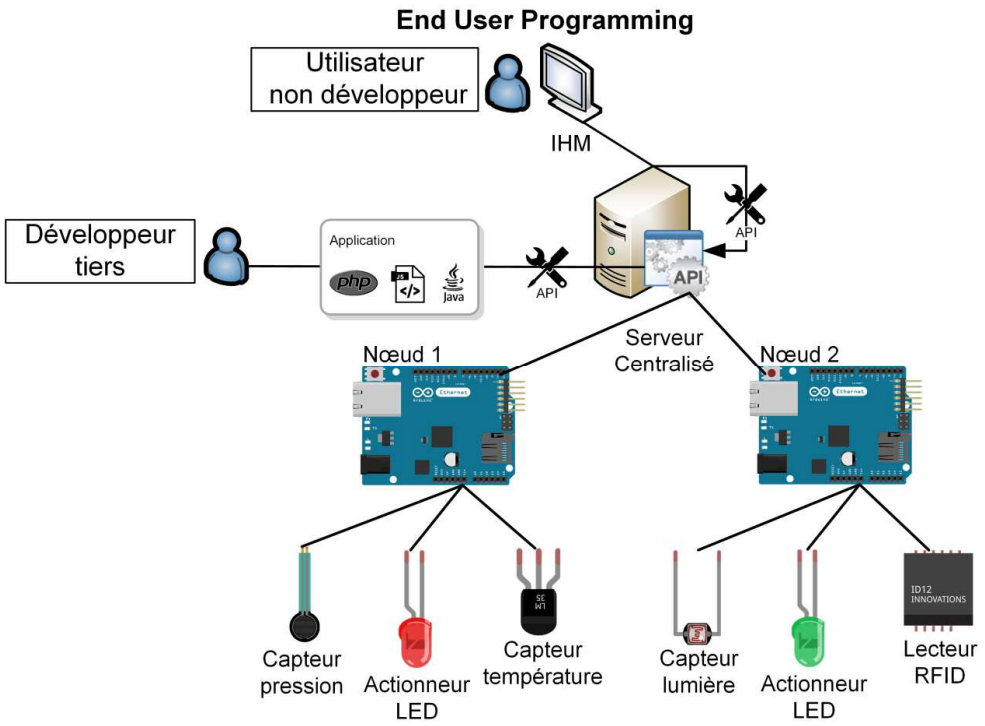


Figure 9 - End User Programming

2.4 Web des objets et REST

2.4.1 Présentation

Le Web des objets désigne l'intégration d'appareils électroniques avec lesquels on peut communiquer pour récupérer des données ou encore pour les contrôler à distance à travers le World Wide Web. Au niveau des fonctionnalités, notre proof of concept apporte une couche réseau et transport avec les protocoles TCP/IP pour permettre une communication entre les objets capteurs et actionneurs, les utilisateurs et le serveur centralisé. On parle ici de notion d'internet des objets : nos objets du monde physique échangent des informations à travers un réseau internet.

Avec l'utilisation du Web pour nos objets connectés, on rajoute une couche application permettant d'apporter la notion de service à travers le Web. Sur ce grand réseau distribué, nous pouvons proposer des services pour interagir avec les objets de nœuds Arduino en utilisant les principales technologies du Web telles que le protocole HTTP. Le but est de créer une architecture autour de nos objets en ce sens. Le développement de notre API a pour but d'offrir des services à travers le Web qui sont facilement intégrables dans un développement tiers. Pour cela nous utilisons le style d'architecture REST appliqué au Web.

2.4.2 Architecture REST

2.4.2.1 Définition

REST est l'acronyme de « REpresentational State Transfer » créé par Roy T. Fielding en 2000 dans sa thèse de doctorat « Architectural Styles and the Design of Network-based Software Architectures ». Il y décrit un style d'architecture logicielle pour systèmes hypermédia distribués. L'idée centrale de REST est la notion de ressource décrivant n'importe quelle information d'une application que l'on peut nommer et accessible via un identifiant [FIELDING – 2000]. Ces ressources peuvent être physiques, comme les objets connectés aux nœuds Arduino ou une personne, mais aussi abstraites comme une ressource décrivant une collection de ressources. Pour qu'une architecture soit dite REST, elle doit respecter plusieurs contraintes définies par son concepteur.

Le style d'architecture REST est grandement utilisé dans la création d'applications orientées services à travers le Web avec le couple URIs (Uniform Resource Identifier) et protocole HTTP. En effet, pour la conception d'une architecture REST, ces deux standards répondent à toutes les contraintes énoncées par Roy T. Fielding dont nous présentons les plus importantes.

2.4.2.2 Contraintes

Identification de ressource

Chaque ressource de notre application doit être identifiée uniquement. Dans le cas de REST appliqué au Web, nous utilisons les URIs qui vont nous servir pour cette identification unique. Une ressource de l'application est ainsi identifiée avec une URI au même titre qu'une page web, une vidéo ou une image.

Interface uniforme

Les ressources sont accessibles via les URIs grâce à une interface uniforme utilisant une sémantique définie pour interagir avec elles. Cette sémantique doit être la même pour tout type d'application architecturée avec REST. L'utilisation du protocole HTTP apporte cette interface avec quelques méthodes utilisées pour l'interaction avec les ressources. Cela permet un découpage entre l'interface et l'implémentation des services. Ainsi il est possible de mettre à jour les fonctions apportant le service sans avoir à changer l'interface. Cette interface uniforme, avec HTTP, utilise 4 méthodes principales que l'on appelle aussi verbes :

Verbes HTTP (REST)	CRUD	Utilisation
POST	Create	Création d'une nouvelle ressource
GET	Read	Récupération de l'état d'une ressource
PUT	Update	Mise à jour de l'état d'une ressource
DELETE	Delete	Suppression d'une ressource

Tableau 4 - Méthodes HTTP de l'interface uniforme REST

Ces quatre méthodes s'appliquent parfaitement dans le cadre d'une application CRUD (Create, Read, Update, Delete) ou l'interface permet de définir clairement les actions en base de données.

Messages auto-descriptifs

A chaque interaction avec une ressource par l'interface uniforme, une représentation de l'état actuel de la ressource est renvoyée par le serveur au client par le protocole HTTP. Ce message est dit « auto-descriptif » car il contient toute l'information nécessaire pour être interprété par le client. Ceci est rendu possible par les métadonnées présentes dans l'en-tête du message HTTP. Par exemple la métadonnée « Content-Type » indique dans quel langage interpréter le message renvoyé.

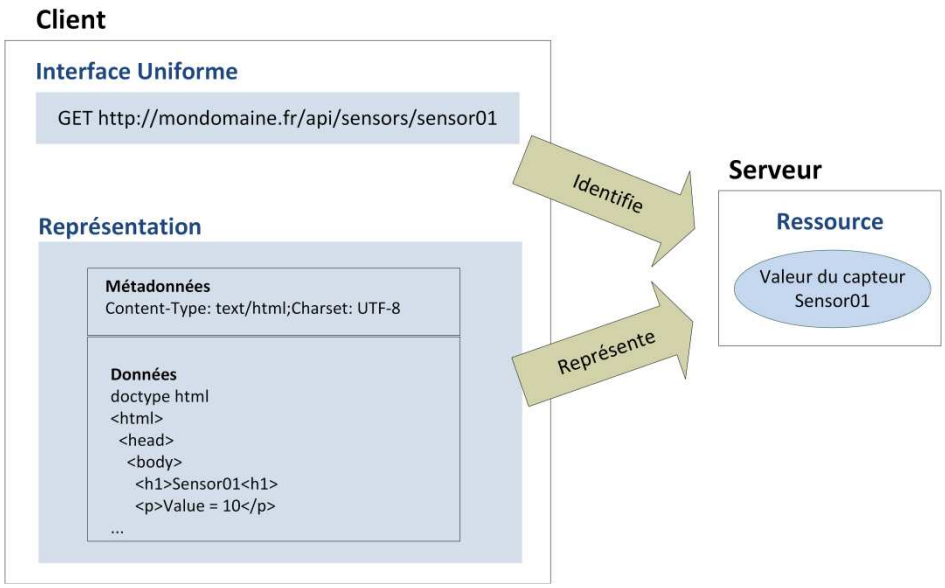


Figure 10 - Représentation REST

La *figure 10* décrit l'interrogation d'un capteur identifié par l'URI `api/sensors/sensor01` avec le verbe `GET` pour en récupérer sa valeur. Le serveur fait le lien entre l'interface uniforme REST et la méthode interne pour récupérer la valeur du capteur et ensuite en crée une représentation. Cette représentation est envoyée sous forme de texte avec la métadonnée `Content-Type : text/html` qui indique au client que les données du corps de la réponse sont à interpréter en HTML 5. Le navigateur peut alors afficher le résultat dans le langage attendu et cela avec un seul échange.

Hypermedia as the engine of the application state (HATEOAS)

Cette contrainte signifie en français l'hypermédia comme moteur d'état de l'application. C'est à dire que l'application fournit aux clients des liens hypermédia pour trouver les ressources et interagir avec leurs services. C'est dans la représentation d'une ressource que nous trouvons ces liens qui indiquent comment naviguer dans l'application à travers l'ensemble des

ressources. Comme dans le cas d'un site web indiquant les liens pour changer de pages et accéder à de nouvelles ressources. En respectant cette contrainte, l'API REST devient pratiquement auto-documenté.

Par exemple l'accès à la ressource `/api/capteurs/` doit nous donner dans sa représentation les liens pour accéder à la représentation de chaque capteur de l'application.

2.4.2.3 Intérêt de REST

L'intérêt principal du style d'architecture REST est qu'il a été pensé pour construire une application sur un système hypermédia distribué. Il s'adapte donc bien au Web grâce au duo URIs et le protocole HTTP. Avec l'interface uniforme basé sur l'utilisation des verbes HTTP, l'accès au service est simple, les interactions avec les ressources intuitives, et facilement compréhensible sans documentation.

Pour notre projet, cette architecture nous permet de rendre nos capteurs et actionneurs communicants en les transformant en ressources accessibles via une interface uniforme au même titre que des ressources « standards » du Web.

2.5 Synthèse

Nous avons identifié les technologies et fonctionnalités que doit comporter notre proof of concept pour proposer un système d'intelligence ambiante. L'aspect important de celui-ci est de rendre les objets connectés aux Arduino communicants en proposant plusieurs niveaux d'abstraction pour interagir avec eux suivant le type d'utilisation souhaité. Le type d'utilisation le plus important étant sous forme d'API, pour offrir des outils à des développeurs leurs permettant d'intégrer des objets connectés dans leurs développements d'applications.

L'élaboration de cette API nécessite d'abord d'établir les fonctionnalités applicatives que doit couvrir le serveur centralisé. Ensuite nous pouvons la concevoir avec l'architecture REST en établissant l'interface uniforme qui fera appel à ces fonctionnalités.

Chapitre 3 : Contenu du projet

3.1 Modélisation de l'applicatif

Nous établissons le socle applicatif de notre serveur centralisé sur lequel va se greffer notre API. Pour cela nous utilisons la modélisation orientée objet pour élaborer un diagramme de classes et un modèle entité-relation pour concevoir notre base de données.

3.1.1 Définition des objets

3.1.1.1 Arduino

L'objet **Arduino** représente une carte microcontrôleur Arduino qui sert de nœud de capteurs et d'actionneurs. Ses propriétés définissent son nom, son adresse IP et sa localisation pour savoir où se trouve physiquement la carte. Il possède des méthodes pour ajouter ou retirer un composant. Ces méthodes représentent de manière conceptuelle le choix des composants que comporte l'Arduino au moment de son paramétrage avec le fichier « modèle » de code ainsi que l'ajout manuel de ces composants.

Méthode	Description
addComponent()	Ajout d'un composant au niveau de la configuration d'un Arduino avec le fichier de code « modèle »
deleteComponent()	Suppression d'un composant au niveau de la configuration d'un Arduino avec le fichier de code « modèle »
updateComponent()	Mise à jour d'un composant au niveau de la configuration d'un Arduino avec le fichier de code « modèle »

Tableau 5 - Méthodes de l'objet Arduino

3.1.1.2 Component

L'objet **Component** représente un capteur ou un actionneur qui est rattaché à un Arduino. Ce composant possède les propriétés nom, type définissant si c'est un composant *sensor* (capteur) ou *actuator* (actionneur) et une propriété sous-type pour identifier à quelle famille de capteur ou d'actionneur il appartient (exemple : LED, capteur de pression, capteur de température). L'objet component ne comporte pas de méthodes.

3.1.1.3 Method

L'objet **Method** représente une méthode de traitement qui s'applique à un composant d'un Arduino. Cette méthode est définie par un nom, la famille de composant auquel elle s'applique (par exemple LED) et enfin le type de méthode. Ce type peut être *standard* pour définir une méthode qui retourne un résultat suite à une demande d'exécution utilisateur ou alors le type peut être *event* pour définir un envoi direct d'un résultat sous forme de notification. L'objet comporte des méthodes pour interagir avec le composant, traiter leurs données, envoyer le résultat de ce traitement à l'utilisateur et enfin enregistrer ce résultat en base de données.

Méthode	Description
getComponentData()	Récupération des données d'un composant généralement de type capteur pour ensuite pouvoir les traiter
sendComponentData()	Envoie de données généralement à destination d'un composant de type actionneur pour le piloter
compute()	Traitement des données d'un composant
sendResult()	Envoie du résultat traité à l'utilisateur
saveResultToDb()	Sauvegarde du résultat traité en base de données dans la partie journalisation

Tableau 6 - Méthodes de l'objet Method

3.1.1.4 User

L'objet **User** représente un compte utilisateur du système. Ce compte utilisateur est défini par un nom, un mot de passe et des droits d'accès aux nœuds. Cet objet possède des méthodes pour récupérer les informations des nœuds Arduino, obtenir la liste des composants et de leurs méthodes respectives. Une méthode est prévue pour exécuter une méthode de traitement

appliquée à un composant. Enfin il possède aussi des méthodes pour consulter l'historique des résultats renvoyés par les méthodes de traitement qu'il a exécutées et pour les supprimer.

Méthode	Description
isAuthorized()	Vérifie si l'utilisateur a le droit d'accéder à un nœud Arduino donné
getArduino()	Récupère toutes les informations d'un ou plusieurs nœuds Arduino dont l'utilisateur a les droits
getComponent()	Récupère toutes les informations concernant un ou plusieurs composants d'un nœud Arduino dont l'utilisateur a les droits
getMethod()	Récupère toutes les informations concernant une ou plusieurs méthodes pour pouvoir ensuite en exécuter une ou la mettre à jour
updateMethod()	Mise à jour des paramètres de la méthode pour changer son mode de traitement des informations ou son pilotage d'un composant
executeMethod()	Exécution d'une méthode d'un composant pour par exemple récupérer des données ou actionner ce composant.
getHistoric()	Récupération dans l'historique, d'un résultat d'une méthode d'un composant exécuté par l'utilisateur
deleteHistoric()	Suppression dans l'historique, d'un résultat d'une méthode d'un composant exécuté par l'utilisateur

Tableau 7 - Méthodes de l'objet User

3.1.1.5 Historic

L'objet **Historic** représente une sauvegarde en base de données d'un résultat retourné par une méthode. Il est défini par un id, une date d'exécution de la méthode et le résultat de cette méthode. Cet objet ne possède pas de méthodes

3.1.1.6 Administrator

L'objet **Administrator** représente l'administrateur du système. Il est défini par un nom et un mot de passe. Il possède plusieurs méthodes pour manager les nœuds Arduino comme leurs référencements dans le système (création d'un objet arduino), leurs mises à jour et leurs suppressions du système. De même il possède des méthodes pour la gestion des utilisateurs avec la création et la suppression de ceux-ci dans le système ainsi que de l'attribution de droits d'accès aux Arduino.

Méthode	Description
addArduino()	Ajoute au système un nœud Arduino préalablement configuré
updateArduino()	Met à jour un nœud Arduino dans le système suite à une reconfiguration de celui-ci
deleteArduino()	Suppression d'un nœud Arduino du système
getArduino()	Récupération de toutes les informations d'un nœud Arduino
enabledArduinoComponent()	Activation d'un composant d'un nœud Arduino permettant aux utilisateurs de pouvoir interagir avec
disabledArduinoComponent()	Désactivation d'un composant d'un nœud Arduino interdisant aux utilisateurs d'interagir avec. Utile en cas d'opération de maintenance sur un composant
createUser()	Création d'un compte utilisateur
deleteUser()	Suppression d'un compte utilisateur
getUser()	Récupération de toutes les informations d'un compte utilisateur
addArduinoToUser()	Ajout des droits à un utilisateur sur un nœud Arduino pour qu'il puisse interagir avec tous les objets composants connectés et activés
deleteArduinoToUser()	Suppression des droits à un utilisateur sur un nœud Arduino. Par conséquent il ne pourra pas interagir avec ses les objets composants connectés

Tableau 8 - Méthodes de l'objet Administrator

3.1.2 Diagramme de classes

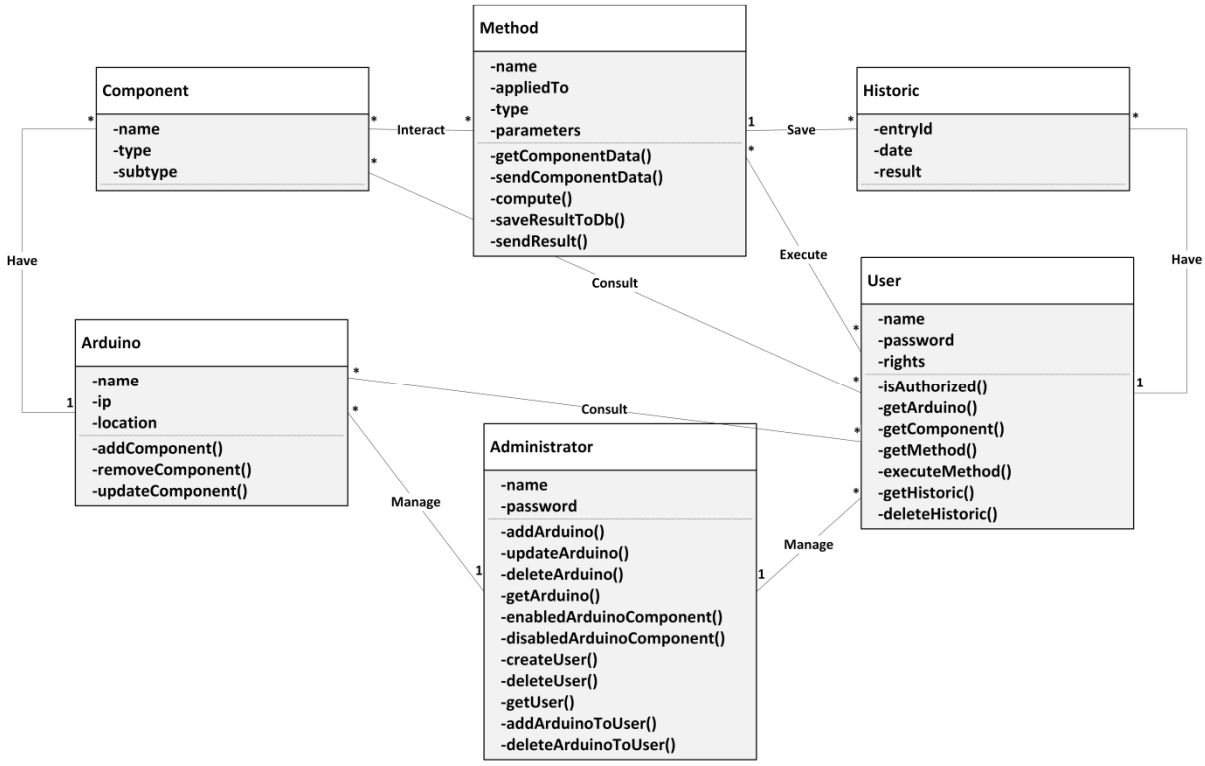


Figure 11 - Digramme de classes

La réflexion sur les objets et leurs méthodes de l'applicatif du serveur nous permet de créer ce diagramme de classe de la *figure 11* dont nous détaillons les relations entre les classes :

- La relation *Manage* entre la classe **Administrator** et **Arduino** sert à indiquer qu'un objet Administrator instancié par la classe **Administrator** gère tous les nœuds Arduino, représentés dans l'applicatif par des objets Arduino. Ces objet Arduino sont instanciés par l'action d'un objet Administrator avec la méthode *addArduino()* mais aussi supprimés ou encore mis à jour par d'autres de ses méthodes.
- La relation *Have* entre la classe **Arduino** et la classe **Component** sert à indiquer qu'un objet Arduino peut avoir plusieurs composants. Cela permet de décrire qu'un nœud physique Arduino a plusieurs composants de type *sensor* ou *actuator* reliés à lui.
- La relation *Interact* entre la classe **Component** et la classe **Method** montre qu'une méthode peut s'exécuter sur plusieurs composants et qu'un composant peut avoir plusieurs méthodes de traitement qui s'appliquent à lui. Un objet Method va pouvoir traiter les informations provenant de plusieurs objets component si leurs propriétés *subtype* sont de même valeur que la propriété *appliedTo* de l'objet Method. Une méthode s'applique donc seulement sur une certaine famille de composant.
- La relation *Save* entre la classe **Method** et la classe **Historic** décrit le fait qu'un objet Method peut instancier plusieurs objet Historic lors de l'exécution de sa méthode *saveResultToDb()*. Cela décrit la fonctionnalité de journalisation par le serveur centralisé.
- Les trois relations entre la classe **User** et les classes **Arduino**, **Componen**t et **Method** ne sont possibles que lorsqu'un objet user a les droits sur l'objet Arduino concerné. Une vérification est effectuée avec la méthode *isAuthorized()* de l'objet user.

  La relation *Consult* entre la classe **User** et **Arduino** décrit le fait qu'une fois l'autorisation obtenue, il peut consulter un objet Arduino. Ensuite l'objet user peut consulter les objets Component d'un objet Arduino ainsi que les objets Method d'un objet Component. Enfin il pourra initier l'exécution des différentes méthodes d'un objet Method (par exemple : *getComponentData()*, *compute()*, *saveResultToDb()* et *sendResult()*) pour interagir avec un composant physique d'un nœud Arduino, d'où la relation *Execute*.
- La relation *Have* entre la classe **User** et la classe **Historic** indique qu'un objet User possède plusieurs objets Historic. Cela décrit l'accès de chaque utilisateur à l'historique de toutes les interactions qu'il a effectuées sur les composants.

3.1.3 Diagramme entité-relation

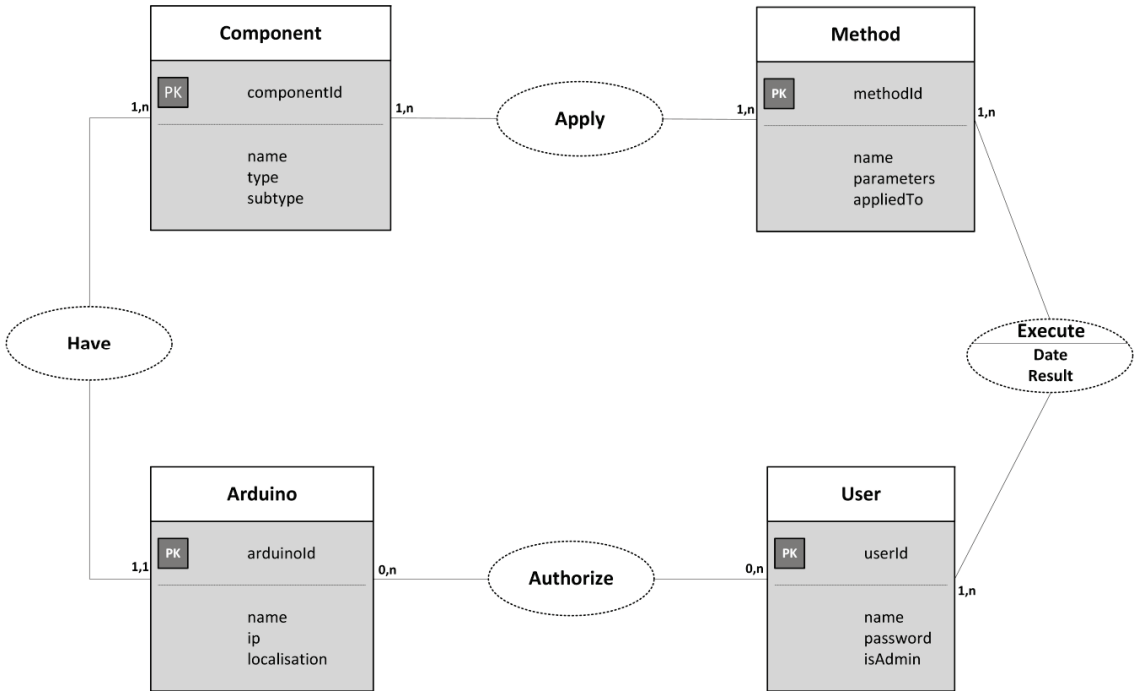


Figure 12 - Modèle entité-relation

Nous remarquons dans le modèle entité-relation de la *figure 12* qu'il n'y a pas d'entités **Administrator**, car un administrateur du système est une occurrence de la table **User** avec la propriété booléenne *isAdmin* de valeur *true*. Lors de l'authentification d'un utilisateur, la base de données est interrogée et si l'occurrence utilisateur a cette propriété alors il bénéficie des droits administrateur.

Pour mieux comprendre ce modèle entité-relation nous en détaillons les relations :

- La relation *Apply* a des cardinalités *n,n* entre les entités **Component** et **Method** car une méthode peut s'appliquer sur plusieurs composants et sur un composant peut s'appliquer plusieurs méthodes. Il en découle la création d'une entité **Apply** regroupant les clés primaires des entités **Component** et **Method**.
  Il en est de même entre les entités **User** et **Arduino** avec la relation *Authorize*.
- L'association *Execute* a aussi des cardinalités *n,n* entre les entités **User** et **Method** ce qui engendre la création d'une entité **Execute** dans laquelle nous avons les clés primaires de ces entités, mais avec en plus les propriétés *date* et *result*. Ceci décrit la création d'une entrée en base pour la fonctionnalité de journalisation. Lorsqu'un utilisateur cherche en base un résultat d'une méthode qu'il a exécutée, c'est cette entité créé par cette relation *Execute* que l'on interroge.

3.2 Architecture de l'API

3.2.1 Identification des ressources

Pour la conception de notre API REST, nous devons identifier les ressources à manipuler, celles-ci pouvant représenter des objets physiques ou virtuels. Ensuite nous définissons nos différentes URIs et les différents verbes HTTP pour interagir avec ces ressources. C'est ce qui constitue notre interface uniforme. Ces ressources sont hiérarchisées et constituent une arborescence dans laquelle naviguer. Pour nommer nos ressources et garder une logique dans cette hiérarchie nous utilisons le choix de noms au singulier pour identifier une ressource et de noms au pluriel pour identifier une collection de ressources.

Une fois nos ressources définies et identifiées par des URIs, nous pouvons faire correspondre ces URIs avec les méthodes applicatives décrites lors de la modélisation objet. Les méthodes applicatives sont appelées suite à une utilisation de l'API.

3.2.1.1 Administration Arduino

La ressource de type collection nommée **admin** regroupe toutes les ressources faisant partie de l'aspect gestion des Arduino et des utilisateurs de l'API. Elle est identifiée par l'URI `/admin`. L'accès aux ressources se fait en suivant une arborescence qui va constituer nos URIs pour gérer les nœuds Arduino et les utilisateurs.

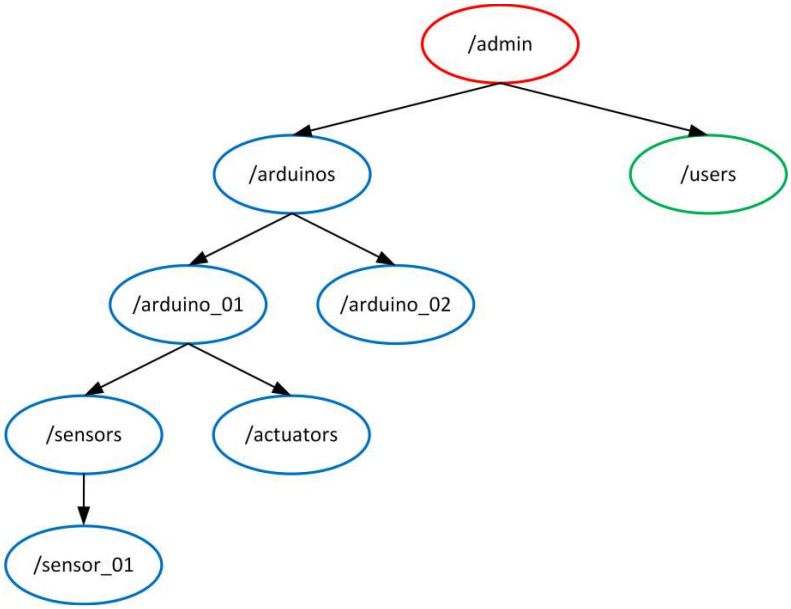


Figure 13 - Hiérarchisation administration Arduino

La *figure 13* détaille la ressource collection nommée **arduinos** identifiée par l'URI `/admin/arduinos`. C'est à partir de cette URI que sont appelées les méthodes applicatives pour ajouter, mettre à jour ou supprimer un Arduino dans le système et aussi récupérer les informations des Arduino déjà référencés. Le *tableau 9* présente l'ensemble des URIs de l'API REST pour la partie gestion administrative des Arduino.

Verbe http	URI	Méthode Applicative
POST	/admin/Arduinos	addArduino()
GET	/admin/Arduinos	getArduino()
GET	/admin/Arduinos/{nom_Arduino}	getArduino()
PUT	/admin/Arduinos/{nom_Arduino}	updateArduino()
DELETE	/admin/Arduinos/{nom_Arduino}	deleteArduino()
GET	- /admin/Arduinos/{nom_Arduino}/sensors - /admin/Arduinos/{nom_Arduino}/actuators	getComponent()
GET	- /admin/Arduinos/{nom_Arduino}/sensors/{nom_sensor} - /admin/Arduinos/{nom_Arduino}/actuators/{nom_actuator}	getComponent()
POST	- /admin/Arduinos/{nom_Arduino}/sensors/{nom_sensor} - /admin/Arduinos/{nom_Arduino}/actuators/{nom_actuator}	enabledComponent()
DELETE	- /admin/Arduinos/{nom_Arduino}/sensors/{nom_sensor} - /admin/Arduinos/{nom_Arduino}/actuators/{nom_actuator}	disabledComponent()

Tableau 9 - Interface uniforme : administration Arduino

3.2.1.2 Administration utilisateurs

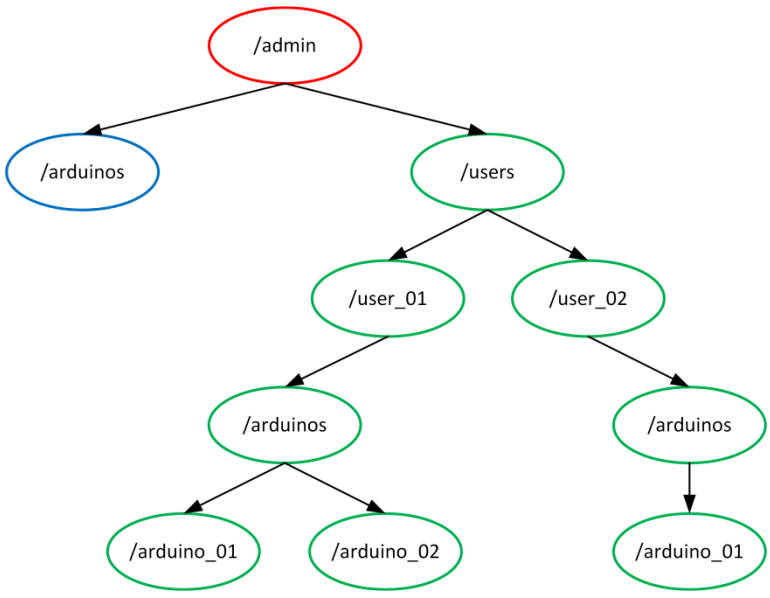


Figure 14 - Hiérarchisation administration utilisateur

La *figure 14* détaille la ressource de type collection nommée **users** identifiée par l'URI `/admin/users`. C'est à partir de cette URI que sont appelées les méthodes applicatives pour ajouter ou supprimer un utilisateur dans le système. Dans la hiérarchie, une ressource *user* possède une collection de ressources nommée **arduinos**. Cela correspond aux ressources **arduino** dont l'utilisateur a les droits pour interagir avec ses objets.

Le *tableau 10* présente l'ensemble des URIs de l'API REST pour la partie gestion administrative des utilisateurs.

Verbe HTTP	URI	Méthode Applicative
POST	*/admin/users*	createUser()
GET	*/admin/users*	getUser()
GET	*/admin/users/{nom_user}*	getUser()
DELETE	*/admin/users/{nom_user}*	deleteUser()
POST	*/admin/users/{nom_user}/Arduinos*	addArduinoToUser()
DELETE	*/admin/users/{nom_user}/Arduinos/{Arduino_name}*	deleteArduinoToUser()

Tableau 10 - Interface uniforme : administration utilisateurs

3.2.1.3 Gestion composants

Cette partie de l'API concerne la manipulation des composants par les utilisateurs. En partant du principe qu'ils sont authentifiés, la *figure 15* décrit un exemple de hiérarchisation de ressources pour un utilisateur.

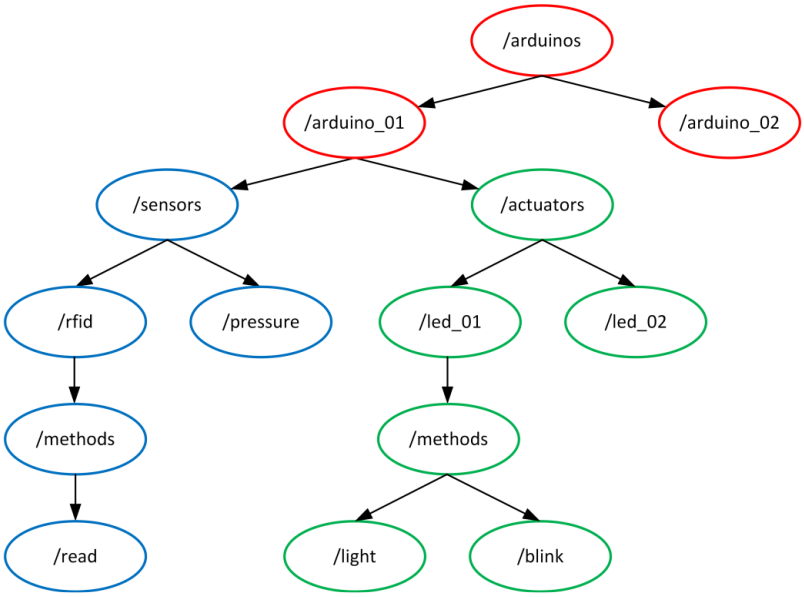


Figure 15 - Hiérarchisation composants

La collection permettant de gérer les Arduino d'un utilisateur et par conséquent de leurs objets, est identifiée par l'URI `/arduinos`. A partir de ce point d'entrée, l'arborescence comprend la liste de toutes les ressources **arduino** qui elles-mêmes contiennent des collections regroupant les ressources **sensor** et **actuator**. Chaque ressource **sensor** ou **actuator** possède une collection **methods** comprenant les ressources méthodes que propose un composant pour interagir avec lui. Le *tableau 11* détaille l'ensemble des URIs mises à disposition des utilisateurs pour manipuler les objets d'un nœud Arduino.

Méthode http	**URI**	**Méthodes Applicative**
GET	*/Arduinos*	getArduino()
GET	*/Arduinos/{nom_Arduino}*	getArduino()
GET	- */Arduinos/{nom_Arduino}/sensors* - */Arduinos/{nom_Arduino}/actuators*	getComponent()
GET	- */Arduinos/{nom_Arduino}/sensors/{nom_sensor}* - */Arduinos/{nom_Arduino}/actuators/{nom_actuator}*	getComponent()
GET	- */Arduinos/{nom_Arduino}/sensors/{nom_sensor}/methods* - */Arduinos/{nom_Arduino}/actuators/{nom_actuator}/methods*	getMethod()
GET	- */Arduinos/{nom_Arduino}/sensors/{nom_sensor}/methods/{nom_method}* - */Arduinos/{nom_Arduino}/actuators/{nom_actuator}/methods/{nom_method}*	getMethod() executeMethod() getData() ou sendData() compute() saveResultToDb() sendResult()
PUT	- */Arduinos/{nom_Arduino}/sensors/{nom_sensor}/methods/{nom_method}* - */Arduinos/{nom_Arduino}/actuators/{nom_actuator}/methods/{nom_method}*	updateMethod() executeMethod() getData() ou sendData() compute() saveResultToDb() sendResult()

Tableau 11 - Interface uniforme : composants

Pour agir physiquement sur un objet d'un nœud Arduino nous utilisons les URIs des deux dernières colonnes du *tableau 11* avec la méthode `GET` ou `PUT`. `GET` est plutôt utilisé pour obtenir la valeur d'un objet et `PUT` pour modifier le fonctionnement d'un objet. Ces deux URIs impliquent l'exécution de plusieurs méthodes applicatives que nous avons décrites au *3.1 Modélisation de l'applicatif* avant de retourner un résultat à l'utilisateur.

Prenons l'exemple d'un actionneur LED sur un nœud Arduino nommé arduino_01 que nous voudrions allumer. Il faut que l'utilisateur ayant le droit d'interagir avec l'arduino_01 utilise

l'API du système avec la requête `PUT /Arduinos/Arduino_01/actuators/led_01/methods/light`.

3.2.1.4 Journalisation

Comme pour la gestion des composants, cette partie de l'API est accessible seulement à un utilisateur authentifié au système. Une fois authentifié, l'utilisateur a accès à une hiérarchisation de ressources **historic** lui appartenant, composées des résultats des méthodes composant qu'il a exécuté antérieurement. La *figure 16* présente un exemple de hiérarchisation de ressources **historic** d'un utilisateur.

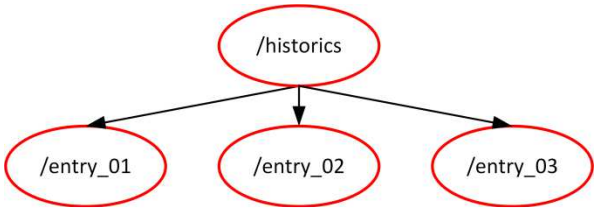


Figure 16 - Hiérarchisation historique

Le *tableau 12* détaille l'ensemble des URIs mis à disposition de l'utilisateur pour manipuler ces ressources **historic**.

Méthode HTTP	URI	Méthode Applicative
GET	*/historics*	getHistoric()
GET	*/historics/{id_historic}*	getHistoric()
DELETE	*/historics*	deleteHistoric()
DELETE	*/historics/{id_historic}*	deleteHistoric()

Tableau 12 - Interface uniforme : historique

3.2.2 Représentation des ressources

Dans une architecture REST appliquée au Web, le client envoie une requête par le biais des URIs et du protocole HTTP pour obtenir une représentation de l'état d'une ressource. A cette étape de l'élaboration de notre API, nous définissons les informations qui doivent être présentes dans la représentation de l'état de nos ressources.

Ces représentations doivent respecter certaines contraintes de style d'architecture pour fournir des informations décrivant la navigation dans l'arborescence des ressources. C'est le principe du **HATEOAS** énoncé au *2.4.2.2 Contraintes* où nous indiquons à chaque ressource comment naviguer dans la hiérarchie en fournissant des liens.

Nous détaillons plusieurs représentations retournées par l'API concernant les ressources utilisateur et composant qui constituent les éléments les plus importants du système. Les représentations des autres ressources sont élaborées suivant les mêmes principes.

3.2.2.1 Représentations utilisateur

Dans le cas d'une ressource utilisateur pour la partie administration, lorsque l'on effectue une requête avec la méthode `GET` sur le point d'entrée `/admin/users`, nous récupérons la liste des utilisateurs du système. La représentation indique à la première ligne le nom de la collection et ensuite une partie nommée `content` contient les informations de chaque ressource. La *figure 17* décrit un exemple de représentation **Users** donnant la liste utilisateurs du système.

```
Users :
content :
            name=user_01
            name=user_02
            name=user_03
            name=user_04
            name=user_05
            name=user_06
```

Figure 17 - Représentation ressource Users

Suivant la contrainte HATEOAS, la représentation de la ressource **Users** doit nous donner les informations sous forme de liens pour connaitre les opérations possibles comme la possibilité de créer un utilisateur ou d'obtenir la ressource d'un utilisateur. Les liens sont présents dans une partie nommée `links` et chaque lien est défini par trois attributs. L'attribut `href` renseigne l'URI, `method` nous donne le verbe HTTP et `rel` donne une brève description du lien. Les attributs `href` et `rel` sont empruntés à HTML.

Les liens concernant la ressource en elle-même sont présents en dessous de son nom avant la partie `content` et les liens concernant le niveau enfant dans la hiérarchie sont présents dans la partie `content`.

La *figure 18* montre la représentation de la ressource collection **Users** retournée par la requête `GET admin/users` intégrant la contrainte HATEOAS

```
Users :
links :
            href = /admin/users
            method = POST
            rel= Create User
content :
            name=user_01
            links :
                  href = /admin/users/user1
                  method = GET
                  rel = Details User

                  href = /admin/users/user1
                  method = DELETE
                  rel = Delete User


            name=user_02
            links :
                  href = /admin/users/user2
                  method = GET
                  rel = Details User

                  href = /admin/users/user2
                  method = DELETE
                  rel = Delete User
```

Figure 18 - Représentation ressource Users avec contrainte HATOEAS

Avec les liens nous avons l'information nécessaire pour obtenir la représentation d'une ressource utilisateur ou pour supprimer une ressource utilisateur.

```
User :
links :
            href = /admin/users/user1
            method = DELETE
            rel= Delete User
content :
            name=user_01
            arduinos=[arduino_01,arduino_02]
            links :
                  href = /admin/users/user1/Arduinos
                  method = GET
                  rel = List Arduino User

                  href = /admin/users/user1/Arduinos
                  method = POST
                  rel = Add Arduino User
```

Figure 19 - Représentation ressource User

La *figure 19* montre la représentation de la ressource **user1** retournée par la requête `GET /admin/user/user1`. Cette représentation nous donne comme information importante les nœuds Arduino dont l'utilisateur possède les droits et le lien pour lui en ajouter de nouveaux.
Pour la suppression d'une ressource, l'API ne retourne pas de représentation. Le fonctionnement de l'API est de répondre à une requête concernant une ressource en donnant une représentation de son état. Lorsqu'il s'agit d'une ressource supprimée, elle ne peut en créer une représentation. Dans ce cas un message indiquant la bonne exécution de la requête est envoyé. La *figure 20* montre le retour envoyé suite à la requête `DELETE /admin/users/user_01`.

```
name = user_01
deleted = true
```

Figure 20 - Message de suppression de ressource

3.2.2.2 Représentations composant et méthode

La navigation dans notre API nous permet d'obtenir les représentations de nos objets Component et de connaitre les méthodes d'interactions qu'ils proposent. Au fil des différentes requêtes, l'utilisateur va pouvoir communiquer avec un objet proposé par un nœud Arduino.

Pour expliquer le cheminement des différentes représentations pour interagir physiquement avec un composant, nous prenons l'exemple d'une LED appelée **led_01** connectée au nœud **arduino_01**.

L'interrogation du système par le biais de l'API avec la requête `GET /arduinos/arduino_01/actuators` renvoie une représentation de la liste des objets de type *actuator* de l'Arduino **arduino_01**.

```
Actuators :
Content =
       name = led_01
       links :
              href = /Arduinos/Arduino_01/actuators/led_01
              method = GET
              rel = Details User

       name = led_02
       links :
              href = /Arduinos/Arduino_01/actuators/led_01
              method = GET
              rel = Details User
```

Figure 21 - Représentation ressource Actuators de l'arduino_01

Nous identifions dans la représentation décrite dans la *figure 21* que l'objet **led_01** de type *actuator* est bien présent sur le nœud **arduino_01** et nous pouvons donc récupérer la représentation de cette ressource avec la requête `GET /Arduinos/Arduino_01/actuators/led_01` pour obtenir plus d'informations sur cet objet.

```
Actuator :
Content :
       name = led_01
       subtype = LED
       methods = [blink,light]
       links :
              method = GET
              href = /Arduinos/Arduino_01/led_01/methods
              rel = List actuator method
```

Figure 22 - Représentation ressource led_01

La représentation de l'actionneur **led_01** de la *figure 22* nous indique les méthodes d'interactions proposées par l'objet et nous donne le lien pour les lister. Pour y accéder nous exécutons la requête `GET /arduinos/arduino_01/actuators/led_01/methods`.

```
Methods :
Content :
       name = light
       links :
              method = GET
              href = /arduinos/arduino_01/led_01/methods/light
              rel = Method Status

              method = PUT
              href = /arduinos/arduino_01/led_01/methods/light
              rel = Switch On/Off LED
              parameters :
                     name = action
                     value = on|off

       name = light
       links :
              method = GET
              href = /arduinos/arduino_01/led_01/methods/blink
              rel = Method Status

              method = PUT
              href = /arduinos/arduino_01/led_01/methods/blink
                     rel = Blink in seconds LED
                     parameters :
                            name = interval
                            value = integer
```

Figure 23 - Représentation ressource Methods de led_01

La représentation de la ressource **Methods** contenant la collection de méthodes de l'objet **led_01** de la *figure 23* nous montre qu'une méthode s'utilise de deux façons. Le lien avec la méthode `GET` nous renvoie le `status` de celle-ci. Le `status` regroupe les informations de la

méthode en fonction du composant sur laquelle elle s'applique. Le lien avec la méthode `PUT` permet de modifier la méthode ce qui change son `status`. Par exemple l'URI `GET /arduinos/arduino_01/led_01/methods/light` appelle une fonction applicative pour interroger la sortie digitale sur laquelle la LED est connectée. La représentation de la ressource méthode renvoyée par l'URI nous indique dans son `status` si la LED est allumée ou non.

```
Method :
       links :
              method = PUT
              href = /Arduinos/Arduino_01/led_01/methods/light
              rel = Switch On/Off LED
              parameters :
                     name = action
                     value = on|off
Content :
       name = light
       status = off
```

Figure 24 - Représentation ressource Method light

La représentation de la ressource **light** de la *figure 24* nous indique que la LED est `off`. Le lien nous donne la requête à effectuer pour allumer la LED et nous indique qu'elle nécessite un paramètre nommé *action* dont la valeur est soit *on* soit *off*. Ces paramètres désignent les paramètres contenus dans le corps d'une requête HTTP. Ce genre de paramètres est très utilisé dans les formulaires HTTP. Pour allumer la LED, nous exécutons donc la requête `PUT /arduinos/arduino_01/actuators/led_01/methods/ligth` avec le paramètre *action=on*. La représentation retournée sera la même que la *figure 24* mais avec la valeur `status` à `on`.

3.2.3 Communication des objets

Avant qu'un utilisateur ayant initié une interaction avec un objet ait un retour de celui-ci, plusieurs messages sont échangés. Suivant nos objets la communication entre l'utilisateur et ceux-ci n'est pas la même. Le système prévoit deux types de communications avec les objets.

3.2.3.1 Communication unidirectionnelle

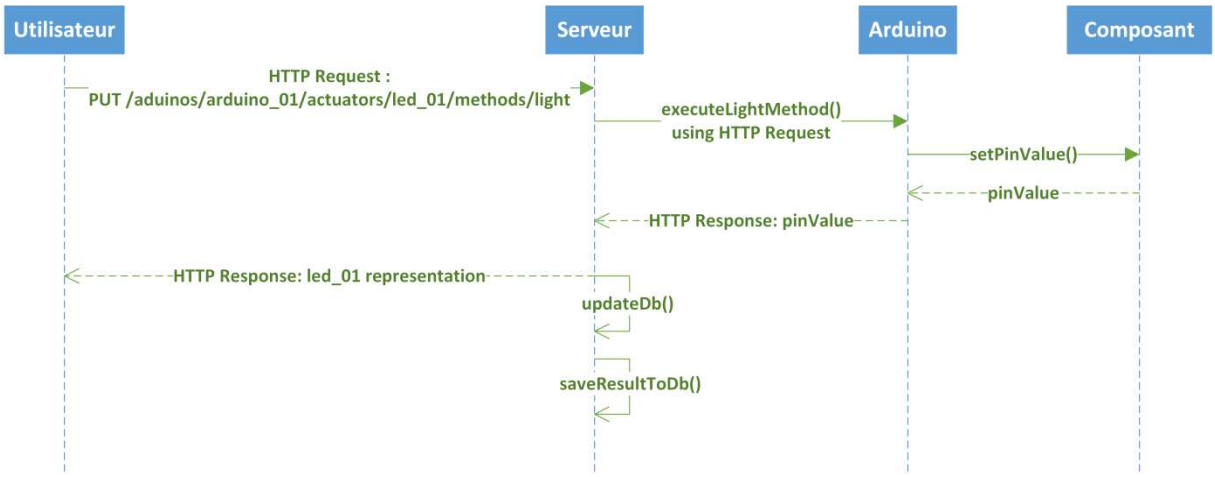


Figure 25 - Diagramme de séquence : communication unidirectionnelle

Le diagramme de séquence de la *figure 25* décrit les différents messages du système suite à l'exécution de la requête `PUT /arduinos/arduino_01/led_01/methods/light` qui sert à allumer la led_01. Chaque message est basé sur le modèle requête/réponse du protocole HTTP, on parle alors de communication unidirectionnelle. Il ne peut pas y avoir de réponse sans requête.

Cette communication fonctionne bien dans le cas d'une interaction avec requête-réponse sur le modèle client-serveur comme allumer ou éteindre une LED ou encore demander des informations à un capteur. Par contre lorsque l'on veut qu'un objet communique sans qu'il y ait de requête au préalable ce modèle ne fonctionne plus. Par exemple l'utilisation courante d'un lecteur RFID est de remonter l'identifiant du transpondeur qu'il a lu. Or le client ne peut pas en permanence envoyer une requête pour savoir si le lecteur RFID a lu un identifiant, cela surchargerait le réseau et engendrerait beaucoup de traitements. Le lecteur doit donc envoyer lui-même l'information. Il faut trouver un moyen d'envoyer l'information directement à travers le Web depuis un objet communicant sans requête du client au préalable.

3.2.3.2 Communication bidirectionnelle

Le protocole WebSocket est un standard du Web normalisé par l'IETF (Internet Engineering Task Force). Il utilise le protocole de transport TCP et permet une communication bidirectionnelle entre le client et le serveur. Une interface de programmation est en cours de

standardisation par le W3C et ajoute cette communication bidirectionnelle entre un navigateur et un serveur. En l'utilisant, notre serveur centralisé peut envoyer directement des données à destination des utilisateurs.

Dans notre architecture le serveur centralisé récupère les notifications en provenance des objets des nœuds Arduino envoyés via HTTP avec le modèle requête/réponse ; l'Arduino agissant comme un client qui envoie une notification dans une requête HTTP. Avec l'apport du protocole WebSocket, le serveur centralisé envoie ensuite directement l'information au client. Ainsi il peut recevoir des notifications depuis des objets en quasi temps réel.

L'ouverture d'un canal de communication bidirectionnel avec le protocole WebSocket nécessite un premier échange classique requête/serveur par le protocole HTTP. Le client envoie une demande de connexion WebSocket au serveur que celui-ci autorise en lui fournissant les données nécessaires pour y parvenir. Ensuite le client, avec la fonctionnalité de mise à jour de la connexion prévue par le protocole HTTP change de protocole à utiliser pour communiquer avec le client ce qui permet l'ouverture d'un canal WebSocket. Le serveur peut donc envoyer directement les messages provenant des objets connectés vers le client dès réception.

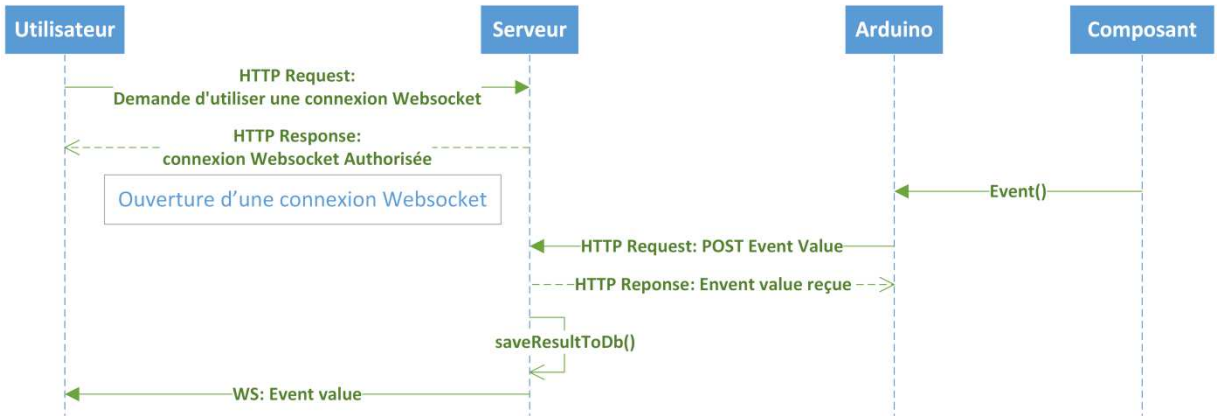


Figure 26 - Digramme de séquence : communication bidirectionnelle

Au niveau de l'API, pour prendre en compte cette communication bidirectionnelle, nous prévoyons des méthodes pour souscrire à la réception des notifications des objets communicants des nœuds Arduino.

En prenant l'exemple d'un objet lecteur RFID nommé rfid_01 d'un nœud nommé arduino_02, la requête `GET /arduinos/arduino_02/sensors/rfid_01/methods/reading` nous donne la représentation, montrée à la *figure 27*, de la méthode *reading* pour souscrire aux lectures des identifiants des tags RFID.

```
Method :
Content :
       name = reading
       state = unsubscribe
       links :
              method = put
              href = /arduinos/arduino_02/sensors/rfid_01/methods/reading
              rel = Un/Subscribe Method
              parameters :
                     name = action
                     value = subscribe|unsubscribe
```

Figure 27 - Représentation ressource Method reading

La souscription à la méthode *reading* se fait avec la requête `PUT /arduinos/arduino_02/sensors/rfid_01/methods/reading` contenant le paramètre *action* = *subscribe*. Au niveau de l'applicatif l'utilisateur est référencé dans une table et peut recevoir via WebSocket chaque lecture d'identifiant en provenance du lecteur RFID rfid_01.

3.2.4 Arduino et REST

Pour que nos nœuds Arduino communiquent à travers le réseau avec le serveur centralisé, nous appliquons aussi le modèle REST avec une interface uniforme pour appeler ses services. Le modèle d'architecture est appliqué de manière plus légère et ne respecte pas toutes les contraintes, notamment celle du HATEOAS. Les représentations sont justes sous forme d'un message court donnant l'essentiel de l'information. De plus, en raison de contraintes techniques, les arborescences sont simplifiées et utilisent des paramètres dans l'URI avec la méthode `GET`.

3.2.4.1 Identification des ressources Arduino

Une carte Arduino possède, sur l'ensemble de ses broches, une majorité d'entrées et de sorties analogiques et digitales car beaucoup de composants fonctionnent avec ce type d'interface. Le fonctionnement que nous appelons standard est la lecture de ces broches et l'écriture sur ces

broches. Ces entrées et sorties digitales et analogiques sont identifiées comme ressources du modèle REST appliqué à Arduino.

Pour le reste des composants qui n'utilisent pas ce fonctionnement standard car ils fonctionnent sur d'autres types d'interfaces ou ont un fonctionnement n'utilisant pas juste la lecture ou l'écriture sur une broche analogique ou digitale, nous développons des méthodes spécifiques qui sont appelées depuis l'API. Dans notre modèle d'architecture, ces méthodes sont identifiées comme ressources.

Le *tableau 13* décrit l'interface uniforme de l'architecture REST mise en place sur un nœud Arduino.

Method	URI	Description
GET	http://ip_arduino	Réception des informations complètes d'un Arduino
GET	http://ip_arduino/analogs	Réception de la valeur d'une entrée analogique. Nécessite le paramètre pin dans l'URI pour définir le numéro de pin dont on souhaite recevoir la valeur.
POST	http://ip_arduino/analogs	Changement de la valeur d'une sortie analogique (PWM). Nécessite les paramètres pin et value dans le corps de la requête pour le numéro de pin et la valeur qu'on lui donne
GET	http://ip_arduino/digitals	Réception de la valeur d'une entrée digitale. Nécessite le paramètre pin dans l'URI pour définir le numéro de pin dont on souhaite recevoir la valeur.
POST	http://ip_arduino/analogs	Changement de la valeur d'une sortie digitale. Nécessite les paramètres pin et value dans le corps de la requête pour le numéro de pin et la valeur qu'on lui donne
POST	http://ip_arduino/methods	Exécution d'une méthode Arduino. Nécessite les paramètres name et value dans le corps de la requête pour désigner la méthode appelée et lui donner une valeur pour effectuer son traitement.

Tableau 13 - Interface uniforme : Arduino

3.3 Synthèse

Pour concevoir la partie applicative de notre système nous avons utilisé la modélisation objet dans l'optique d'identifier les principaux composants et d'en définir leurs principales méthodes. Ensuite nous avons défini l'architecture de notre API pour créer l'interface qui sera

mise à dispositions des utilisateurs. C'est par cette interface que sont appelées les méthodes de nos objets.

A partir de ces spécifications nous pouvons réaliser notre application avec plusieurs outils en commençant par créer les différentes méthodes définies dans la modélisation objet et ensuite construire l'API suivant notre architecture REST. Nous utilisons le même cheminement pour le développement sur Arduino.

Chapitre 4 : Réalisation du projet

4.1 Architecture matérielle

4.1.1 Choix Arduino

Pour l'élaboration de notre système d'intelligence ambiante, nous avons choisi d'utiliser des cartes Arduino. Ce choix a été motivé par la popularité de ces cartes qui sont utilisées par une très grande communauté notamment grâce à sa philosophie open source basée sur le partage. La famille de cartes proposées par le projet Arduino s'est grandement étoffée au fil des années. Les cartes sont proposées sous différentes tailles, les plus grandes sont utilisées pour du prototypage avec des cartes prévues pour connecter facilement des fils sans soudure et effectuer des allers-retours entre codage, chargement du code et test. Les cartes plus petites sont plutôt pensées pour la création d'un projet fini nécessitant de souder aux bornes des interfaces de l'Arduino et de ne plus changer de configuration.

Notre choix s'est porté sur deux cartes Arduino étant prévues pour du prototypage pour pouvoir facilement changer de configuration dans l'élaboration de nos nœuds de capteurs et d'actionneurs : l'**Arduino Ethernet** et l'**Arduino Mega**.

4.1.1.1 Arduino Ethernet

L'Arduino Ethernet est équivalent à une carte Arduino Uno avec un shield Ethernet sauf que celle-ci est plus compacte. L'Arduino Uno est la carte plus populaire du monde Arduino car elle est peu couteûse et possède un nombre d'interfaces suffisant pour couvrir l'utilisation d'une multitude de types de capteurs et d'actionneurs différents. La plupart des ouvrages sur

Arduino sont réalisés avec la carte Uno, ce qui permet d'avoir un grand nombre d'exemples d'utilisation que nous pouvons utiliser avec la carte Ethernet.

Le modèle Ethernet possède une interface RJ45 pour accéder à un réseau filaire Ethernet. Cette carte peut aussi comporter un module PoE (Power over Ethernet) est ainsi être alimentée à travers l'Ethernet si les équipements réseaux, tel que des switchs, le permettent.

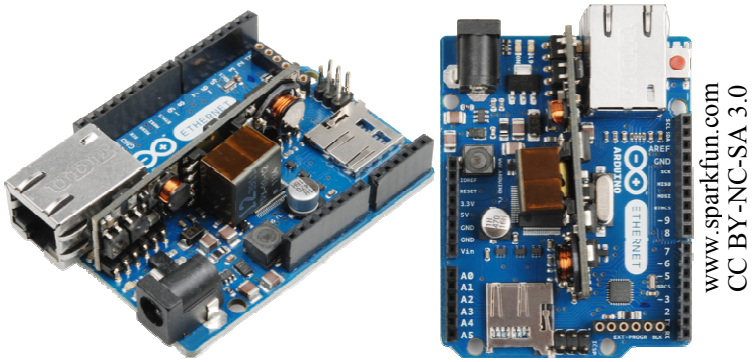


Figure 28 - Arduino Ethernet avec PoE

Ce modèle comporte un microcontrôleur ATmega328 muni d'un processeur cadencé à 16MHz. Cela peut sembler peu par rapport aux processeurs d'ordinateur mais c'est amplement suffisant pour gérer l'ensemble des fonctions que l'Arduino propose. Par contre il faut être vigilant sur l'utilisation de la mémoire SRAM (Static Random Access Memory) lors du développement de programmes car celle-ci est seulement de 2KB.

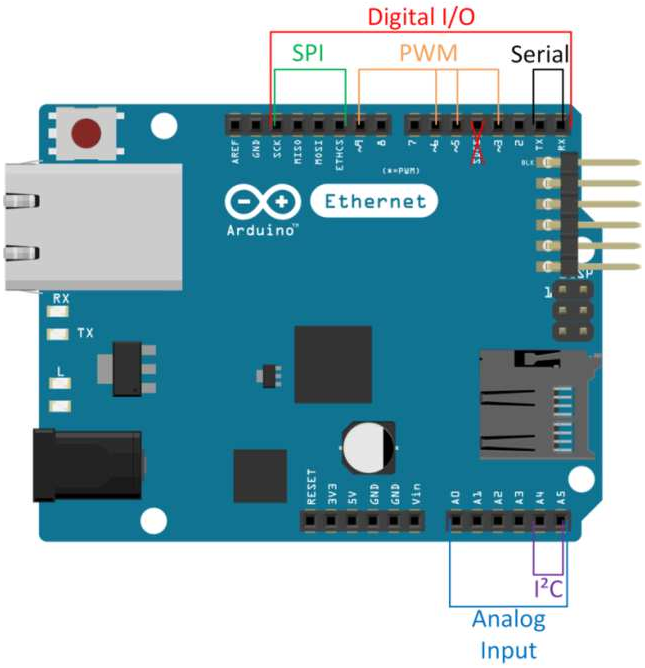


Figure 29 - Interfaces Arduino Ethernet

La *figure 29* montre où sont situés les différents types de broches permettant la communication entre l'Arduino Ethernet et nos différents objets. L'ensemble des broches entrées/sorties digitales (Digital I/O) peuvent avoir un autre fonctionnement (SPI, Serial et PWM) et les entrées analogiques (Analog Input) A4 et A5 peuvent être utilisées pour dialoguer avec le protocole I²C. Une broche ne peut fonctionner que d'une façon, il faut donc choisir son utilisation, soit entrée/sortie digitale ou soit sa spécification. Nous n'utilisons pas les possibilités de communications en SPI et I²C car nous n'avons pas de composants utilisant ces protocoles.

4.1.1.2 Arduino Mega

L'Arduino Mega ressemble à l'Arduino Uno sauf qu'il est prévu pour prendre en compte beaucoup plus d'objets en même temps, grâce à un grand nombre d'interfaces. Pour notre utilisation nous l'avons équipé d'un shield Ethernet pour pouvoir le relier en réseau.

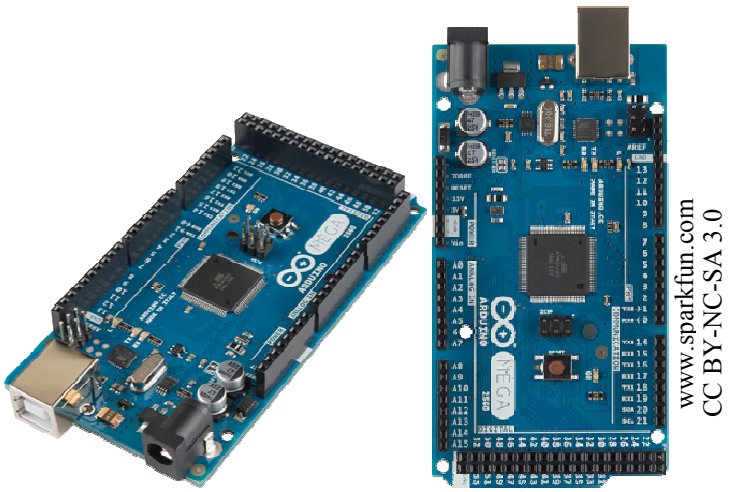


Figure 30 - Arduino Mega

L'Arduino Mega est architecturé autour d'un microcontrôleur ATmega2560 dont la fréquence du processeur est la même que celui de l'Arduino Ethernet. Par contre à la différence de l'ATmega328, celui-ci peut gérer beaucoup plus d'interfaces. De plus la partie mémoire SRAM est elle aussi plus grande : 8KB.

Tous les programmes développés sur Arduino Ethernet fonctionnent sur l'Arduino Mega. Il faut juste adapter aux numéros de broches utilisées.

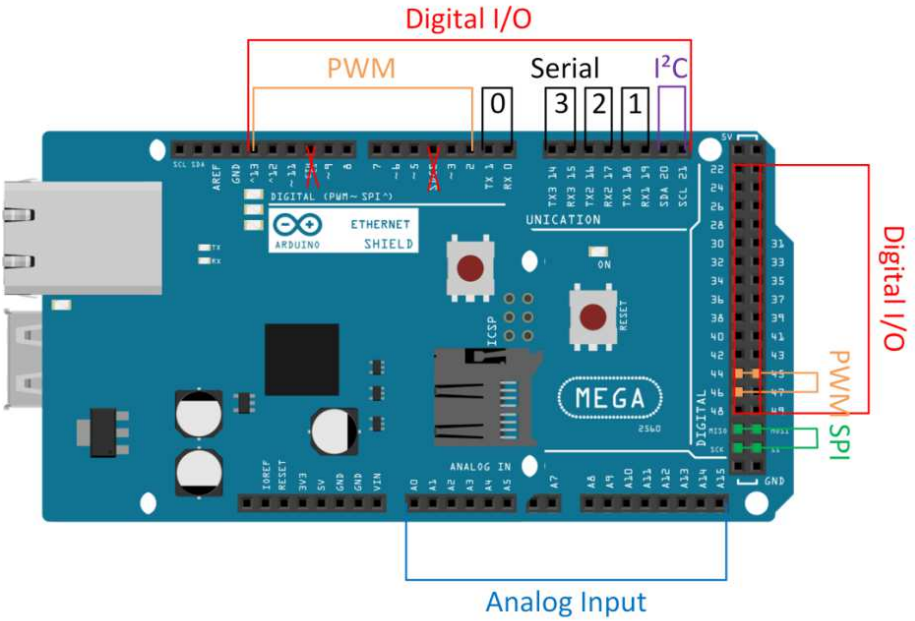


Figure 31 - Interfaces Arduino Mega et shield Ethernet

La *figure 31* détaille le placement des différents types de broches permettant la communication entre l'Arduino Mega et nos différents objets. Le modèle Mega possède un grand nombre d'entrées/sorties digitales (Digital I/O) dont 8 broches peuvent aussi fonctionner en liaison Série (4 liaisons Série sur 4 couples de broches), 13 peuvent fonctionner en PWM et un couple de broche apporte la communication I²C. Comme pour l'Arduino Ethernet une broche entrée/sortie digitale ne peut fonctionner que d'une façon et nous n'utilisons pas les protocoles SPI et I²C.

4.1.2 Choix du serveur centralisé

Nous nous sommes intéressés au choix d'un matériel qui soit aussi miniaturisé pour héberger notre applicatif et assurer la fonction de serveur centralisé. Cela nous permet de déplacer facilement notre système, et d'expérimenter dans plusieurs endroits en créant un petit réseau privé à l'aide d'un switch.

Dans un premier temps, nous avons considéré le Raspberry PI qui est de la même taille qu'un Arduino et dont le fonctionnement est plus similaire à un ordinateur même s'il possède quelques interfaces permettant le physical computing. Son architecture est faite autour d'un processeur ARM1176 cadencé à 700Mhz, mais cela ne suffit pas à faire fonctionner notre applicatif avec un temps de réponse convenable.

Nous nous sommes donc orientés vers une configuration plus standard avec le mini PC NUC (Next Unit of Computing) produit par Intel. Ce mini PC utilise des processeurs classiques du marché des ordinateurs mais avec une carte mère très petite par rapport au standard : 10cm sur 10cm. Cette carte possède, entre autre, une interface RJ45 pour être reliée à un réseau Ethernet. Notre modèle a un processeur double cœur Intel Core i5 de quatrième génération cadencé à 1.3Ghz par cœur avec 3Mo de cache L3. Ceci est amplement suffisant pour héberger notre applicatif dans de bonnes conditions.

Figure 32 - Intel NUC

4.1.3 Architecture retenue

Dans le cadre des tests de notre proof of concept nous avons retenu une architecture simple pour se concentrer que sur l'aspect transformation des capteurs et actionneurs en objets communicants, sans se soucier des aspects sécurité du réseau et de son utilisation à l'internet public. Les nœuds Arduino, le serveur centralisé et les utilisateurs sont sur un même réseau relié à un switch et utilisent le même plan d'adressage.

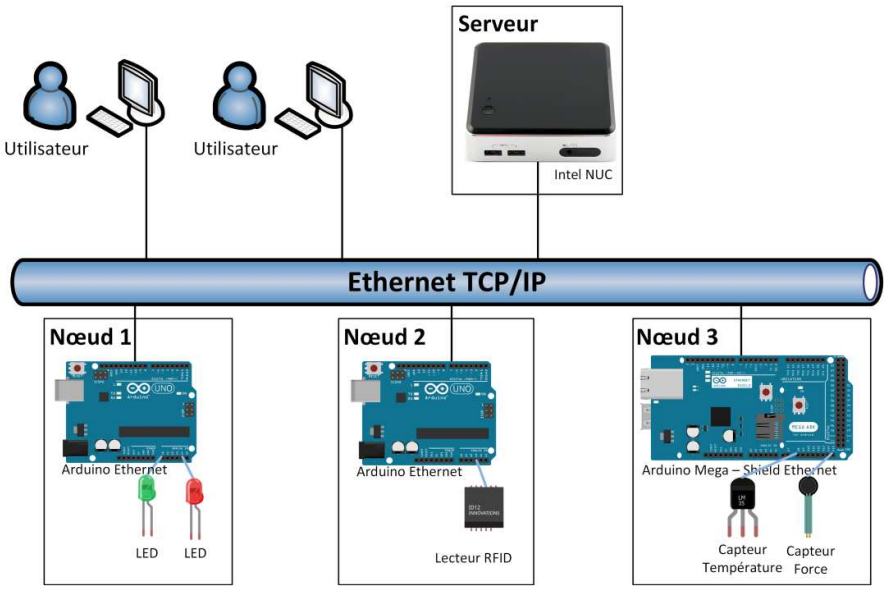


Figure 33 - Architecture matérielle

4.2 Arduino à travers Internet

4.2.1 Entrées/sorties digitales et analogiques

4.2.1.1 Structure de base d'un code Arduino

Un code Arduino est structuré par deux fonctions principales et obligatoires : `setup()` et `loop()`. La fonction `setup()` est exécutée une seule fois au démarrage et à chaque reset. Elle est utilisée à des fins de configuration de l'Arduino. C'est dans cette fonction que sont déclarés les modes de fonctionnement de chaque pin digital (entrée ou sortie), ou encore que sont initialisées certaines méthodes de librairies. La fonction `loop()` est appelée après la fonction `setup()` et tourne en boucle. C'est cette fonction qui permet au programme de répondre et de contrôler la carte Arduino en s'exécutant en permanence.

4.2.1.2 Méthodes pour entrées/sorties digitales et analogiques

La plupart des composants que nous rendons communicants sur nos nœuds Arduino fonctionnent avec les broches entrées/sorties digitales et analogiques. Pour agir sur un composant fonctionnant sur une broche digitale il faut déterminer s'il doit être utilisé en sortie (exemple : LED) ou en entrée (exemple : bouton poussoir). Cette déclaration s'effectue au sein de la méthode `setup()` avec la méthode `pinMode()` qui prend en paramètre le numéro de la broche (pin en anglais) et le mode choisi qui est `INPUT` (entrée) ou `OUTPUT` (sortie). Une fois le mode déclaré, il est possible de lire l'état de la broche digitale dans le cas d'une broche en mode `INPUT` ou `OUTPUT` avec la méthode `digitalRead()`. Cette méthode prend en paramètre le numéro de broche et renvoi la valeur `HIGH` et `LOW` correspondant à la tension 5v ou 0v. Dans le cas d'une broche en mode `OUTPUT`, en plus de la lecture ; nous pouvons aussi changer son état, passer de `HIGH` à `LOW` et inversement, avec la méthode `digitalWrite()`. Dans le cas d'une LED, cela permet de l'allumer ou de l'éteindre.

Pour les entrées analogiques qui permettent la lecture des différentes tensions d'un composant, il n'y a pas besoin de définir un mode de fonctionnement de la broche car ces broches ne fonctionnent que dans un seul mode. La lecture se fait avec la méthode `analogRead()` qui prend en paramètre le numéro de la broche. Le résultat retourné est un entier compris entre 0 et 1023 car l'Arduino utilise un convertisseur analogique vers digital

qui code la tension sur 10 bits. Ainsi la tension lue entre 0v et 5v correspond à une valeur numérique entre 0 et 1023.

Pour simuler une sortie analogique, une carte Arduino propose plusieurs broches digitales pouvant fonctionner avec la méthode PWM (Pulse Width Modulation). En sortie de la broche digitale, la tension va passer plus ou moins vite de 0v à 5v. Le durée où la tension est à 5v est appelée pulse width (largeur d'impulsion), et c'est celle-ci qui permet de simuler une tension envoyée à une valeur comprise entre 0v et 5v. La fonction `analogWrite()` prend en paramètre un entier de 0 à 255. La valeur 255 signifie que sur un intervalle de temps la tension est toujours à 5v et la valeur 127 signifie que sur ce même intervalle de temps la tension est à une valeur de 5v la moitié de cet intervalle ce qui simule une tension de 2.5v. Cette fonction `analogWrite()` permet, par exemple, de contrôler l'intensité lumineuse d'une LED ou encore un servomoteur.

Ce sont les méthodes `digitalRead()`, `digitalWrite()`, `analogRead()` et `analogWrite()`que nous utilisons à travers notre architecture REST pour Arduino. La *figure 34* résume l'utilisation de ces quatre méthodes pour dialoguer avec des composants interagir avec deux composants LED, un capteur de luminosité et un bouton poussoir.

```
//led01 connectée sur pin digitale 1
int led01 = 1;
//led02 connectée sur pin digitale 2
int led02 = 2;
//lightSensor connectée sur pin analogique A0
int lightSensor = A0;
//pushButton connecté sur pin digitale 3
int pushButton = 3;

void setup() {
  //pin led01 déclaré en mode Sortie pour changer d'état
  pinMode(led01, OUTPUT);
  //pin pushButton déclaré en mode entrée pour lecture
  pinMode(pushButton, INPUT);
}

void loop() {
  //Allume led01
  digitalWrite(led01, HIGH);
  //Lit état pushButton (Retourne LOW pour non pressé et HIGH pour pressé)
  digitalRead(pushButton);
  //Lit valeur capteur de lumière (Retourne entier entre 0 et 1023)
  analogRead(lightSensor);
  //Envoie tension de 2.5v sur led02 (PWM)
  analogWrite(led02, 127);
}
```

Figure 34 - Méthodes Arduino pour broches analogiques et digitales

4.2.2 Difficultés

La prise en charge d'un réseau TCP/IP par Arduino que ce soit avec le shield Ethernet ou directement avec un modèle Arduino Ethernet permet de transformer la carte en un petit serveur web ou alors en client web. L'IDE Arduino propose une librairie pour y parvenir et le site d'Arduino décrit chaque méthode fournie par cette librairie avec en plus des exemples [Ethernet Library – 2015].

Pour notre projet nous réalisons une interface de type API simplifié sur le modèle élaboré en phase de conception. Sur l'Arduino, pour mettre en place une API sur le modèle REST, nous nous sommes heurtés à plusieurs difficultés.

L'architecture REST sur Arduino n'est pas aisée à mettre en place à cause de la faible capacité de la mémoire SRAM du modèle Arduino Ethernet qui est de 2 KB. L'envoi d'information retournée dans une réponse nécessite la création de variables contenant des chaines de caractères. Ces chaines prennent rapidement beaucoup d'espace mémoire.

Pour contourner cette difficulté, il est possible de stocker des variables dans la mémoire du programme qui est de 32KB ce qui laisse de la marge si le programme développé ne prend pas tout cet espace. Cette fonctionnalité est apportée par la bibliothèque du langage C nommée PROGMEM que l'on peut utiliser dans un développement Arduino. Par contre les variables stockées dans la mémoire programme ne peuvent être que statiques car accessibles qu'en lecture. Nous utilisons cette technique pour des chaines de caractères utilisées dans la création des messages renvoyant les informations.

Une autre difficulté apparait car le traitement des requêtes réceptionnées par l'Arduino se fait caractère par caractère. Pour que l'API de l'Arduino réponde, il faut récupérer la requête complète dans une variable tampon et identifier le verbe HTTP et l'URI pour ensuite créer des conditions de traitements qui génèrent les actions et retournent une réponse. Le traitement des chaines de caractères en langage C n'est pas optimisé comme dans les langages récents et nécessite de bonnes connaissances en ce langage. De plus nous nous heurtons encore à des problèmes de gestion de la mémoire. C'est pour répondre à ce genre de problématique que plusieurs développeurs ont créé la librairie Webduino.

4.2.3 Librairie Webduino

Cette librairie a été conçue par des développeurs tiers pour ajouter à un serveur web Arduino plusieurs fonctionnalités. Les plus intéressantes pour notre projet sont la gestion des verbes HTTP, des différentes URIs et des paramètres contenus dans le corps des requêtes [Webduino – 2015]. Ces fonctionnalités sont apportées par un code optimisé consommant au minimum la mémoire SRAM de l'Arduino, grâce à des mécanismes utilisant la mémoire programme et vidant les variables tampons contenant les requêtes HTTP une fois traitées. La *figure 35* présente un exemple simplifié d'utilisation de Webduino dans notre projet pour répondre aux URIs concernant les entrées/sorties analogiques.

```
// Appel de la librairie Ethernet fournie par Arduino
#include "Ethernet.h"
// Appel de la librairie Webduino
#include <WebServer.h>

//Adresse MAC et IP de la carte Arduino
static uint8_t mac[] = {0x90 , 0xA2, 0xDA, 0x00, 0x6B, 0x96};
static uint8_t ip[] = {192, 168, 1, 10};

/* Instanciation d'un objet Webserver qui créé un serveur web répondant sur
   le port 80 à l'adresse racine (utilisation de "") */
WebServer webserver( "" , 80 ) ;

/* fonction appelée par les méthodes de la librairie Webduino pour traiter
   les demandes concernant les broches analogiques */
void analogsCommand( WebServer &webserver , WebServer::ConnectionType connType ,
char *tailUrl , bool tailComplete ) {
  if (connType == WebServer::GET) {
    /* Traitement pour une requête GET */
    analogRead()
  }

  if (connType == WebServer::POST) {
    /* Traitement pour une requête POST */
    analogWrite()
  }
}

void setup() {
  // Initialisation du contrôleur Ethernet d'Arduino
  Ethernet.begin(mac, ip);

  // Appel de la fonction analogsCommand lors d'une requête sur /analogs
  webserver.addCommand("analogs", &analogsCommand);

  // Démarrage du serveur web
  webserver.begin() ;
}

void loop()
{
  char buff[64];
  int len = 64;

  /* Traite en boucle les connections au serveur web. */
  webserver.processConnection(buff, &len);
}
```

Figure 35 - Code Webduino

4.2.4 Méthodes composant coté Arduino

L'interface uniforme REST pour les entrées/sorties analogiques et digitales fonctionne bien pour des composants simples utilisant ces interfaces. Dans le cas d'une LED, une requête `GET http://ip_arduino/digitals?pin=2` donne l'état 0 ou 1 de la sortie nous permettant de savoir si elle est éteinte ou allumée. La requête `POST http://ip_arduino/digitals` avec les paramètres `pin=3` et `value=1` écrit l'état 1 dans sortie 3 ce qui allume la LED connectée sur la sortie 3. Pour les composants de type capteur fonctionnant sur une entrée digitale, nous prévoyons d'en récupérer la valeur brute avec `analogRead()` à travers l'API et de transformer cette valeur par le serveur centralisé afin d'effectuer le moins de traitement sur l'Arduino tant que cela est possible. Quand ce n'est pas le cas, nous développement des méthodes sur Arduino spécifiques au composant.

4.2.4.1 Méthodes de notification

Lorsqu'un composant doit envoyer de l'information sans être interrogé, nous mettons en place un système de notification. Un bouton poussoir ou un capteur de pression utilisent une entrée digitale pour le premier et une analogique pour deuxième. Si nous souhaitons savoir s'ils sont actionnés nous pouvons très bien utiliser les URIs de l'API concernant les entrées digitales et analogiques. Le problème est qu'il faudrait envoyer en permanence des requêtes pour espérer avoir un résultat où l'un des deux composants répond. Pour apporter une solution, nous créons une méthode pour chaque composant qui va surveiller son utilisation et envoyer un retour au serveur central via une requête HTTP dès que le composant interagit. Cela est possible grâce à la librairie Ethernet pour Arduino qui lui permet de fonctionner en client HTTP.

4.2.4.2 Méthodes nécessitant des paramètres

Ce type de méthode peut s'appliquer à tous les composants si nous souhaitons un fonctionnement plus spécifique ou plus élaboré. Si nous voulons qu'un capteur de pression envoie une notification après une pression de plusieurs secondes définie par l'utilisateur il faut que la méthode accepte des paramètres depuis le serveur. Pour notre proof of concept nous réalisons un exemple de ce type de méthode pour faire fonctionner un servomoteur avec un débattement de 180°. Ce type de servomoteur fait tourner son moteur suivant une rotation comprise entre 0 à 180 degrés que nous devons lui indiquer. La méthode associée à cet objet

prend en compte un paramètre avec le degré de rotation choisi. La valeur est envoyée depuis le serveur centralisé à la demande d'un utilisateur.

4.3 Architecture logicielle de l'API

Pour réaliser notre API, nous avons fait le choix de technologies orientées web dans un même langage : le JavaScript. Ce langage fait partie des plus utilisés pour les applications web et a été adapté à plusieurs niveaux d'une architecture logicielle, que ce soit coté serveur, client ou encore pour la base de données. L'applicatif et l'API sont développés avec Node.JS et son module Express.JS, et la partie base de données est gérée par MongoDB. Les échanges de données de l'API sont au format JSON.

4.3.1 Node.JS

4.3.1.1 Présentation de Node.JS

Node.JS est une plateforme, créée en 2008, utilisant le langage JavaScript coté serveur pour créer des applications réseaux. Les applications sont construites sur le modèle évènementiel avec des entrées/sorties non-bloquantes. Son créateur, Ryan Dahl, après avoir testé plusieurs technologies pour une plateforme de développement non-banquant, a choisi le JavaScript car Google venait de développer un nouveau moteur d'exécution de JavaScript open source pour son navigateur Chrome. Ce moteur nommé V8 est 10 à 20% plus rapide pour interpréter du JavaScript par rapport ce qui se faisait auparavant sur les autres navigateurs [RAI – 2013]. Node.JS est construit à partir de ce moteur et tire parti de la programmation évènementielle du JavaScript que l'on rencontre coté client avec par exemple la gestion des évènements clavier ou souris dans une interface Web. Cette plateforme utilise un principe de module ou packages pour rajouter des fonctionnalités. Node.JS propose un gestionnaire de package nommé NPM (Node Package Manager) et permet l'installation automatique de modules, en ligne de commande, à la manière d'un gestionnaire de packages d'une distribution Linux. L'élaboration d'une application nécessite un certain nombre de modules pour fonctionner et l'avantage du gestionnaire de packages est de référencer tous les modules utilisés dans un même fichier. Lorsque l'on veut déployer l'application sur une machine il suffit de demander à NPM d'installer tous les modules à partir de ce fichier.

La philosophie de la programmation de Node.JS est d'utiliser des fonctions ou des méthodes asynchrones avec un mécanisme de callback (rappel en français). Chaque méthode de traitement comporte une fonction callback qui est appelée dès que le traitement est terminé. Cette fonction callback contient le retour du traitement (résultats ou erreurs). Pendant le traitement en cours d'une méthode, une autre opération peut être effectuée sans attendre le retour de la première. A aucun moment il n'y a d'opérations bloquantes. Node.JS utilise une boucle d'évènement qui dispatche les opérations nécessitant des entrées/sorties comme des traitements sur le système de fichiers ou sur la base de données. Dès que les opérations sont terminées, cette même boucle envoie le résultat dans la fonction callback. Ceci permet des traitements en quasi parallèle appelés hautement concurrentiel.

```
/* Méthode asynchrone pour récupérer les données d'un capteur */
getSensor(arduinoName, sensorName, function(result, error) {
    if (error) {
        console.log('Erreur récupération données capteur');
    }
    if (result) {
        console.log(result);
    }
});

/* Méthode asynchrone pour créer un utilisateur */
createUser(userName, userPassword, function(result, error) {
    if (error) {
        console.log('Erreur création user');
    }
    if (result) {
        console.log(result);
    }
}) ;
```

Figure 36 - Methodes Node.JS asynchrones

La *figure 36* montre la construction de méthodes asynchrones sous Node.JS. Chaque méthode accepte des arguments et possède une fonction anonyme `function(result, error)` qui est le callback appelé à la fin du traitement de la méthode. Ce callback contient deux objets `result` et `error` pour traiter le retour de la méthode. Dans l'exemple nous retournons le résultat dans la sortie standard avec l'instruction `console.log`. La première méthode cherche en base de données le capteur d'un Arduino donné en paramètre et pendant cette recherche l'autre méthode s'occupe de créer un utilisateur. Les retours des méthodes sont traités dans la fonction callback.

4.3.1.2 Méthodes d'organisation des traitements des ressources

Pour structurer notre code Node.JS nous regroupons nos méthodes asynchrones dans plusieurs fichiers différents suivant leurs actions. Par exemple les méthodes pour la gestion des utilisateurs par l'administrateur sont regroupées dans un fichier `admin_user_handling.js`. Avec un mécanisme d'exportation des méthodes, il possible de les appeler à partir d'un autre fichier de code. La *figure 37* montre l'exportation de la méthode de création d'un utilisateur à partir du fichier `admin_user_handling.js` et son utilisation dans un autre fichier de code `exemple.js`.

FICHIER admin_user_handling.js

```
exports.create = function(userName, userPassword, callback) {
    /* opération de création de l'utilisateur
       en base de données */
       CREATE USER

    /* Envoi du callback avec un objet result et error pour
       traiter le retour de cette fonction */

    return callback(error, result);
}
```

FICHIER exemple.js

```
// Appel du fichier admin_user_handling.js
var admin_user_handling = require('admin_user_handling.js');

// Exécution de la méthode create
admin_user_handling.create('user1', 'secret', function(error, result) {
    if (error) {
        console.log(error);
    }
    if (result) {
        console.log(result)
    }
});
```

Figure 37 - Export et appel de méthodes

Lors de la conception de l'applicatif nous avions identifié plusieurs méthodes avec la modélisation objet que notre système devait implémenter. Ces méthodes sont créées sous le modèle asynchrone et sont organisées dans des fichiers suivant les ressources manipulées.

Nom	Méthodes	Description
admin_arduino- user_handling.js	- list() - retrieve() - add() - del()	Méthodes administrateur pour gérer les Arduino d'un utilisateur. Ajouter les droits ou retirer les droits (add et del). Voir la liste des arduino ou un arduino dont il possède les droits
admin_arduino_handling.js	- list() - retrieve() - create() - del()	Méthodes administrateur pour gérer les Arduino. Récupérer la liste de tous les Arduino ou les informations d'un arduino. Créer ou supprimer un Arduino.
admin_component_handling.js	- listSensors() - listActuators() - retrieveSensor() - retrieveActuator() - activateSensor() - activateActuator() - desactivateSensor() - desactivateActuator()	Méthodes administrateur pour gérer les objets composants d'un Arduino. Lister les composants ou récupérer les informations d'un composant. Activer ou désactiver un composant.
admin_user_handling.js	- list() - retrieve() - create() - del()	Méthodes administrateur pour gérer les utilisateurs. Récupérer la liste de tous les utilisateurs ou les informations d'un utilisateur. Créer et supprimer un utilisateur
arduino_post_handling.js	- process()	Méthode pour traiter l'envoi d'information depuis un Arduino. Utiliser pour traiter les notifications.
user_arduino_handling.js	- list() - retrieve()	Méthodes utilisateur pour lister les Arduino avec lesquels il peut interagir ou récupérer toutes les informations d'un Arduino
user_component_handling.js	- listSensors() - listActuators() - retrieveSensor() - retrieveActuator()	Méthodes utilisateur pour lister tous les composants d'un Arduino avec lesquels il peut interagir ou récupérer toutes les informations d'un composant.
user_componentmethod_ handling.js	- listSensorMethods() - listActuatorMethods() - retrieveSensorMethod() - retrieveActuatorMethod() - executeSensorMethod() - executeActuatorMethod()	Méthodes utilisateur pour interagir avec les méthodes composant. Lister toutes les méthodes d'un composant ou récupérer toutes les informations d'une méthode composant. Exécuter une méthode composant.
user_historic_handling.js	- listByUser() - retrieveByUser() - delItemByUser() - delAllByUser()	Méthodes utilisateur pour gérer son historique. Lister toutes les entrées ou une entrée. Supprimer une entrée ou toutes ses entrées.

Tableau 14 - Méthodes Node.JS de l'applicatif

La plupart des méthodes du *tableau 14* prennent des paramètres en entrée que nous n'avons pas détaillé pour ne pas alourdir le tableau. Elles ont chacune une fonction callback retournant un objet résultat contenant la représentation et un objet erreur contenant le message d'erreur.

4.3.1.3 Méthodes de traitement des données Arduino

Nous appelons **computeMethods** ou méthodes de traitements, les méthodes qui s'occupent de la communication avec un nœud Arduino dans le but d'agir avec une famille précise de composants et d'en traiter leurs données. Dès qu'il nous est possible nous récupérons juste les données brutes des composants de l'Arduino et nous les traitons depuis l'applicatif du serveur centralisé pour les transformer en données lisibles pour l'utilisateur. Cela permet d'utiliser au maximum l'Arduino pour des opérations de lecture et d'écriture avec les composants

Par exemple, l'interrogation d'un capteur de température d'un nœud Arduino renvoie un entier de 0 à 1023 qui correspond avec une tension de 0v à 5v. Pour élaborer la computeMethod pouvant transformer les valeurs en provenance du nœud en une unité de température il faut obtenir l'information dans la documentation du constructeur du composant. Le capteur utilisé pour le besoin de notre proof en concept est un TMP36. Sa documentation nous donne la courbe de la tension retournée par le capteur en fonction de la température.

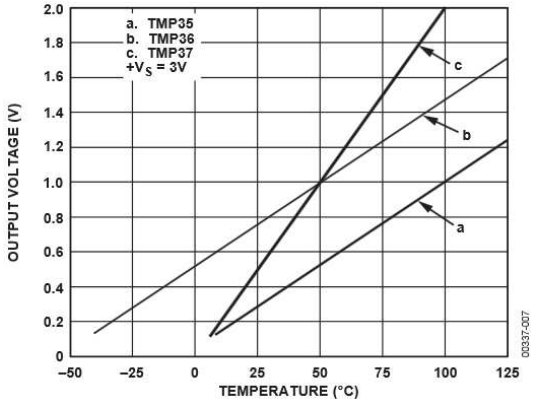


Figure 38 - TMP36 : voltage en fonction de la température [TMP36 Data Sheet – 2015]

En suivant la courbe b de la *figure 38* qui est une fonction affine nous en déduisons la formule pour obtenir la température en fonction du voltage :

Température_celcius = (Voltage – 0,5) / 0,01

Comme nous récupérons le voltage à partir de l'entrée digitale de notre Arduino sur un entier codé sur 10 bits nous devons faire la conversion pour transformer cet entier en volt. La description de la fonction `analogRead()` nous indique qu'une unité (entre 0 et 1023)

correspond à 0.0049V [Arduino analogRead() – 2015]. Ainsi la formule finale pour interpréter la valeur récupérée par l'Arduino depuis la computeMethod du serveur centralisé est :

Température_celcius = (Valeur_entrée_analogique * 0,0049 – 0,5) / 0,01

Lors de notre projet nous avons développé plusieurs méthodes décrites dans le *tableau 15*. Chaque méthode couvre une manière différente de dialoguer avec un type de composants. Elles sont regroupées dans le fichier de code `computeMethod.js` et exportées pour être appelées dans les autres fichiers de l'applicatif.

Méthode	Description
ledOnOff	Pilote l'allumage d'une LED en envoyant une requête POST http://ip_arduino/digitals/{digital_pin} au nœud Arduino avec le paramètre value à 0 ou 1 pour indiquer l'état demandé de la sortie. Cette méthode permet aussi d'interroger l'état de la sortie d'une LED sans la modifier.
servoRotation	Pilote la rotation d'un servomoteur par l'envoi d'une valeur comprise entre 0 et 180° à une méthode Arduino.
tmp36Temperature	Retourne la température d'un capteur TMP36 en degrés Celcius ou Fahrenheit
methodSubscription	Crée à une souscription à un utilisateur pour des notifications de capteurs. Cette méthode écrit en base la souscription avec le nom du composant et les utilisateurs devant recevoir les notifications celui-ci via WebSocket.

Tableau 15 - Liste des computeMethods

4.3.2 JSON

4.3.2.1 Présentation de JSON

JSON (JavaScript Object Notation) est un format de données basé sur le langage de programmation JavaScript. Il est très utilisé dans l'échange de données au même titre que XML (eXtensible Markup Language). JSON reprend le formalise d'écriture d'un objet JavaScript ce qui facilite grandement son utilisation dans une application codée dans ce

langage. Il n'est pas pour autant dépendant de JavaScript et il existe un grand nombre d'interpréteurs pour l'utiliser dans plusieurs langages. JSON est de plus en plus utilisé pour l'échange de données car il est plus simple à utiliser et plus léger que XML.

Ce format est construit suivant deux structures : un objet contenant une collection de paires nom/valeur désordonnée et un tableau contenant une liste de valeurs ordonnées. Une valeur peut être de différents types : string, number, object, array, boolean et null [JSON – 2015]. La *figure 39* décrit la construction de ces deux structures.

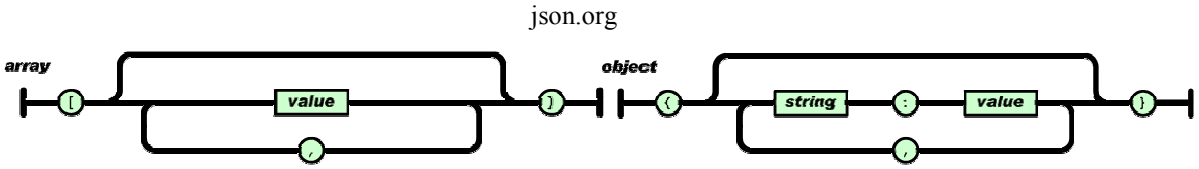


Figure 39 - Objet et tableau JSON

4.3.2.2 Représentation au format JSON

Notre API utilise le formalisme JSON pour l'échange de données. Nous l'appliquons à nos représentations. La *figure 40* décrit un exemple de représentation d'une ressource utilisateur au format JSON.

```
{
  "resource": {
    "type": "collection",
    "name": "users",
    "content": [
      {
        "_id": "54a80f9a80f23b0c1cf2e50e",
        "name": "user1",
        "arduinos": [
          "arduino01",
          "arduino02"
        ],
        "links": [
          {
            "href": "admin/users/user1",
            "method": "GET",
            "rel": "details user"
          },
          {
            "href": "admin/users/user1",
            "method": "DELETE",
            "rel": "delete user"
          }
        ]
      }
    ]
  }
}
```

Figure 40 - Exemple d'une représentation Users au format JSON

4.3.3 MongoDB

4.3.3.1 Présentation de MongoDB

Créé en 2007, MongoDB est un système de gestion de bases de données orienté documents qui fait partir de la famille des bases dite NoSQL (Not Only SQL) ; c'est-à-dire qu'elles ne sont pas relationnelles et n'utilisent pas le langage SQL. Une base de données orientée documents structure ses données avec le modèle clé-valeur. Un document comprend donc plusieurs clés et valeurs et un ensemble de document est appelée une collection. Une collection pouvant être comparée à une table dans le modèle relationnel. Une collection de documents ne comporte pas de schémas prédéterminés, c'est-à-dire qu'il est possible d'avoir plusieurs documents dans une même collection n'ayant pas du tout les mêmes clés et les même types de valeurs.

MongoDB structure ses données au format BSON, signifiant Binary JSON. Lors de l'interrogation de la base tous les documents seront retournés avec le formalisme JSON. Si MongoDB est un système de base de données surtout utilisé pour la gestion de grand volume de données ne nécessitant pas de jointures entre les tables, nous l'avons choisi pour son utilisation orientée JavaScript avec sa structure de données. De plus cette base est souvent couplée à Node.JS car des drivers performants ont été développés pour fonctionner avec. L'optique de notre proof of concept n'étant pas forcément d'obtenir la meilleure performance dans la gestion des données, nous aurions pu choisir n'importe quel outil de gestion de bases de données.

4.3.3.2 Module Mongoose

Mongoose est un module pour Node.JS visant à faciliter l'utilisation d'une base de données MongoDB. C'est ce qu'on appelle un ODM (Object Data Mapping) pour signifier un outil qui fait le lien entre les données d'une base et les objets de l'application. Dans le cas de Mongoose, l'outil apporte la notion de schéma pour définir une structure de données dans les collections et une notion de modèle pour créer des instances de données comme un objet, les manipuler et les sauvegarder dans les documents [HOLMES – 2013].

L'avantage de l'utilisation de schéma est d'ajouter une structure de données aux collections, c'est-à-dire que chaque document comporte les mêmes clés et que les valeurs sont du même type. Les schémas Mongoose apportent aussi un outil de validation des données pour renvoyer des erreurs en cas de non-respect du schéma.

Le modèle apporte à l'application une classe proposant plusieurs méthodes pour interroger la base de données et retourner des objets JavaScript représentant des documents. Les clés d'un document sont les propriétés de l'objet. Ainsi, nous pouvons manipuler ces objets dans notre code et ensuite faire appel à la méthode `save` du modèle qui lance la mise à jour en base de données du document correspondant à l'objet. Mongoose s'occupe alors du dialogue avec MongoDB pour générer les entrées sorties pour cette mise à jour.

La définition des schémas et modèles se fait en un seul point dans notre application et est séparé du reste du code, ce qui est plus pratique dans sa gestion. Le modèle créé à partir du schéma, et ensuite exporté pour utiliser à n'importe quel endroit de notre application.

```
// Appel du module mongoose
var mongoose = require('mongoose');

// Création du schéma pour la collection user
var userSchema = mongoose.Schema({
    // Nom de type string unique et obligatoire
    name: {type: String, unique: true, required: true},
    // Password de type string obligatoire
    password: {type: String, required: true}
});

/* Céation du modèle User à partir du schéma
   L'export permet d'utiliser le modèle n'importe où
   dans nos fichiers de code de l'application */
module.exports = mongoose.model('User', userSchema);
```

Figure 41 - Mongoose : schéma et modèle User

La *figure 41* décrit un exemple simple de schéma pour la collection User où sont stockés nos documents utilisateur. Le modèle créé à partir de ce schéma apporte les méthodes pour créer un document utilisateur et faire des requêtes au sein de la collection User. A chaque sauvegarde suite à une création ou à une modification ou encore à chaque requête de recherche en base avec notre modèle, Mongoose s'occupe de l'ensemble de la communication avec MongoDB. La *figure 42* montre deux exemples d'utilisation de méthodes proposées par un modèle Mongoose pour la création d'un utilisateur et la recherche d'un utilisateur.

```
/* Appel du modèle user exporté depuis le fichier
   de code user.js */
var User = require('./models/user.js');

// CREATION USER
// Instanciation d'un objet user avec le modèle
var user = new User({name: 'user_01', password: 'secret'})

/* L'objet user possède une méthode pour
   être sauvegardé en base pas mongoose.
   La méthode save est asynchrone avec une fonction
   de callback pour gérer le résultat ou l'erreur. */
user.save(function(result, err) {
    if (err) {
        // Envoi message d'erreur JSON
    }

    if (result) {
        // Envoi représentation User crée
    }
});

// RECHERCHE USER
/* Le modèle User possède une méthode de recherche dans
   sa collection.
   La méthode find est asynchrone avec une fonction
   de callback pour gérer le résultat ou l'erreur. */
User.find({name: 'user_01'}, function (result, error) {
    if (err) {
        // Envoi message d'erreur JSON
    }

    if (result) {
        // Envoi représentation User trouvé
    }
});
```

Figure 42 - Modèle Mongoose : création et recherche utilisateur

4.3.3.3 Collections de l'applicatif

Pour notre applicatif, nous adaptons le modèle entité-relation de la *figure 12*, élaboré lors de la conception, pour une base MongoDB non relationnelle. Nous créons les collections **Arduinos**, **Users**, **Subscriptions**, **Historics** et **componentmethods**.

- La collection **Arduinos** comporte les documents décrivant les nœuds Arduino. Chaque document *Arduino* comporte des sous-documents *Component* qui eux-mêmes comportent des sous-documents *Method*. La *figure 43* montre un exemple de document *Arduino* avec un composant comportant une méthode.

```
{
    "_id" : ObjectId("54ad4ad9deff38002445b1a8"),
    "name" : "arduino_01",
    "ip" : "192.168.1.10",
    "location" : "salle_01",
    "components" : [
        {
            "name" : "redLed1",
            "pin" : 22,
            "type" : "actuator",
            "subtype" : "led",
            "activated" : true,
            "methods" : [
                {
                    "name" : "light",
                    "type" : "standard",
                    "state" : "off",
                    "description" : "Parameter 'action' with value 'on' or 'off'",
                }
            ]
        }
    ]
}
```


Figure 43 - Document Arduino

Avec un document *Arduino* nous pouvons effectuer toutes les requêtes concernant le nœud Arduino, ses composants et leurs méthodes.

- La collection **User** décrit les utilisateurs avec les clés `name`, `password` et `arduinos`. La clé `arduinos` a pour valeur un tableau avec le nom des Arduino avec lesquels l'utilisateur peut interagir.
- La collection **Subscription** contient les souscriptions aux notifications des objets. Ses documents sont décrits par les clés `name`, `componentName` et `users`. La clé `users` a pour valeur un tableau d'utilisateurs recevant ces notifications.
- La collection **Historic** contient les résultats retournés par chaque méthode. Ses documents sont décrit par les clés `id`, `date` (date d'exécution de la méthode), `methodName` (méthode exécutée) et `user` (utilisateur qui a exécuté la méthode).
- La collection **ComponentMethod** contient le descriptif des méthodes s'appliquant aux composants. Un document componentMethod est décrit par les clés `name`, `type` (*event* ou *standard*), `appliedTo` (famille de composant avec laquelle la méthode s'applique), `description` et `computeMethod` (méthode de traitement appelée).

4.3.4 Express.JS

4.3.4.1 Présentation d'Express.JS

Node.JS fournit un module nommé HTTP pour fonctionner en tant que serveur web gérant les connexions des clients. Grâce à ce module, une application Node.JS ne nécessite pas de serveur web comme Apache ou IIS pour fonctionner. Express.JS est un module créé par des développeurs tiers se basant sur le module HTTP de Node.JS mais apportant de nombreuses fonctionnalités pour développer une application web. Pour notre projet, nous nous servons principalement des fonctionnalités permettant de mettre en place notre API grâce à une gestion simple des différences URIs que propose le module.

4.3.4.2 Organisation du code de l'applicatif

Le module Express.JS laisse le développeur libre dans l'organisation du code de son applicatif mais propose un générateur d'applications qui va créer un squelette pour aider à mieux l'organiser. Le squelette propose une organisation sur le modèle MVC (Modèle Vue Contrôleur) avec en plus la gestion des routes. Le modèle MVC sert à séparer une application en trois principaux composants. La partie *modèle* s'occupe du traitement des données, la partie *contrôleur* s'occupe de la gestion d'évènements mettant à jour le modèle et la partie *vue* sert à gérer les interactions entre l'utilisateur et l'application. C'est ce que nous avons utilisé pour notre API comme base.

La *figure 44* représente l'arborescence de notre code avec en commentaire la description de chaque dossier. En plus de ces dossiers on remarque la présence du fichier `app.js` qui est le fichier principal du code Node.JS/Express.JS à partir duquel tous les autres fichiers vont être appelés. Le fichier `package.json` sert à décrire l'ensemble des modules (ou packages) nécessaires au bon fonctionnement de l'application. C'est à partir de ce fichier que le gestionnaire de package NPM télécharge et installe tous les modules. Cela s'avère très pratique pour déployer l'application.

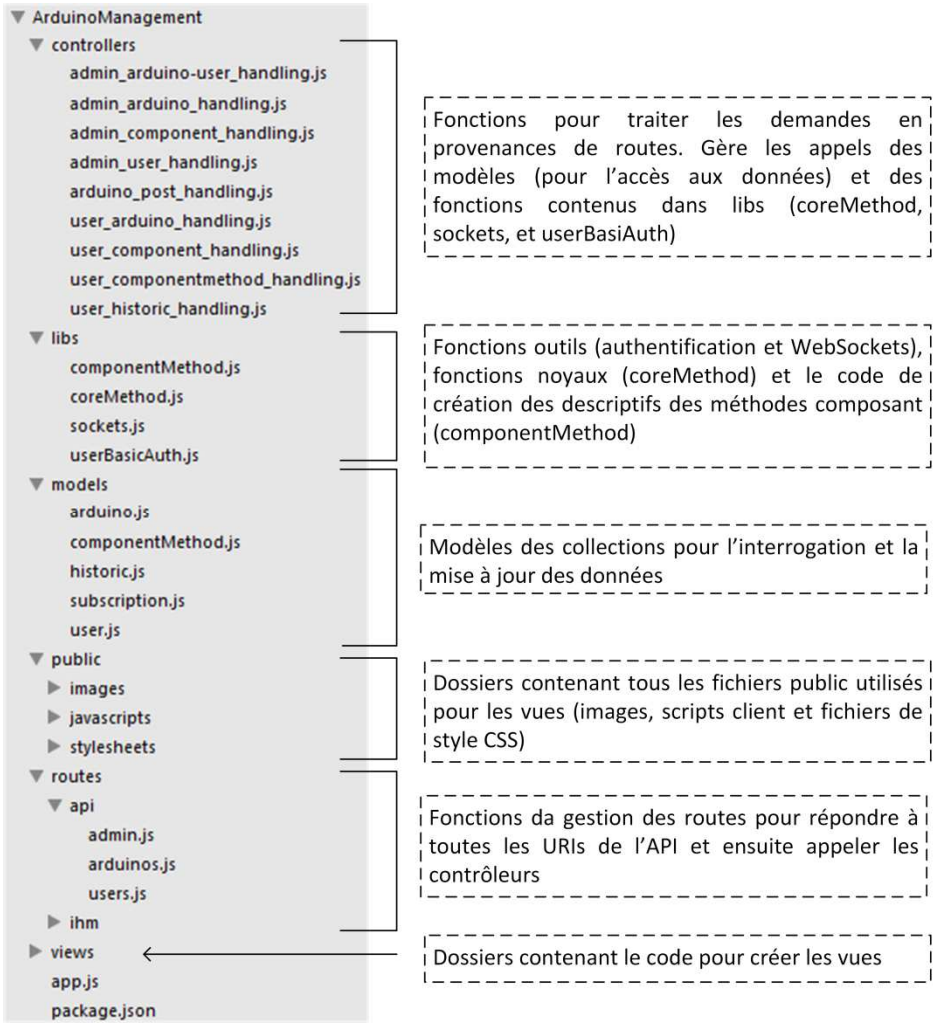


Figure 44 - Organisation du code de l'applicatif

4.3.4.3 Gestion des routes

Le terme route d'Express.JS désigne l'interface uniforme du modèle REST avec les URIs et les verbes HTTP. Le module permet de créer plusieurs fonctions de routage à partir desquels nous appelons une ou plusieurs méthodes de traitements [Routing Express – 2015]. Dans notre architecture les fonctions de routages gèrent l'identification de la requête et appellent les fonctions contrôleurs. Express.JS apporte une manière simple de les mettre en place et les gère dynamiquement.

La gestion dynamique permet de créer une fonction de routage répondant à plusieurs routes construites de la même façon. Nous prenons en exemple deux requêtes désignant deux capteurs de deux nœuds différents : `GET api/arduinos/arduino_01/sensors/`

sensor_01 et GET api/arduinos/arduino_02/sensors/sersor_02. Sur ces requêtes, il y a juste le nom de l'Arduino et le nom du capteur qui diffèrent. Avec Express.JS, pour répondre à ces deux requêtes, la syntaxe de la route est api/arduinos/:arduino_name/ sensors/:sensor_name. Chaque nom précédé de « : » désigne la partie dynamique avec laquelle Express.JS nous crée une variable. Dans le cas de l'exemple, deux variables sont créées contenant le nom de l'Arduino et le nom du capteur avec le formaliste : req.params.arduino_name et req.params.sensor_name. Nous utilisons ensuite ces variables pour appeler les fonctions contrôleurs.

La *figure 45* montre l'exemple d'un bout de code d'une fonction de routage avec la création dynamique de paramètres. L'objet express app propose des méthodes correspondant aux verbes HTTP.

```
// Appel du module express
var express = require('express') ;
// Création d'un objet Express pour notre application
var app = express() ;

// Gestion des requêtes GET api/arduinos/{arduino_name}/sensors/{sensor_name}
app.get('api/arduinos/:arduino_name/sensors/:sensor_name', function (req, res) {
    /* :arduino_name = req.params.arduino_name
       :sensor_name = req.params.sensor_name */

    //Appel de la fonction contrôleur avec les paramètres générés précédemment
    retrieveSensor(req.params.arduino_name, req.params.sensor_name, callback);
});
```

Figure 45 - Exemple d'une route Express.JS

La gestion des routes répondant aux requêtes HTTP constituant notre interface est divisée en trois fichiers:

- Le fichier **admin.js** pour répondre à toutes les requêtes commençant par l'URI api/admin. Ce sont les routes qui appellent les fonctions contrôleurs de gestion administratives.
- Le fichier **user.js** pour toutes les URIs commençant par api/arduinos accessibles par un utilisateur authentifié. Ce sont les routes pour interagir avec les nœuds Arduino auxquels ils ont les droits. Ce fichier comprend aussi les méthodes de routage pour les URIs commençant par api/historics pour accéder à leur historique.
- Le fichier **arduino.js** qui s'occupe d'une route répondant à l'URI POST api/arduino pour traiter les notifications en provenance des nœuds Arduino.

4.3.4.4 Envoie des représentations et des erreurs

Chaque fonction asynchrone contrôleur retourne un objet résultat ou un objet erreur. L'objet résultat comporte la représentation au format JSON de l'état de la ressource demandée. L'objet erreur envoie un message aussi au format JSON indiquant le type d'erreur, son code, son message et le code retour HTTP.

```
{
    type: 'invalid_request_error',
    code: 'actuator_invalid_name',
    httpCode: 404,
    message: 'Arduino "arduino_01" does\'t have actuator "led_01"'
}
```

Figure 46 - Exemple d'un message d'erreur au format JSON

Le protocole HTTP prévoit une nomenclature de codes retours que nous utilisons avec nos propres messages. Cela permet d'indiquer rapidement à un outil utilisant une librairie HTTP le succès ou l'échec des requêtes exécutées. Le *tableau 16* indique les codes retours HTTP utilisés pour notre API

HTTP		
Code	**Message**	**Appliqué à l'API**
200	OK	Indique que la requête a été effectuée. La représentation ou un message signalant le traitement effectué est envoyé
400	Bad Request	Indique que les paramètres pour une requête POST ou PUT ne sont pas corrects. Cela peut signifier qu'il en manque au moins un ou alors qu'ils n'existent pas
403	Forbidden	Echec de l'authentification avec HTTP Basic
404	Not Found	Requête sur une ressource qui n'existe pas.
500	Internal Server Error	Problème réseau lors de la communication avec les nœuds Arduino ou alors erreur non prévue par l'API. Les erreurs non traitées proviennent d'un problème de traitement de la base de données.

Tableau 16 - Codes retours HTTP utilisés par l'API

L'envoi de ces messages vers le client se fait dans la fonction de gestion de la route. Cette fonction asynchrone comporte une fonction callback retournant deux objets. Un objet requête avec toutes les informations sur la requête cliente, et un objet réponse que nous met à disposition Express.JS pour envoyer une réponse au client. La *figure 47* montre un exemple de code qui traite le retour d'une méthode d'un contrôleur pour ensuite envoyer la réponse au client grâce à l'objet `rep` et sa méthode `send()`.

```
/* POST Route /admin/arduinos pour créer un objet arduino */
/*Paramètre POST attendu :
    ip | Adresse ip de l'arduino
*/
app.post('/arduinos', function (req, res) {
    adminArduinoHandling.create(req.param('ip'), function(error, result) {
        if (error) {
            /* En cas d'erreur, utilisation de res avec la méthode send avec
               en paramètre le code http (contenu dans l'objet error de la fonction
               adminArduinoHandling.create) et le message JSON d'erreur complet */
            res.send(error.httpCode, error);
        }
        if (result) {
            /* En de succès, utilisation de res avec la méthode send avec
               en paramètre la représentation JSON
               Si non spécifié res.send envoie un code retour 200 */
            res.send(result);
        }
    });
});
```

Figure 47 - Envoi de la réponse au client avec Express.JS

4.3.5 Temps réel avec Socket.IO

4.3.5.1 Présentation de Socket.IO

Socket.IO est un module pour Node.JS créé par des développeurs tiers. Il a été créé pour apporter à une application un fonctionnement temps réel où le serveur et le client web peuvent dialoguer de manière bidirectionnelle. Socket.IO utilise le protocole WebSocket en apportant une abstraction de son fonctionnement pour créer un canal de communication très simplement. Il implémente une partie serveur proposant une API et une partie cliente utilisée pour dialoguer avec l'API à partir d'un code exécuté dans un navigateur.

La force de Socket.IO, outre sa facilité de mise en place et d'utilisation du protocole WebSocket, est d'utiliser aussi d'autres méthodes de communication en temps réel quand WebSocket n'est pas supporté. En effet, WebSocket n'est supporté que par les navigateurs

récents. Dans le cas où le navigateur ne supporte pas le protocole alors Socket.IO utilise à défaut, des mécanismes plus anciens et moins performants mais fonctionnant avec un plus grand panel de navigateurs. Il prend en compte les mécanismes de transports : FlashSocket, XHR long pooling, XHR multipart streaming, XRH pooling, JSONP pooling et iframe ce qui assure une compatibilité jusqu'à Internet Explorer 6 [Rohit RAI – 2013]. Ce changement de méthode de communication est transparent pour le développeur qui utilise les méthodes de l'API de la même manière quel que soit celui choisi. Cela évite de prévoir plusieurs morceaux de code pour gérer plusieurs navigateurs.

4.3.5.2 Notification quasi temps réel avec l'API

Pour notre application nous utilisons Socket.IO pour envoyer les notifications des objets directement au client. Un utilisateur souscrit au préalable à une méthode d'un objet proposé par l'API, lui permettant d'être notifié. Cette souscription est sauvegardée en base et contient les noms des utilisateurs abonnées. Lorsqu'un objet envoie une notification, le nœud Arduino agit en client HTTP et envoie cette notification au serveur centralisé. Le serveur centralisé traite la demande avec la méthode `process` contenu dans `arduino_post_handling.js` (cf *tableau 14*). Cette méthode détermine quels sont les utilisateurs abonnés à cette souscription et leur envoie la notification avec Socket.IO.

Au niveau SocketIO, pour établir la connexion avec le serveur, le client envoie une demande de connexion vers le serveur ce qui entraine la création d'un objet `socket`. Cet objet est l'interface de connexion du client. Nous utilisons cet objet pour créer une écoute via sa méthode `on` sur un évènement que l'on nomme *arduinoPooling*. Dès qu'un message de notification est envoyé par le serveur avec l'évènement *arduinoPooling*, la méthode `on` récupère le message qui peut être ensuite interprété par le client. Coté serveur, nous utilisons l'objet `sockets` et sa méthode d'écoute `on` sur l'évènement `connection`. Dès qu'un client veut se connecter, la méthode d'écoute se déclenche et créé un objet `socket` pour dialoguer avec lui. Avec cet objet `socket`, nous envoyons les notifications par la méthode `emit` avec l'évènement nommé *arduinoPooling*. La *figure 48* résume le fonctionnement de Socket.IO mis en place.

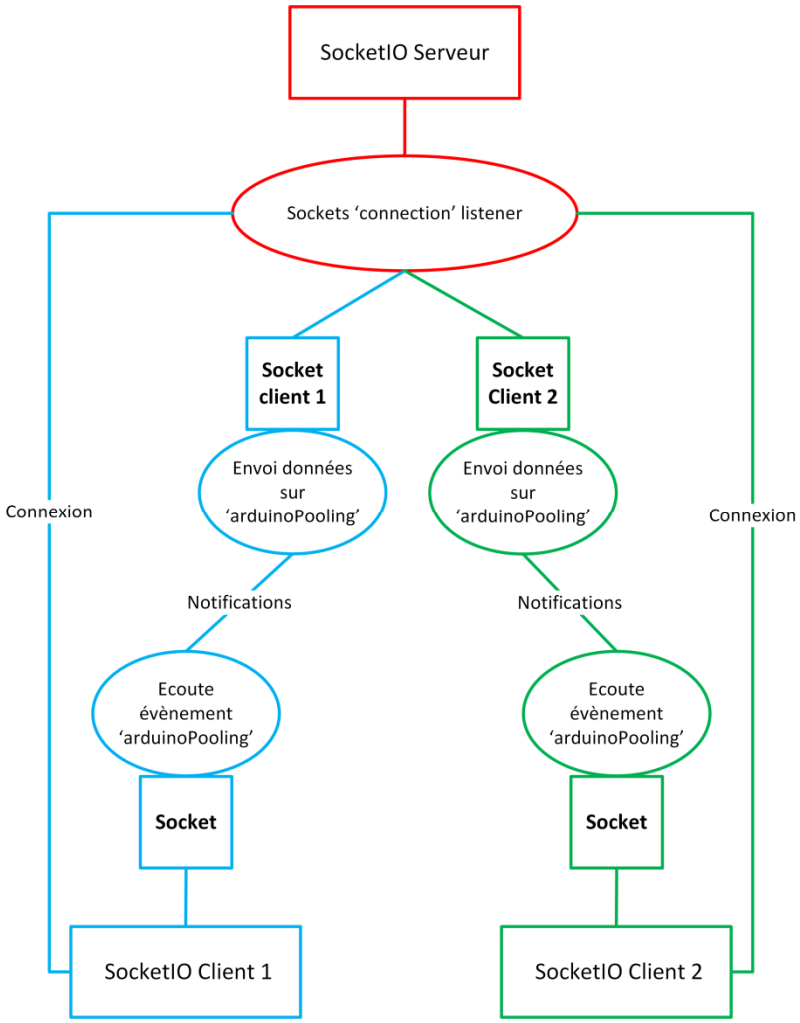


Figure 48 - Fonctionnement Socket.IO pour les notifications Arduino

4.3.6 Schémas résumé de l'applicatif

4.3.6.1 Fonctionnement standard

La *figure 49* propose un résumé des principaux composants de notre architecture logicielle mise en place pour son fonctionnement standard. On appelle standard le fonctionnement client-serveur utilisant l'architecture REST. Les étapes concernant l'authentification ne sont pas représentées.

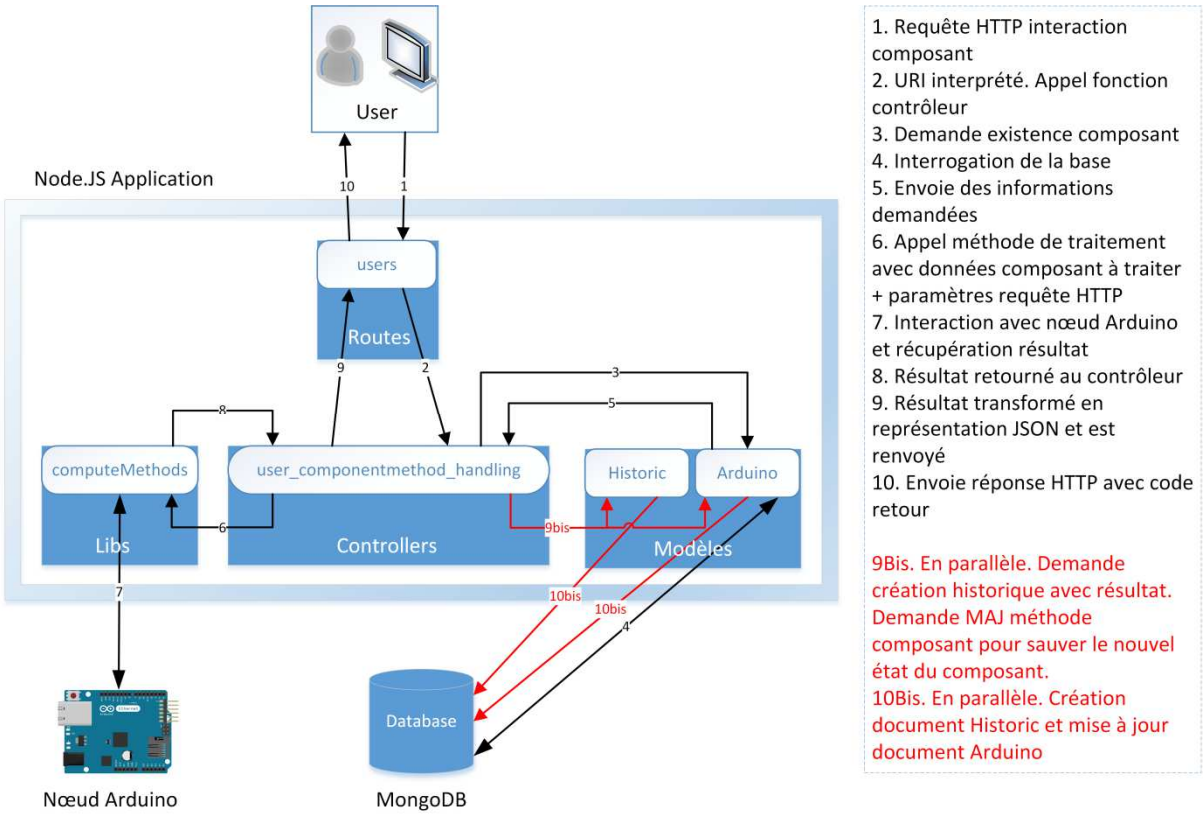


Figure 49 - Schéma d'architecture logicielle pour le fonctionnement standard

4.3.6.2 Fonctionnement avec évènement

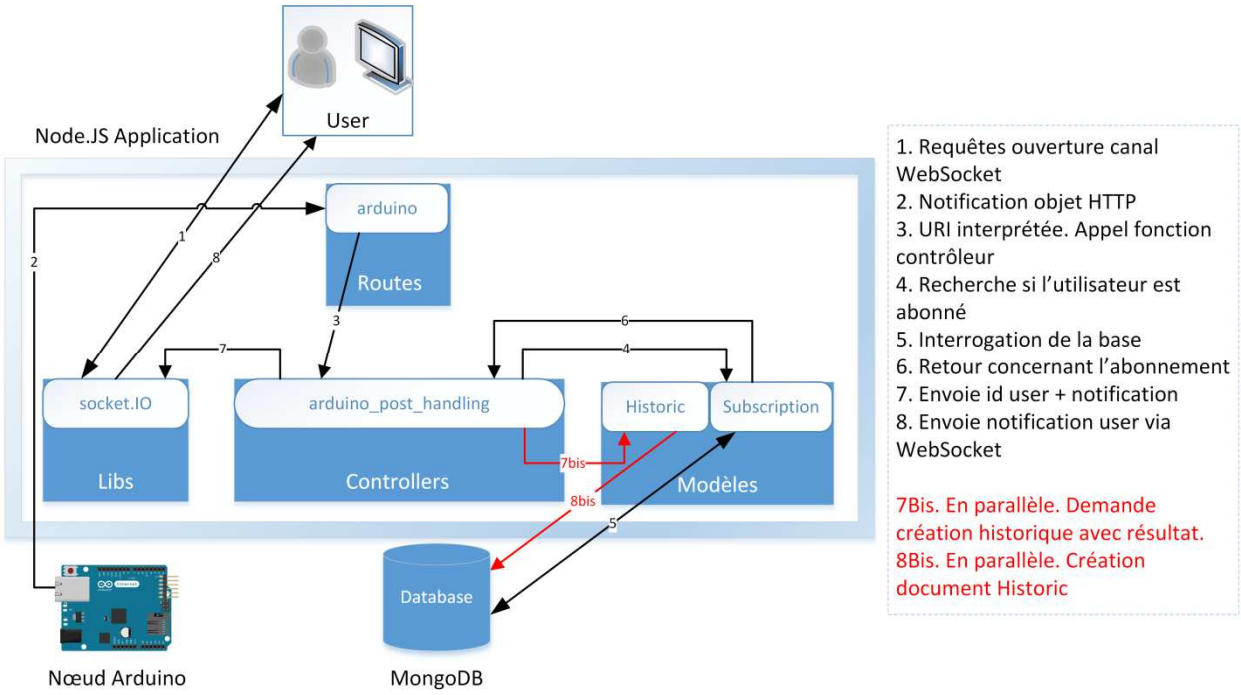


Figure 50 - Schéma d'architecture logicielle pour le fonctionnement avec évènements

La *figure 50* propose un résumé des principaux composants de notre architecture logicielle mise en place pour son fonctionnement avec évènement. On parle d'évènement lorsque les objets envoient des notifications via WebSocket. L'étape de souscription et d'authentification n'est pas représentée.

4.4 Fonctionnalités de l'API

4.4.1 Prérequis

4.4.1.1 Préparation d'un nœud

Avant d'utiliser l'API de notre système nous devons effectuer plusieurs opérations en prérequis. Tout d'abord nous nous occupons de la préparation des nœuds Arduino. Cela consiste à connecter nos composants qui vont devenir les objets communicants du système. Cette étape s'effectue par une personne ayant un minimum de notion en électronique. Pour faciliter cette opération nous créons une documentation avec des schémas réalisés avec l'outil nommé Fritzing qui propose des illustrations pour une grande partie des cartes Arduino ainsi que pour une multitude de composants (cf *Annexe 1*). La *figure 51* montre un exemple de schéma Frtizing pour indiquer comment relier une LED sur une carte Arduino. On remarque l'utilisation d'une résistance de 220 Ohms pour limiter le courant qui traverse la LED et éviter de l'endommager.

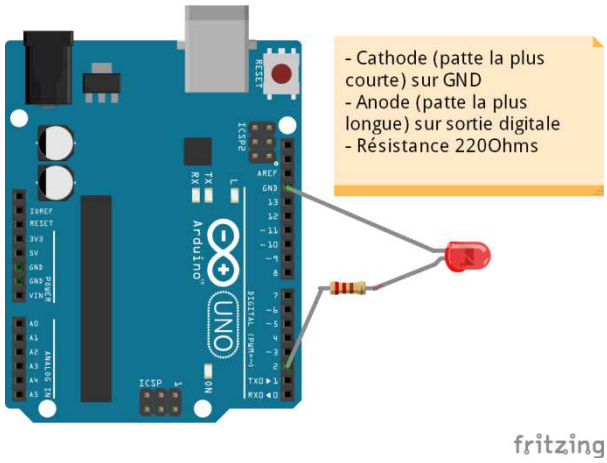


Figure 51 - Schéma Fritzing pour LED

Une fois les composants connectés, nous modifions le fichier de code « modèle » pour Arduino et nous renseignons son nom, son adresse IP, sa localisation, le nom des composants,

le type (*sensor* ou actuator) des composants et la famille à laquelle font partie les composants (LED, Capteur de pression ou de température). Suivant leurs fonctionnements, certains composants nécessitent le paramétrage d'une méthode qui leur est associée. Ce sont les méthodes décrites dans *4.2.4 Méthodes composant coté Arduino*. Toutes les informations nécessaires à cette mise en place se retrouvent dans le code modèle commenté et dans une documentation.

Enfin la dernière opération, si l'applicatif vient juste d'être installé sur le serveur centralisé, consiste à remplir la collection de documents décrivant les méthodes coté serveur qui vont permettre de faire le lien entre les composants et les méthodes de traitement. Cette opération s'effectue par l'administrateur avec la requête `POST http://ip_serveur/api/admin/componentmethods` qui entraine la création des documents de la collection *componentMethods*.

4.4.1.2 Outil client HTTP

Pour dialoguer avec notre API par l'envoi de requêtes HTTP, nous utilisons un outil open source nommé cURL (abréviation de Client URL Request Library). C'est un outil en ligne de commande de transfert de données via URL. Il supporte un grand nombre de protocoles basé sur les URIs comme par exemple HTTP, HTTPS, FTP, LDAP et bien d'autres [cURL – 2015]. De plus une librairie est aussi proposée pour l'utiliser dans plusieurs langages. La *figure 52* montre comment nous utilisons cet outil de ligne de commande avec deux exemples. Nous ne faisons pas apparaitre le retour des commandes mais celui-ci est envoyé dans la sortie standard.

```
// Syntaxe de cURL
curl --user [user:password] -X [GET|POST|PUT|DELETE] --data ["parameter=value"]
[URI]

// Requête user1 pour connaitre les nœuds Arduino dont il a les droits d'accès
curl --user user1:secret -X GET URI http://ip_server/api/arduinos

// Requête user1 pour allumer la LED led_01 sur le nœud arduino_01
curl --user user1:secret -X PUT --data "action=on" URI
http://ip_server/api/arduinos/arduino_01/actuators/led_01/methods/light
```

Figure 52 - Utilisation de cURL

4.4.2 Administration

Toutes les opérations de gestion des nœuds Arduino et des utilisateurs se fait à partir du point d'entrée `/admin` de l'API. Pour notre système nous n'avons pas pris en compte l'aspect authentification de l'administrateur, car pour notre proof of concept, la sécurité n'est pas l'objectif principal. Pour administrer, il suffit d'exécuter les requêtes à partir du point d'entrée afin d'accéder aux fonctionnalités.

Les premières opérations de l'administrateur sont de référencer les nœuds Arduino et leurs objets au sein du serveur centralisé. Pour cela il utilise la requête `POST http://ip_serveur/api/admin/arduinos` avec dans le corps de la requête le paramètre `ip` contenant l'adresse IP de l'Arduino. Cette requête déclenche un ensemble de fonctions qui, dans un premier, récupère la description au format JSON de l'Arduino. Cette description regroupe la définition de l'Arduino et de ses composants. A partir de cette description, un document Arduino est créé en base avec les informations de l'Arduino, de ses composants et des méthodes qui sont attribuées aux composants.

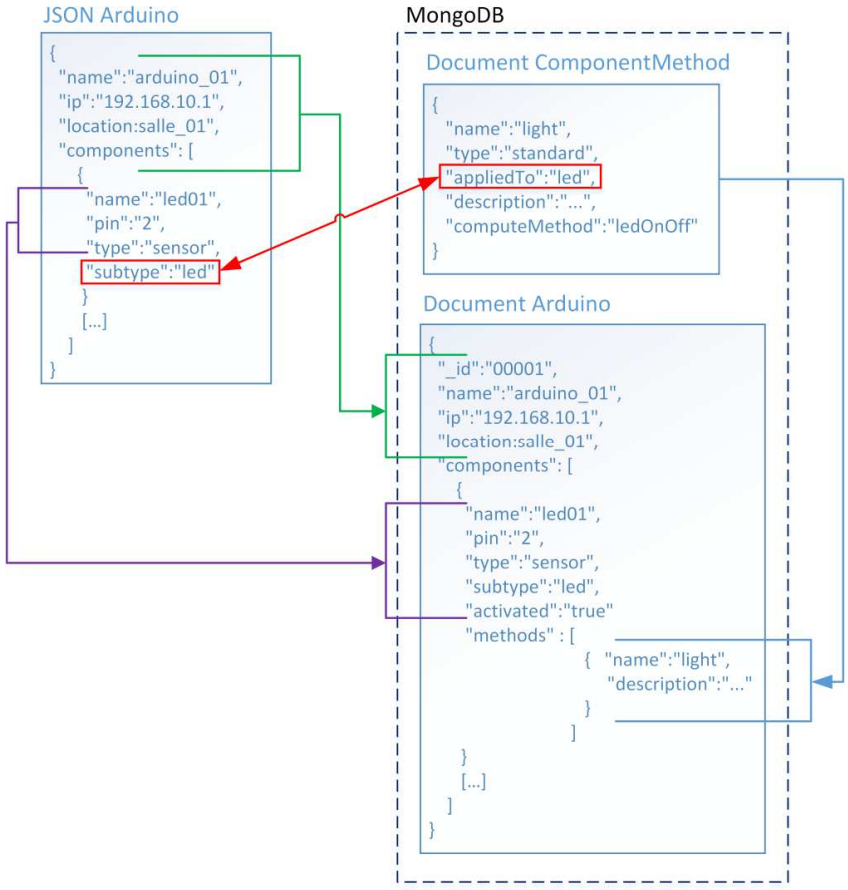


Figure 53 - Création d'un document Arduino

La *figure 53* montre que le document Arduino est créé à partir du JSON de description du nœud, et, pour les composants et leurs méthodes, à partir des documents de la collection ComponentMethod. Pour chaque composant du JSON Arduino, l'application recherche la méthode composant à appliquer en faisant correspondre les valeurs de `subtype` et `appliedTo`. A partir de cela les informations concernant la méthode sont insérées dans le document Arduino pour le composant. Lorsqu'un utilisateur exécute une méthode d'un composant récupérée depuis un document Arduino, l'application fait une recherche du nom de la méthode dans la collection ComponentMethod pour récupérer le document correspondant et ensuite exécuter la méthode de traitement indiquée par la clé `computeMethod`.

Ensuite l'administrateur utilise l'API peut créer des comptes utilisateurs avec la requête `POST http://ip_serveur/api/admin/users` avec dans le corps de la requête les paramètres `name` et `password` pour indiquer le nom et le mot de passe de l'utilisateur. A ces utilisateurs, il leur ajoute les droits d'utilisation d'un Arduino et par conséquent de ses objets s'ils sont activés. Pour cela il se sert de la requête `POST http://ip_serveur/api/admin/users/{nom_user}/arduinos` avec dans le corps de la requête le paramètre `arduino` ayant pour valeur le nom de l'Arduino. L'API offre aussi à l'administrateur, les possibilités de retirer les droits d'un Arduino et de supprimer un utilisateur.

4.4.3 Utilisateurs

Les fonctionnalités de l'API offertes à l'utilisateur nécessitent que celui-ci s'identifie pour chaque requête qu'il devra effectuer. Si la gestion de l'authentification n'est pas l'aspect le plus important de notre proof of concept, nous avons quand même mis en place une authentification avec la méthode basique d'HTTP. Cette méthode du protocole HTTP permet de transporter dans la requête le couple identifiant/mot de passe que le serveur va pouvoir ensuite analyser en interrogeant la collection user. Si les identifiants sont bons alors la requête est traitée. Cette authentification n'est pas bien sécurisée car les identifiants sont cryptés avec un algorithme facilement déchiffrable. Dans notre système cette authentification nous sert surtout pour identifier l'utilisateur qui effectue la requête.

L'utilisateur a accès à l'ensemble des informations d'un document d'un nœud Arduino créé par l'administrateur. Les informations sont découpées en trois parties : Arduino, composant et

méthode. Une requête `GET http://arduinos/{nom_arduino}` interroge le document Arduino et construit une représentation JSON sans inclure la liste de composants et de leurs méthodes. Pour connaitre les composants, l'utilisateur rajoute à sa requête `/sensors` ou `/actuators` pour avoir la liste des objets de type capteur ou actionneur que comporte le nœud Arduino. Cette liste comporte le nom des méthodes attribuées à chaque composant. Pour avoir le détail des méthodes d'un composant l'utilisateur envoie la requête `GET http://ip_serveur/api/arduinos/{nomarduino}/actuators/{nomactuator}/methods`. Ensuite il peut communiquer avec le composant grâce à ses méthodes avec les requêtes `GET` et `PUT` sur une URI identifiant une méthode.

Enfin l'API propose à l'utilisateur une gestion de l'historique des données remontées par les objets. La requête `GET http://ip_serveur/api/historics` lui renvoie la liste de tout son historique où il peut récupérer la date, le nom de l'objet et le résultat de toutes les méthodes qu'il a exécutées. Il a aussi la possibilité de supprimer des entrées dans l'historique une à une ou toutes d'un coup.

4.4.4 Historisation

L'historisation ou journalisation des résultats s'effectue à chaque exécution d'une méthode par un utilisateur. Cette historisation est traitée quasiment en parallèle de l'envoi du résultat à l'utilisateur par une méthode de traitement d'un composant. Par contre elle ne renvoie pas d'erreur directement à l'utilisateur en cas d'échec car celui-ci reçoit déjà un retour de la méthode exécutée. L'historisation s'effectue aussi lors d'une notification d'un objet. Elle a lieu en parallèle avec l'envoie du résultat par WebSocket.

4.4.5 Mise à jour du système

Mettre à jour le système pour qu'il propose de nouveaux objets communicants utilisables dans un environnement ambiant intelligent nécessite plusieurs opérations. Premièrement il faut que le nouvel objet fonctionne avec une carte Arduino au niveau électrique et au niveau de l'interface de communication. Si son fonctionnement n'est pas une simple utilisation des entrées/sorties digitales ou analogiques, une méthode coté Arduino doit être créée.

Coté API du serveur centralisé, nous devons dans un premier temps créer une méthode de traitement (dans `computeMethod.js`) qui contient le code pour communiquer avec l'objet et transformer ses résultats pour l'utilisateur. Ensuite nous devons créer une description de l'utilisation de cette méthode en indiquant à quelle famille d'objet elle est destinée dans un document contenu dans la collection ComponentMethods. Enfin la dernière opération consiste pour l'administrateur à faire une mise à jour des nœuds Arduino proposant ce nouvel objet.

4.5 Synthèse

Pour nos Arduino, nous avons réalisé une mini API pour essayer au maximum de retourner les informations des objets en données brutes. Le but étant de traiter les informations coté serveur centralisé. Côté serveur centralisé, nous avons orienté le développement de l'application sur des technologies utilisant le langage JavaScript. Le coupe Node.JS et Express.JS nous a permis de réaliser une API traitant dynamiquement les requêtes. Enfin Socket.IO nous a apporté la gestion de la communication temps réel.

Notre système répond à toutes les spécifications que nous avons réalisées lors de la conception de notre proof of concept. Nous pouvons le tester et le valider dans un scénario nécessitant l'apport d'une intelligence ambiante.

Chapitre 5 : Validation du proof of concept

5.1 Cas allocation dynamique des voieries

5.1.1 Présentation

Au sein de l'équipe Silex, des travaux concernent le domaine des « smart cities ». Ce terme définit les villes utilisant les technologies de l'information et de la communication pour améliorer leurs gestions. Cela concerne plusieurs domaines tels que la gestion de l'énergie, des transports, des services d'urgence ou encore la gestion des déchets et du tri.

Le doctorant Chen Wang oriente ses travaux dans le domaine de la gestion du trafic dans une « Smart City ». Le sujet de recherche est l'utilisation de l'internet des objets et des services basés sur la localisation pour gérer dynamiquement le trafic routier. L'internet des objets apporte la mise en place de capteurs et d'actionneurs dialoguant sur un même réseau pour remonter des informations et agir sur l'environnement. Les services basés sur la localisation apportent des services aux utilisateurs en fonction de leur localisation géographique. Le cas d'étude des recherches du doctorant Chen Wang est d'utiliser ces deux principales techniques pour concevoir un système d'allocation dynamique des voieries.

La *figure 54* présente les idées derrière l'allocation dynamique des voieries dans le cas d'une route avec un arrêt de bus. Tous les véhicules et les utilisateurs sont équipés de périphériques avec une HCI (interface homme machine) leur remontant des informations concernant le trafic. Lorsqu'un bus approche d'un arrêt, les services basés sur la localisation informent les

usagers que le bus est en approche. Les voitures sont informées de sa présence et libèrent la voie de droite pour le bus. Un panneau d'affichage en bord de voie (EDB) indique que celle-ci est réservée.

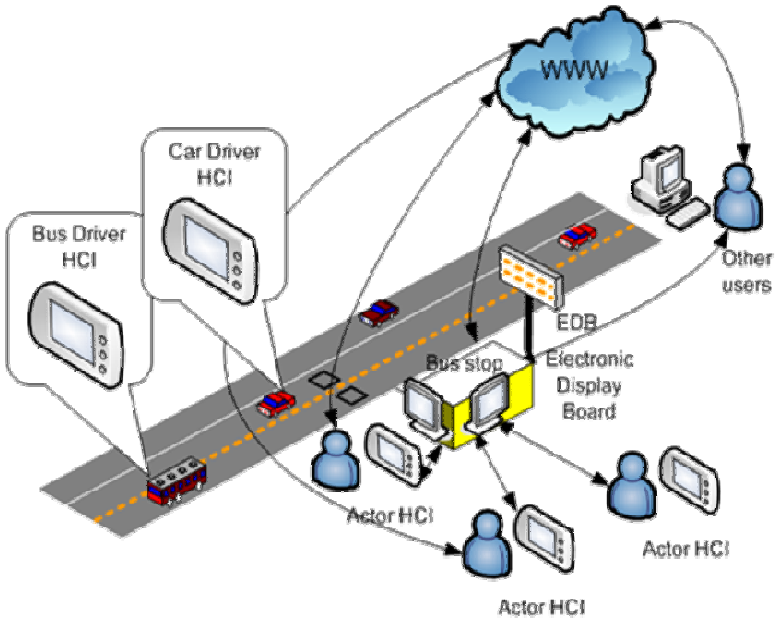


Figure 54 - Vue d'ensemble de l'allocation dynamique des voieries

Dans ce contexte de recherches, nous établissons un scénario pour valider notre système d'intelligence ambiante. Notre système apporte une solution d'utilisation de l'internet des objets pour ce cas d'étude.

5.1.2 Scénario

Nous souhaitons réaliser technologiquement l'augmentation des voieries pour apporter une détection de véhicule. Le scénario est d'identifier l'arrivée d'un véhicule prioritaire et de lui réserver un tronçon de voie qui sera interdite aux usagers « normaux ». Lorsque le véhicule est détecté à l'entrée du tronçon, un panneau d'affichage électronique indique que la voie est fermée par un signale rouge. Lorsqu'il est détecté en sortie du tronçon le panneau affiche un signal vert et la voie est de nouveau disponible.

Pour réaliser ce scénario nous créons une maquette que nous équipons avec notre système de création d'intelligence ambiante.

5.1.3 Réalisation de la maquette

Nous construisons la maquette avec une planche de carton sur laquelle nous dessinons trois voies. Nous équipons la voie de droite avec deux lecteurs RFID servant à lire un transpondeur au début et à la fin de la voie. Nous fabriquons un panneau d'affichage surplombant les trois voies avec du carton dans lequel nous intégrons des LED. Chaque voie a ainsi deux LED, une rouge et une verte, pour signaler si l'on peut l'emprunter ou non. Pour faire communiquer tous ces dispositifs nous utilisons trois cartes Arduino : deux Arduino Ethernet s'occupant chacun d'un lecteur RFID et un Arduino Mega pour le panneau d'affichage.

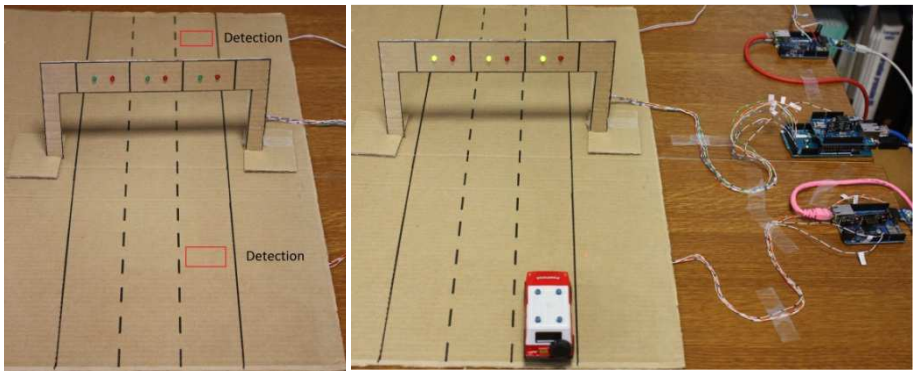


Figure 55 - Vues d'ensemble de la maquette

Nous disposons d'une voiture miniature sur laquelle nous fixons un transpondeur RFID qui peut être lu par les lecteurs RFID de marque Parallax. Ces lecteurs sont positionnés sous la maquette aux emplacements `Dectection` indiqués dans la *figure 55*.

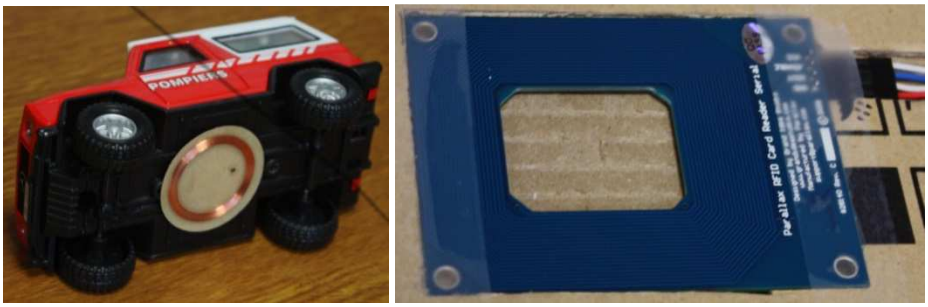


Figure 56 - Transpondeur et lecteur RFID

5.1.4 Utilisation du système d'intelligence ambiante

5.1.4.1 Administration

Avec le code modèle Arduino nous paramétrons nos trois cartes que nous nommons arduinorfid1, arduinorfid2 et arduinoled. Pour chaque code Arduino nous décrivons chaque composant connecté aux cartes. Ces trois Arduino sont reliés au serveur centralisé Intel NUC sur le même réseau filaire via un switch. Tous les quatre sont configurés sur le même plan d'adressage.

Au niveau de l'API du serveur centralisé, nous ajoutons nos trois nœuds en exécutant trois requêtes `POST http://ip_serveur/api/admin/arduinos` avec pour chacune d'entre elles l'adresse IP envoyée dans le paramètre `ip`. Ensuite nous créons un compte utilisateur nommé *userScenario* qui exécutera les requêtes sur les objets communicants de la maquette. Ceci est réalisé par la requête `POST http://ip_serveur/api/admin/users` avec dans le corps de la requête les paramètres `name` et `password` contenant le nom et le mot de passe du compte. La dernière étape d'administration est d'attribuer les droits sur nos trois nœuds Arduino à l'utilisateur *userScenario*. Pour cela nous envoyons au serveur trois requêtes `POST http://ip_serveur/api/admin/users/userScenario/arduinos` avec pour chacune d'entre elles le nom de l'Arduino envoyé dans le paramètre `arduino`.

Pour chacune de ces opérations nous vérifions que les codes et les messages retours n'indiquent aucun problème. Pour apporter une deuxième vérification nous interrogeons l'API pour vérifier que les ressources ont bien été créées avec la méthode `GET` sur les URIs `admin/arduinos`, `admin/users` et `admin/users/userScenario/arduinos`.

Enfin il nous reste à demander au serveur de nous envoyer les messages en provenance des objets lecteur RFID. Pour cela nous utilisons la méthode de l'API proposé par un objet RFID nommé *reading* qui est de type *event*. La méthode inscrit l'utilisateur *userScenario* dans la liste des utilisateurs qui seront notifiés en cas de lecture. Nous exécutons la requête `PUT http://ipserveur/arduinos/arduinorfid1/sensors/rfid1/methods/reading` avec le paramètre `action = subscribe`. Ainsi nous serons notifiés par l'objet rfid1 du nœud arduinorfid1. On répète cette opération pour l'objet rfid2 du nœud arduinorfid2.

5.1.4.2 Développement du scenario

La phase d'administration étant terminée, nous développons une petite application Web coté client se servant de l'API pour dérouler le scénario. Nous choisissons de la développer en JavaScript pour rester dans l'utilisation d'un même langage.

Gestion des LED

Pour piloter nos LED nous utilisons la librairie jQuery. Cette librairie dont le slogan est « write less, do more » [jQuery – 2015], que l'on peut traduire par « écrire moins, faire plus », propose une multitude de fonctions pour faciliter le développement JavaScript coté client. Nous l'utilisons pour ses fonctions AJAX (Asynchronous JavaScript and XML) qui nous permettent d'exécuter des requêtes HTTP pour dialoguer avec notre API et récupérer les messages JSON. Ce sont ces fonctions que nous utilisons pour dialoguer avec les objets LED et ainsi les allumer ou les éteindre.

```
/* Fonction Ajax utilisant les paramètres:
   - url : URI de la ressource
   - type : verbe HTTP
   - data : paramètre du corps de la requête
   - async : asynchrone ou non */
$.ajax({
    url : "http://" + ip_server +
"/api/arduinos/arduinoled/actuators/greenLed1/methods/light",
    data: {action: 'on'},
    async: false,
    type: 'PUT',
}).fail(function (error) {
        alert(error.message);
    }
});
```

Figure 57 - Fonction AJAX en jQuery pour allumer une LED

La *figure 57* montre la construction d'une fonction AJAX pour interagir sur une LED nommée *greenLed1* avec sa méthode *light*. On remarque le paramètre `async` qui a la valeur `false` ce qui oblige la fonction à terminer son traitement avant que le code interprète la suite. Nous utilisons ce fonctionnement car quand nous enchainons les fonctions de manières asynchrones pour allumer toutes nos LEDs, le nœud Arduino n'arrive pas à traiter assez vite les demandes qui arrivent quasi simultanément. Ce paramétrage permet d'enchainer les fonctions pour allumer ou éteindre les LED une à une.

Nous créons deux fonctions nommées *reserverVoie1()* et *libererVoie1()*. La première contient deux fonctions AJAX pour éteindre la LED verte et allumer la LED rouge de la voie de droite. La deuxième effectue la même opération dans le sens inverse.

Gestion des lecteurs RFID

Nous connaissons l'identifiant du tag de la voiture miniature et nous devons le récupérer dès sa lecture, c'est-à-dire au moment où la voiture miniature roule au-dessus du lecteur. Pour récupérer ces notifications en « temps réel » nous utilisons la librairie Socket.IO cliente. Dans un premier, nous établissons la connexion via la méthode `io.connect(ip_serveur)`. Puis nous émettons un message au serveur contenant le nom de l'utilisateur pour que celui-ci connaisse l'identité de l'utilisateur connecté au canal WebScoket. Le serveur peut alors envoyer les notifications en utilisant le principe des évènements.

La *figure 58* montre le bout code mettant en place l'écoute sur un évènement nommé *arduinoPooling* qui sera appelé à chaque envoie de message du serveur et comment sont traités ces messages.

```
// Connexion avec le serveur
var socket = io.connect('ip_server');

// Ecoute de l'évènement arduinoPooling
socket.on('arduinoPooling', function (data) {
    if (isScenarioStarted) {
        // Reception tag RFID 010CE098E5 depuis l'arudinorfid1
        if (data.return.value == '010CE098E5' && data.arduino == "arduinorfid1") {
            /* Appel de la fonction pour changer la signalisation du panneau
               d'affichage pour réserver la voie */
            reserverVoie1();
        }
        // Reception tag RFID 010CE098E5 depuis l'arudinorfid2
        if (data.return.value == '010CE098E5' && data.arduino == "arduinorfid2") {
            /* Appel de la fonction pour changer la signalisation du panneau
               d'affichage pour libérer la voie */
            libererVoie1();
        }
    }
});
```

Figure 58 - Reception de l'identifiant RFID avec Socket.IO

Exécution du code scenario

Pour exécuter le code nous avons juste créé un bouton HTML nommé *Start* pour mettre le panneau central contenant les LED dans un état initial ou toutes les LED vertes sont allumées

et où le canal WebSocket avec le serveur est ouvert. Ensuite nous faisons avancer la voiture pour obtenir le résultat escompté.

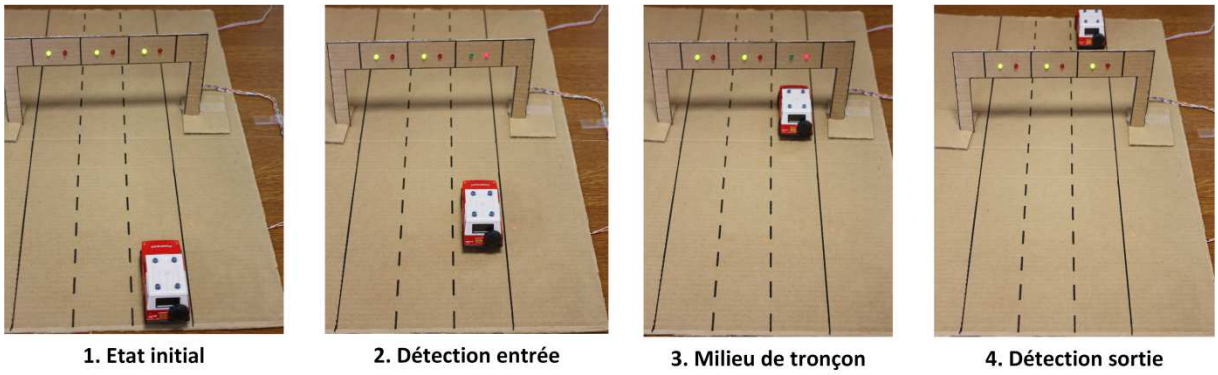


Figure 59 - Déroulement du scénario

5.2 Bilan et perspectives

5.2.1 Bilan

5.2.1.1 API REST

Notre système présente une idée d'architecture apportant une intelligence ambiante à l'aide d'objets de type capteur et actionneur basé sur l'utilisation d'une API. Le modèle REST utilisé pour la construction de celle-ci offre une interface uniforme proposant des services d'utilisations de ces objets de manière simple et leur permet de communiquer à travers le Web avec des messages JSON. Un utilisateur s'intéressant un peu au fonctionnement du Web n'aura aucun mal utiliser l'API pour la partie découverte des objets car les requêtes `GET` fonctionnent sur un navigateur. Les retours au format JSON s'afficheront sur celui-ci et il sera possible pour l'utilisateur de les interpréter.

L'API est aussi facile d'utilisation pour un profil développeur grâce à son architecture REST sur HTTP. La totalité des langages de programmation Web intègrent une librairie HTTP couvrant tous les besoins techniques pour utilisant l'API. Nous l'avons montré avec l'utilisation de la librairie jQuery, une simple fonction permet d'exécuter une requête HTTP pour piloter des LED. Nous pouvons imaginer la création d'une interface pour la domotique

permettant de piloter à distance son éclairage, la mise en marche de divers appareils ou la récupération de la température dans différentes pièces.

5.2.1.2 Gestion des requêtes HTTP par l'Arduino

L'Arduino offre un moyen de rendre communiquant n'importe quel composant qu'il peut faire fonctionner avec l'architecture que nous avons mis en place. Il est peu couteux et bénéficie d'une grande communauté très active. Cela permet de voir arriver de nouvelles librairies et de nouveaux matériels agrandissant ses possibilités. Par contre, avec les modèles choisis et notre configuration pour notre proof of concept, il s'avère qu'ils ne gèrent pas très bien un grand nombre de requêtes sur un court instant. Lors de la réalisation de notre scénario nous avons dû enchainer les requêtes pour allumer les LED contenu dans les fonctions AJAX une à une, car l'exécution asynchrone les envoyant quasiment simultanément ne permettait pas à l'Arduino de toutes les traiter.

5.2.1.3 Temps réel avec le protocole Websocket

L'accès au temps réel avec Socket.IO et les WebSocket apporte un moyen très intéressant aux objets d'envoyer directement l'information mais n'est pas aussi simple d'utilisation que l'utilisation d'HTTP sur le modèle REST. Dans notre scenario l'utilisation d'un objet fonctionnant par évènements nécessite d'utiliser la librairie cliente Socket.IO ce qui limite les possibilités d'intégration dans divers développements. En effet cette librairie est développée en JavaScript et est donc prévue pour une application cliente dans ce langage.

L'utilisation des WebSockets via Socket.IO nécessite de connaitre le fonctionnement des différentes méthodes pour se connecter au serveur et aussi de connaitre le nom des évènements à écouter. Une fois cela mis en place, la récupération de données d'un évènement fonctionne très bien et cela en un temps rapide.

5.2.2 Perspectives

5.2.2.1 Sécurité

Durant l'élaboration de ce proof of concept l'aspect sécurité a été mis de côté pour se concentrer sur la capacité à rendre nos composants reliés à l'Arduino communicant. Néanmoins nous avons réfléchi à plusieurs pistes d'implémentation de la sécurité pour plusieurs niveaux.

D'un point de vue réseau, tous les nœuds Arduino doivent faire partie d'un réseau privé dans lequel aucune personne ne peut accéder depuis l'internet public. Avec un routeur nous pouvons faire le lien entre le réseau public et privé. Pour sécuriser notre serveur centralisé, celui-ci doit être placé dans une DMZ (DeLimitarized Zone ou zone démilitarisé) grâce à un firewall. Ce matériel apporte une politique de filtrage des données. Notre serveur centralisé reçoit seulement les requêtes prévenant de l'internet public utilisant le port spécifique de l'application Node.JS. Le firewall interdit tout accès aux nœuds Arduino sauf depuis le serveur centralisé.

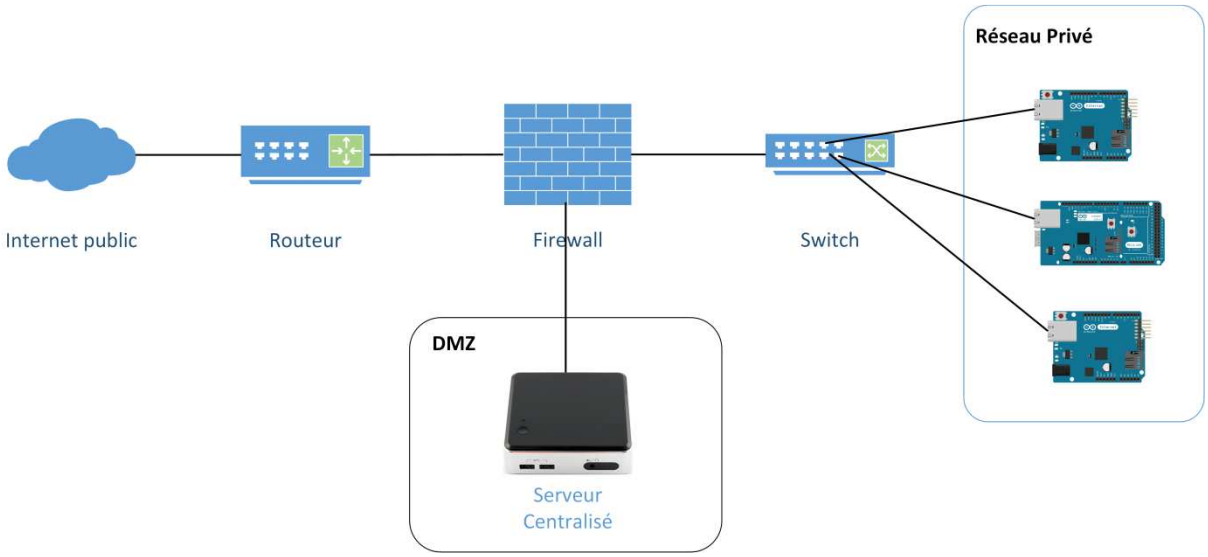


Figure 60 - Sécurité réseau pour le système

Au niveau des échanges de données, il existe le protocole SSL (Secure Sockets Layer) très utilisé dans le Web qui permet de crypter les échanges. Appliqué à HTTP, cela a donné le protocole HTTPS, le « S » rajouté signifiant Secure. L'utilisation de SSL apporte une couche

de cryptage qui garantit la confidentialité des données échangées. Avec ce protocole, l'utilisation de l'authentification basique d'HTTP devient plus intéressante car il n'est pas possible de récupérer l'identifiant et le mot de passe des clients en clair. Le protocole WebSocket peut lui aussi s'utiliser avec la couche SSL afin de crypter ses échanges

5.2.2.2 Interface end-user programming

La finalité d'utilisation de ce proof of concept était de proposer aux utilisateurs non développeur un accès aux objets connectés de notre système pour réaliser des routines simples pour créer leur intelligence ambiante. C'est le dernier niveau d'abstraction que le système doit proposer. Le développement d'une interface end-user programming n'a pas pu être réalisé pendant la durée du stage. Cependant, nous proposons des pistes pour faire évoluer notre système en incluant ce type d'interface.

Nous pensons à une programmation visuelle basée sur des blocs. Ces blocs pourront s'assembler comme des pièces d'un puzzle et créer une routine. Les blocs seront répartis en catégories dont une catégorie regroupe les opérations contrôle pour démarrer, arrêter le programme, créer des boucles et des conditions. D'autres catégories contiendront les blocs capteurs et actionneurs proposant leurs méthodes d'interaction associées. Tous ces blocs possèderont des inscriptions claires dans la langue de l'utilisateur pour définir leur rôle. L'interface contiendra plusieurs panneaux dont un avec toutes les catégories de blocs, un contenant tous les blocs de la catégorie choisie et un dernier où l'on fera glisser les blocs de nos catégories pour les assembler et créer notre programme.

Ce type d'interface existe déjà dans le monde de la programmation visuelle et a été adapté pour différentes cartes de physical computing. Par exemple Scratch propose un outil de programmation visuelle utilisant les blocs. Il a été conçu pour offrir aux enfants un outil de réalisation d'histoires interactives, de jeux et d'animations [Scratch – 2015]. Ce logiciel est sous licence libre et les sources sont disponibles, ce qui a permis à une équipe de chercheurs de créer S4A (Scratch for Arduino). La *figure 61* présente l'interface de S4A avec un programme changeant l'état d'une sortie numérique 10 fois de suite. Ce programme permet, par exemple de faire clignoter une LED connectée à cette sortie.

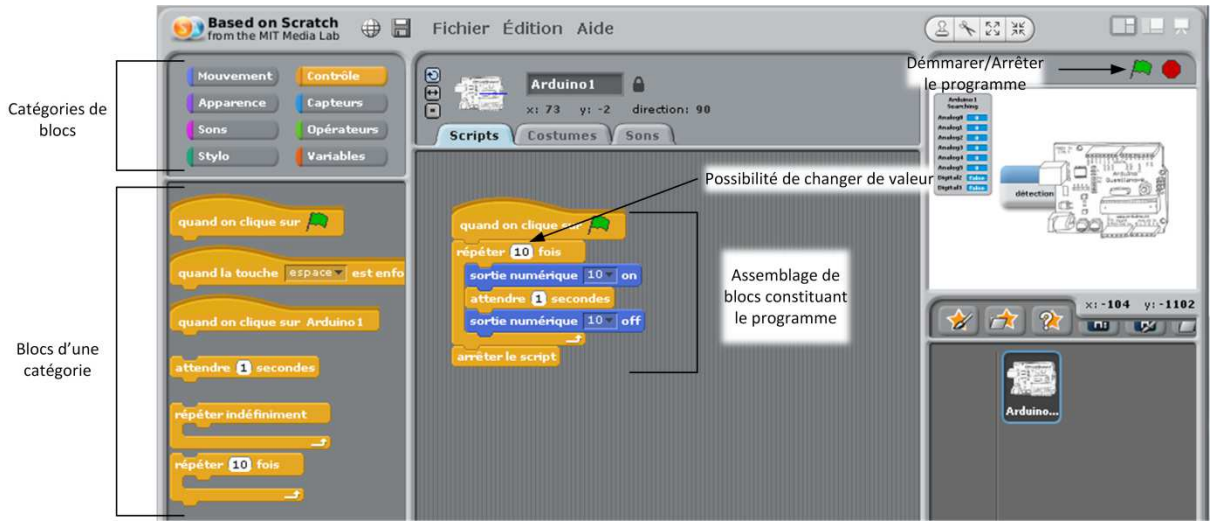


Figure 61 - Programmation visuelle avec S4A

S4A est prévu pour fonctionner en direct depuis un PC et l'Arduino en liaison série et ne convient pas en l'état actuel à notre proof of concept où les échanges sont réalisés via HTTP et TCP/IP. Il faudrait étudier la possibilité de l'adapter à notre besoin en récupérant les sources du programme afin de l'adapter à une interface web.

Il existe un autre projet de ce type nommé Wyliodrin. Il est compatible avec plusieurs cartes du monde du physical computing comme l'Intel Edison, l'Intel Galileo, le RaspberryPi et l' Beaglebone Black [Wyliodrin – 2015]. Wyliodrin permet aussi de programmer pour Arduino mais celui-ci droit être relié à un Raspberry PI. Ce projet n'est pas gratuit et ne propose pas de code source. Par contre il a été développé à partir de la librairie open source nommée Blocky, créée par Google pour construire des éditeurs de programmation visuelle [Blocky – 2015].

La librairie Blocky offre un développement orienté web, coté client, avec l'utilisation de JavaScript. Elle semble être un choix viable pour créer une interface de programmation visuelle à base de blocs qui soit basé sur l'utilisation de notre API. Il faudrait créer des catégories par famille d'objets. Chaque bloc objet serait présenté avec son nom et le nœud auquel il appartient. Nous pourrions utiliser ces blocs avec des blocs méthodes s'emboitant qu'avec les blocs objets pour lesquels ils sont prévus. Ces blocs méthodes s'exécuteraient suivant des paramètres et retourneraient les résultats dans des variables. Par exemple, pour profiter des notifications temps réel d'un objet lecteur RFID, nous aurions un bloc méthode prenant en paramètre l'identifiant attendu pour ensuite déclencher un autre bloc. Ou alors

l'identifiant serait attribué dans une variable que l'on pourra réutiliser avec des blocs conditions afin de traiter plusieurs actions suivant plusieurs identifiants.

5.2.2.3 Amélioration des traitements

Lors des tests réalisés sur notre proof of concept, nous nous sommes rendu compte que l'Arduino Ethernet et l'Arduino Mega avec shield ethernet supportaient mal la montée en charge des requêtes. L'utilisation de notre système pour l'élaboration de la maquette a montré que plusieurs requêtes envoyées rapidement ne pouvaient être traitées.

Nous avons réfléchi à plusieurs manières d'améliorer le traitement d'un plus grand nombre de requêtes. Nous pourrions utiliser un Raspberry PI dont la fonction est d'être un serveur web assurant la gestion des requêtes et de la communication TCP/IP. Il serait relié à un Arduino via USB et communiquerait sur une liaison série. Cette solution est utilisée par le projet Wyliodrin pour programmer visuellement sur Arduino. L'intérêt du Raspberry PI est qu'il bénéficie de caractéristiques techniques plus adaptées pour être utilisé comme un serveur web. Son processeur est plus rapide, il comporte plus de mémoire vive et son contrôleur réseau est plus performant que celui d'un shield Arduino. De plus son système d'exploitation est un noyau Linux et permet l'installation de logiciels de serveur web tels qu'Apache ou Nginx. Ces logiciels proposent de très grandes possibilités de configurations pour s'adapter au matériel. Pour valider cette solution il faudrait effectuer des tests de charge sur la vitesse de traitement et de communication entre un Raspberry PI et un Arduino via liaison série. I faudrait ensuite étudier la rapidité du serveur Web suivant les différents logiciels proposés.

Un des objectifs du proof of concept était d'utiliser les cartes Arduino, car celles-ci sont actuellement la référence dans le monde du physical computing, mais nous pourrions aussi adapter notre système à des cartes proposés par d'autres constructeurs. Il existe plusieurs types de cartes reprenant le principe d'Arduino qui ont été développés, surfant sur leurs succès. Nous pouvons citer celles de la famille Intel comme le modèle Galilélo ou Edison qui fonctionnent avec des processeurs x86 et supportent le code Arduino. Intel propose un site pour promouvoir leur matériel en orientant leur communication sur l'Internet des objets. Le site fournit les informations nécessaires pour créer des projets sur leurs cartes avec l'IDE Arduino, Eclipse ou encore avec Wyliodrin pour la partie programmation visuelle [Intel IoT –

2015]. Enfin nous citons le constructeur BeagleBoard qui propose une carte nommée BeagleBoard Black fonctionnant sous linux et offrant un très grand nombre d'interfaces pour connecter des objets (cf *Annexe 2*). Ces cartes pourraient être testées dans notre système.

5.2.2.4 Communication entièrement bidirectionnelle

A l'heure actuelle, des travaux sont en cours pour la spécification d'une API pour HTML5 permettant d'utiliser le protocole WebSocket [The WebSocket API - 2015]. Socket.IO s'appuie sur ces travaux en apportant des fonctionnalités supplémentaires comme l'utilisation d'autres méthodes de communication en cas de non support du protocole WebSocket. Nous avons vu que cette communication bidirectionnelle fonctionnait très bien dans notre système et permettait de couvrir la réception d'évènements ; ce que ne peut pas couvrir le modèle REST avec HTTP. Par contre si les objets peuvent envoyer des évènements, l'utilisateur peut aussi en envoyer à destination des objets.

Il serait intéressant d'étudier la création d'un système fonctionnant entièrement de manière évènementielle avec une communication bidirectionnelle. L'utilisation de WebSocket est plus rapide que le modèle client-serveur [WebSocket Benchmark – 2015], cela permettrait de gagner en responsivité avec les objets intelligents. Cette technologie semble gagner du terrain ; de plus en plus d'applications web utilisent l'aspect temps réel du protocole WebSocket. Pour l'implémenter correctement dans notre système, il faut que les nœuds puissent eux aussi communiquer via un canal Websocket. Actuellement dans notre architecture ce n'est pas le cas pour des raisons techniques. En effet, nous avons testé l'utilisation d'envoi de données depuis l'Arduino avec WebSocket mais ses performances sont trop limitées.

Les échanges se feront entièrement au formalisme JSON. Pour le contrôle d'un objet, l'utilisateur enverra un message JSON contenant le nom du nœud, le nom de l'objet, le nom de la méthode et ses paramètres. Le serveur interprétera le message et s'occupera de la communication avec le nœud via WebSocket avec des messages JSON. Les souscriptions aux notifications pourront aussi se faire suivant ce modèle ou alors en gardant le fonctionnement avec l'API REST.

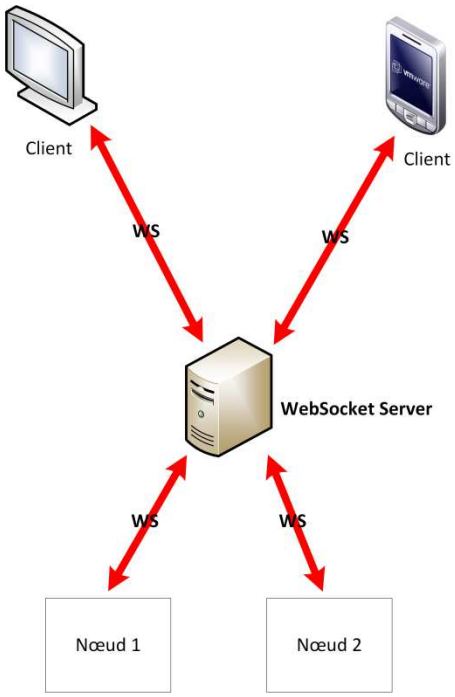


Figure 62 - Système communicant entièrement avec WebSocket

Avec cette architecture nous gagnerons surement en performance mais nous perdrons en simplicité d'utilisation par rapport au modèle REST qui s'intègre parfaitement au Web.

Conclusion

Les travaux réalisés sur ce proof of concept nous ont permis de faire ressortir deux principales techniques pour la création d'un système d'intelligence ambiante. La première consiste à élaborer une architecture REST pour considérer les objets communicants comme des ressources Web classiques que nous pouvons interroger avec le protocole HTTP. Cela apporte un haut niveau d'abstraction pour l'utilisateur qui n'a pas à se soucier du fonctionnement technique de l'objet pour interagir avec lui. Il doit juste utiliser une interface uniforme qui est la même quel que soit l'objet. A partir de cette interface il peut naviguer dans une arborescence de ressources d'objets intelligents et connaitre les différents services proposés pour communiquer avec eux. Cette communication réalisée avec le formaliste JSON permet d'interpréter les messages tels quels ou alors dans une multitude de langages informatiques.

La deuxième technique est celle du Web temps réel avec le protocole WebSocket. Celle-ci est utilisée en complément de l'API pour apporter la communication vers le client directement depuis les objets. Ainsi les objets peuvent envoyer des notifications ou évènements et enrichir notre système.

Nous avons montré avec la réalisation de notre maquette, qu'avec notre système, des développeurs pouvaient concevoir une application web utilisant des objets communicants suivant les deux techniques énoncées. Nous n'avons pas pu mettre en place une interface pour les utilisateurs non-développeurs mais nous avons disserté sur les outils de programmation visuelle existants que nous pourrions adapter pour notre système.

Une piste d'amélioration intéressante de notre système est d'envisager une évolution vers l'utilisation des objets intelligents de manière évènementielle avec la communication bidirectionnelle permise par WebSocket. Le modèle client-serveur d'HTTP convient bien pour des applications où les interactions sont initiées par l'utilisateur. Par contre il n'est pas prévu les applications impliquant des objets déclenchant des évènements à traiter en temps réel. Il faudrait envisager un fonctionnement entièrement évènementiel via WebSocket ou l'utilisateur contrôle les objets par des évènements et où les objets remontent des informations par évènement. Nous utiliserions toujours notre API pour la découverte des objets mis à disposition et pour administrer le système.

Bibliographie

Ouvrages imprimés

WALDNER J-B., 2007, *Nano-informatique et intelligence ambiante – inventer l'ordinateur du XXIème siècle*, Lavoisier, France, 300p

ROHIT R., 2013, *Socket.IO Real-time Web Application Development*, PACKT Publishing, Angleterre, 124p

HOLMES S., 2013, *Mongoose for Application Development*, PACKT Publishing, Angleterre, 124p

Travaux universitaires

FIELDING RT., 2000, *Architectural Styles and the Design of Network-based Software Architectures* [en ligne], Thèse de doctorat, University of California – Irvine
Disponible sur : <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (consulté le 13/10/14)

Articles

WEISER M., 1991, *The Computer for the 21st Century*, *Scientific American*

Sites web

Ubiquité, wikipedia.org, [en ligne].
Disponible sur : <http://fr.wikipedia.org/wiki/Ubiquit%C3%A9> (consulté le 15/10/14)

Ambient intelligence, In : wikipedia.org, [en ligne]
Disponible sur : <http://en.wikipedia.org/wiki/Ambient_intelligence> (consulté le 16/10/14)

Equipe Silex, liris.cnrs.fr, [en ligne]
Disponible sur : <https://liris.cnrs.fr/equipes?id=44> (consulté le 17/10/14)

Ethernet library, arduino.cc, [en ligne]
Disponible sur : <http://arduino.cc/en/Reference/Ethernet> (Consulté le 9/12/14)

Node.JS, nodejs.org, [en ligne]
Disponible sur : <http://nodejs.org/> (Consulté le 10/12/14)

Routing, expressjs.com, [en ligne]

Disponible sur : <http://expressjs.com/guide/routing.html> (26/01/15)

Webduino, github.com, [en ligne]
Disponible sur : <https://github.com/sirleech/Webduino> (consulté le 15/08/14)

TMP36 Data sheet, analog.com [en ligne]
Disponible sur : <http://www.analog.com/media/en/technical-documentation/data-sheets/TMP35_36_37.pdf?> (Consulté le 17/12/14)

Introduction JSON, json.org [en ligne]
Disponible sur : <http://json.org/> (Consulté le 9/01/15)

jQuery.ajax(), jquery.com [en ligne]
Disponible sur : <http://api.jquery.com/jQuery.ajax/> (Consulté le 13/01/15)

cURL, curl.haxx.se [en ligne]
Disponible sur : <http://curl.haxx.se/> (Consulté le 13/01/15)°

A propos de Scratch, scratch.mit.edu [en ligne]
Disponible sur : <http://scratch.mit.edu/about/> (Consulté le 14/01/15)

Wyliodrin, wyliodrin.com [en ligne]
Disponible sur : <https://www.wyliodrin.com/> (Consulté le 14/01/15)

Intel IoT, intel.com [en ligne]
Disponible sur : https://software.intel.com/en-us/iot (Consulté le 14/01/15)

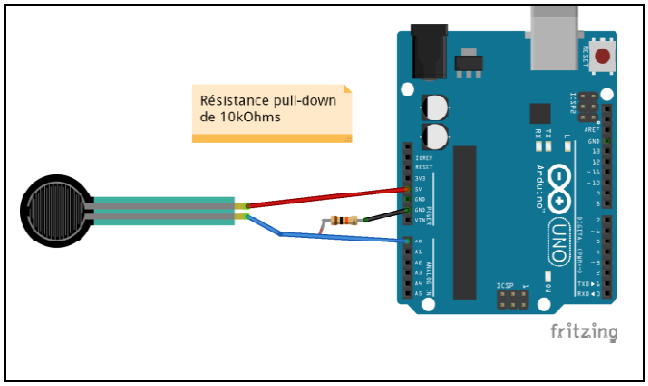
The Web Socket API, w3c.org [en ligne]
Disponible sur : <http://dev.w3.org/html5/websockets/> (Consulté le 14/01/15)

REST vs WebSocket Comparaison Benchmarks, blog.arungupta.me [en ligne]
Disponible sur :<http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/> (Consulté le 16/01/15)

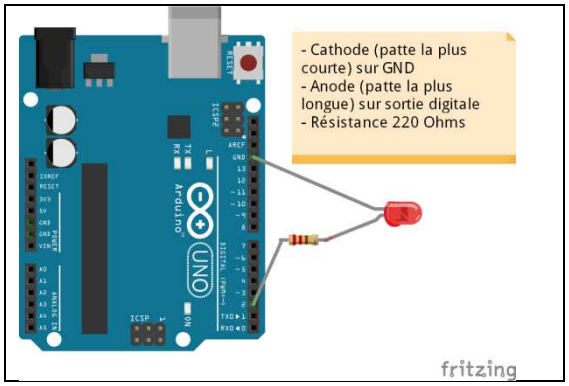
Table des annexes

Annexes

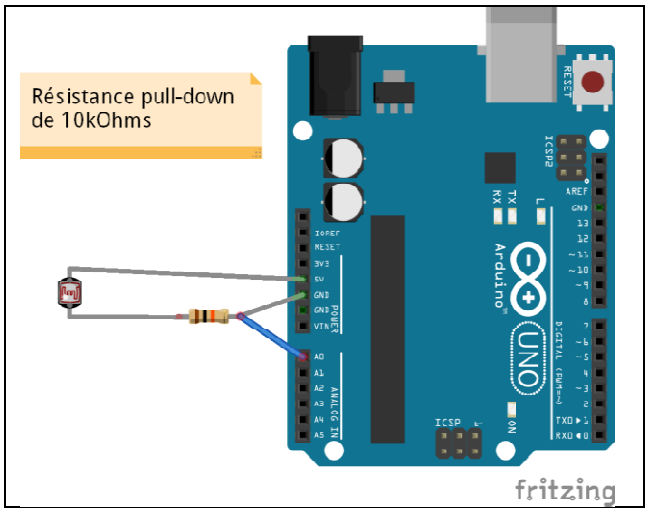
Annexe 1 Schémas Fritzing issus de la documentation



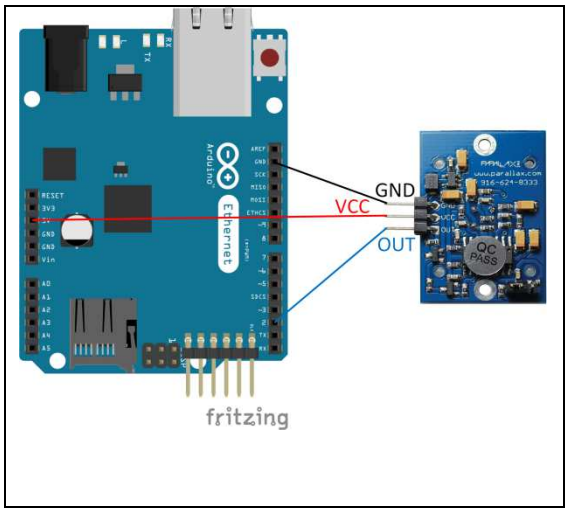
Capteur de pression



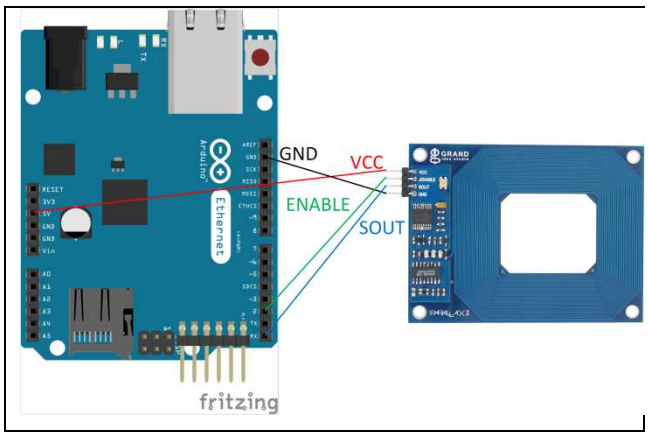
LED



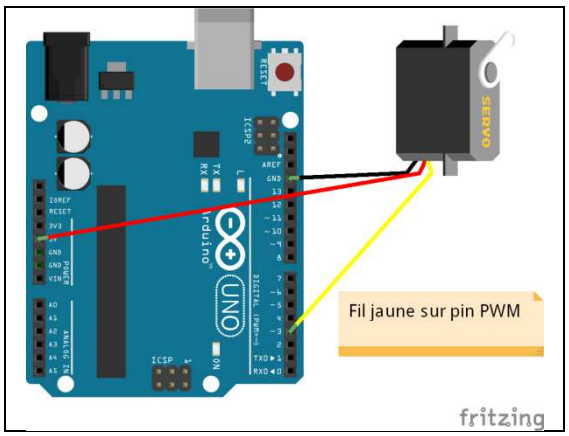
Capteur de luminosité (photoressistance)



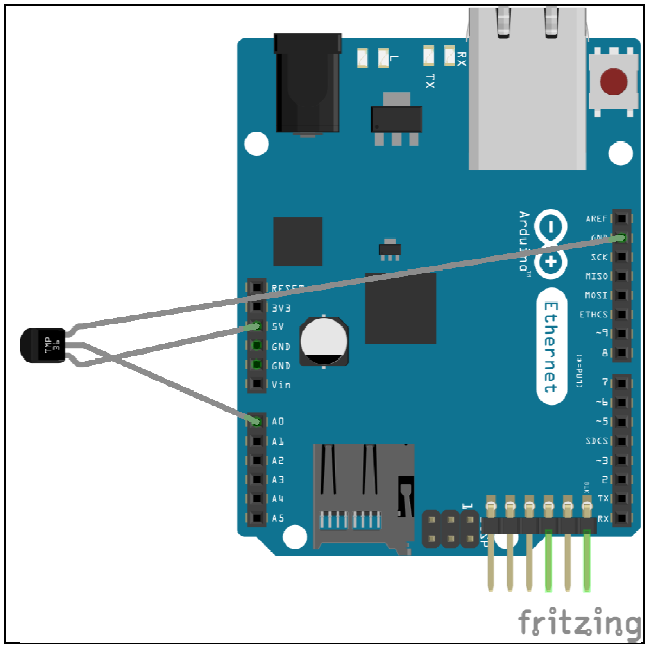
Détecteur de mouvement (Parallax PIR Rev B)



Lecteur RFID (Parallax 28140)



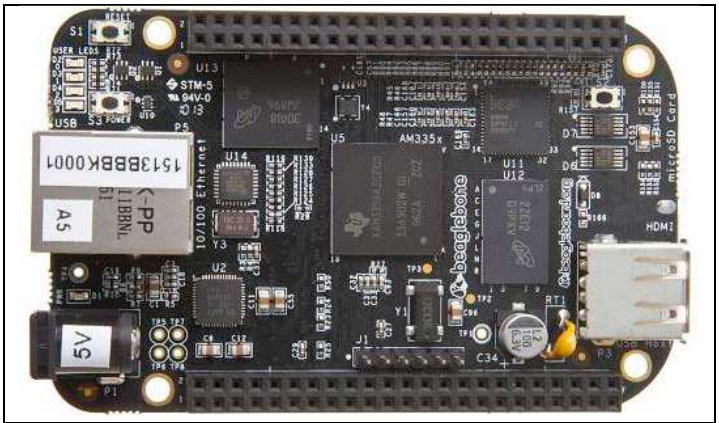
Servomoteur



Capteur de température (TMP36)

Annexe 2 **Alternatives à Arduino**

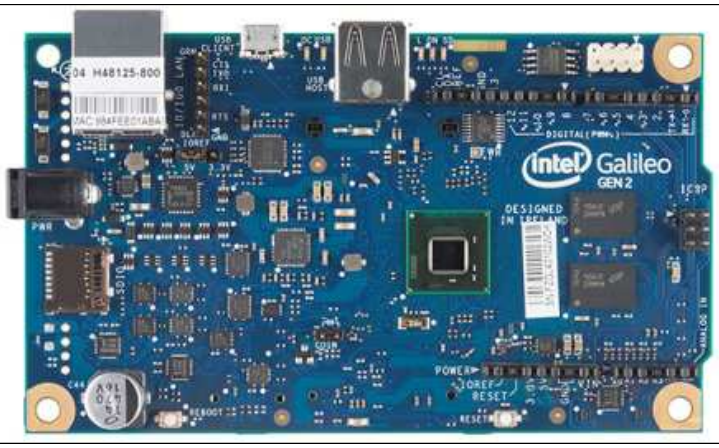
BeagleBone Black



Caractéristiques	
Processeur	AM335x 1GHz ARM Cortex-A8
Mémoire	512 MB DDR3 RAM
Analog Input	7 (12 bits)
Digital I/O	64
PWM	8
I²C	2
SPI	2
Serial UART	4

La carte BeagleBone Black fonctionne avec plusieurs distributions Linux (Ex :Debian, Ubuntu, Gentoo, Fedora). Il est possible de programmer cette carte avec JavaScript, Python, C et Ruby. La carte possède un controleur Ethernet.

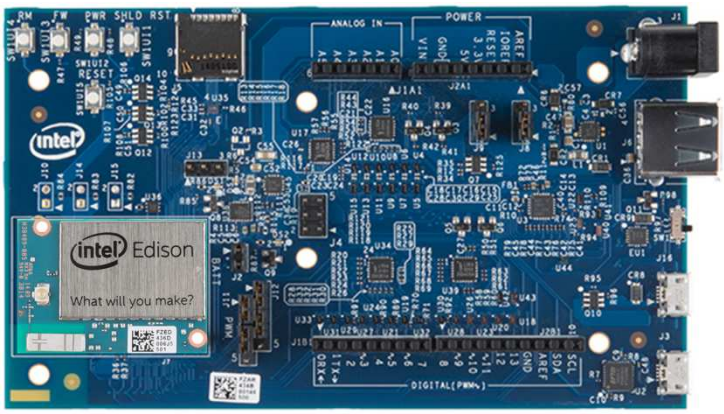
Intel Galileo Rev 2



Caractéristiques	
Processeur	Intel® Quark SoC X1000 400Mhz
Mémoire	512 KB SRAM + 256 MB DRAM
Analog Input	6 (12 bits)
Digital I/O	14
PWM	6
I²C	1
SPI	2
Serial UART	1

La carte Galileo est construite pour intégrer le marché du physical computing par le géant Intel. Elle possède un SoC (System on Chip) avec un processeur x86. Elle a été conçu en partenariat avec Arduino et propose quasiment le même nombre d'interfaces qu'un Arduino Uno. De plus elle est compatible avec les shields Arduino. Elle fonctionne avec une distribution Linux et ses interfaces sont programmables avec le langage Arduino mais aussi avec Python ou JavaScript. Elle possède un contrôleur Ethernet mais aussi un port mini PCI-E au-dessous de la carte acceptant une carte Wifi.

Intel Edison



Caractéristiques	
Processeur	Intel Atom Silvermont DualCore 500Mhz Microcontrôleur Intel Quark 100Mhz
Mémoire	512 KB SRAM + 256 MB DRAM
Analog Input	6 (12 bits)
Digital I/O	20
PWM	4
I²C	1
SPI	1
Serial UART	1

L'Edison est la carte la plus récente de ce que propose Intel pour le physical computing. Elle est très petite et a été conçu pour s'intégrer sur plusieurs cartes de prototypage. Sur la 2ème photo elle est intégrée sur une carte de prototypage Arduino et ses caractérises sont issus de cette association, mais l'Intel Edison peut accepter 40 entrées/sorties avec des fonctions multiples. Cette carte possède deux unités de traitements : un microcontrôleur dont le but est de collecter et prétraiter les données avant de les envoyer au processeur pour effectuer d'autres traitements. Elle fonctionne avec un système d'opération Linux et permet un développement avec Python, C, C++, JavaScript et Arduino. Elle comporte un contrôleur Wifi et Bluetooth indépendamment de la carte avec laquelle est associée.

Liste des figures

Liste des tableaux

Conception d'un système d'intelligence ambiante à l'aide de cartes Arduino

Mémoire d'Ingénieur C.N.A.M., Lyon 2015

RESUME

Le système d'intelligence ambiante est un proof of concept dont le but est de proposer des objets intelligents, simples d'utilisation, afin de créer des environnements augmentés. Les objets intelligents sont des capteurs et des actionneurs rendus communicants par le système.

Le système s'articule autour de cartes Arduino fonctionnant comme des nœuds d'objets intelligents. Ces nœuds sont reliés en réseau à un serveur centralisé. Le serveur propose une API REST pour piloter les objets intelligents depuis les nœuds, et offre la possibilité aux développeurs d'intégrer leur utilisation dans leurs projets.

Le système apporte aussi une communication en temps réel via le protocole WebSocket et permet aux objets intelligents d'envoyer des évènements à l'utilisateur.

Mots clés : Intelligence Ambiante, Web des objets, objets intelligents, API REST, Arduino, capteurs, actionneurs, WebSocket

---

SUMMARY

The ambient intelligence system is a proof of concept which goal is to provide smart things for building augmented environments. Those smart things are sensors and actuators interacting with each other within the system.

The system is built around Arduino cards, used as smart things nodes. Each node is connected to a centralized server over the network. The server uses a REST API which enables smart things to be controlled via the nodes themselves, allowing developers to use them for their projects.

Real time communication is assured by the Websocket protocol in order for the smart things to send events back to the end-user.

Key words : Ambiant Intelligent, Web of Things, smart things, API REST, Arduino, sensors, actuators, WebSocket