Rapport de projet EI5 AGI

Projet Domotique
Serveur REST NodeJS

Projet réalisé par :

Baptiste Gauduchon
Yann Jajkiewicz

Projet encadré par :

Serge Tahé
Sébastien Lagrange

Remerciements

Nous tenons à remercier dans un premier temps M. Serge TAHE et M. Sebastien LAGRANGE pour leur soutien tout au long du projet ainsi que pour l'aide qu'il ont pu nous apporter. Nous les remercions aussi pour nous avoir permis de découvrir les technologies et méthodes sous-jacentes au sujet traité.

Nous tenons à remercier dans un second temps l'ensemble du corps enseignant de l'ISTIA, et en particulier M.Hassan BOULJROUFI sans l'aide de qui ce projet n'aurait pu voir le jour.

Enfin, nous tenons à remercier l'Université d'Angers et en particulier l'ISTIA, sans le financement desquels ce projet serait resté couché sur le papier.

Table des matières

Introduction

C'est dans le cadre de la 3ème année du cycle ingénieur de l'ISTIA que nous avons choisi comme sujet de projet "Projet domotique", et plus particulièrement le développement d'un serveur REST NodeJS. Notre intérêt prononcé pour le mélange d'informatique embarquée, de réseau, de technologies web et mobiles de ce sujet nous a permis d'approfondir nos connaissances dans certaines technologies et de mettre en place des solutions correspondantes aux besoins demandés.

I. Le projet

L'objectif global du projet est de réaliser une preuve de concept pour une application de domotique permettant de contrôler et superviser des objets du quotidien au sein d'une habitation (lampe, chauffage, porte de garage, etc.) avec un ordinateur, un smartphone ou une tablette.

L'interêt principal est de mettre en place les fonctionnalités de communication entre les objets connectés et les utilisateurs. Cette démarche s'inscrit selon plusieurs sous-projets permettant pour chacun d'eux la réalisation d'une fonctionnalité grâce à une technologie, afin de pouvoir finalement les tester et les confronter.

Notre travail initial était de mettre en place la partie serveur grâce à la technologie Node.JS[1]. Nous verrons par la suite que nous avons pu aller plus loin avec la mise en place de deux clients, l'un natif sur iOS[2] et l'autre web grâce à jQuery Mobile[3].

1 - Présentation du sujet

Pour la suite de ce rapport, nous nous intéresserons uniquement aux parties que nous avons traité dans le cadre de ce projet, à savoir le serveur REST et les clients (iOS et jQuery mobile). L'ensemble de notre travail est basé sur le **TP "Mobilité et Domotique"[4]** rédigé par M. Serge TAHE. Le schéma d'application représentatif de notre version du projet est le suivant:

[1] **Node.JS** : voir chapitre concernant le serveur REST
[2] **iOS** : voir chapitre concernant le client natif iOS
[3] **jQuery Mobile** : voir chapitre concernant le client web
[4] Document disponible à cette adresse : http://tahe.developpez.com/android/arduino/

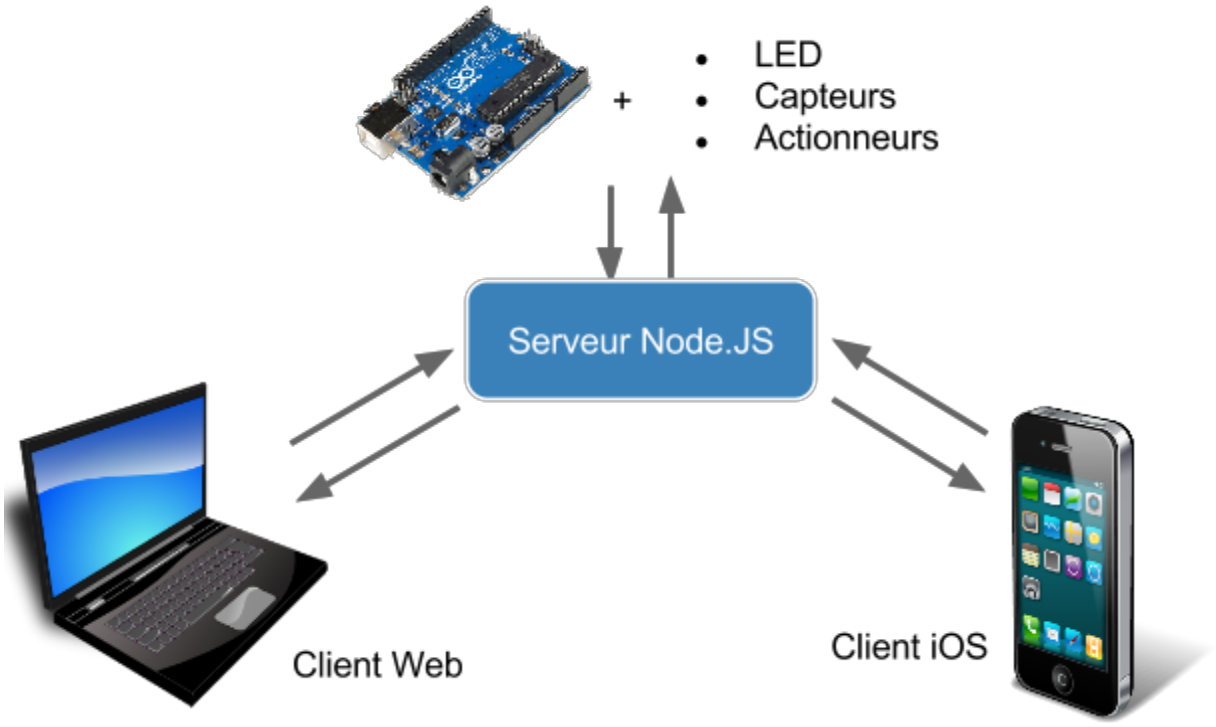


Figure 1 : schéma d'application du projet

Au centre se situe le serveur REST[5], dévelopé avec le langage Javascript grâce à Node.JS. Il permet de faire le lien entre les cartes embarquées et les applications de supervision. Un client quelconque peut alors utiliser l'API[6] fournie par le serveur REST afin de récupérer ou envoyer des données à une carte Arduino[7] connectée au serveur.

Les langages de programmations utilisés sont les suivants:

- carte arduino: C
- serveur: Javascript avec le framework NodeJS
- clients:
  - web : HTML, CSS, JQuery Mobile
  - iOS : Objective-C

Les ressources matérielles utilisées sont les suivantes:

- PC serveur
- Macbook Pro (développement iOS)
- Cartes Arduinos
- Câble communication USB
- Câble Ethernet
- Capteur de température
- LED
- Switch ethernet
- Clients (PC, Mobile, Tablette)

[5] **REST** : *REpresentational State Transfer* (architecture de service web)
[6] **API** : *Application Programming Interface* (Interface de programmation)
[7] **Arduino** : voir chapitre du même nom

Les deux ressources humaines utilisées pour la répartition des tâches :

- Yann Jajkiewicz
- Baptiste Gauduchon

2 - Méthodologie

Afin d'aborder ce sujet de la meilleure manière possible, et pour mettre à profit l'expérience acquise lors du projet réalisé l'année précédente, nous avons suivi la méthodologie suivante :

1. Prise en main de notre partie du projet : prise de connaissance des objectifs souhaités ainsi que des contraintes fonctionnelles et de temps ;

2. Mise en place de notre gestion de projet pour cadrer et suivre notre travail sur la durée du projet :
    a. Création d'un cahier des charges dès la première séance regroupant les fonctionnalités souhaitées ainsi que les contraintes du projet ;
    b. Mise en place et maintien d'une veille technologique dès la première séance et sur toute la durée du projet. Celle-ci permet de faire un état des lieux initial des technologies disponibles, de noter toutes nos éventuelles découvertes relatives à ces technologies ainsi que toutes les ressources que nous avons pu utiliser pour avancer notre travail ;
    c. Mise en place et maintien d'un suivi de séances pour garder une trace des séances passées, pour planifier les séances futures et pour gérer notre temps de travail ;

3. Mise en place d'une division des tâches selon le cahier des charges pour pouvoir travailler en parallèle en maximisant nos ressources humaines.

3 - Cahier des charges du projet

3.1 - Identification du besoin

Nous avons utilisé la représentation sous forme de bêtes à cornes afin d'identifier clairement les besoins du projet :

- Point du vue des clients :

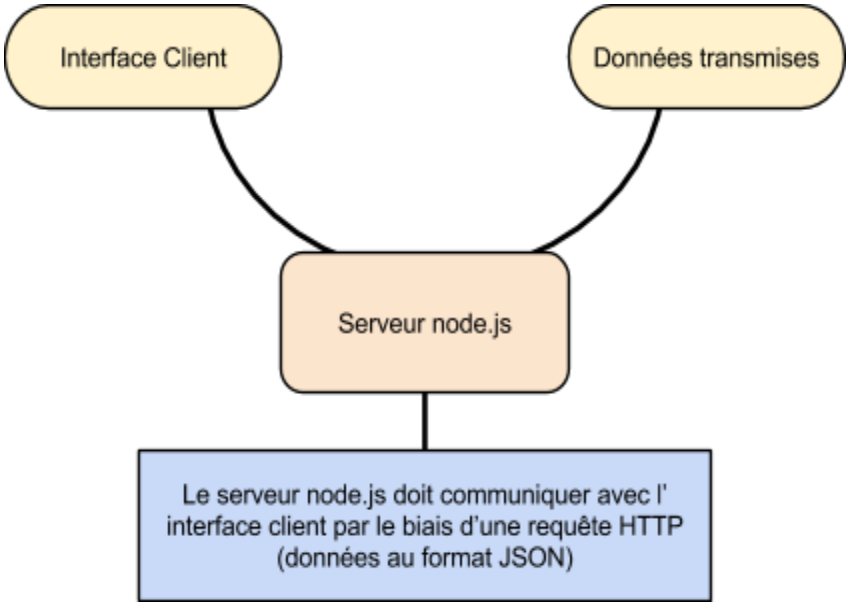


Figure 2 : besoin des clients

- Point de vue des arduinos :

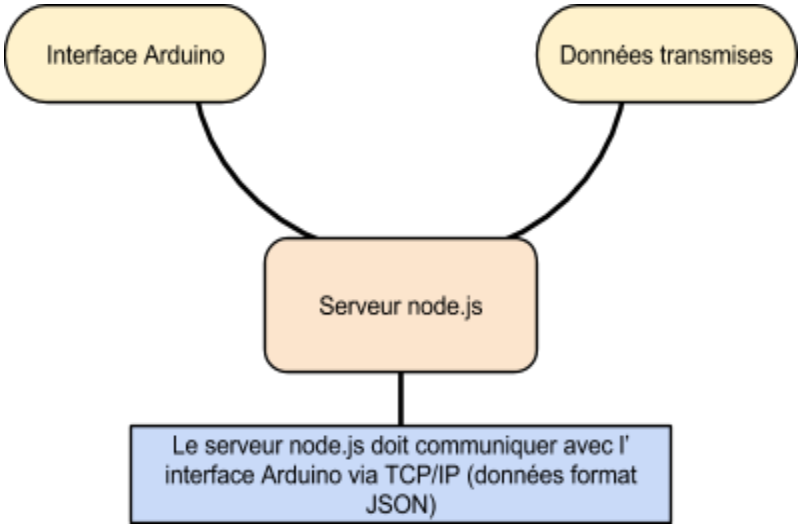


Figure 3 : besoin des arduinos

3.2 - Identification des fonctionnalités

La représentation sous forme de diagramme pieuvre nous a permis de définir les fonctions principales ainsi que les contraintes relatives au projet :

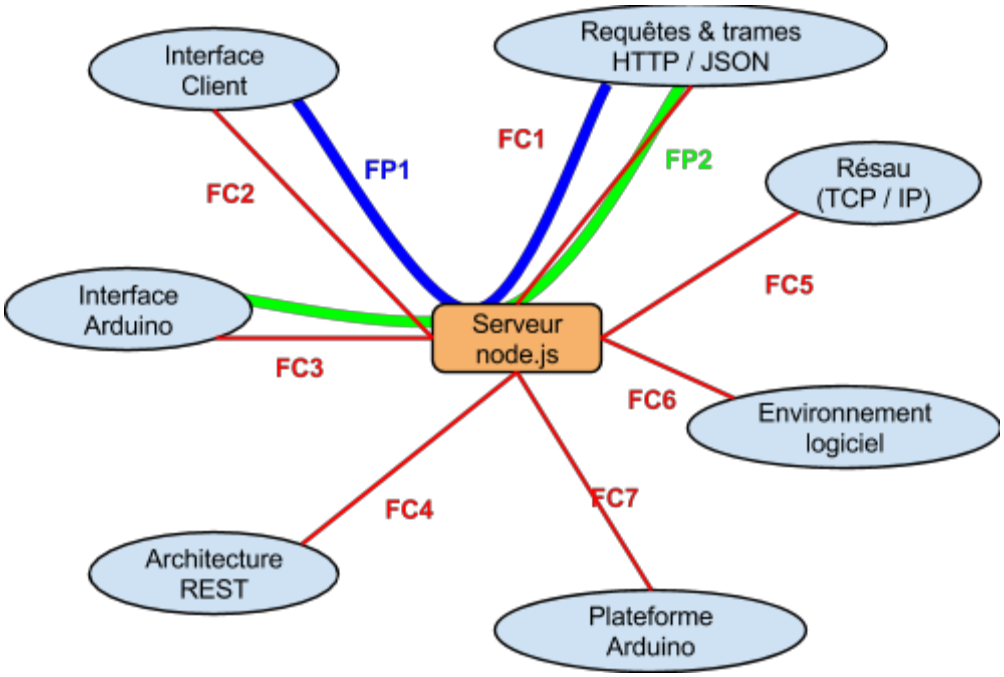


Figure 4 : fonctions principales et contraintes

Classification	Description
FP1	Répondre aux requêtes HTTP[8] de l'interface client par des trames JSON[9]
FP2	Récupérer les information de la carte Arduino grâce à des trames TCP/IP[10] au format JSON
FC1	Respecter l'interface imposée pour la communication avec le client
FC2	Respecter l'interface imposée pour la communication avec le module Arduino
FC3	Respecter le format des requêtes HTTP et des trames JSON fournis
FC4	Respecter l'architecture REST
FC5	Être compatible avec le réseau TCP/IP
FC6	S'intégrer dans un environnement logiciel de type serveur
FC7	S'intégrer sur le module de prototypage Arduino

[8] **HTTP** : protocole de communication utilisé pour le web
[9] **JSON** : format de données spécifique
[10] **TCP/IP** : protocole de communication réseau

3.3 - Cahier des charges : interfaces de programmation

Afin de garantir une interopérabilité entre notre serveur REST et les autres client ainsi qu'entre nos clients et les autres serveur REST, nous avons défini les interfaces de communication entre les différentes parties de ce projet.

3.3.1 - Interface client du serveur

Le serveur doit respecter une API commune à tous les serveurs :

1. Les point d'entrée sont les suivant :

   - URL de base fixe : "**`http://ip:port/rest`**"
   - Pour obtenir la liste des arduinos (méthode GET) : "**`/arduinos/`**"
   - pour effectuer une lecture sur un pin (méthode GET) :
     "**`/arduinos/pinRead/{idCommande}/{idArduino}/{pin}/{mode}`**"
   - pour effectuer une écriture sur une pin (méthode GET) :
     "**`/arduinos/pinWrite/{idCommande}/{idArduino}/{pin}/{mode}/{valeur}`**"
   - pour effectuer un clignotement (méthode GET) :
     "**`/arduinos/blink/{idCommande}/{idArduino}/{pin}/{duree}/{nombre}`**"
   - pour envoyer une commande (méthode POST) :
     "**`/arduinos/commands/{idArduino}`**",
     contenu de la commande au format "application/json"

   Remarques :
   - **`{idArduino}`** = identifiant de l'arduino, contenu dans la réponse de la liste des arduinos ;
   - **`{pin}`** = numéro de pin sur l'arduino : numérique [1-13] ou analogique [0-5] (attention, le pin 0 n'est pas disponible en lecture pour le mode numérique) ;
   - **`{mode}`** = analogique ("a") ou numérique("b"), pour choisir entre les entrées / sortie numérique ou analogique ;
   - **`{valeur}`** = valeur retournée lors de la lecture d'une pin par l'arduino (numérique [1/0], analogique [0-255]).

2. Les réponses sont les suivantes :

   - Réponse lors d'une erreur d'une commande

```
{"data":{"id":"X", "erreur":"Y", "etat":"", "json":""}}
```

   - Réponse lors d'une erreurs de fonctionnement (serveur REST)

```
{"data":{"message":"message comportant les indications des erreurs"}}
```

   - Réponse lors d'une demande des arduinos connectées

```
{"data":[{JSON infos arduino 1}, {JSON infos arduino 2}, ..., {JSON infos
                                arduino X}]}
```

avec par exemple : “JSON Info arduino X” = {"id":"Nom de l’arduino", ”ip”:”255.255.255.255", desc":"Uno_Projet5", "mac":"90:A2:DA:00:1D:A7", "port":102}

- Réponse pour le bon fonctionnement d’une commande

```
{“data”:{“id”:”X”, “erreur”:””, “etat”:””, “json”:””}}
```

- Réponse pour la lecture d’une pin

```
{“data”:{“id”:”X”, “erreur”:””, “etat”:””, “json”:{"pinX":"XXX"}}}
```

3.3.2 - Interface arduino du serveur

Le serveur communique avec les arduinos en respectant les conventions suivantes :

1. Trames JSON attendues par l’arduino et qui doivent être sérialisées :
   - lecture d’une pin

```
{"id":"1", "ac":"pr", "pa":{"pin":"8", "mod":"a"}}
```

   - écriture sur une pin

```
{"id":"2", "ac":"pw", "pa":{"val":"0", "pin":"8", "mod":"b"}}
```

   - clignotement

```
{"id":"3", "ac":"cl", "pa":{"pin":"8", "dur":"100", "nb":"10"}}
```

Remarques :

- **id** = identification de la commande (peut être un chiffre ou un texte, il faudra penser à verifier la taille au niveau de l’arduino pour des soucis d’optimisation mémoire)
- **ac** = action (pr, pw, cl)
- **pa** = parameters (valeur, pin, mode, + durée , nombre pour le clignotement)

L’arduino envoie ses informations (id, ip, mac, port, desc) lors de sa première connection. Il faudra cependant gérer au niveau du serveur REST la mise à jour de la liste des arduinos réellement connectés (en cas de déconnection intempestive d’un arduino) de ceux-ci et de savoir quel arduino est connecté à l’instant t.

2. Trames JSON retournées par l’arduino
   - réponse sur erreur

```
{"id":"X","er":"XXX","et":{}}
```

   - réponse sur bon fonctionnement

```
{"id":"X","er":"0","et":{}}
```

- réponse pour la lecture d'une pin

```
{"id":"X","er":"0","et":{"pinX":"XXX"}}
```

Remarques :

- **id** = écrit en dur dans le code de l'arduino (idéalement un texte court) ;
- **er** = un code spécifique à l'Arduino, ou 0 si pas d'erreurs ;
- **et** = valeur de la pin lors d'une lecture (pin X = valeur)

4 - Gestion de projet

Comme le projet concerne le serveur REST d'une part et les plateformes arduinos d'autre part, le projet fut naturellement découpé ainsi :

- Une partie développement serveur avec NodeJS (décomposé en sous parties correspondant aux couches web, métier, et dao) ;
- Une partie développement carte arduino.

Les temps de développement furent plus court que prévus, ce qui nous a permis de réaliser deux clients :

- Un client JQuery Mobile ;
- Un client iOS ciblé iPhone.

4.1 - Répartition du temps de travail

Voici notre diagramme de répartition globale du temps de travail sur toute la durée du projet :

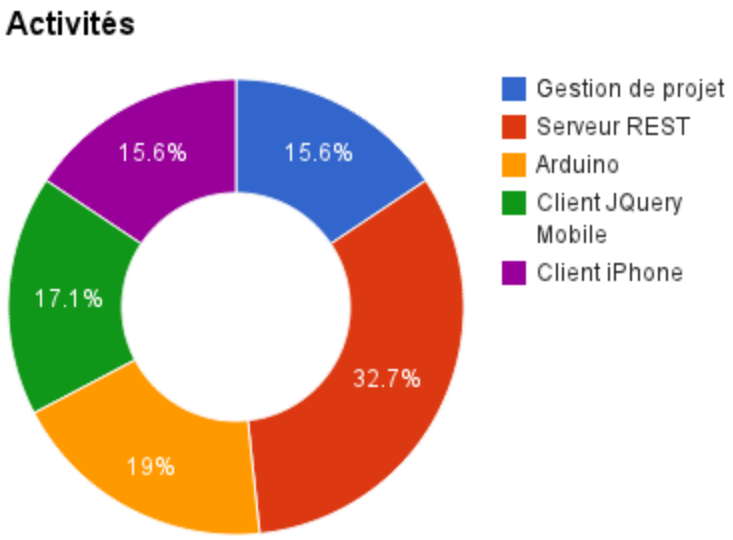


Figure 5 : activités du projet

Sur ce diagramme circulaire sont représentés en pourcentage les temps de réalisation réels des différentes activités du projet. On remarque que la réalisation du serveur REST est prédominante, suivie par l'Arduino et la gestion de projet. Ceci s'explique par la priorité importante de ces réalisations afin de répondre aux besoins du sujet, mettant en second plan la réalisation des clients.

4.2 - Répartition des tâches

Ormis pour la gestion de projet, les activités présentées précédemment peuvent être décomposées en trois types de tâche:

- développement : la création et réalisation de code nouveau implémentant de nouvelles fonctionnalités
- refactoring : la modification de code ou algorithme existant dans le but d'améliorer une fonctionnalité
- divers: rédaction de spécifications, veille technologique, tests, débuggage, validation, etc.

Les tâches ont été repartie de la manière suivante :

Yann	Baptiste
➔ Plateforme Arduino ➔ Client natif iOS	➔ Serveur REST node.JS ➔ Client web jQuery Mobile

Il important de noter que les tâches ne sont pas cloisonées, nous avons souvent mis à profit notre travail en équipe pour avancer lorsqu'un problème survient.

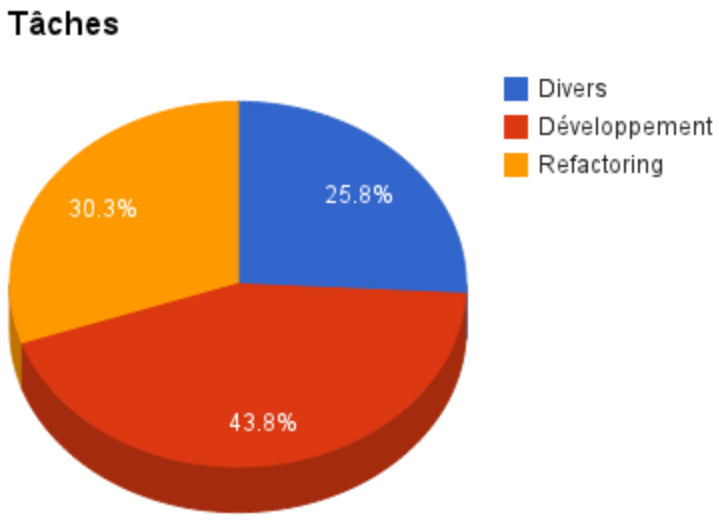


Figure 6 : proportion des tâches (toutes activités confondues)

Détail des tâches par activité:

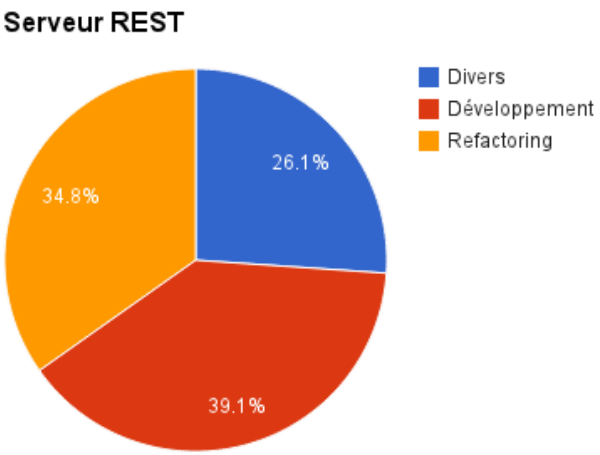


Figure 7 : détail tâches serveur REST

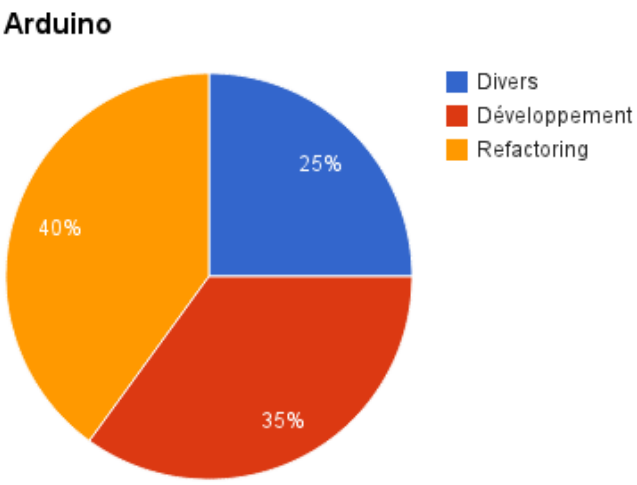


Figure 8 : détail tâches arduino

On remarque que pour les activités principales, qui sont la réalisation du serveur REST et de la carte Arduino, les temps de développement et de refactoring sont proches. Ceci s'explique par le fait qu'une fois que l'on obtenait un produit proche du fonctionnel souhaité, il devait constamment être testé et débuggé afin de pouvoir être amélioré, dans le but d'obtenir des codes maintenables et dont l'exécution correspondaient exactement aux comportements demandés.

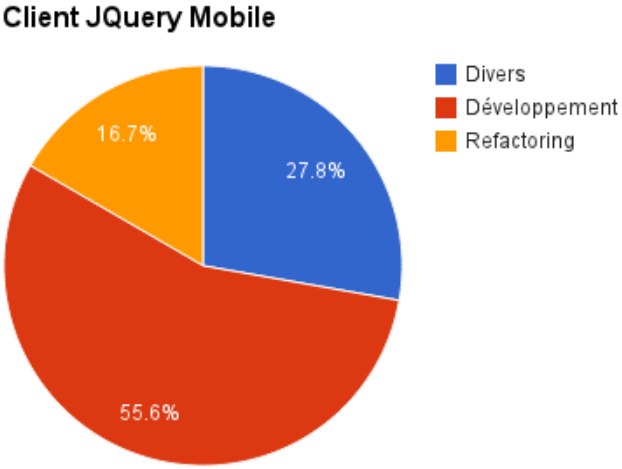


Figure 9 : détail tâches client JQuery Mobile

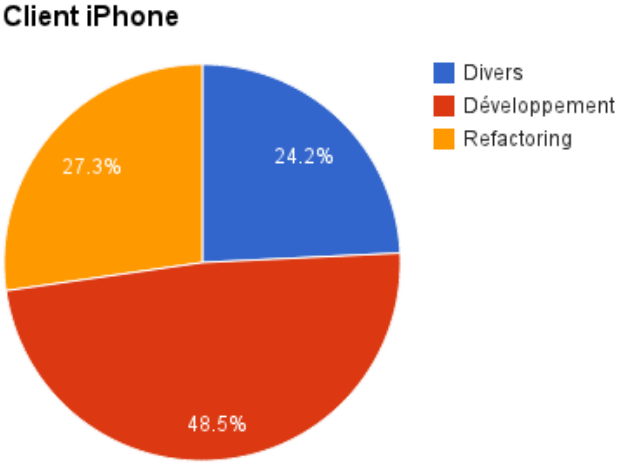


Figure 10 : détail tâches client iPhone

Pour la réalisation des clients, qui sont des activités annexes, on remarque que les temps développement sont prédominants et que les tâches de refactoring le sont beaucoup moins. Ceci s'explique par des cahiers des charges initiaux de ces activités moins exigents et par l'utilisation de technologies moins contraignantes.

II. Arduino

Une carte arduino est un circuit imprimé qui contient un microcontrôleur. Une fois programmé, ce dernier permet d'effectuer des opérations de lecture et d'écriture de signaux électriques permettant de mettre en place par exemple une communication réseau ou d'effectuer des tâches diverses de domotique. Concrètement pour ce projet, le programme pour notre carte peut par exemple lire la valeur d'un capteur de température ou encore allumer, éteindre ou faire clignoter une LED.

1 - Avantages et inconvénients

Les plateformes de prototypage arduino comme cartes embarquées pour ce projet étaient un choix imposé du cahier des charge. Les **avantages** d'utiliser ce type de carte sont les suivants :

- **langage de développement** : les programmes pour plateformes arduino doivent être écrits en C, qui est un langage bas niveau simple et sans surprises ;
- **accent sur le développement** : le développeur n'a pas à s'occuper du fonctionnement interne du microcontrôleur et des composants contenus dans la carte. Ainsi il peut se concentrer sur l'objectif à atteindre et mettre en place le programme qu'il souhaite en n'utilisant que les fonctions proposées par arduino pour contrôler ces composants ;
- **communauté importante** : les plateformes arduino sont open source et attirent un grand nombre de développeurs, de ce fait la communauté d'utilisateurs est importante. Cela est un avantage car cette communauté permet de trouver facilement des ouvrages, aides et solutions à certains problèmes sur internet. De plus, grâce à cette communauté, de nombreuses librairies sont disponibles, permettant ainsi d'atteindre divers objectifs plus facilement ;
- **IDE cross-platform** : le logiciel d'édition et de programmation des cartes est réalisé en Java, permettant d'être installé et utilisé sous Windows, Mac et Linux.

Ces avantages permettent aux développeurs une rapidité de programmation et un déploiement simple puisque le côté matériel des cartes n'a pas à être compris en profondeur ni même programmé.

Néanmoins les plateformes arduino possèdent également quelques inconvénients qui ne facilitent pas la tâche du développeur :

- **Comportements aléatoires** : les cartes arduino possèdent une mémoire de type RAM[11] faible (comme la plupart des cartes embarquées, surtout low-cost). Le problème est que lorsque la mémoire libre devient faible, les comportements de la carte deviennent aléatoire, rendant difficile de trouver la source du problème ;

- **Débuggage difficile** : il n'est pas évident de débugger un programme arduino car l'IDE[12] fournit ne permet pas d'exécution "pas par pas" ou de scrutage du contenus des

[11] **RAM** : Random Access Memory, en français "mémoire vive"
[12] **IDE** : Integrated Development Environment (logiciel de développement)

variables par exemple. La solution possible est donc d'utiliser le port série et d'afficher sur le PC de développement des affichages de log lorsque que certains points du programme sont exécutés.

2 - Spécifications

Notre carte arduino doit remplir les fonctions principales suivantes :

- client afin de s'enregistrer auprès du serveur REST
- serveur afin de recevoir des trames JSON de la part d'un client (ici notre serveur REST)
- ne jamais planter

L'arduino peut effectuer 3 commandes :

- pinRead: lecture analogique ou binaire d'une broche de la carte
- pinWrite: écriture analogique ou binaire d'une broche de la carte
- doClignoter: clignotement (écriture répétée sur une broche de la carte)

Lorsque l'arduino démarre, il se place en tant que client ethernet et essaye d'établir une connexion avec le serveur REST. En cas d'échec l'arduino continue d'établir la connexion, s'il y arrive il envoit la chaîne JSON d'enregistrement au serveur, considère que l'enregistrement a été effectué et se place alors en tant que serveur.

En tant que serveur, l'arduino écoute en continu et attend qu'un client se connecte. Lorsque c'est le cas, le programme reçoit une commande de la part du serveur REST sous la forme d'une chaîne JSON.

Une fois la chaîne JSON reçue, elle est traitée:

- la présence des paramètres nécessaires est vérifiée
- si une action reconnue est détectée on appelle la fonction de commande adéquate (pinRead, pinWrite ou doClignoter)
- la fonction de commande vérifie la validité des paramètres et effectue l'action nécessaire
- une réponse est envoyée au client (ici notre serveur REST) avant de clôturer sa connexion

En cas de paramètres non présents ou non valides, l'arduino arrête le traitement de la commande et envoie au client une réponse d'erreur au format JSON dont les codes d'erreur sont les suivants:

Code	Fonction	Description
100	traiterCommande	json == null
101	traiterCommande	id == null
102	traiterCommande	action == null
103	traiterCommande	parametres == null

104	traiterCommande	action non reconnue
201	doClignoter	pin == null
202	doClignoter	pin not [1,13]
203	doClignoter	duree == null
204	doClignoter	duree not [100,2000]
205	doClignoter	nb == null
206	doClignoter	nb < 2
301	pinWrite	pin == null
302	pinWrite	val == null
303	pinWrite	mod == null
304	pinWrite	mod différent de “a” ou “b”
305	pinWrite	mod == a && (pin not [0,5]) or mod == b && (pin not [1,13])
306	pinWrite	mod == b && (val not [0,1]) or mod == a && (val not [0,255])
401	pinRead	pin == null
402	pinRead	mod == null
403	pinRead	mod différent de “a” ou “b”
404	pinRead	mod == a && (pin not [0,5]) or mod == b && (pin not [1,13])

3 - Difficultés rencontrées

Le squelette du programme Arduino était donné dans le document “TP Mobilité et Domotique 2013” de Serge Tahé, le travail à réaliser était le suivant:

- mémoriser la commande reçue par un client connecté
- parser la commande reçue en objets JSON
- mémoriser les différents paramètres reçus selon le type de variable (ex: transformer l’objet JSON “nb” en entier int)
- vérifier la validité des paramètres (qui induit la création de nouveaux codes erreur présentés précédemment)
- effectuer les actions souhaitées en fonction des paramètres

Le programme permettant d'obtenir un arduino fonctionnel fut réalisé sans grandes difficultés, néanmoins le programme comportait des "failles" pouvant causer le plantage de la carte arduino et qui donc ne respectait pas les spécifications demandées.

3.1 - Fuites mémoires

Le problème que le programme rencontrait fréquemment était qu'au moment de répondre au client connecté, l'arduino ne possédait plus assez de mémoire pour construire et envoyer la réponse. Le cas se présentait souvent lorsque l'attribut "idCommand", souvent un nombre entier chez les clients, était une chaîne de caractères assez longue comme une adresse IP par exemple.

La solution première était donc de tronquer cette "idCommand" reçu pour éviter d'alourdir la mémoire arduino consommée, mais cette solution peut poser problème dans le cas où le client souhaite utiliser un "idCommand" précis pour une quelconque raison et ne souhaite donc pas qu'il soit tronqué au sein de l'arduino et donc au sein de la réponse. Cette solution fut testée mais abandonnée.

Pour éviter que la mémoire de l'arduino soit trop utilisée inutilement et que ce problème survienne, le programme a été modifié afin d'utiliser autant que possible des pointeurs. En effet, le programme est découpé en fonctions afin de le structurer, or certaines fonctions possèdent comme paramètres des chaînes de caractères de type String. De ce fait, lorsque ces fonctions sont utilisées les variables passées en paramètres sont recopiées en mémoire inutilement. Ainsi en utilisant des pointeurs sur ces variables on évite la recopie en mémoire de leur valeur, évitant une surconsommation inutile de mémoire.

3.2 - Longueur de la commande

L'arduino ne doit pas planter, or s'il se trouve que la mémoire libre devient faible, son comportement deviendra alors imprévisible et donc non fonctionnel.

Dans le cas où un client se connecte à l'arduino et envoie une commande contenant un grand nombre de caractères, notre carte ayant une mémoire libre très limitée plantera et devra être réinitialisé pour être à nouveau fonctionnel. Pour éviter ce soucis, lors de la réception de la commande qui se fait caractère par caractère, le programme va limiter la taille de la commande reçue à 100 caractères, évitant un abus d'utilisation de la mémoire de l'arduino.

3.3 - Paramètres numérique

Lors du traitement de la commande reçue, l'arduino doit vérifier la validité de chaque paramètre. Un code erreur pour chacun de ces paramètres a été créé et présenté dans la partie précédente dans le cas où ils ne sont pas conformes à nos attentes.

Lorsqu'un paramètre est reçu, il est au format JSON avant d'être mémorisé dans le type attendu (String, int, char, etc.). La librairie JSON utilisée avec le programme arduino permet de récupérer les attributs de la trame sous forme de chaîne de caractères String avant d'être mémorisé dans le type souhaité.

Or, une chaîne de caractères ou un caractère simple possède une valeur en tant que nombre entier. Si l'on prend l'exemple d'un paramètre correspondant au nombre de clignotements, nous souhaitons recevoir un nombre entier. Si un client envoit comme valeur pour ce paramètre un caractère ou une chaîne de caractères, le passage de String à int va nous renvoyer un nombre entier alors que le paramètre n'est pas conforme à nos attentes.

Le framework fourni par arduino contient une fonction "isDigit" permettant de savoir si un caractère est numérique, mais elle ne contient pas de fonction permettant de savoir si tous les caractères d'une chaîne String contient que des caractères numériques. Nous avons donc utilisé une fonction qui parcourt un objet JSON et vérifie si tous ses caractères le sont afin de vérifier si le paramètre est bien un nombre entier:

```
//vérifie que la valeur json en paramètre est numérique
boolean isValidNumber(aJsonObject *json) {
   boolean isNum = false;
   for(byte i=0; i < String(json->valuestring).length(); ++i) {
     isNum = isDigit(String(json->valuestring).charAt(i));
     if(!isNum) return false;
   }
   return isNum;
}
```

Pour éviter la redondance de variables entraînant une utilisation trop importante de la mémoire de la carte, cette fonction reçoit comme paramètre un pointeur sur une variable de type "aJsonObject" qui est comme son nom l'indique un objet JSON et effectue les tests directement sur cet objet.

III. Serveur REST

1 - Node.js

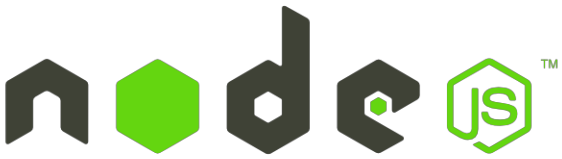


Figure 11 : logo node.js

Figure 12 : logo Google V8 Engine

Node.JS est un framework javascript basé sur le moteur google V8[13] qui s'éxécute coté serveur. Il permet donc l'utilisation de toute les fonctionnalité du js[14] sur un serveur, excepté la navigation dans le DOM[15] (qui n'a plus aucun sens dans ce cas). Ses caractéristiques sont les suivantes :

- Mode d'éxécution à processus unique : pas de thread, c'est node.js qui gére la répartition de charge en arrière plan ;
- Modèle évènementiel asynchrone, qui permet la gestion d'un grand nombre de requêtes en simultané. Ce modèle est basé sur une boucle qui gére une queue d'événnements. L'idée est qu'a un instant *t*, le programme ne reste pas sans rien faire. techniquement, cela se traduit de la façon suivant : lorsque l'on éxécute une fonction, le programme principale la "détache" et continu son éxécution. le retour de la fonction est idéalement passée via un callback[16], puis on peut traiter les données. Ce fonctionnement est similaire aux tâches asynchrones rencontrées en programmation android, à la différence qu'il est géré par node.js.

[13] **Google V8** : moteur javascript Open Source (https://code.google.com/p/v8/)

[14] **js** : abréviation pour *JavaScript*, langage de programmation basé sur *ECMAScript* (*http://www.ecma-international.org/publications/standards/Ecma-262.htm*)

[15] **DOM** : Document Object Model, convention de représentation d'objets en HTML et XML

[16] **callback :** fonction de rappel qui est passé en argument d'une fonction.

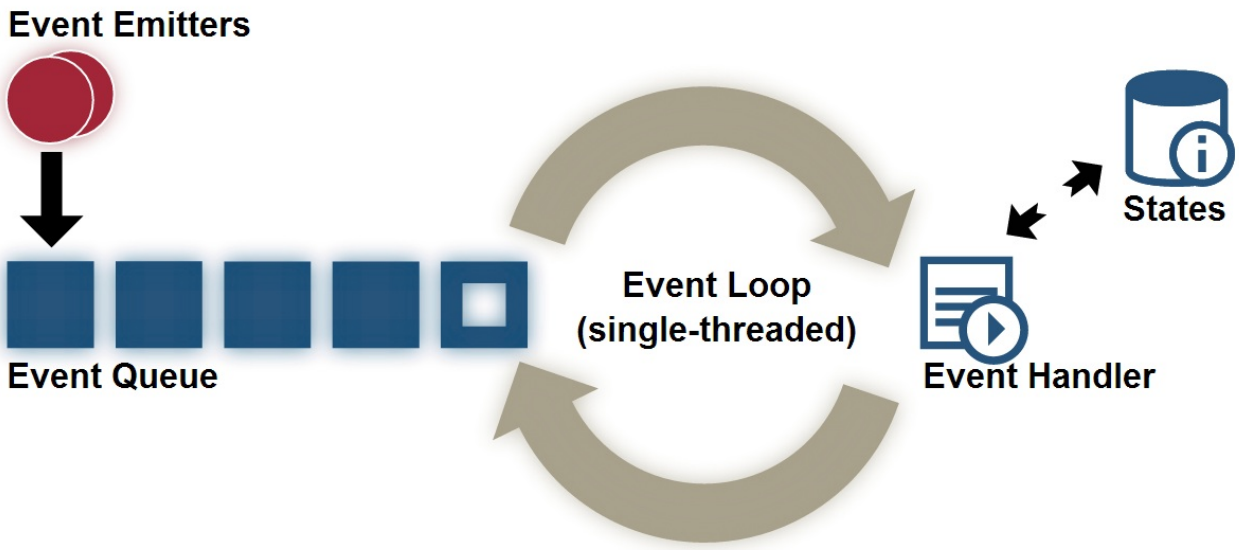


Figure 13 : modèle évènementiel asynchrone de Node.js

- Orienté application réseau : permet le déploiment rapide d'application web ;
- Système intégré de gestion des paquets (type "apt[17]" sur Ubuntu Linux) simplifiant grandement la gestion des dépendances.

Ce framework posséde différents avantage et inconvenients qui le rende avantageux dans certains cas d'utilisation, et inutile dans d'autres.

Avantages :

- "Scalable" : bonne capacité à supporter la montée en charge
- Unification du langage client / server dans le cas d'une application web basée sur javascript (développement simplifié) ;
- Grande communauté, donc grand nombre de modules[18] disponibles ;
- Pas de compilation donc simplification des tests et de la mise en oeuvre ;
- Multi-plateforme : déploiement fortement facilité par le moteur V8 dont les sources en C peuvent être compilées pour tous types de système d'exploitation.

Inconvénients :

- Nécéssite l'apprentissage du javascript, qui est aujourd'hui un langage de programmation à par entière avec ses propres conventions et caractéristiques ;
- Évolution très rapide : cycle de mise à jour des sources hebdomadaire, ce qui implique une constante évolution des fonctionnalité (et pas forcement de la documentation en conséquence...);
- Framework jeune : projets d'intégration en cours dans les IDE existantes donc débuggage encore complexe ;
- Pas prévu pour du traitement de données gourmand en ressources (type encodage / décodage vidéo), il faudra préférer un langage compilé type C ou java.

[17] **apt :** système de gestion des paquets (briques logicielles) utilisé par les distributions Linux de type *Debian.*

[18] **module** : extension qui permet d'ajouter simplement une fonctionnalitée à un projet (ex : framework web Express)

Concrètement, node.js est très puissant lorsqu'il s'agit de transit de données de points à points, comme entre une base de donnée à un client par exemple. Il permet, grâce à sa rapidité, de mettre en place des applications de type temps réel "mou"[19]. Par contre, celui-ci n'est clairement pas adapté pour du traitement de de donnée, de par sa nature interprété. À titre comparatif, voici les principales différences entre Node.JS et Apache[20] + PHP[21] :

	Apache + PHP	Node.js
Implementation technology	C	Google's V8 JavaScript Engine; C++
Architecture	Process-per-client	Event-loop
Non-blocking	No	Yes
Asynchronous operation	No	Yes
Memory requirement per client connection	15-30MB	1-2KB
Practical simultaneous connections per server	100-200	20,000-100,000

Figure 14 : Comparaison des caractéristiques entre Apache+PHp et Node.js

Ce framework trouve parfaitement sa place pour notre cas d'utilisation, à savoir faire transiter les messages entre les clients et les arduinos et proposé une API aux clients. Il est à prescrire en particulier pour des applications à grand nombre de clients.

Nous avions déjà eu l'occasion d'utiliser ce dernier lors de notre projet l'année précédente, ce qui nous a epargné la période de découverte initiale. Une période d'adaptation est cependant nécéssaire à cause de la logique de programmation sensiblement différentes des langages orientés objets.

2 - Méthodologie de développement

Comme avec les autres langages, le développeur est libre de procéder comme bon lui semble en javascript, on peut donc trouver de tout et n'importe quoi sur internet. Il est donc necessaire de se renseigner sur les bonnes pratiques de développment pour produire un code qui soit lisible, compréhensible et surtout maintenable.

Cela implique une étape préliminaire de lecture de tutoriels, cours et design pattern afin de s'imprégner du langage. Comme node.js s'appuie sur javascript, largement utilisé pour le développement web, on trouve de nombreuses ressources de qualité sur internet. (voir

[19] **Temps réel "mou" :** systémes temps réel pour lesquels l'incidence d'une durée de traitement ne respectant pas les contraintes est faible ou sans importance.

[20] **Apache** : serveur web

[21] **PHP** : language de programmation côté serveur pour le web

bibliographie). Nous avons aussi la chance de disposer d'une documentation officielle bien plus fournie que l'année passée, dans laquelle nous pouvons trouver de nombreux exemples d'application. Il en est de même pour les modules additionnel utilisés.

Nous avons choisis de suivre le modèle traditionnel en couche que nous avons utilisé de nombreuses fois lors de notre cursus scolaire. Cela nous permet de conserver l'avantage de la séparation des différentes fonctions, facilitant le développement ainsi que les tests. Cela nous a grandement facilité le travail lors des différents refactoring de l'application !

La logique de validation par itération de type "cycle en V" à été appliquée lors du développement :

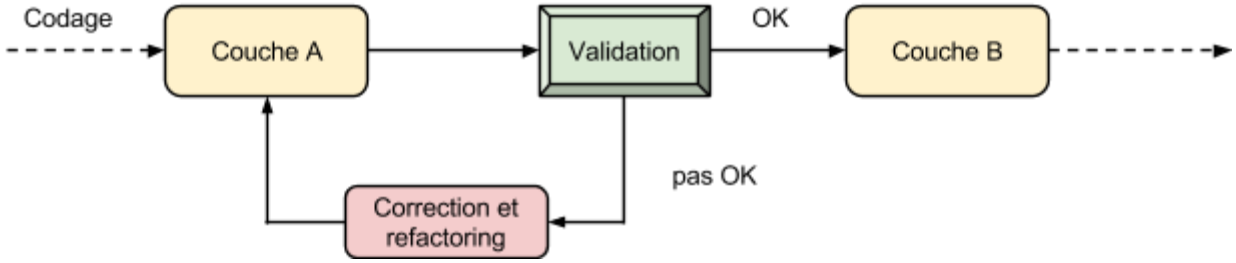


Figure 15 : logique de validation appliquée

3 - Spécifications

Les spécifications du serveur REST sont issues d'une réfléxion préliminaire, ainsi que d'une amélioration continue. Il est à noter que le serveur répond aux standards de l'architecture REST-like et non REST-full, car nous gérons uniquement les requêtes de type "GET" et "POST" ("PUT" et "DELETE" ne sont pas utilisés dans notre cas).

3.1 - Architecture

Outre l'application du modèle en couche,s nous avons définin l'architecture suivante pour le serveur :

- Une couche [WEB] pour la communication avec les clients web ;
- Une couche [DAO] pour la communication avec les arduinos ;
- Une couche [METIER] pour le traitement des données, en particulier la construction des requêtes pour l'arduino et des réponses client ainsi que la validation des paramètres reçus.

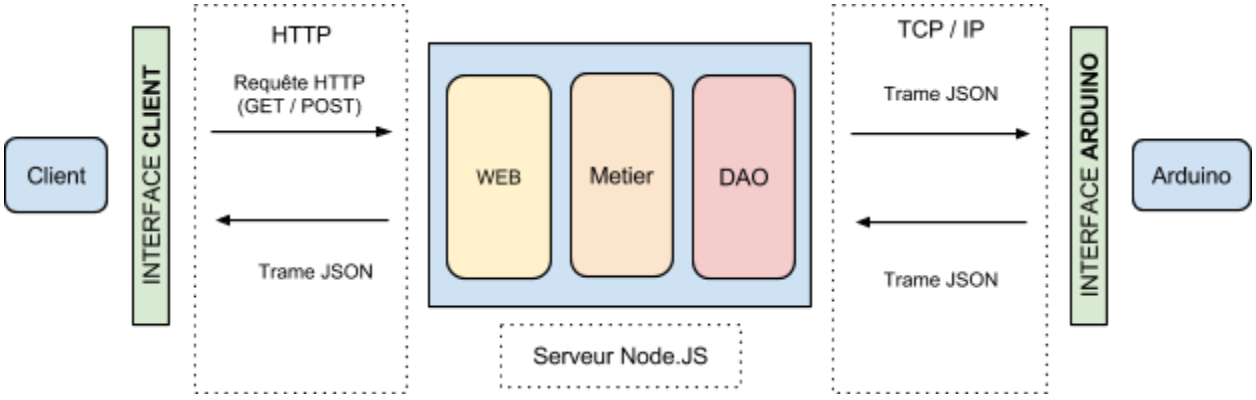


Figure 16 : schéma du serveur REST

3.2 - Fonctionnement

On peut voir ces couches comme une sorte de “pile”, par laquelle vont transiter les requêtes entre le client et l’arduino. C’est la couche [WEB] qui, une fois éxécutée dans le terminal, lance le serveur REST et appelle les autres couches en cascade.

Cela nous permet de répartir les fonctionnalités du serveur dans chacune des couches de la manière suivante :

1. Couche [**WEB**] :

- Récéption des requêtes des clients ;
- Si la requête respecte l’API client, routage de la requête ;
- Si la requête ne correspond pas, reponse au client avec un message d’erreur ;
- Extraction des données de l’URL (GET / POST) et des données transmises (POST) ;
- Appel de la fonction de la couche [METIER] correspondant à la requête avec les paramètres ;
- Récéption de la réponse de la couche [METIER] ;
- Sérialisation de l’objet JSON reçu ;
- Réponse au client.

2. Couche [**METIER**] :

- Réception des données de la couche [WEB] dans la fonction apropriée pour la requête cliente en cours de traitement ;
- Vérification de la validité des paramètres ;
- Si les paramètres sont valides : construction de l’objet JSON correspondant et passage à la douche [DAO] ;
- Si un ou des paramètres sont érronés, construction et retour de l’erreur correspondante (objet JSON) à la couche [WEB] ;
- Réception de la réponse de la couche [DAO] (JSON sérialisé en provenance de l’Arduino ou message d’erreur objet JSON);
- Si on a reçu une erreur, on fait remonter la réponse à la couche [WEB] directement ;
- Si la réponse vient d’un arduino : dé-sérialisation et validation de la réponse ;
- Construction d’une nouvelle réponse au format attendu par le client (spécifié par l’interface client) ;
- Retour à la couche [WEB] de l’objet JSON ;

3. Couche [DAO] :

- Partie Cliente (gère la communication avec l’arduino) :
    - Réception de l’objet JSON de la couche [METIER] ;
    - Sérialisation de celui-ci ;
    - Pas d’erreur possible normalement car les paramètres ont été vérifié en amont et l’objet JSON a été construit par la couche [METIER], mais on remonte l’erreur à cette couche-ci si la sérialisation a échoué ;
    - Si l’arduino sélectionné n’est pas connecté au server, on fait remonter l’erreur correspondante à la couche [METIER] ;
    - Envoi de la chaîne JSON à l’arduino ;

- Si une erreur de communication survient, on fait remonter l'erreur correspondante à la douche [METIER] et on supprime l'arduino de notre collection (permet d'éviter les erreurs futur, un reset de l'arduino est alors nécessaire pour l'ajouter à nouveau à notre collection);
- Si on obtient une réponse de l'arduino, on remonte cette chaine JSON à la couche [METIER] ;

- Partie serveur (gère l'enregistrement des arduino) :
  - Réception d'une chaine JSON d'enregistrement d'un arduino ;
  - Sérialisation de cette chaine ;
  - Si une erreur survient, on l'affiche dans les log du serveur ;
  - Si la sérialisation abouti et que l'arduino n'est pas dans notre collection, on l'enregistre ;
  - Sinon, on affiche dans les log du serveur un message d'avertissement.

L'ordre de développement à été le suivant :

1. la couche [WEB], avec validation des points d'entrée ;
2. la couche [DAO], avec validation de la communication avec les arduinos ainsi que leur enregistrement ;
3. la couche [METIER], avec validation des tâches de construction, de la gestion des erreurs et de la montée & descente des données entre les clients et les arduinos au travers des trois couches.

Remarques :

Les résultats intermédiaire ainsi que toute les erreurs rencontrées par le serveur sont loguées dans la console et permettent de suivre le déroulement des requêtes en cours.

Il est important de noter que nous pouvons tout à fait recevoir plusieurs requêtes de clients différents en même temps sur le serveur, grâce à l'aspect asynchrone non-bloquant de node.js. C'est l'objectif même de Node.js et c'est ce qui fait sa force.

Cependant, si un arduino est sollicité par plusieurs requêtes simultanées, il constitue un goulet d'étranglement de par son comportement séquentiel. Le serveur node.js va construire une file d'attente de requêtes, qu'il va envoyer au fur et à mesure que l'arduino lui répond.

3.3 - Scénario de fonctionnement

Afin de mieux comprendre le fonctionnement du serveur lors d'une requête, observons ce qu'il se passe lors d'un cycle sans erreurs. nous prenons deux requêtes :

1. une requête de type "GET" pour une action "blink" ;
2. une requête de type "POST" avec en paramètre un tableau de requête vers un arduino.

- Voici le schéma illustrant le transit des données lors de la requête "blink" suivante : http://ip:port/rest/arduinos/blink/idComand/idArduino/pin/nombre/durée/

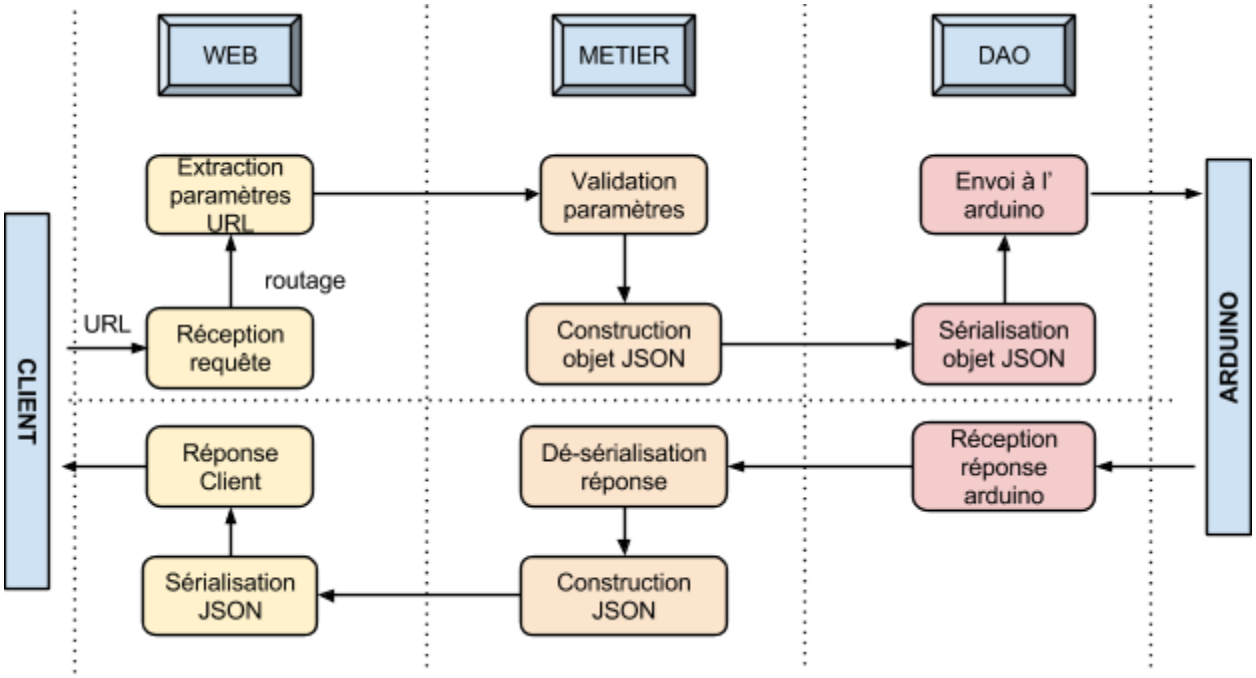


Figure 17 : schéma de transit des données pour une requête “blink”

- Voici le schéma illustrant le transit des données lors de la requête “command” avec l’URL et le POST suivants :
  http://ip:port/rest/command/idArduino/

POST data = [{"id":"1","ac":"pw","pa":{"val":"0","pin":"8","mod":"b"}}, {"id":"2","ac":"pr","pa":{"pin":"8","mod":"a"}}, {"id":"3","ac":"cl","pa":{"pin":"8","dur":"100","nb":"10"}}]

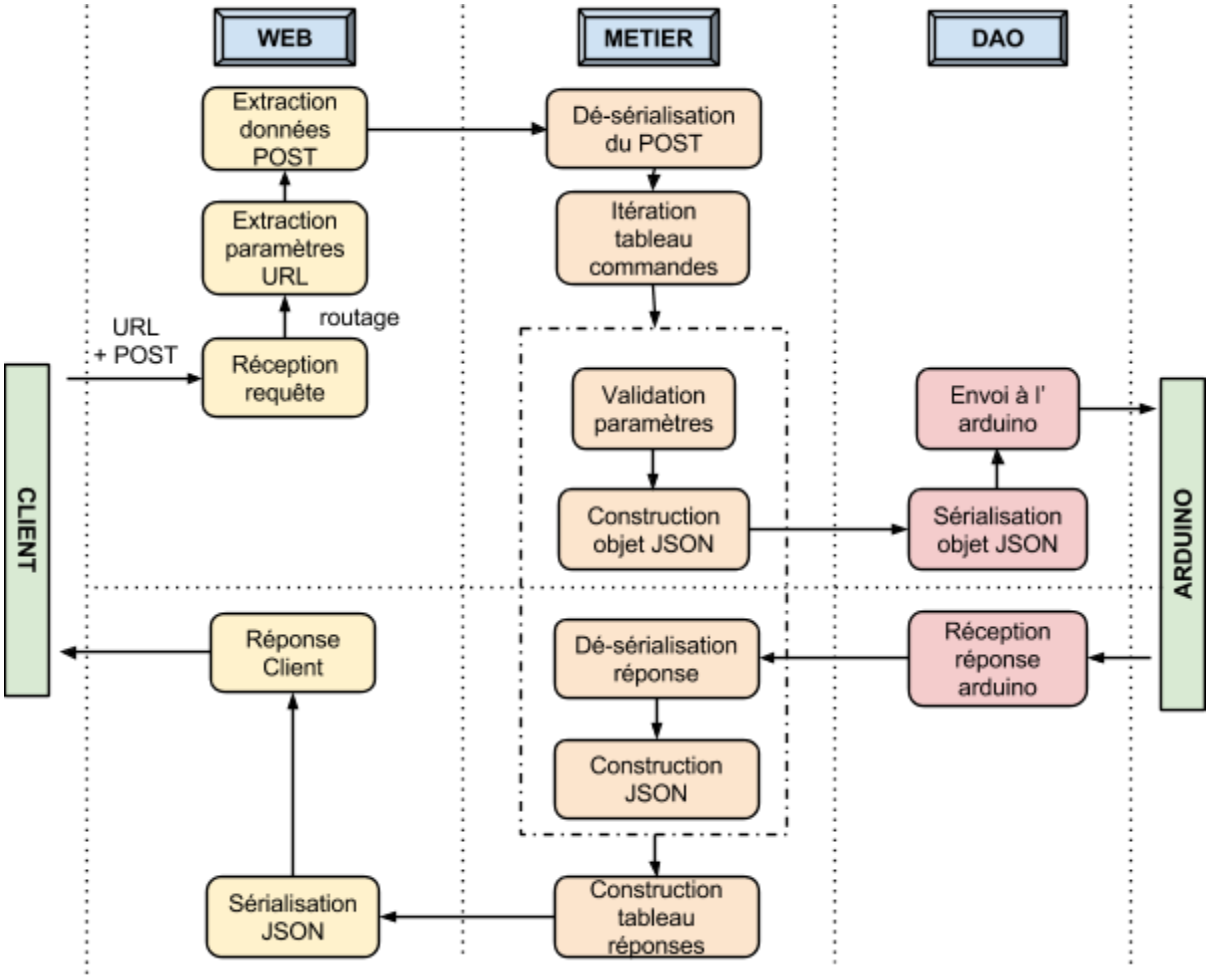


Figure 18 : schéma de transit des données pour une requête “command”

Pour avoir un aperçu de ces représentations avec la gestion des erreurs, il suffit d’imaginer que l’on ajoute, pour chacune des cases d’action, une flèche vers la couche précédente ou le client si l’action courante rencontre une erreur. Pour des raisons évidente de clareté, elle ne seront pas représentées ici.

3.4 - Interfaces internes

Nous avons défini la spécification des interfaces entre les couches internes du serveur comme il suit :

- La couche [WEB] respecte l’API client ;

- La couche [METIER] :
  1. arduinos(): récupère la liste des différents arduinos connectés et les retourne dans une trame au format json ;
  2. blink(idCommand, idArduino, pin, duree, nombre): vérifie que l’arduino existe,

vérifie la validité des paramètres, construit la trame json et appelle la fonction dao send avec cette trame ;

3. read(idCommand, idArduino, pin, mode): vérifie l'existence de l'arduino et la validité des paramètres, appelle la fonction dao send avec les paramètres au format json ;
4. write(idCommand, idArduino, pin mode, valeur): vérifie l'existence de l'arduino et la validité des paramètres, appelle la fonction dao send avec les paramètres au format json ;
5. cmd(idArduino, parametres): vérifie l'existence de l'arduino, vérifie la validité des paramètres (données du POST), appelle la fonction dao send avec les paramètres au format json ;

- La couche [DAO] respecte l'interface arduino en externe, et propose l'interface suivante en interne :
    1. sendToArduino(idArduino, jsonObject): vérifie la présence de l'arduino dans notre collection, sérialise la trame json, envoi celle-ci à l'arduino.

Remarques :

Si le serveur rencontre une erreur, il renvoit la trame JSON suivante :

```
{"data":{"message":"some text go there"}}
```

Dans le cas ou l'arduino n'est pas ou plus connecté, si un problème de sérialisation ou dé-sérialisation survient, ou encore si une erreur réseau survient.

3.5 - Choix technologique

Afin de simplifier le développement et pour ne pas de ré-inventer la roue, nous avons choisis d'utiliser certains modules (ou frameworks) pour node.js. Ces choix ont été effectués en prenant soins de vérifier la maturité et la maintenabilité de chacun de ces modules.

3.5.1 - Gestion des routes

Afin de simplifier la couche web et surtout la gestion du routage des requêtes, nous avons choisis d'utiliser **Express.js**[22], qui est un framework robuste et minimaliste mettant à disposition les fondamentaux pour déployer une application web.

Il faut noté que node.js est très bas niveau en ce qui concerne les paramèters des requêtes et réponses: il faut gérer "à la main" les données du header[23] ainsi que tous les paramètres de la transaction. L'utilisation d'Express permet d'automatiser une partie de ce traitement pour ce concentrer sur la fonctionnalité. Son utilisation est la suivante :

On créé et on lance le serveur :

```
// Import required modules
var express = require('express');
var server  = new express();
// start server
server.listen(PORT, function() {
```

[22] **Express.js**: framework web pour Node.JS (http://expressjs.com/)

[23] **header :** En-tête des données transitant entre un client et un serveur.

```
    // log
    util.log('[WEB] Server launched on port ' + PORT);
});
```

On peut ensuite facilement ajouter un point d'entrée (une route) :

```
// Arduino list query: http://localhost:8080/rest/arduinos
.get(BASE_PATH, function(req, res) {
    util.log('[WEB] Query : Arduino list');
    metier.arduinos(function(arduinos) {
        // serialize & send answer
        sendToClient(arduinos, function(jsonString) {
            util.log('[WEB] ARDUINOS sending back : ' + jsonString);
            res.send(jsonString);
        });
    });
})
```

On voit clairement la simplicité d'utilisation. Nous laissons le soin au lecteur de se renseigner sur la création d'un serveur sans framework, le nombre de ligne est rapidement multiplié avec un code complexe.

3.5.2 - log console

Pour pouvoir avoir des logs horodatés, nous avons utilisé le module **util.js**[24], qui ajoute automatiquement la date et l'heure dans les sortie console. Ce module est très pratique pour le débuggage. Il est à noté qu'il founit d'autre fonctionnalité que nous n'avons pas utilisé comme l'inspection d'objet, et qu'il est maintenant intégré à node.js.

Voici un exemple de sortie console suite à différentes requète, qui montre l'évolution des données au travers des différentes couches :

[24] **Util.js** : utilitaire pour node.js (http://nodejs.org/api/util.html)

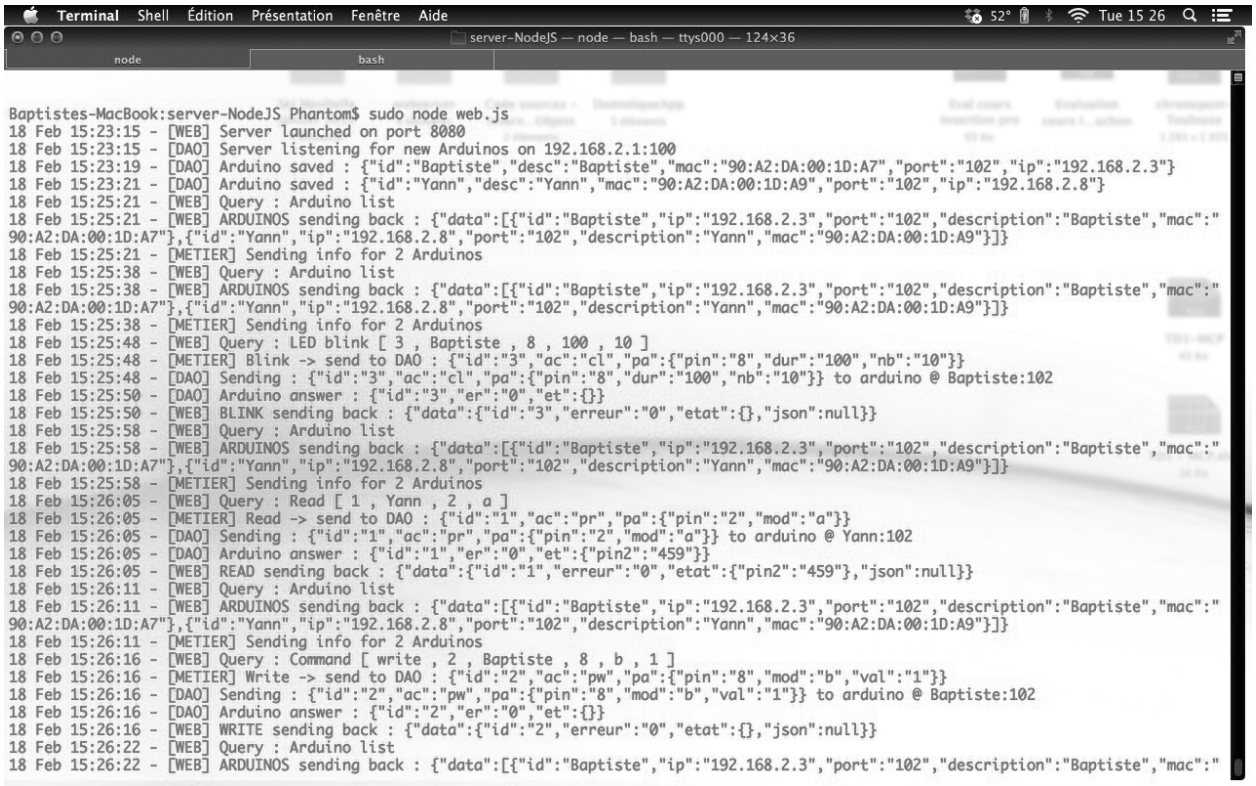


Figure 19 : sortie terminal lors de l'exécution du server

3.5.3 - Synchronisation des tâches

Il n'est pas évident de controler un flux de données avec node.js, en particulier pour une éxecution synchrone de tâche qui soit non-bloquante vis-à-vis du reste du serveur. En effet, la nature asynchrone de node.js complique cela. Afin de palier à ce problème, nous avons utilisé le module **Async.js**[25] (ne pas se fier à son nom trompeur) qui permet de controler un flux de données de différentes manières. C'est indispensable lorsque nous recevons un tableau de commandes en POST : il faut en effet effectuer les tâches et récupérer les réponses dans l'ordre, avant de répondre au client. Sans cela on peut avoir des réponses qui ne sont pas dans le même ordre que les commandes de la requête, ou pire ne pas retourner tous les élements de réponses.

Son utilisation est la suivante :

```
// start Synchronous call on an array of objects
async.eachSeries(
    // the array to iterate over
    ObjectArray,
    // the iterator function
    function(object, callback) {
        // do something with current object
    },
    // the final callback (or when error occure)
    function(err) {
        // do something when all tasks are over, or on error
    }
);
```

[25] **Async.js** : modules pour node.js permettant de simplifier la gestion des flux de données

Cette structure est nécéssaire afin :

- D'effectuer toute les tâches une par une dans le cas d'un POST avec un tableau de JSON ;
- De récupèrer dans l'ordre les réponses de l'arduino ;
- De renvoyer notre tableau de réponses JSON seulement quand nous avons fini toutes les requête sur l'arduino.

4 - Difficultés rencontrées

Plusieurs difficultés ont emmergés lors du processus de développement. Voici une liste non exhaustive des plus importantes d'entre elles :

- **L'utilisation de javascript**, et en particulier des "callback" s'est révélée déroutante au début.
  Pour rappel, un callback est une fonction que l'on passe en paramètres d'une autre fonction. Ce callback permet de retourner de manière asynchrone une ou des variable issues de la fonction appellée, vers la fonction appelante.
  L'effet principale est le suivant : à la différence de l'instruction "return", un programme node.js n'attend pas le retour d'un callback pour continuer son éxécution. Il faut alors faire très attention à l'enchainement d'action, et en particulier au données que l'on utilise suite à l'appel d'une fonction.
  Cela a un effet secondaire : on se retrouve avec plusieurs fonctions imbriquées les unes dans les autres, ce qui compléxifie grandement le code. Ce problème peut être compensé par l'utilisation d'une architecture en couches.

- Il n'est pas évident d'appliquer les **bonnes pratique** de codage et les **design pattern** en javascript, à cause du problème vu précédemment ;

- Le **debuggage** en javascript n'est pas évident sans utiliser de plugin externe (pour l'IDE Eclipse par exemple). Lors du plantage d'un programme dans la console, on obtient une pile d'erreurs plus ou moins compréhensible en fonction du code ;

- La **synchronisation** du flux de données est complexe sans utiliser un module externe comme Async.js, de part l'aspect asynchrone de node.js. Par exemple, si on n'utilise pas le modèle vu ci dessus, on peut se retrouver avec le comportement suivant :

```
// array of command
var cmdArray = [{json1}, {json2}, {json3}];

// array of answer
var responseArray = [];

// iterate trought array
cmdArray.forEach(function(json) {
    // send command
    dao.send(json, callback(answer) {
```

```
        // save answer
        responseArray.push(answer);
    });
})

// send back answer
callback(responseArray);
```

Ici, lorsque l'on retourne notre tableau de réponse à la fonction parente, celui-ci est **vide**! En effet, Node.js n'attend pas que la couche [DAO] réponde pour passer à l'instruction suivante (asynchronisme). Au contraire, il va continuer l'éxécution du programme et traitera l'instruction "responseArray.push(answer)" uniquement lors de la réponse de la couche [DAO], et remplira le tableau de réponses en conséquence. Il est donc nécéssaire de traiter ces itérations selon le modèle synchrone vu précédement sous peine de soucis de fonctionnement.

- Nous utilisons la ligne de commande pour exécuter notre serveur. Le **déploiement** de l'application en mode production n'est pas évident, à moins d'avoir recours à un service en ligne (ce qui peut être problèmatique pour notre application avec les arduinos) ;

- Nous avons eu un soucis avec l'utilisation d'Express.js pour récupérer les données contenues dans le POST. En effet, le parser intégré du module vide le post si il ne rencontre pas certains "datatype" spécifique. Nous avons eu du mal à isoler le problème, mais avons pu facilement mettre en place un parser manuel une fois notre soucis identifié. Ce parser va récupèrer toutes les données du body lors de la réception de données POST afin d'en extraire ensuite la trame JSON. Voici son intégration :

```
// hand-made body parser
function rawBody(req, res, next) {
    req.setEncoding("utf8");
    req.rawBody = '';
    req.on("data", function(chunk){
        req.rawBody += chunk;
    });
    req.on("end", function() {
        next();
    });
}
```

À la récéption de la requête, on extrait les données du body que l'on insére dans une nouvelle propriété de la requête ("rawBody"). on peut ensuite y accéder comme une propriété de notre requête.

5 - Axes d'amélioration

Une suite interessante à donner à ce projet serait le déploiement sur un serveur privé de l'application, pour y accéder à distance au travers d'internet.

IV. Clients jQuery Mobile

Ce projet a été l'occasion d'utiliser la technologie jQuery pour mobile avec laquelle nous n'avions pas travaillé précédement.

1 - jQuery Mobile

jQuery Mobile est une librairie javascript dont l'objectif et la création d'application web ou de site web à destination des plateformes mobile comme les smartphones ou les tablettes. Il apporte une syntaxe javascript simplifiée, un système d'évènements relatif aux plateformes mobiles, ainsi qu'une manipulation plus simple du DOM.

L'intéret principal de cette librairie est son aspect esthétique, qui permet de déployer rapidement une application web sans perdre de temps sur la partie visuel. La blibliothèque de composants graphiques disponible ainsi que les démos et la documentation officielle facilite l'apprentissage ainsi que le développement.

L'objectif est de mettre en place une application web de type APU[26] qui puisse communiquer avec un serveur REST en respectant l'API mise en place, dans le but de piloter les arduinos connectés au serveur.

2 - Spécifications

L'application permet d'intéragir avec le serveur, et doit en particulier apporter les fonctionnalités suivantes :

- Configuration de la connection au serveur ;
- Récupération de la liste des arduinos connectés au serveur ;
- Paramétrer et lancer les différentes actions possibles avec l'arduino (blink, pin read, pin write, command).

Cela représente un total de six vues : une pour le menu principale, une pour la configuration du serveur REST, quatre pour les actions.

2.1 - Architecture

Il aurait été intressant de mettre en place une architecture de type MVC[27], mais avec le peu de temps à notre disposition et de ce que j'ai pu en voir sur internet, je n'ai pas pris le risque d'essayer. Comme l'application reste simple (peu de vue), cela ne pose pas vraiment de problème.

Le projet est structuré de cette manière :

- Un fichier "**index.html**" qui contient le code html des vues ;
- Un dossier "**js**" qui contient le script javascript de l'application, ainsi que les scripts locaux de jquery mobile. Les serveurs de contenu seront utilisé à la fin du

[26] **APU** : *Application à Page Unique* (Single Page Application)
[27] **MVC** : Modèle Vue Contrôleur, design pattern répondant aux besoin d'applications interactives

développment (les liens vers les CDN[28] de jQuery sont commentés dans le code HTML, pour pouvoir utiliser le client sans accès internet) ;
- Un dossier "**content**", qui contient le gif utilisé pour afficher l'attente et les éventuelles images à afficher sur les pages.

2.2 - les pages

Le design des pages de l'application est sobre, afin de privilégier les fonctionnalités. Je ne prétend pas que le visuel de l'application est ce qu'il y a de mieux, et suis bien conscient de la nécéssité du travail d'un designer lorsque l'application est fonctionnelle. Le menu principal permet de naviguer entre les différentes pages de commande et la page de réglage.

Chaque page a un contenu qui lui est propre mais qui suit une organisation d'ordre général. On retrouve dans toutes les pages liées aux actions les éléments suivants :

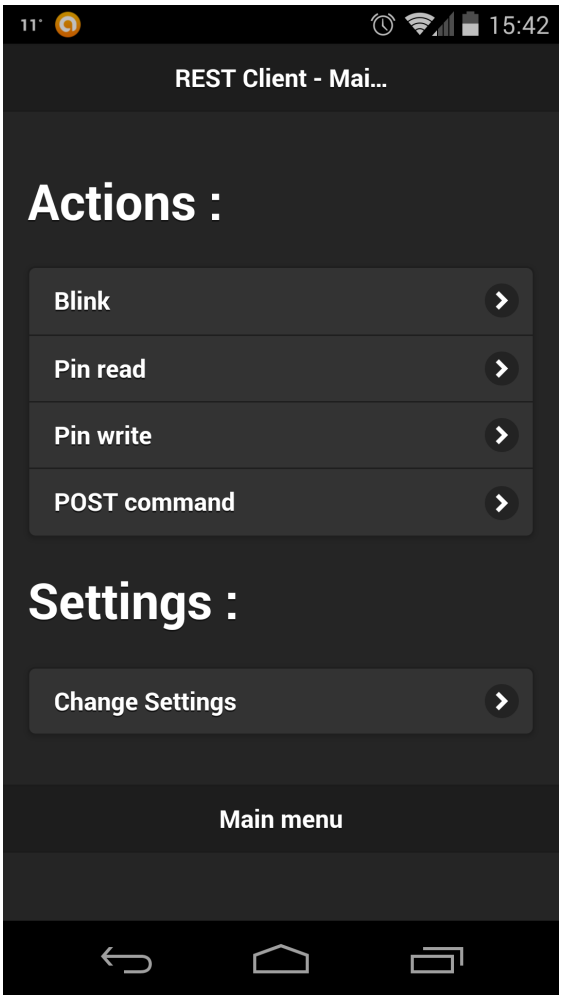
1. Un "header" :
    a. Le titre ;
    b. Un lien vers le menu principal.
2. Une zone de contenu :
    a. Une liste déroulante permettant de séléctionner l'arduino sur lequel on souhaite agir ;
    b. Une section de paramétrage relatif à l'action ;
    c. Une section de boutons d'action ;
    d. Une section permettant l'affichage du résultat et de l'indicateur de chargement ;
3. Un "footer" (pied de page) :
    a. Une barre de navigation entre page avec
    b. Un second lien vers le menu principal.

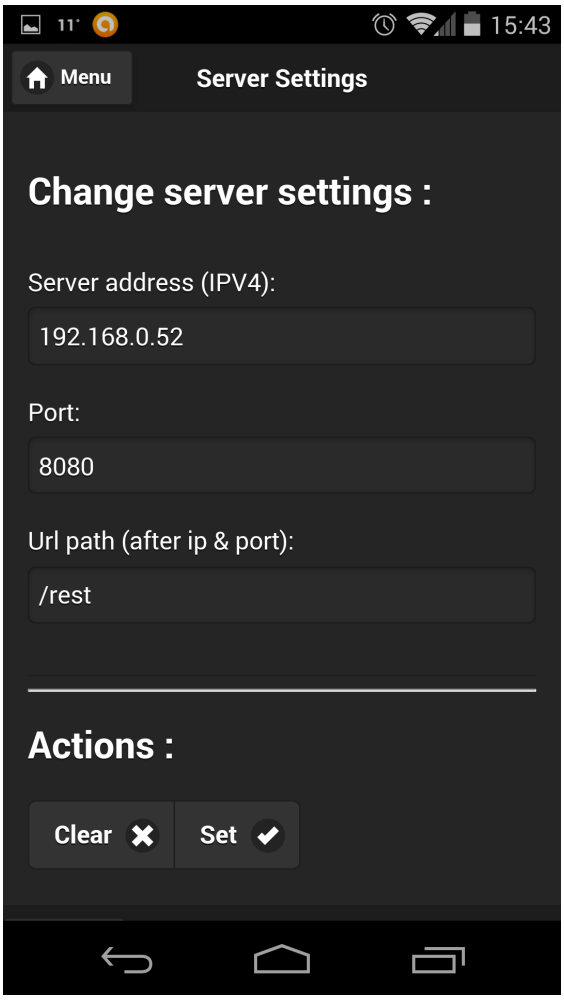
La page de réglage comporte uniquement les champs suivant : 1, 2.b, 2.c et 3. La section de paramètrage comporte les champs de saisie des paramètres suivant :

- ip ;
- port ;
- url de base du serveur.

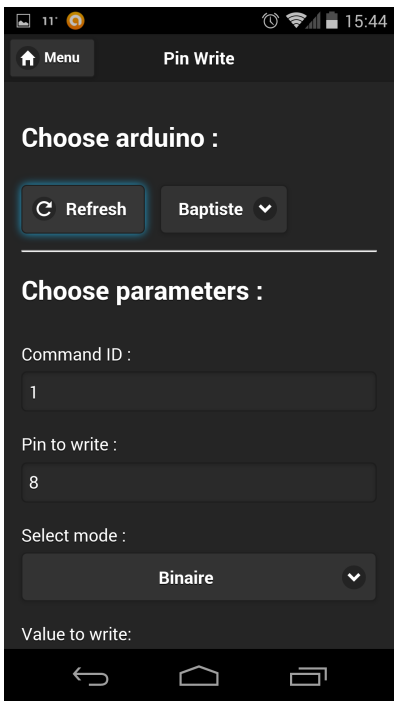
Voici différentes vue de l'application lors d'une utilisation sur un mobile :

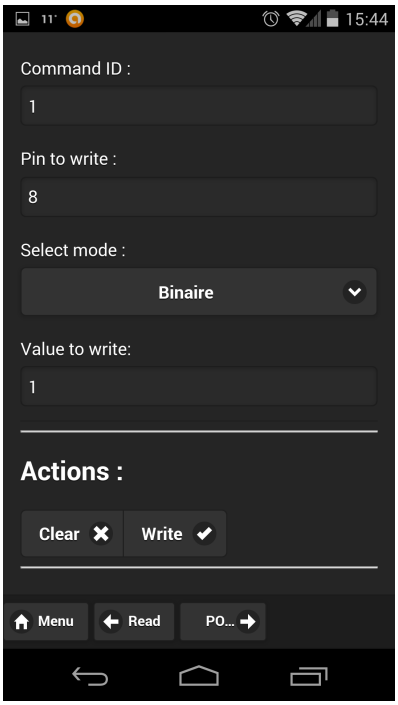
[28] **CDN** : Content Delivery Network, système permettant de servir du contenu avec une grande disponibilité et de grandes performances

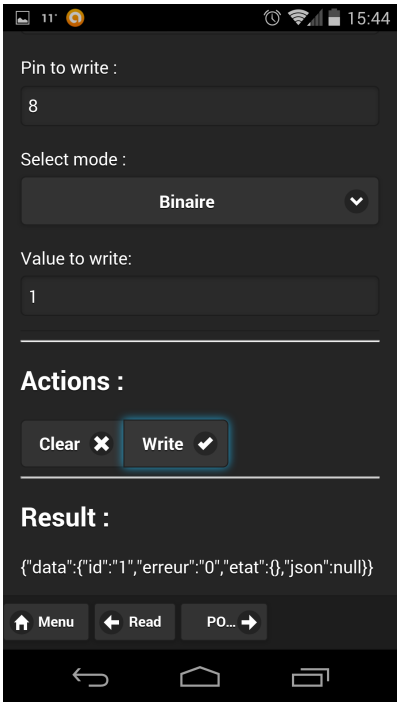




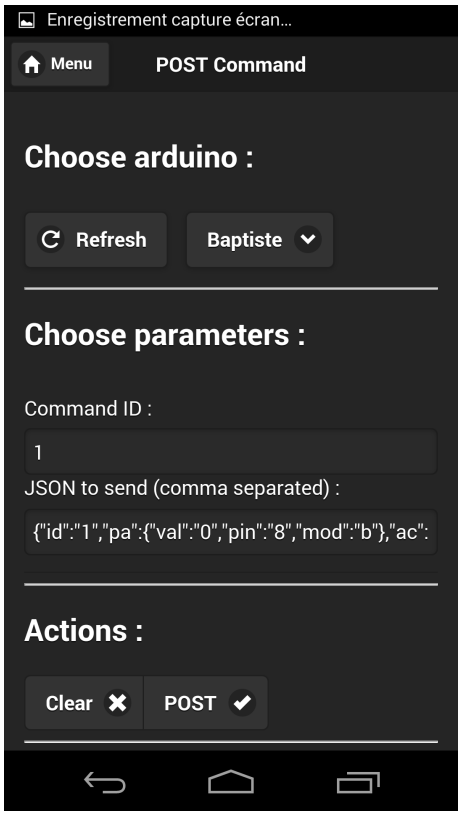
Figures 20 - 21 : Menu principal de l'application et page de réglage

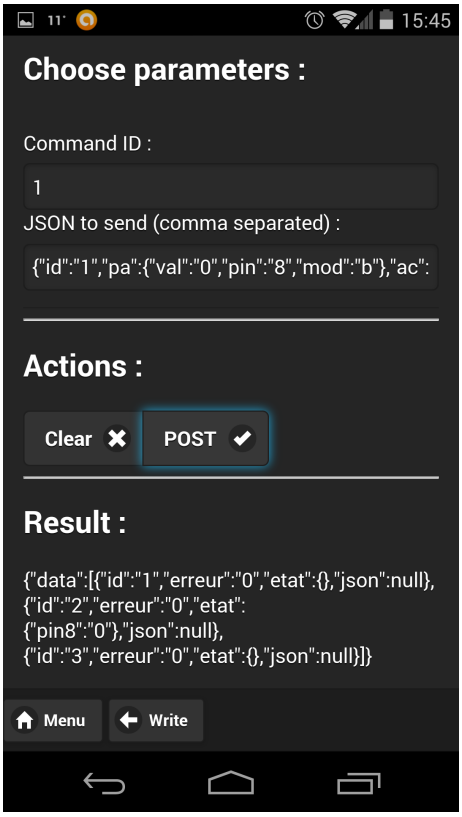






Figures 22-23-24 : page de commande "Write" avant et après l'envoi





Figures 25 - 26 : page de commande "Post" avant et après l'envoi

2.3 - le comportement

Initialement, l'application est configurée pour chercher d'abord un serveur en local sur le port 8080 (paramètres par default modifiables dans le code javascript).

Lors du lancement de l'application :

- On affiche la page de menu. Cette dernière permet de choisir entre les différentes actions possibles ou le paramétrage ;
- On effectue une requête ajax[29] GET pour recupérer la liste des arduino sur le serveur.

Lors d'un changement de page :

- La section d'affichage des résultats et de l'indicateur de chargement est cachée (avant d'effectuer une autre action, évite d'afficher le résultat précédent).

Pour toutes les actions :

- les paramètres saisis par l'utilisateur dans un formulaire sont validés avant d'envoyer la requête, ce qui permet d'effectuer une première validation coté client ;
- L'indicateur de chargement est affiché dans la section de résultat lors de l'évènement "ajaxStart", et caché lors de l'évènement "ajaxComplete" ;
- La section d'affichage du résultat et de l'indicateur de chargement est rendu visible.

Lors de la validation d'une action exploitant un point d'entrée de type GET :

[29] **ajax** : asynchronous javascript and xml, technique permettant de récupérer des informations de manière asynchrone

- Une requête AJAX de type GET est lancée sur l'URL spécifique de l'action ;
- Si une erreur est rencontrée, une fenêtre de dialogue s'ouvre et affiche les détails de l'erreur ;
- Si pas d'erreur, on désérialize le JSON pour vérifier que le format et bon, puis on l'affiche dans la section de résultat.

Lors de la validation de l'action "Command" exploitant un point d'entrée de type POST :

- Même chose que pour le "GET", à la différence que l'on effectue une requête ajax de type POST, avec en donnée le tableau de commandes JSON sérialisé.

3 - Service depuis le serveur REST

J'ai décidé de mettre à profit une fonctionnalité du framework Express que nous utilisons pour le serveur node.js : le service de fichiers statiques. Grâce à une simple ligne de code, on peut demander à express de servir les fichiers d'un répertoire spécifique, comme un serveur web traditionnel type Apache.

Il suffit ensuite d'ajouter un point d'entrée à la racine pour rendre possible le chargement du client web jQuery ("index.html)" directement depuis le serveur REST, lorsque le client charge l'url principale du serveur dans son navigateur. On se trouve ainsi avec un service tout-en-un ou une simple adresse URL permet de piloter les arduinos.

4 - Problèmes rencontrés

Jquery est relativement facile à prendre en main, mais il subsite quelques zones d'ombre qu'il a fallu éclairer lors de son utilisation:

- jQuery Mobile est une extension qui ajoute principalement des éléments graphique a la libraire jQuery, mais Il existe de légères différences qu'il faut prendre en compte.
  Comme JQM effectue des opérations suplémentaires (principalement de mise en forme) avant l'affichage du DOM, l'initialisation des éléments graphiques va être différente : il faudra utiliser l'évènement **$(document).bind('pageinit')** au lieu de **$(document).ready()**. Il faut faire attention à la structure du code javascript car l'ordre et le placement du code à un impact important.

- Au début de l'utilisation, l'architecture à fichier HTML unique est troublante, mais on s'y fait rapidement ;

- Certaines fonctionnalités intéressantes proposées par jQuery mobile ne permettent pas de faire remonter certaines erreurs.
  Si on veut effectuer une requête de type AJAX à laquelle on attend un résultat JSON, il existe une méthode **getJson(...)**. Il est cependant préférable de lancer soit-même une requête AJAX personnalisée afin de gérer manuellement les erreurs (communication, sérialisation, etc) ;

- Le client marche sans soucis en local, mais des problèmes surviennent dès lors d'une utilisation en reseau. Les navigateurs actuels limitent en effet les requètes ajax au

contexte d'éxécution local du navigateur, par mesure de sécurité. Cela complique la tâche du client lorsqu'il doit effectuer ses requêtes vers un serveur distant. Pour palier à ce problème, il est necessaire de régler le serveur lui-même en lui indiquant qu'il doit accepter les requêtes ayant une origine externe : on parle alors de CORS[30]. voici l'implémentation coté serveur :

```
// Allow CORS, tricky ! (needed when working under different Domains
Names)
serveur.all('/*', function(req, res, next) {
    // allow all domaines as Oringin of the request
    res.header("Access-Control-Allow-Origin", "*");
    // specify the supported CORS methodes
    res.header("Access-Control-Allow-Methods","GET, POST");
    // set content type
    res.setHeader('Content-Type', 'application/json');
    // call next middleware
    next();
})
```

La configuration précédent s'applique à toutes les requêtes entrantes du serveur (paramètre "/*").

5 - Axes d'amélioration

De nombreux axes d'amélioration sont envisageables pour ce client. Ce sont, pour la plupart, des fonctionnalités additionnelles que je n'ai pas eu le temps d'implémenter.

- Utilisation de liste à sélection multiple pour envoyer la même requête à plusieurs arduino. C'est en réalité une erreur de compréhension de ma part, car cette foncitonnalité était explicitement attendu des clients dans le sujet de TP.
- Gestion de la gestuelle pour l'utilisation sur un mobile (changement de page par glissement de doigt, etc) ;
- Envoyer une seule requête pour mettre à jours les différentes listes déroulantes. Le code relatif à cette fonctionnalité est présent mais commenté dans le fichier javascript. Pour une raison qui m'échappe, quelque chose se passe mal lors du remplissage des liste déroulante ;
- Conserver les réglages de la page settings pour chaque utilisateur, en utilisant les cookies par exemple ;
- Utilisation d'un modèle de type MVC pour gérer les vues et les données
- Refonte de la navigation en remplaçant le menu principal par la page de configuration directement (à l'instar des TD de mobilité). C'est bien plus logique et évite de lancer des requête GET vouées à l'echec vers un serveur qui n'existe pas ;
- Refonte du design, en gardant à l'esprit l'utilisation mobile.

[30] **CORS** : *Cross Origin Ressource Sharing*, permet les requêtes de navigateurs qui ne sont pas sous le même nom de domaine.

V. Client iOS

Mon stage Ei4 avait été l'occasion de découvrir le développement mobile et plus particulièrement pour les cibles iOS. Le développement des parties principales (serveur REST et Arduino) ayant pris moins de temps que prévu, j'ai souhaité utiliser les séances projet afin de créer un client iOS ciblé iPhone afin d'approfondir des fonctionnalités du framework que je n'ai pas eu le temps d'aborder durant ma période de stage.

1 - Avantages et inconvénients

Dans le cadre de ce projet, on pourrait être amené à se questionner sur les intérêts de développer une application iPhone ou iPad par rapport à un client web responsive design[31] tel que le client JQuery Mobile présenté précédemment.

Les applications pour les systèmes iOS et leur développement possède des avantages non négligeable :

- une expérience utilisateur plus approfondie : une application iOS se "fond" dans le décor du terminal qui l'accueille, c'est à dire que son fonctionnement et son interface graphique sont cohérentes avec la manière dont l'utilisateur est habitué à utiliser son smartphone ;
- les temps de développements sont réduis lorsqu'il s'agit de créer une application pour iPhone et pour iPad étant donné que le code peut rester le même et que seules les interfaces visuelles sont amenées à être recrées ;
- les temps de calcul de données sont plus rapide sur une application native que lorsqu'il sont exécutés avec javascript dans un navigateur.

Néanmoins son développement est plutôt contraignant :

- l'application doit être validée par Apple, ce qui peut être long, surtout si elle ne correspond pas aux standards d'exigence d'Apple ;
- la license développeur permettant de tester l'application sur un vrai terminal au lieu d'un simulateur et permettant de proposer son application sur l'AppStore[32] est chère (environ 100$) ;
- l'application ne doit pas planter afin que son utilisation reste agréable et que l'utilisateur ne soit pas frustré, le programme doit donc capturer toutes les exceptions possibles, ce qui est plus long à mettre en place et pas forcément nécessaire à faire dans le cas d'un client web.

2 - Spécifications

Le client iOS, ciblé iPhone, doit assurer les fonctions suivantes:

- proposer les vues des commandes (pinRead, pinWrite, doClignoter)

[31] **responsive design** : approche de création de vues qui s'adapte au format d'affichage (taille écran, résolution, etc.)

[32] **AppStore** : boutique d'applications iOS et Mac gérée par Apple

- proposer une vue permettant d'envoyer directement une commande JSON
- vérifier la validité des paramètres saisis
- afficher la réponse reçue par le serveur REST

Le client iOS doit communiquer uniquement avec le serveur REST via ses points d'entrée web, recevoir les réponses au format JSON et les traiter.

3 - Travail réalisé

L'application développée a été réalisée pour les cibles iPhones iOS 7, la dernière version du système. Ce choix se justifie par notre curiosité à vouloir distinguer les différences que cette nouvelle version du SDK[33] proposait par rapport aux versions précédentes. Il en ressort que, pour les fonctionnalités dont l'application avait besoin, seule l'apparence des éléments graphiques avait évolué.

3.1 - Vues d'accueil

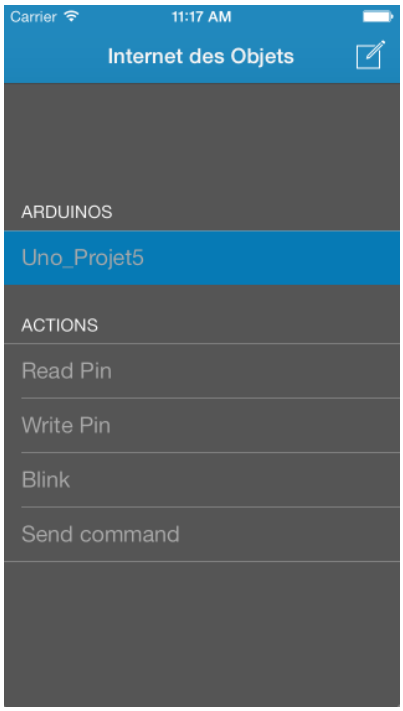


Figure 27 : vue d'accueil

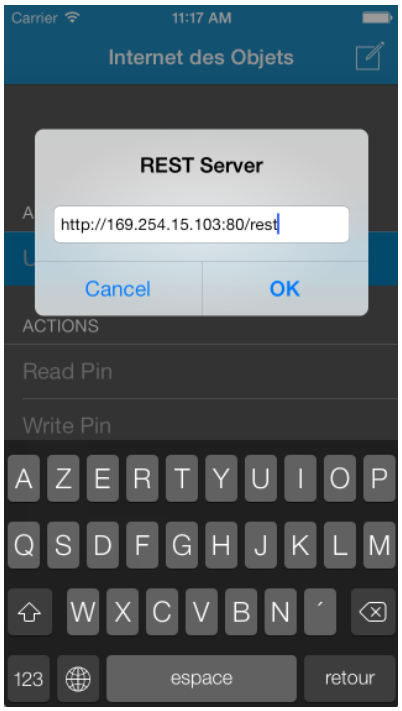


Figure 28 : personnalisation de l'adresse du serveur REST

Au démarrage, l'application utilise l'adresse du serveur REST par défaut (http://localhost:8080/rest) afin de récupérer la liste des arduinos et l'afficher au dessus du menu de commandes. Si le serveur ne répond pas la liste est simplement vide.

Au dessous de cette liste d'arduinos, un menu contenant les différentes actions (read pin, write pin, blink, send command) est proposé à l'utilisateur.

[33] **Software Development Kit** : ensemble d'outils permettant la création d'applications

Si l'utilisateur souhaite modifier l'adresse du serveur REST, il lui suffit d'appuyer sur le bouton en haut à droite, représenté par une icône d'édition, afin qu'une fenêtre de type pop-up se présente lui demandant d'indiquer l'URL du serveur à utiliser.

L'utilisateur a également la possibilité de rafraîchir la liste des arduinos avec la fonction iOS "pull down to refresh". Ce qui signifie qu'un appui sur la liste suivi d'un mouvement vers le bas de l'écran va provoquer une requête de la liste des arduinos au serveur REST dont la finalité sera la mise à jour de la liste présentée à l'utilisateur.

3.2 - Vues de commande

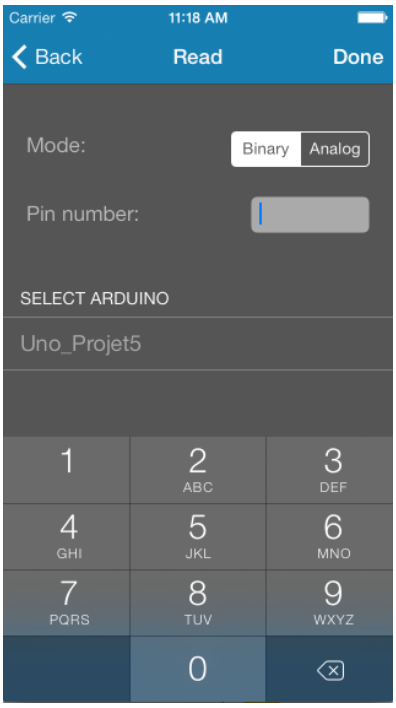


Figure 29 : vue de lecture d'une pin

Figure 30 : vue d'envoi de commande JSON

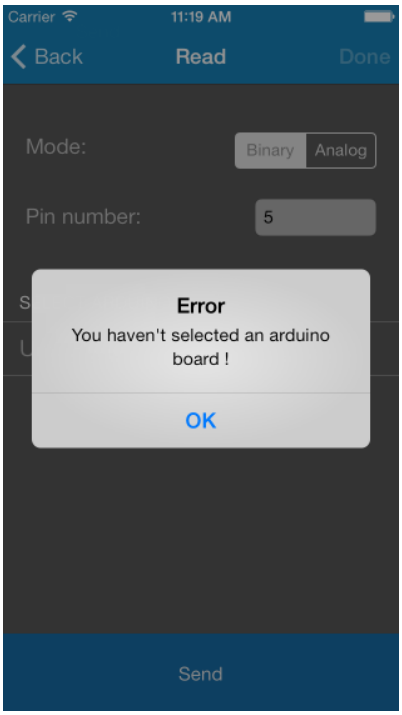


Figure 31 : Message d'erreur lors d'une mauvaise saisie

Les vues d'actions sont toutes sensiblement similaire: l'utilisateur saisi les paramètres de l'action, sélectionne un arduino cible, et appui sur le bouton "Send" présenté en bas de l'écran. Lors de cet appui, l'application va alors vérifier la validité des paramètres, en fonction parfois d'autres paramètres comme par exemple la validité du numéro de pin en fonction de si le mode sélectionné par l'utilisateur est binaire ou analogique. Si les paramètres sont valides, alors une requête asynchrone est envoyée au serveur REST, sinon un message d'erreur de type pop-up est présenté.

Bien que les vues soient similaires, il faut noter que certaines saisies de paramètres sont différentes: lors de la saisie d'un paramètre numérique un clavier ne présentant que des chiffres est présenté, et lors de la saisie d'une commande JSON par exemple c'est un clavier alphanumérique qui est proposé.

La vue propose en haut à gauche de l'application un bouton "back" permettant à l'utilisateur de

retourner sur la vue d'accueil. Ce bouton et le retour à la vue précédente est géré automatiquement par le type de contrôleur utilisé pour créer et commander la vue.

3.3 - Vue de réponse

Lorsque la requête envoyée par une des vues de commande reçoit une réponse, une vue contenant les informations reçues par le serveur REST est affichée à l'utilisateur.

Figure 32 : vue de réponse

Cette vue présente à l'utilisateur les attributs reçus: erreur, etat et id. Le cadre situé en bas de ces éléments est une zone de texte non éditable de type UITextView[34] permettant d'afficher l'attribut "json" reçu. Cette zone permet à l'utilisateur de pouvoir sélectionner le texte présenté et de le copier s'il le souhaite.

Tout comme les vues de commande, cette vue propose un bouton "back" permettant à l'utilisateur de revenir à la vue précédente, en l'occurrence la vue de commande qui l'a amené à la vue de réponse.

3.4 - Modèles et contrôleurs

La programmation pour les plateformes iOS repose sur le design pattern MVC: Modèle, Vue, Contrôleur. Son schéma de fonctionnement est le suivant:

[34] **UITextView** : élément iOS qui représente une zone de texte

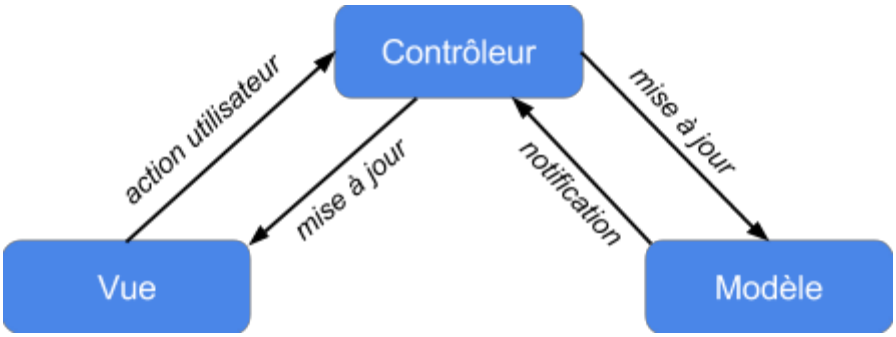


Figure 33 : Schéma MVC de la programmation iOS

Pour afficher une vue à l'utilisateur, c'est tout d'abord le contrôleur de la vue qui est initialisé et exécuté afin de mettre à jour la vue. Le contrôleur contient des modèles, qui sont concrètement des classes et utilisés par le contrôleur comme des objets, qui peuvent être mis à jour par le contrôleur et utilisés par ce dernier afin de mettre à jour la vue. À chaque action de l'utilisateur sur la vue ce cycle est à nouveau réitéré: certaines fonctions du contrôleur sont appelées, qui peuvent mettre à jour le ou les modèles nécessaires, et utiliser ces modèles pour mettre à jour la vue et afficher le résultat de l'action à l'utilisateur.

Dans notre cas nous possédons les 6 vues présentées précédemment: accueil, lecture de pin, écriture de pin, clignotement de pin, envoi de commande, réponse. Chacune de ces vues possède son propre contrôleur. Afin de mieux organiser le code, j'ai donc créé deux classes modèles: Arduino et Response. Ces deux classes sont très simples car elles ne contiennent que des attributs et des constructeurs.

La classe Arduino contient les informations relatives à une carte arduino qui sont envoyées par le serveur REST: id, ip, mac, description, port. La classe Response contient les informations contenues dans la réponse envoyée par le serveur REST: erreur, id, etat, json. C'est par exemple cette classe Response qui est utilisée comme Modèle pour contenir les informations affichées dans la vue de réponse présenté dans la sous-partie précédente.

Les constructeurs de ces classes reçoivent comme paramètre un objet JSON, qui est en fait la trame envoyée par le serveur REST à l'application, afin d'être parsé et validé avant de construire et mettre à jour les propriétés de la classe.

Tous les contrôleurs de commande (read pin, write pin, blink) contiennent les algorithmes suivants :

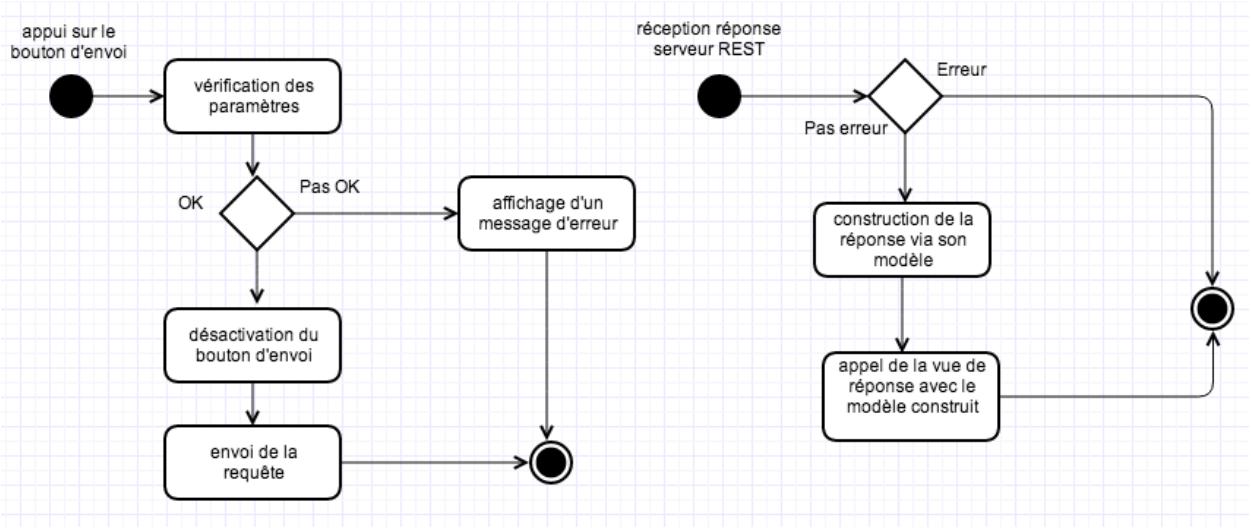


Figure 34 : diagramme d'activité d'un contrôleur de commande

Sur la gauche se trouve le diagramme d'activité de la fonction du contrôleur qui gère le bouton "Send" de la vue, de son appui jusqu'à l'envoi d'une requête au serveur REST.

Sur la droite se trouve le diagramme de la fonction appelée au sein du contrôleur lors de la réception d'une réponse du serveur REST à la requête envoyée via le bouton "Send".

3.5 - Difficultés rencontrées

Le développement de cette application ne fut pas très compliqué, néanmoins plusieurs difficultés sont apparus, nous obligeant à mettre en place des solutions plus ou moins adaptées.

Le principal problème est de faire disparaître le clavier virtuel de iOS une fois qu'il est affiché. La solution préconisée par Apple consiste à déclarer une fonction de délégation associée à chaque champs de saisie et de gérer la disparition du clavier pour chacun de ces champs. Cette solution peut vite devenir difficile à gérer et à mettre en place si le nombre de champs de saisie est important. Même si cela n'est pas vraiment le cas pour cette application, une autre solution plus facile à mettre en place serait appropriée.

La première solution est d'utiliser une fonction qui détecte un appui sur l'écran : à l'appel de cette fonction on vérifie si l'appui a été fait sur un élément de type champ de saisie et si ce n'est pas le cas on fait disparaître l'écran. De cette manière, lorsque l'utilisateur a terminé sa saisie et souhaite faire disparaître le clavier, il doit alors appuyer sur un élément de l'application qui n'est pas un champ de saisie. Cette méthode fonctionne uniquement dans certains cas pour des raisons qui semblent inconnues.

La seconde solution a donc été d'ajouter un bouton "done" en haut à droite de l'application. Ce bouton est désactivé par défaut. Au sein du contrôleur de vue, on peut enregistrer l'évènement d'apparition du clavier et l'associer à une fonction. Ainsi lorsque le clavier apparaît, on peut utiliser une fonction qui va activer le bouton "done" qui, une fois appuyé, va effectuer l'action de faire disparaître le clavier et de redésactiver le bouton.

Le développement de cette application iOS m'a permis de découvrir des fonctionnalités du framework de développement qui m'étaient inconnues :

- la personnalisation d'interfaces visuelles

- l'utilisation de UINavigationController[35] permettant de gérer automatiquement la hyérarchisation et l'enchaînement des vues, générant automatiquement un bouton de retour à la vue précédente
- la fonction "pull to refresh" permettant de "tirer" sur un tableau afin d'effectuer des actions de mise à jour sur un tableau

4 - Axes d'amélioration

L'application réalisée fonctionne, néanmoins étant donné le peu de temps de développement disponible elle pourrait être améliorer.

4.1 - Faire hériter les contrôleurs

Les différentes vues d'actions proposées à l'utilisateur sont très similaires visuellement mais également très similaire au sein du code de leur contrôleur. Si un élément du fonctionnement "interne" de l'application devait à être changer, les 4 contrôleurs de vues devraient être modifiés.

Afin d'éviter cela, il serait intéressant de créer une classe contrôleur "mère" contenant les fonctions récurrentes aux 4 contrôleurs. Ainsi ces derniers pourraient hériter de cette classe, unifiant leur code commun.

4.2 - Préférence permanente

L'URL de base du serveur REST est écrite "en dur" au sein d'un fichier "Global.m" qui est inclus à la compilation dans chaque contrôleur. De ce fait, la modification de cette adresse dans ce fichier se répercutera sur toute l'application, permettant au développeur de modifier cette adresse à son bon vouloir.

L'application propose sur la vue d'accueil un bouton affichant une popup permettant de modifier cette adresse. Cette fenêtre récupère l'adresse que l'utilisateur a saisi et la place dans le contenu de la variable enregistré dans le fichier "Global.m". De cette manière, tout au long de l'utilisation de l'application cette variable utilisée par les contrôleurs contient la nouvelle adresse saisie par l'utilisateur. Le problème de cette méthode est que lorsque l'application est "tuée" par l'utilisateur ou par le système, le contenu de cette variable est réinitialisé à la réouverture de l'application. Autrement dit, la préférence utilisateur n'est pas sauvegardée.

Il serait judicieux de stocker l'adresse saisie par l'utilisateur au sein d'une variable de préférence, facilement fournie par le SDK iOS, afin de pouvoir sauvegarder et réutiliser cette URL même lorsque l'application est tuée.

4.3 - Sélectionner plusieurs arduinos

L'application réalisée permet à l'utilisateur d'envoyer une commande à un seul arduino. Il serait judicieux de permettre à l'utilisateur de sélectionner plusieurs arduinos auxquels la commande serait envoyée.

[35] **UINavigationController** : classe implémentant des contrôleurs de vue

La difficulté de cette amélioration à apporter réside dans la réception et l'affichage des différentes réponses reçues. En effet, lorsqu'un seul arduino est en jeu, une seule réponse est attendue et reçue, mais lorsque plusieurs arduinos sont sélectionnés, alors plusieurs réponses sont attendus. L'enjeu est donc de trouver un affichage des différentes réponses qui reste aussi visuel et explicit pour l'utilisateur que l'affichage d'une seule réponse. De plus, en dehors de l'aspect visuel des réponses, il faut également prendre en compte le fait que les requêtes envoyées au serveur REST sont asynchrones, impliquant que les différentes réponses attendues ne parviendront probablement pas en même temps.

Pour cela, deux choix sont possibles : soit l'application attend toutes les réponses avant de les présenter à l'utilisateur, ce qui semble assez facile, soit l'affichage évolue à chaque arrivée de réponse, ce qui semble plus intéressant pour l'utilisateur mais plus délicat à mettre en place.

4.4 - Gestion des langues

L'application réalisée contient un fichier "Localizable.strings" qui contient toutes les chaînes de caractère utilisée par l'application et présentées à l'utilisateur. L'avantage de centraliser ces messages est de pouvoir facilement modifier une de ces chaînes dans un seul fichier afin de répercuter ces changements sur l'ensemble de l'application, facilitant le travail du développeur.

Le second avantage de ce fichier est également la gestion des langues : il doit exister un fichier "Localizable.strings" par langue et qui doit donc être traduis, ainsi lorsque le terminal de l'utilisateur est dans une langue spécifique, l'application utilisera automatiquement les messages dans la langue appropriée.

Pour le moment, ce fichier contient uniquement les chaînes de caractères en anglais et il faudrait donc le dupliquer et le traduire en français afin que les utilisateurs français puissent obtenir l'affichage de l'application dans leur langue d'origine.

Actuellement ces messages sont utilisés au sein des contrôleurs de vue pour l'affichage de message par programmation, c'est-à-dire pour la majorité des messages affichés par la vue. Or les vues sont construites avec Storyboard[36] ne permettent donc pas l'initialisation automatique de labels avec des messages du fichier "Localizable.strings". Pour palier à ce problème, il faudrait au sein de chaque contrôleur récupérer les labels de la vue gérée et leur attribuer la valeur du message souhaité par programmation.

[36] **StoryBoard** : fonctionnalité de l'IDE XCode facilitant la construction des vues pour iOS et les liens entre elles

Conclusion

Ce projet nous a permis de mettre en pratique des technologies telles que nodeJS, JQuery Mobile ainsi que le développement d'applications native sous iOS. C'était par ailleurs une excellente occasion de travailler selon le paradigme client / server. Nous avons pu appréhender le fonctionnement de la carte embarquée Arduino ainsi que l'interêt d'utiliser cette plateforme de prototypage. La simplicité de mise en oeuvre ainsi que le gain en terme de temps de développement sont une commodité pour notre projet.

Au delà des compétences techniques apprises, ce projet nous a surtout permis de réfléchir sur les problèmes de communication, de traitement et d'affichage auxquels nous étions confrontés. Cette réfléxion nous à permis de mettre en place les solutions techniques répondant aux besoin de l'utilisateur, en correlation avec la définition même du travail d'un ingénieur.

Les difficultés rencontrées nous ont mises à l'epreuve dans un contexte d'entreprise avec une gestion de projet ainsi qu'un suivi des travaux primordial. Cette confrontation à une situation proche de notre secteur d'activité nous prépare à notre arrivée sur le marché du travail.

Bibliographie

Architecture *REST*

- Roy Thomas Fielding, créateur de l'architecture REST
  http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Roger L. Costello, introduction à REST
  http://www.xfront.com/REST-Web-Services.html
- HATEOAS, **Hypermedia as the Engine of Application State**,
- Comparaison : REST vs. WebAPI
  http://blog.restlet.com/2013/05/02/how-much-rest-should-your-web-api-get/
- REST-full API Server
  http://blog.mugunthkumar.com/articles/restful-api-server-doing-it-the-right-way-part-1/

Node.js

Ressources

- Site officiel : http://nodejs.org/
- Documentation : http://nodejs.org/api/
- Dépôt officiel des package utilisables : https://npmjs.org/
- Node.js IRC : irc.freenode.net / #node.js
- "Google developper group" pour node.js :
  https://groups.google.com/forum/#!forum/nodejs

Apprentissage

- "Bulletproof Node.JS coding" (référence de programmation) :
  http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/
- "Mastering Node.JS" : http://visionmedia.github.io/masteringnode/
- "Felix's Node.JS guide" : http://nodeguide.com/
- "Mixu's node book" (très bonne ressource) : http://book.mixu.net/node/index.html
- "The Node.JS beginer book" (payant) : http://www.nodebeginner.org/
- "Node : Up and running" (utilisation d'express, NoSQL,etc) :
  http://chimera.labs.oreilly.com/books/1234000001808/index.html
- How to Node : http://howtonode.org/
- "Node School" : http://nodeschool.io/
- Site du Zero, programmation Node.JS :
  http://fr.openclassrooms.com/informatique/cours/des-applications-ultra-rapides-avec-node-js
- Site du Zero, programmation JavaScript :
  http://fr.openclassrooms.com/informatique/exportPdf/dynamisez-vos-sites-web-avec-javascript
- Gestion des problèmes de CORS ("Cross-Origin Ressources Sharing") :
  http://stackoverflow.com/questions/7067966/how-to-allow-cors-in-express-nodejs

Cas pratiques

- “Bac à sable” permettant le test de code javascript (et autre langages) en ligne: http://jsfiddle.net/
- Introduction à Express.js : http://webapplog.com/intro-to-express-js-parameters-error-handling-and-other-middleware/
- Utilisation des tableaux, objets, fonction et JSON : http://book.mixu.net/node/ch5.html
- Express : utilisation du “datatype” JSON (RESTfull) : http://tech.pro/tutorial/1094/nodejs-and-express-setting-content-type-for-all-responses
- Création d’objet de type collection: http://www.i-programmer.info/programming/javascript/1504-j.html
- Vérification de la présence de propriétés dans un objet : http://www.nczonline.net/blog/2010/07/27/determining-if-an-object-property-exists/
- Blog référençant différents cas pratique avec Node.JS : http://www.atinux.fr/

Déploiement

- Liste des différents services de déploiement d’une application node.js : https://github.com/joyent/node/wiki/Node-Hosting
- Service d’hébergement open-source Nodester : https://github.com/nodester/nodester
  exemple d’utilisation : http://www.youtube.com/watch?v=jwsP1Ejv-_w&feature=youtu.be
- Déploiement sur Heroku : http://code4fun.fr/deployer-application-nodejs-heroku/
- Déploiement sur serveur linux : https://github.com/Nesk/jQueryum-Annihilationem/blob/master/articles/deployer-et-gerer-ses-applications-node.markdown
- Empaqueter son app : http://mesinfos.fing.org/espace-developpeurs/aller-plus-loin/deployer

JSON

- Site officiel, exemple d’utilisation pour différents languages : http://www.json.org/
- Application au javascript (natif) : http://www.json.org/js.html

Arduino

- Exemples et librairies : http://arduino.cc/en/Tutorial/HomePage
- Guide de référence : http://arduino.cc/en/Reference/HomePage
- Playground : http://playground.arduino.cc/
- Librairie aJSON pour Arduino : https://github.com/interactive-matter/aJson
- Aides sur le capteur de température: http://learn.adafruit.com/tmp36-temperature-sensor
- Utilisation & conversion de valeur du capteur TMP36 : http://learn.adafruit.com/tmp36-temperature-sensor/using-a-temp-sensor

jQuery (+ mobile)

Ressources

- Dépôt GitHub : https://github.com/jquery/jquery
- Librairie hébergée sur les serveurs Google (CDN : 'Content Delivery Network") : https://developers.google.com/speed/libraries/devguide?hl=fr-FR&csw=1#jquery
- Code source commenté : http://robflaherty.github.io/jquery-annotated-source/
- Forum : http://forum.jquery.com/
- Support des navigateurs : http://jquery.com/browser-support/
- Documentation de l'API jQuery : http://api.jquery.com/

Apprentissage

- Les bases de jQuery (demos officille) : http://learn.jquery.com/about-jquery/how-jquery-works/
- "Code School" (pour débuter): http://try.jquery.com
- "jQuery4U", différents articles sur le développement de jQuery : http://www.jquery4u.com/popular-articles/
- "Code Schoo"l (fonction avancées) : https://www.codeschool.com/courses/jquery-the-return-flight?utm_medium=null&utm_campaign=jquery&utm_source=tryjquery

Application

- Validation formulaire : http://www.pierrefay.fr/jquery-validate-formulaire-validation-tutoriel-455
- Update Drop-Down list : http://stackoverflow.com/questions/2043572/jquery-update-text-in-a-select-dropdown-list
- "AJAX, 5 ways to do it" gestion des requêtes AJAX avec jQuery : http://net.tutsplus.com/tutorials/javascript-ajax/5-ways-to-make-ajax-calls-with-jquery/
- Gestion de l'évènement "click" sur un bouton : http://stackoverflow.com/questions/5676258/jquerymobile-add-click-event-to-a-button-instead-of-changing-page
- Bibliothèque d'icônes disponible directement sous jQuery (personnalisatin des boutons) : http://demos.jquerymobile.com/1.1.2/docs/buttons/buttons-icons.html
- Accès au DOM avec jQuery : http://api.jquery.com/category/manipulation/
- Utilisation de multiple instances d'un menu déroulant : http://stackoverflow.com/questions/12673077/toggle-only-one-dropdown-from-multiple-with-same-class

iOS

- Ressources de développement : https://developer.apple.com/devcenter/ios/index.action
- Apprentissage développement iOS avec “iOS 6 Programming Cookbook” : http://shop.oreilly.com/product/0636920027683.do
- Conception MVC de iOS : https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html
- Implémenter la fonctionnalité “pull to fresh” sur une TableView : http://www.techrepublic.com/blog/software-engineer/better-code-implement-pull-to-refresh-in-your-ios-apps/
- Utilisation des navigation controller : http://www.appcoda.com/use-storyboards-to-build-navigation-controller-and-table-view/
- Tester son application sans certification développeur Apple : http://www.securitylearn.net/tag/develop-your-own-iphone-app-without-developer-certificate/
- Personnalisation de la barre de status : http://www.appcoda.com/customize-navigation-status-bar-ios-7/

Résumé. Dans le cadre de la 3ème année du cycle ingénieur de l'ISTIA, le projet réalisé est un serveur REST écris avec NodeJS permettant de lier des clients à des cartes arduino à des fins domotiques. Par URL, les clients peuvent envoyer des commandes avec paramètres au serveur afin de les envoyer à la carte souhaitée. Les cartes arduinos sont connectées au serveur via ethernet et en reçoivent les commandes. Pour réaliser cela, les fonctionnalités du serveur ont été découpées en 3 couches : une interface web pour recevoir les commandes des clients, une couche métier pour transformer les paramètres en une chaîne JSON, et une couche de communication permettant d'enregistrer les différentes cartes arduinos connectées et de leur envoyer les commandes utilisant le protocol TCP/IP. Lorsqu'une commande est reçue, la carte analyse la chaîne JSON pour récupérer les paramètres et effectuer l'action souhaitée. La carte répond alors au serveur avec une chaîne JSON qui est reconstruite et renvoyée au client. L'utilisation de la technologie NodeJS et du format JSON permettent au serveur de respecter une architecture REST tout en étant robuste et capable de gérer un grand nombre de requêtes simultanément.

Mots clé: REST, JSON, Arduino, serveur, commande, web, TCP/IP, NodeJS.

Abstract. As part of the 3rd year of engineering studies at ISTIA, the completed industrial project is a REST Server written in NodeJS linking clients and arduino boards for domotic purposes. By URL, clients have to be able to pass commands with parameters to the server in order to send them to the requested board. The arduino boards are connected to the server via ethernet and receive the commands from it. To achieve that, the server's functionalities have been split in 3 layers: a web interface to receive commands from clients, a work layer that transforms the parameters in a JSON string, and a communication layer allowing to register the different arduino boards connected and to send the commands to them using TCP/IP protocol. When receiving a command, the board analyzes the JSON string to get the parameters and performs the action required. Then the arduino board responds to the server with a JSON string which is rebuild and sent back to the web client. Using the NodeJS technology and the JSON format allows the server to respect the Representational State Transfer architecture while being robust and able to handle a large amount of simultaneous requests.

Keywords: REST, JSON, Arduino, server, command, web, TCP/IP, NodeJS.