

Le Web des Objets, de la théorie à la pratique

Implémentation de plusieurs cas d'utilisation, utilisation et création d'outils pour développer des applications RESTful du Web des Objets

TRAVAIL DE BACHELOR

NUMA DE MONTMOLLIN

Août 2014

Supervisé par :

Prof. Dr. Jacques PASQUIER-ROCHA
et
Andreas RUPPEN
Software Engineering Group

Remerciements

Je tiens à remercier tout d'abord M. Andreas Ruppen pour son accompagnement tout au long de ce travail. Il m'a laissé une grande liberté, tout en m'offrant un encadrement bien structuré et des discussions intéressantes. Ainsi que Magali Chatelet pour ses encouragements et sa grande aide pour le dessin 4.2 du rideau de fer. Merci également à mes parents, André de Montmollin et Isabelle Mamie, pour la relecture du travail et leurs conseils avisés.

Résumé

Dans ce travail, trois objets connectés au Web ont été modélisé grâce au méta-modèle défini par M. Andreas Ruppen. De plus, le premier objet modélisé, une porte, a également été réalisé du tout au tout. C'est-à-dire, construction mécanique, électronique et programmation incluant notamment un service Web et une interface client.

Après une base théorique, nous décrivons les caractéristiques de chaque objet et comment nous les avons implémentés. Finalement, un dernier chapitre est consacré aux bibliothèques que nous avons développées dans le cadre de ce travail et à différents développements futurs.

Keywords : Web des objets, objets connectés, méta-modèle, Arduino, Jersey, RESTful Web services

Table des matières

1. Introduction	2
1.1. Motivations et objectifs	2
1.2. Organisation	2
1.3. Notations et conventions	3
2. Bases théoriques, principes et technologies	4
2.1. Internet et Web des Objets (WoT)	4
2.1.1. Internet des Objets	5
2.1.2. Le Web des Objets, un niveau plus élevé	6
2.1.3. Internet versus Web des Objets	7
2.2. REST architecture pour le Web	8
2.2.1. Principes	8
2.2.2. Verbes	9
2.2.3. Transmissions des informations	9
2.3. Interaction entre le monde physique et virtuel	10
2.3.1. Arduino	10
2.3.2. Raspberry Pi	11
2.4. Autres technologies	11
2.4.1. Sunspot	11
2.4.2. Arduino Galileo	12
2.4.3. Composants Arduino sur Raspberry Pi	12
2.5. Méta-modèle	12
2.5.1. Définition du xWoT méta-modèle	13
3. Cas d'utilisation	15
3.1. Rideau de fer	15
3.1.1. Motivations	15
3.1.2. Description du problème	17

3.2. Aquaponie	18
3.2.1. L'aquaponie explications	19
3.2.2. Motivations	19
3.2.3. Description du problème	20
3.3. Machine à caramels	21
3.3.1. Motivations	21
3.3.2. Description du problème	22
4. Implémentation	26
4.1. Jersey, un framework web	26
4.1.1. Implémenter une ressource avec Jersey	27
4.2. Rideau de fer	30
4.2.1. Modélisation	30
4.2.2. Discussion au sujet du XSD	32
4.2.3. Implémentation de l'Arduino	32
4.2.4. Obtenir et modifier l'état de l'Arduino	34
4.2.5. Implémentation d'un publisher	37
4.2.6. Tests	40
4.2.7. Implémentation de l'interface client	43
4.2.8. Construction physique de la porte	48
4.3. Aquaponie	51
4.3.1. Modélisation	51
4.4. Machine à caramels	53
4.4.1. Modélisation	54
4.4.2. Discussion au sujet du XSD	55
4.4.3. Création du projet	56
5. Développements annexes	57
5.1. Bibliothèques, tests unitaires, et gem	57
5.1.1. Protocole de communication	58
5.1.2. ArduinoCommunication	60
5.1.3. ArduinoComponents	65
5.1.4. Simulation de l'Arduino	66
5.1.5. Icwot	71
5.1.6. Archétype Maven	74
5.2. Futurs développements	75
5.2.1. Amélioration de la bibliothèque de tests	76
5.2.2. Implémentation du concept des publisher dans un navigateur web	76
5.2.3. Communication entre l'Arduino et l'application Web	78
5.2.4. Modélisation de composants	78
5.2.5. Langage de gestion des événements d'un publisher	79

6. Conclusion	80
A. Common Acronyms	81
B. License of the Documentation	83
C. CD-Rom	84
C.1. Description rapide	84

Liste des figures

2.1. Un Raspberry PI. Depuis [33]	12
2.2. Modélisation physique d'une porte	14
3.1. Schéma d'un rideau de fer sans les potentiomètres linéaires	16
3.2. Diagramme simplifié des cas d'utilisation de la porte	18
3.3. Schéma du principe de l'aquaponie. Depuis [14]	19
3.4. Schéma du système pour la production de caramels	23
3.5. Diagramme des cas d'utilisation de la machine à caramels	25
4.1. La modélisation physique de la porte sous eclipse	30
4.2. La modélisation virtuelle de la porte sous eclipse	31
4.3. Les deux moteurs et les deux potentiomètre linéaires connectés à l'Arduino grâce à une carte Tinkershield	33
4.4. Requête avec le client HTTP curl sur la ressource <code>GET /door</code> et demandant de retourner la réponse XML	43
4.5. Diagrammes des modèles de l'interface client pour la gestion de rideaux de fer	46
4.6. Notre construction d'un « rideau de fer »	48
4.7. Auquaponie, modélisation de la partie physique sous Eclipse	53
4.8. Machine à caramels, modélisation de la partie physique sous eclipse	54
5.1. Hiérarchie des différents composants ayant été implémentés	58
5.2. Structuration des différentes fonctionnalités de ArduinoCommunication	60
5.3. Hiérarchie des classes pour la communication avec l'Arduino	61
5.4. Principe de simulation de connexions sur un port série	68
5.5. Hiérarchie des classes utilisées pour la simulation	69
5.6. L'application Sunrise Calendar utilisant Chrome	77

Liste des codes sources

2.1. Exemple de code permettant le clignotement d'une Led en Arduino	10
4.1. Code simplifié de l'implémentation du livre d'or	29
4.2. Exemple de clignotement d'une Led avec Jarduino. Cf le listing 2.1 pour l'exemple en Arduino	35
4.3. Comment connaître la valeur d'un composant de l'Arduino en Java grâce à notre bibliothèque	37
4.4. Modification de l'état d'un composant Arduino afin de faire tourner un moteur à vitesse maximale pour déverrouiller le rideau de fer	37
4.5. Retourner tous les clients enregistré pour un certain type d'événement . .	38
4.6. Enregister ou mettre à jour l'inscription d'un client pour un certain type d'événement	39
4.7. Configuration du projet Maven pour des tests d'intégration	40
4.8. Test d'intégration correspondant à l'action GET /door/lock	41
4.9. Test de la transformation de la classe LinearPotentiometer en classe JAXB	42
4.10. Exemple d'une requête AJAX avec Rails	44
4.11. Implémentation de la méthode <code>fetch</code> pour le modèle Open	47
4.12. URIs de notre interface client	47
5.1. Obtenir des objets Java à partir d'informations de l'Arduino	63
5.2. Encoder des objets et les envoyer à l'Arduino	63
5.3. Pseudo-code pour l'envoi de notifications	64
5.4. Pseudo-code permettant de vérifier que le message est bien transmis au port	69
5.5. Factory pour simuler différents états de l'Arduino	70
5.6. Le serveur Sinatra reçoit des requêtes POST à l'URI /	72
5.7. Les différentes options de l'exécutable icwot	73
5.8. Ce à quoi pourraient ressembler les tests d'envoi de notifications.	75
	76

1

Introduction

1.1. Motivations et objectifs	2
1.2. Organisation	2
1.3. Notations et conventions	3

1.1. Motivations et objectifs

L'Internet des objets (IoT) est un domaine en pleine évolution. D'ici à six ans il est prévu que 50 milliard de machines [18] soient connectées, pour la plupart, des objets de la vie de tous les jours.

Le web des objets (WoT) se base sur IoT tout en utilisant les technologies du web classique. Actuellement, les technologies nécessaires au web des objets existent, sont de faibles coûts et faciles à utiliser. Comme par exemple, l'Arduino [15] ou le Raspberry Pi [32], ou également frameworks et serveurs Web existants. Le défi pour les années à venir n'est pas de développer de nouvelles technologies, mais, au contraire, de développer des nouveaux concepts, outils, courants de pensée afin d'utiliser au mieux ces technologies existantes.

La thèse de doctorat, en cours de rédaction, de M. Andreas Ruppen¹ cherche à apporter une réponse à cette problématique. M. A. Ruppen propose un méta-modèle appelé xWoT(eXtended Web Of Things) qui définit une manière de modéliser la connexion d'un objet au Web. Le but de ce travail de Bachelor est d'implémenter des cas concrets afin de prouver par l'usage les concepts et outils développés dans le cadre du xWoT proposé par M. A. Ruppen.

Personnellement, le WoT m'était totalement inconnu avant de commencer ce travail. Cela m'a permis découvrir un nouvel aspect de l'informatique tout en approfondissant mes connaissances dans d'autres domaines tels que le Web et la programmation de micro-contrôleurs.

1.2. Organisation

Chapitre 1 : Introduction

¹Doctorant à l'université de Fribourg - Département d'informatique - Groupe de génie logiciel

L'introduction pose les motivations et les objectifs de ce travail. Elle explique également comment ce travail est organisé.

Chapitre 2 : Bases théoriques, principes et technologies

Ce chapitre, théorique, pose les bases nécessaires à la réalisation de ce travail. De plus, il décrit les technologies utilisées et donne un court aperçu d'autres technologies disponibles.

Chapitre 3 : Cas d'utilisation

Ce chapitre décrit les caractéristiques de chaque objet implémenté, les motivations à implémenter un tel objet et les spécifications de notre implémentation.

Chapitre 4 : Implémentation

Ce chapitre explique comment les trois cas d'utilisation ont été implémentés, y compris la création d'une interface client et l'écriture de tests, comment les différentes technologies sont utilisées et quels ont été les différents problèmes rencontrés.

Chapitre 5 : Développements annexes

Ce dernier chapitre décrit tous les développements nécessaires à l'implémentation des trois cas d'utilisation. De plus, il propose plusieurs pistes pour des développements futurs.

Chapitre 6 : Conclusion

Finalement, ce dernier chapitre clôture ce travail par une conclusion personnelle.

Appendix

Les appendix contiennent la liste des abréviations utilisées, la description du contenu du CD-ROM fourni avec le projet et la licence.

1.3. Notations et conventions

- Le travail est divisé en cinq chapitres, tous divisés en sections, sous-sections, sous-sous-sections et paragraphes.
- Le genre masculin utilisé dans tout le travail pour des raisons de simplifications désigne aussi bien le féminin que le masculin.
- Les exemples de codes et les images sont tous référencés et listés au début du travail.
- Le code est formaté de la manière suivante :

```
1 puts "hello w#{'o'*10}rld"
```

- Ce formatage est utilisé pour indiquer des noms de fichiers, de classes, etc

2

Bases théoriques, principes et technologies

2.1. Internet et Web des Objets (WoT)	4
2.1.1. Internet des Objets	5
2.1.2. Le Web des Objets, un niveau plus élevé	6
2.1.3. Internet versus Web des Objets	7
2.2. REST architecture pour le Web	8
2.2.1. Principes	8
2.2.2. Verbes	9
2.2.3. Transmissions des informations	9
2.3. Interaction entre le monde physique et virtuel	10
2.3.1. Arduino	10
2.3.2. Raspberry Pi	11
2.4. Autres technologies	11
2.4.1. Sunspot	11
2.4.2. Arduino Galileo	12
2.4.3. Composants Arduino sur Raspberry Pi	12
2.5. Méta-modèle	12
2.5.1. Définition du xWoT méta-modèle	13

2.1. Internet et Web des Objets (WoT)

Selon la loi de Moore [30], la puissance des machines informatiques double chaque deux ans. Un autre effet de cette loi est la diminution du prix des composants informatiques. Ainsi, il devient de plus en plus facile d'ajouter des composants informatiques aux objets de tous les jours et de les relier entre eux.

Une manière de relier ces objets entre eux est de les connecter à Internet, leur permettant ainsi de communiquer avec des humains et d'autres machines. Le Web des Objets se base sur Internet, mais avec un niveau d'abstraction plus élevé. L'idée fondatrice du WoT étant d'utiliser les technologies du Web classique pour l'Internet des objets.

Un objet est n'importe quelle partie du monde physique que nous connaissons et avec lequel nous interagissons. Il peut être, une simple affiche publicitaire contenant un code QR, une porte, ou, plus complexe, un patient sur un lit d'hôpital ou un système de culture sous serre.

Dans cette section, nous allons tout d'abord définir ce que sont IoT et WoT, puis nous ferons un court comparatif entre eux, afin de mieux saisir les différences.

2.1.1. Internet des Objets

Tout d'abord, il est important de signaler qu'il n'y a pas de définition standardisée de l'Internet des Objets. Rappelons toutefois ce qu'est Internet. Littéralement, Internet signifie « entre réseau », ou, dans un sens plus large, réseau de réseaux. Ainsi Internet est un réseau regroupant plusieurs réseaux de machines communiquant entre elles grâce à différents protocoles de communication. Le grand succès d'Internet est justement de regrouper plusieurs protocoles de communication non propriétaires, permettant ainsi aux machines de communiquer entre elles. Même à travers d'autres réseaux.

Internet est un réseau composé d'ordinateurs, de câbles et de différents terminaux tous reliés entre eux. Ceci permettant donc d'échanger des informations et à partir de là de créer une réalité différente de la réalité physique. En effet, si nous considérons que le monde qui nous entoure, la réalité physique, est constitué d'informations et que pour une information, un ou plusieurs actes sont possibles, créant de nouvelles informations, le réseau Internet correspond également à cette définition. Des terminaux échangent des informations entre eux et effectuent différents actes en fonction de ces informations. Cependant, ces informations et ces actes existent uniquement à travers un réseau de câbles et de machines, c'est pourquoi nous distinguons ce qui se passe sur Internet de la réalité physique et nous l'appelons réalité virtuelle. Internet des Objets est un cas spécial d'Internet, puisqu'à la place des ordinateurs, il relie des objets du monde physique au monde virtuel. Ainsi, une définition valable pour nous, et dans le cadre de ce travail, est la suivante : Internet des objets est le fait de relier des objets physiques au monde virtuel. Ce faisant, chaque objet est unique et peut communiquer avec d'autres acteurs d'Internet à travers un protocole de communication défini. Dans notre travail, nous connectons des objets au Web, leur ajoutant donc une deuxième réalité. L'objet connecté est présent dans la réalité physique, nous pouvons le voir, le toucher (échange d'informations) et l'utiliser (actes), de même il est présent dans la réalité virtuelle, puisque des informations de l'objet sont visibles à travers le Web (généralement un navigateur Web) et que l'on peut également interagir avec lui.

Il est intéressant de noter que, pour le moment, les protocoles utilisés par Internet des Objets ne sont de loin pas autant unifiés et généralisés que pour Internet. Ceci est une différence importante et va nous amener au Web des Objets dans le prochain chapitre.

Historique

Le concept du IoT est relativement ancien à l'échelle d'Internet. Il est développé à partir de 1991 par Marc Weiser [34] sous le terme d'informatique ubiquitaire (« Ubiquitous computing »). Bien qu'il ne soit fait aucune mention d'Internet ou du Web, les premières bases sont déjà posées. Le terme lui-même est proposé par Kevin Ashton en 1999 [28].

Les premiers balbutiements du IoT se font au début des années 2000, avec l'étiquetage d'objets grâce à des puces RFID qui permettent d'identifier les objets de manière unique et de les tracer. Le premier pas est franchi. Grâce à une adresse unique, une URI (Uniforme Ressource Indicator), chaque objet est ainsi répertorié et accessible. Par la suite, de par le développement croissant des technologies, de plus en plus de possibilités de développements sont données. Localisation, possibilité d'embarquer de véritables systèmes d'exploitation et serveurs dans l'objet, véritable interaction avec l'environnement grâce au développement de capteurs et d'actuateurs performants, miniaturisés et de faible coût.

2.1.2. Le Web des Objets, un niveau plus élevé

L'idée du WoT est d'intégrer des objets au Web, afin de les faire communiquer avec des humains ou d'autres machines. Le Web, ou World Wide Web pour être exact, étant un ensemble de technologies permettant d'interroger des ressources, identifiées par des URI, accessibles sur Internet [37] et [38]. Ces ressources contiennent des informations textes structurées (HTML, JSON, XML) ou binaires (images, vidéos, etc.). Dans bien des cas, le Web est considéré comme un système d'information, composé de ressources liées entre elles par des hyper-liens.

Un point important du WoT est de fournir un protocole de communication commun entre les objets. En effet, l'utilisation du HTTP, le même que pour le Web, permet une unification au niveau de la communication, de l'interrogation de ressources et de la conception. Puisque tous les objets parlent la même langue et répondent donc de la même manière à la même requête.

L'utilisation des technologies Web adaptées aux objets fournit la même facilité d'interaction que pour n'importe quel site Web. Par exemple, l'utilisateur a uniquement besoin d'un navigateur Web, de plus n'importe quel moteur d'indexation, tel que Google, peut y accéder et tous les clients HTTP sont utilisables. Un moteur de recherche pourrait tout à fait permettre la recherche d'objets selon différents critères et la création d'applications clients n'est pas plus compliquée que pour un site classique. L'utilisation d'un protocole différent ou propriétaire ne permettant pas cette facilité d'utilisation, c'est la raison principale d'utiliser le Web des objets.

Le développement d'applications Web pour un objet diffère peu du développement d'une application Web classique, puisque les outils utilisés sont quasiment identiques. C'est là aussi un avantage, le développement d'applications Web est un processus connu et il n'y a pas besoin de réinventer la roue.

Cependant des inconvénients existent également. Notamment une consommation d'énergie accrue pour l'objet et un besoin en ressources informatiques plus important. En effet, la communication par HTTP est plus coûteuse en énergie [25] que certains protocoles spécifiquement développés pour l'Internet des objets. De même qu'un serveur Web peut demander plus de ressources informatiques (mémoire, processeurs, etc.) qu'un objet utilisant un protocole parfaitement adapté à ces besoins.

Le HTTP qu'est-ce ?

Le HTTP est un protocole permettant une communication client-serveur pour des systèmes d'informations hypmédia distribués et collaboratifs [22]. C'est un protocole d'ap-

plication, de haut niveau, utilisant le protocole FTP (File Transfer Protocol) pour le transfert de fichiers.

La façon de communiquer à travers le protocole HTTP est défini de manière précise, à travers la norme RFC 2616 [22], basée sur l'architecture REST. Ceci est détaillé à la section 2.2.1. HTTP suit une logique client-serveur avec un client possédant plusieurs méthodes, ou verbes, pour effectuer différentes actions. Le client peut également passer différents paramètres dans la requête et le serveur peut retourner différentes informations avec la requête. Par exemple, un navigateur Web, le client, effectue une requête HTTP auprès d'un serveur Web. Le serveur Web retournera une ressource HTTP qui peut par exemple être un document HTML, CSS, une image ou autre.

Chaque ressource HTTP est identifiée par une adresse URI (Uniform Resource Indicator) unique. Permettant d'accéder à chaque ressource séparément, mais également de lier une ressource à une autre à travers un lien hypertexte.

La première version (0.9) [23] du HTTP est publiée en 1991, dans le cadre des travaux de l'équipe de Tim Berner Lee afin de mettre en place le Web. Puis la version 1.0 est publiée en 1996 avec la référence RFC 1945 [24]. La version actuelle (1.1) est publiée en juin 1999 avec la référence RFC 2616 [22].

Le HTTP est un protocole populaire de par le fait qu'il est utilisé pour le Web. Mais le HTTP n'est pas inhérent au Web. Il peut théoriquement être utilisé dans d'autres systèmes.

Une spécificité du HTTP est qu'il est simple d'utilisation, rapide et économique en quantité de données envoyées. Les ressources peuvent également être mises en cache, limitant le transfert de données. D'où le choix d'être utilisé pour le Web traditionnel et encore plus pour le Web des Objets.

2.1.3. Internet versus Web des Objets

Vouloir comparer le Web des Objets à l'Internet des Objets ne fait pas de sens. Le niveau d'abstraction du WoT étant plus élevé que celui du IoT, la comparaison devient impossible. Tout comme le Web fait partie d'Internet, avec un niveau d'abstraction plus élevé, le Web des Objets appartient également à l'Internet des Objets. Le WoT est un domaine du IoT, dans lequel la communication se fait à travers le protocole HTTP. Cependant, au vu des développements actuels, certaines personnes tendent à opposer le IoT et le Wot. Dans certains articles et prises de positions, IoT est considéré comme toutes les technologies relatives au monde des objets, exceptées les technologies Web.

Les avantages du Web des Objets sont les faiblesses de l'Internet des Objets, et vice versa. En effet, les avantages du WoT (simplicité d'utilisation, façon de communiquer très normalisée, même langage utilisé entre objets, etc.) peuvent être des désavantages dans certaines situations. Tandis que la grande souplesse du IoT peut aussi être un avantage. Par exemple, un objet fonctionnant sur batterie, ayant peu d'énergie électrique à disposition devra être conçu de manière à économiser cette énergie et utilisant un protocole plus économique que le HTTP.

Tout dépend de l'utilisation que l'on veut en faire, du type d'objet à connecter et le but dans lequel il a été construit. C'est également une affaire de philosophie suivant que l'on veut restreindre l'accès à l'objet en utilisant un protocole propriétaire, ou au contraire, permettre au plus de monde possible d'y accéder.

2.2. REST architecture pour le Web

REST (REpresentational State Transfer) est une architecture pour les systèmes hypermédijs. Au moyen de cinq principes, REST propose une série de contraintes pour les protocoles d'application, dont le HTTP. Donc, le protocole HTTP applique les contraintes REST, mais REST n'est pas dépendant de HTTP. Cependant aucun autre protocole de communication n'utilise REST à ce jour.

Plusieurs définitions de l'architecture REST existent. La première est donnée par Roy Fielding, en 2000, dans sa thèse de doctorat [2]. Par la suite, Bill Burke tente une redéfinition tandis que Erik Wilde essaie de simplifier.

Comme le HTTP est le seul protocole à implémenter l'architecture REST, les sous-sections suivantes traiteront de REST appliquée au Web.

2.2.1. Principes

REST est composé de cinq principes :

1. Client-serveur

Il y a une stricte séparation entre le client et le serveur. Le serveur stocke les données et le client accède ou modifie ces données à travers une interface. Ce qui signifie que le développement du client est indépendant du développement du serveur et que le code du client et du serveur est plus simple.

2. Sans état

Chaque requête effectuée vers le serveur est indépendante des autres requêtes passées ou futures. Cela implique que chaque requête doit contenir toutes les informations nécessaires. C'est au client de conserver ces informations.

3. Mise en cache

Toutes les ressources peuvent être mises en cache du côté du client. Ainsi, chaque ressource doit indiquer, implicitement ou explicitement, si la ressource peut être mise en cache et pour quelle période. Cela permet de réduire potentiellement le nombre de requêtes entre le client et le serveur.

4. Interface uniforme

Ce principe est décomposé en quatre concepts :

- a) Chaque ressource est identifiée par un nom unique (URI)
- b) Chaque ressource peut être manipulée par différentes actions
- c) Le type de contenu de la ressource est nommé explicitement
- d) Les ressources sont du contenu hypermédia reliées entre elles par des liens.

5. Un système hiérarchisé

Chaque ressource peut être divisée en sous-ressources. L'accès à une ressource ne donne pas accès à tout son contenu, mais uniquement à une partie, le reste étant accessible au moyen de liens. De même que le client n'a pas de moyen de savoir sur quel serveur la ressource est localisée. Il peut tout à fait interroger un serveur qui interrogera lui-même un autre serveur et qui renverra la réponse au premier serveur, qui renverra à son tour la réponse au client.

2.2.2. Verbes

Le concept b) du principe quatre de REST à la section 2.2.1 défini que plusieurs actions peuvent s'effectuer sur une ressource. Dans le cas du HTTP, neuf actions sont définies [27], correspondant à neuf verbes, ou méthodes, HTTP. Cela correspond plus ou moins aux opérations que l'on effectue sur une base de données.

Une ressource ne doit pas forcément implémenter toutes les actions, mais au moins une. Dans ce travail, uniquement six actions ont été utilisées :

- GET, Demander au serveur une représentation de la ressource
- HEAD, Demander au serveur uniquement les en-têtes correspondantes à la ressource
- POST, Créer une nouvelle ressource en fonction de données transmises par le client.
- PUT, Modifier ou créer une ressource à partir de données envoyées par le client. La création est uniquement possible si l'on connaît l'URI de la ressource qui sera créée.
- PATCH Permettre la modification partielle d'une ressource
- DELETE Supprimer la ressource spécifiée.

Uniquement POST, PUT et PATCH acceptent que des données soit envoyées dans le corps de la requête.

Les verbes HTTP CONNECT, TRACE, et OPTIONS n'ont pas été décris ci-dessus. Ils n'ont jamais été utilisés dans ce travail.

2.2.3. Transmissions des informations

Le principe deux de REST défini à la section 2.2.1 spécifie que pour chaque requête toutes les informations doivent être transmises. Dans le cas du protocole HTTP, cela est implémenté grâce aux en-têtes HTTP.

Une en-tête HTTP se compose d'une liste de clés-valeurs. Par exemple, pour informer le serveur que l'on désire une réponse en format XML, l'en-tête est la suivante :

¹ `accept: application/xml`

De même que le serveur répond avec certaines en-têtes et un code de réponse afin de donner des informations au client. Par exemple quels types de données le serveur transmet.

Le client reçoit du serveur un code de réponse une fois sa requête exécutée [26]. En fonction du type de code, le client peut savoir si la requête a été exécutée avec succès, a été redirigée vers une autre URI, a été mal formulée ou encore si le serveur a rencontré une erreur interne. HTTP définit cinq catégories de code de réponse :

1. Information : les codes commençant par 1xx sont utilisés pour donner des informations de comment la requête est traitée
2. Succès : les codes commençant par 2xx informent le client que la requête a été effectuée avec succès
3. Redirection : les codes commençant par 3xx indiquent que la requête doit être redirigée vers une autre URI
4. Erreurs de la part du client : les codes commençant par 4xx indiquent que le client a mal formulé sa requête. Le cas le plus connu est l'erreur 404 indiquant que le client cherche à accéder à une ressource dont l'URI n'est pas reconnue par le serveur.

5. Erreurs de la part du serveur : les codes commençant par 5xx indiquent que le serveur a rencontré une erreur et qu'il n'est pas capable de procéder à la requête correctement.

2.3. Interaction entre le monde physique et virtuel

Internet et le Web n'interagissent pour le moment, que sur un monde virtuel, composé de terminaux, de câbles et de serveurs. Alors que nous, êtres humains, nous interagissons avec une réalité physique. Même lorsque nous utilisons Internet ou surfons sur le Web, nous restons et interagissons avec la réalité physique. C'est-à-dire que nous interagissons physiquement avec un terminal qui, lui, fait partie de la réalité virtuelle. Le cas des objets connectés est double. D'une part, comme ce sont des objets, ils font partie de et interagissent avec la réalité physique. D'autre part, comme ils sont connectés à Internet ou au Web, ils font également partie de la réalité virtuelle. Dans cette section, l'intérêt est justement de voir comment faire le lien entre la réalité virtuelle et la réalité physique. Ainsi, cette section détaille avec quels outils l'interaction entre le monde physique et virtuel a été gérée. Nous nous concentrerons en premier lieu sur l'interaction avec la réalité physique puis avec la réalité virtuel.

2.3.1. Arduino

Afin d'interagir avec le monde réel, nous avons besoins de capteurs et d'actuateurs. La nécessité de simplifier au maximum les montages électriques et électroniques nous a conduit vers le choix d'un micro-contrôleur. Un micro-contrôleur est un processeur couplé à de la mémoire et à des entrées/sorties qu'il peut piloter. Ainsi, régler la vitesse d'un moteur en fonction d'une valeur d'un potentiomètre ne nécessite pas un montage électronique compliqué, mais un petit programme. Ce qui est parfait pour un informaticien.

Le choix c'est porté sur l'Arduino[15]. Un micro-contrôleur, possédant un circuit de base sur lequel il est possible d'ajouter d'autres composants électroniques. La carte Tinker-Shield constitue un avantage. En s'ajoutant à l'Arduino, elle facilite grandement le montage de composants qui se branchent simplement grâce à une prise à trois pins. Ainsi, tous les montages électroniques sont évités.

L'Arduino est un produit Open-source, essentiellement construit et développé en Italie. Il est escorté d'une excellente documentation ainsi qu'une grande communauté qui a déjà essayé à peu près tout et n'importe quoi.

La programmation se fait en C++, légèrement modifiée afin de s'adapter aux spécificités d'une architecture de micro-contrôleur. Le langage est particulièrement adapté dans le contexte de la programmation sur micro-contrôleur. Extrêmement facile à utiliser pour coder les fonctions les plus simples, il est également puissant pour développer du code plus complexe. Arduino permet de développer aussi bien des fonctionnalités de très bas niveau que des librairies orientées objet.

Ci-dessous, comment faire clignoter une led en Arduino :

```
1 // Pin 13 est l'emplacement d'une led sur la plus part des arduino
2 int led = 13;
3
4 // la methode setup est appelee a la mise sous tension de l'appareil
```

```

5 // ou quand le bouton reset est presse
6 void setup() {
7     // initialise la pin comme output
8     pinMode(led, OUTPUT);
9 }
10
11 // la methode loop est une boucle infinie
12 void loop() {
13     digitalWrite(led, HIGH); // Allume la led, HIGH est le niveau du voltage
14     delay(1000); // attend une seconde
15     digitalWrite(led, LOW); // eteint la led en descendant le voltage
16     delay(1000);
17 }
```

Listing 2.1 – Exemple de code permettant le clignotement d'une Led en Arduino

2.3.2. Raspberry Pi

Maintenant que nous pouvons interagir avec le monde physique, il ne reste plus qu'à faire le lien avec le monde virtuel. Pour cela, nous avons choisi d'avoir un ordinateur, connecté à Internet, recevant des données de l'Arduino et les rendant accessibles grâce à un serveur Web.

Pour ce faire, nous avons utilisé un ordinateur Raspberry PI. C'est un petit ordinateur de la taille d'un Smartphone, vendu à faible prix. Tout l'intérêt du Raspberry PI est de pouvoir installer une distribution Debian légèrement modifiée, ce qui permet l'exécution de tous les outils disponibles sous distribution Linux (Debian). C'est à dire Java, Maven, Python, Git, etc.

Le processeur est également intéressant, son rapport coût/puissance est bon, par contre, il y a une perte de compatibilité des outils disponibles due à l'architecture Arm.

La connexion entre Arduino et Raspberry PI se fait par un simple câble série USB. Ce câble assure la transmission de données ainsi que l'alimentation en énergie. Cependant, dans notre cas, en raison de l'utilisation de deux moteurs, il est nécessaire de relier l'Arduino à une source d'alimentation externe.

2.4. Autres technologies

Afin de réaliser ce travail, nous avons fait notre choix entre plusieurs technologies. Ci-après, nous présentons d'autres technologies possibles.

2.4.1. Sunspot

Le Sunspot est un produit de Oracle. Programmable en Java, il communique avec d'autres appareils Sunspot « enfants » grâce à un protocole qui lui est propre. Il fait partie du groupe de l'Internet des Objets et ne peut pas avoir de serveur Web embarqué ni utiliser le protocole HTTP. C'est pourquoi nous ne l'avons pas retenu comme solution. Cependant, il offre certains aspects intéressants comme une facilité de communication avec d'autres appareils Sunspot, un environnement pur Java et une plus faible consommation d'énergie réalisée grâce à un protocole minimaliste.



Figure 2.1. – Un Raspberry PI. Depuis [33]

2.4.2. Arduino Galileo

Le processeur Intel Galileo est un processeur 32 bits avec une architecture X_86. De la taille d'une carte SD, il offre autant de possibilités que n'importe quel processeur Intel, avec une puissance moindre. Son grand intérêt vient du fait d'une collaboration entre Arduino et Intel, offrant sur le même appareil, tous les avantages de l'Arduino et d'un ordinateur Intel. Il est parfaitement possible de faire tourner n'importe quel système d'exploitation « Unix-like », ainsi que de lancer des sketches Arduino. Une limitation vient du fait que la communication entre un processus Arduino et un autre est difficile. Il est toujours nécessaire de passer par une simulation de port série. Nous n'avons pas choisi cette technologie, car elle était trop récente au moment de commencer le travail.

2.4.3. Composants Arduino sur Raspberry Pi

Le Raspberry PI supporte le branchement de composants par ports GP/IO. De même que les composants Arduino peuvent être branchés sur ports GP/IO. Ainsi, il existe une librairie écrite en C++ qui permet de contrôler des composants Arduino directement depuis le Raspberry PI. Nous n'avons pas fait ce choix, car la librairie est très limitée et mal documentée, comparée à celle de l'Arduino.

2.5. Méta-modèle

Un méta-modèle est un modèle servant à modéliser des modèles, un modèle étant une représentation d'un système ou d'un sous-système réel.

Un méta-modèle est la troisième couche dans la définition des modèles. La première couche est la réalité physique, c'est-à-dire tout ce qui nous entoure et dont nous pouvons recevoir des informations. Les modèles constituent la deuxième couche, ils servent à représenter de manière abstraite la première couche. Les méta-modèles, la troisième couche, représentent à un plus haut niveau un groupe de modèles. Finalement, comme dernière couche nous avons les méta-méta-modèles qui représentent les méta-modèles. Tout ceci peut être représenté comme une pyramide de quatre étages, dont le premier étage est le monde réel et le quatrième, le méta-méta-modèle.

Un exemple de modèle pourrait être la modélisation technique du rideau de fer d'un magasin. Ce modèle cherche à représenter comment un rideau de fer d'un magasin est perçu dans la réalité. Un méta-modèle est le modèle qui définit comment doit être n'importe quel modèle technique. Un autre exemple est celui des classes en Java. Une classe Java a certaines contraintes et définitions (un nom, des attributs, des méthodes, etc.). Ces contraintes et définitions peuvent être regroupées dans un méta-modèle définissant toutes les classes en Java. De même que chaque classe Java reflète une certaine part d'une certaine réalité. Donc chaque classe est un certain modèle.

2.5.1. Définition du xWoT méta-modèle

Le xWoT est un modèle servant à modéliser des objets connectés au Web, il rentre donc dans la catégorie des méta-modèles. Plus précisément, il définit la façon dont une application Web RESTful et un objet composé de capteurs et d'actuateurs sont modélisés, afin de connecter cet objet au Web.

Le terme Smart-Device défini un objet connecté. Cet objet est composé de deux aspects. L'aspect device qui regroupe des senseurs, des actuateurs, de l'électronique et des algorithmes pour faire fonctionner le tout. L'aspect smart représente les capacités de l'objet à communiquer sur le Web de manière RESTful. C'est là le premier principe du méta-modèle, il divise la modélisation en deux parties. D'une part la modélisation physique de l'objet avec ses capteurs et actuateurs. D'autre part, la modélisation virtuelle de l'objet, qui est de représenter la capacité RESTful de l'objet à communiquer.

À partir de la modélisation physique, une modélisation virtuelle est générée. Puis tout cela est traduit en code Java exécutable, utilisable dans un framework Jersey.

Partie Physique

Dans la partie physique, l'entité ou objet, est composée d'un ou plusieurs devices. Un device peut lui-même être composé d'un ou plusieurs sous-devices, de capteurs ou d'actuateurs. Ainsi la modélisation physique d'un objet se fait du point de vue de ses actuateurs et capteurs, regroupés en devices et sous-devices. Un objet est représenté par un modèle en arbre composé de devices. L'image ci-dessous l'illustre mieux qu'une longue explication. La modélisation de la partie physique a été effectuée sous le logiciel de développement intégré Eclipse¹ grâce à un plugin spécialement développé pour ce méta-modèle.

¹<http://www.eclipse.org>

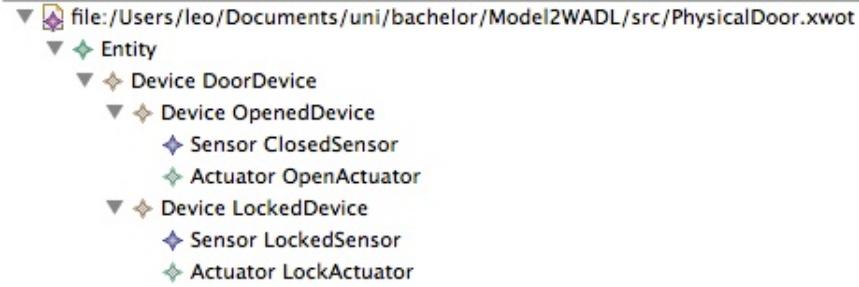


Figure 2.2. – Modélisation physique d'une porte

Partie virtuelle

La partie virtuelle de la modélisation définit trois façons différentes de communiquer pour une ressource Web. Une ressource peut soit répondre à une requête GET, dans ce cas elle transmet des informations au client, soit répondre à une requête PUT, dans ce cas elle est modifiée, ou encore répondre aux deux. Le troisième cas est ce qui est appelé un contexte. Les trois cas décrits ci-dessus correspondent à la modélisation physique de l'entité. Un senseur unique correspond à une ressource répondant exclusivement à une requête GET et un actuateur unique correspond à une ressource répondant exclusivement à une requête PUT.

Le cas de la ressource contexte est plus délicat. Il peut uniquement être appliqué lorsqu'un actuateur et un senseur sont liés de manière intrinsèque. Le senseur mesure directement le changement lié à l'actuateur. Par exemple, l'ouverture/fermeture d'une porte est un contexte, puisque, quand l'actuateur ouvre/ferme la porte, le senseur mesure ce changement.

Un publisher peut être attaché à chaque ressource de contexte ou de capteur. Un publisher permet à un client de s'enregistrer via une requête PUT ou POST, puis d'envoyer aux clients enregistrés une notification via une requête HTTP POST. Le moment où envoyer une notification aux clients enregistrés est différent pour chaque objet.

Tout comme la partie physique permet la modélisation en devices et sous-devices, la partie virtuelle reprend la hiérarchie pour en former des ressources et sous-ressources. Il y a là une correspondance très forte entre devices et ressources. Un device modélisé dans la partie physique, doit correspondre à une ressource de la partie virtuelle.

3

Cas d'utilisation

3.1.	Rideau de fer	15
3.1.1.	Motivations	15
3.1.2.	Description du problème	17
3.2.	Aquaponie	18
3.2.1.	L'aquaponie explications	19
3.2.2.	Motivations	19
3.2.3.	Description du problème	20
3.3.	Machine à caramels	21
3.3.1.	Motivations	21
3.3.2.	Description du problème	22

Cette partie décrit les trois cas d'utilisation que nous avons choisis d'explorer dans le cadre de ce travail. Ils s'appliquent à deux domaines principaux, la domotique, pour le premier et troisième cas, l'agriculture pour le deuxième cas. Ceci permet d'explorer au mieux plusieurs possibilités offertes par le méta-modèle.

3.1. Rideau de fer

Le premier cas choisi est simple de prime abord, il s'agit de relier une porte au Web. Cependant, afin d'implémenter un cas pratique et original, et à cause de contraintes matérielles, un type spécifique de porte a été choisi. Puisque seuls des moteurs à rotation continue étaient disponibles, une porte de type rideau de fer d'un magasin a été implémentée. Nous avons choisi le rideau de fer pour la simple raison que l'idée nous est venue un soir en faisant les courses dans un super-marché à la fermeture. Mais un rideau de fer peut tout aussi bien correspondre à un store d'une fenêtre.

D'aspect simple, ce premier cas peut cependant devenir compliqué si toutes les différentes façons d'utiliser ce rideau, par différentes personnes, sont prises en compte. Ceci sera discuté dans la section suivante.

3.1.1. Motivations

La principale motivation est de créer une porte virtuelle, exposée au Web, représentant le rideau de fer. Ainsi, une nouvelle façon d'interagir avec un rideau de fer apparaît.

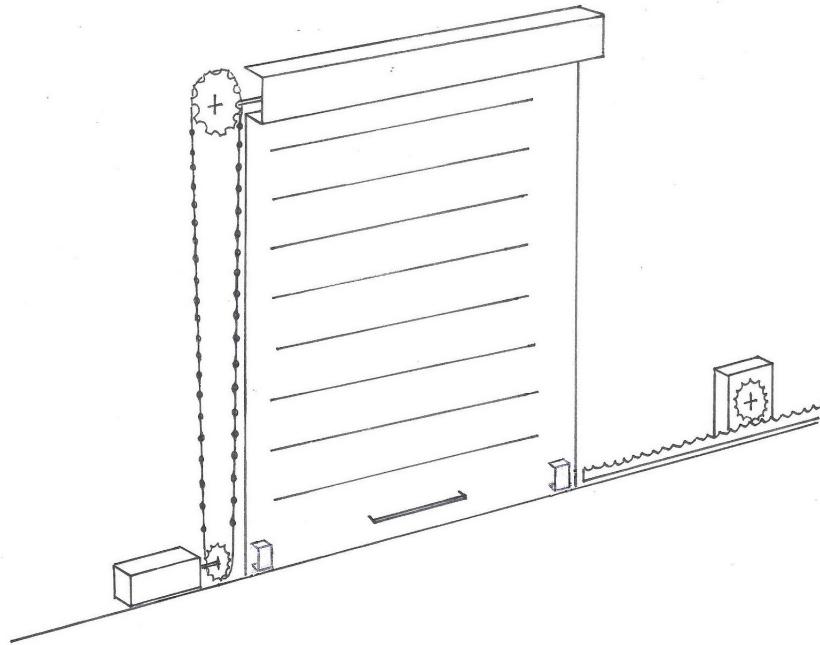


Figure 3.1. – Schéma d'un rideau de fer sans les potentiomètres linéaires

En effet, chaque appareil possédant un navigateur Web peut y accéder depuis n'importe où ; différentes interfaces peuvent être créées et utilisées à volonté pour diriger le même rideau ; il est possible de regrouper tous les rideaux d'un certain type dans une même application, créant ainsi une application composite (Mash-up). Sans compter que le Web nous offre énormément d'autres possibilités. Actuellement, la gestion d'un rideau de fer est basique. Au mieux quelques interfaces peuvent être utilisées pour diriger le même rideau, par exemple, en l'ouvrant manuellement, au moyen de boutons-poussoirs ou avec une télécommande. Bien qu'il existe déjà des moyens de commander à distance un rideau de fer, il est nécessaire de posséder une télécommande dédiée. Mais qui ne s'est pas déjà plaint du nombre de télécommandes envahissantes le quotidien ? De plus les rideaux de fer actuels ont beaucoup de peine à communiquer entre eux ou avec d'autres applications.

Un avantage de transformer le rideau de fer en Smart Device (objet intelligent) est de permettre à une personne, ayant le droit de le faire, de manipuler le rideau depuis n'importe quel Smartphone, tablette ou ordinateur. En effet, la seule contrainte est de posséder un terminal, avec un navigateur web et une connexion Internet. Ce qui, avec le développement technologique des dernières années est le cas de presque tout le monde.

Bien que le rideau soit exposé au Web des Objets, il faut qu'il puisse toujours être manipulé manuellement, en se passant de l'interaction avec le monde virtuel. Ainsi, relier un objet au Web ne crée rien de nouveau, mais au contraire, ajoute une dimension à l'objet, lui permettant de se situer dans deux réalités différentes à la fois. Si bien que, même si le rideau est piloté manuellement, son changement d'état sera toujours retransmis au serveur et donc visible depuis le monde virtuel.

Grâce aux définitions fournies par le xWoT méta-modèle, une application de ce type donne la possibilité de s'enregistrer pour un certain types d'événements. C'est-à-dire :

- ouverture/fermeture
- verrouillage/déverrouillage

Cela permet notamment la création d'alertes si le rideau change d'état alors qu'il ne devrait pas. Par exemple, si quelqu'un relève le rideau pendant la nuit ou si le rideau est baissé pendant l'ouverture du magasin. Cependant, dans notre implémentation, les notifications sont extrêmement basiques. Il est seulement possible de recevoir tous les changements d'état pour un type d'événement.

Finalement, cela offre la possibilité de gérer plusieurs portes d'un magasin. Puisque, chaque rideau possède une adresse IP unique, il est facile de concevoir un site Web regroupant toutes les portes d'un magasin. Nous arrivons ici aux principes des Mash-ups ou applications composites qui sont des applications interrogeant différentes autres applications du même type. Dans le Web actuel, un site de comparateur de prix de billets d'avion est un Mash-up puisqu'il utilise les données de différents autres applications Web afin d'afficher différents prix pour un billet d'avion. Dans le cas du Web des Objets, nous pouvons imaginer un bâtiment dont les stores de chaque fenêtre sont reliés au Web et un Mash-up qui s'occupe de répertorier tous les stores, d'indiquer ceux qui sont ouverts, de permettre la manipulation des stores sous certaines contraintes et même, d'empêcher qu'un trop grand nombre de stores exposés à un fort ensoleillement soient montés en même temps. Permettant ainsi de conserver une température agréable et évitant de recourir à la climatisation.

3.1.2. Description du problème

Un rideau de fer d'un bâtiment est un rideau métallique pouvant se mouvoir, se relever et s'abaisser grâce à un axe rotatif horizontal fixé au sommet. Le rideau peut également être verrouillé/déverrouillé grâce à une barre horizontale empêchant ainsi son ouverture/fermeture.

Le rideau est actionné de manière mécanisée grâce à un moteur à rotation continu, de même que la barre de fermeture. Afin de connaître la position du rideau et son état (verrouillé/déverrouillé), deux potentiomètres linéaires sont utilisés. Toutes les actions effectuées par les moteurs peuvent également être effectuées manuellement, même si le changement d'état du rideau sera toujours transmis au serveur. Le schéma 4.2 donne une idée générale du type de rideau de fer qui a été implémenté et construit.

Chaque porte doit posséder sa propre adresse IP et être contrôlable à travers le protocole HTTP. C'est-à-dire, qu'il est possible de la manipuler depuis un navigateur Web. De plus, le serveur Web doit suivre l'architecture RESTful et retourner les données sous différents formats, XML et JSON. Le serveur doit également être capable d'envoyer des événements aux clients enregistrés. Une interface client Web, utilisant les standards du Web, doit être fournie afin de gérer les différentes portes d'un magasin.

Ci-après, le diagramme 3.2 décrit les différentes actions pouvant être effectuées sur la porte. Pour des questions de lisibilité, nous avons décidé de ne pas illustrer la gestion des notifications et de ne pas différencier la manipulation physique de la manipulation virtuelle. La façon de gérer les notifications différencie peu d'un objet à l'autre, c'est pourquoi nous en discuterons de manière plus détaillée à la section 3.3. La manipulation virtuelle n'a pas été différenciée de la manipulation physique pour deux raisons. Premièrement, les types d'action sont les mêmes, dans les deux cas il est possible d'ouvrir/fermer et verrouiller/déverrouiller la porte, ce qui change la façon de le faire. De même que la façon d'observer l'état de la porte est différentiable s'il s'effectue dans le monde virtuel

ou physique, alors que les résultats de l'observation sont les mêmes. Deuxièmement, différencier la réalité virtuelle de la réalité physique dans le diagramme, l'aurait rendu peu lisible sans pour autant gagner en informations qualitatives.

Il est important de noter l'ordre dans lequel les actions peuvent s'effectuer. En effet, toutes ne sont pas indépendantes les unes des autres. L'action de verrouiller le rideau peut s'effectuer uniquement si le rideau est fermé et l'action d'ouverture peut s'effectuer uniquement si le rideau est déverrouillé. Fermer le rideau reste toujours possible puisqu'il n'est pas interdit de fermer un rideau fermé, ce qui équivaut à ne rien faire. L'ordre dans lequel doivent s'effectuer ces actions n'est pas dû au fait que notre rideau soit un Smart-device, mais est dû à sa conception qui découle de sa fonction même.

Dans notre implémentation, tout le monde peut effectuer toutes les actions, ce qui n'est que très rarement le cas dans la pratique. Cependant, il serait aisé de mettre en place un système de gestion des permissions, afin d'autoriser certaines personnes à effectuer uniquement certaines actions mais cela aurait été hors sujet.

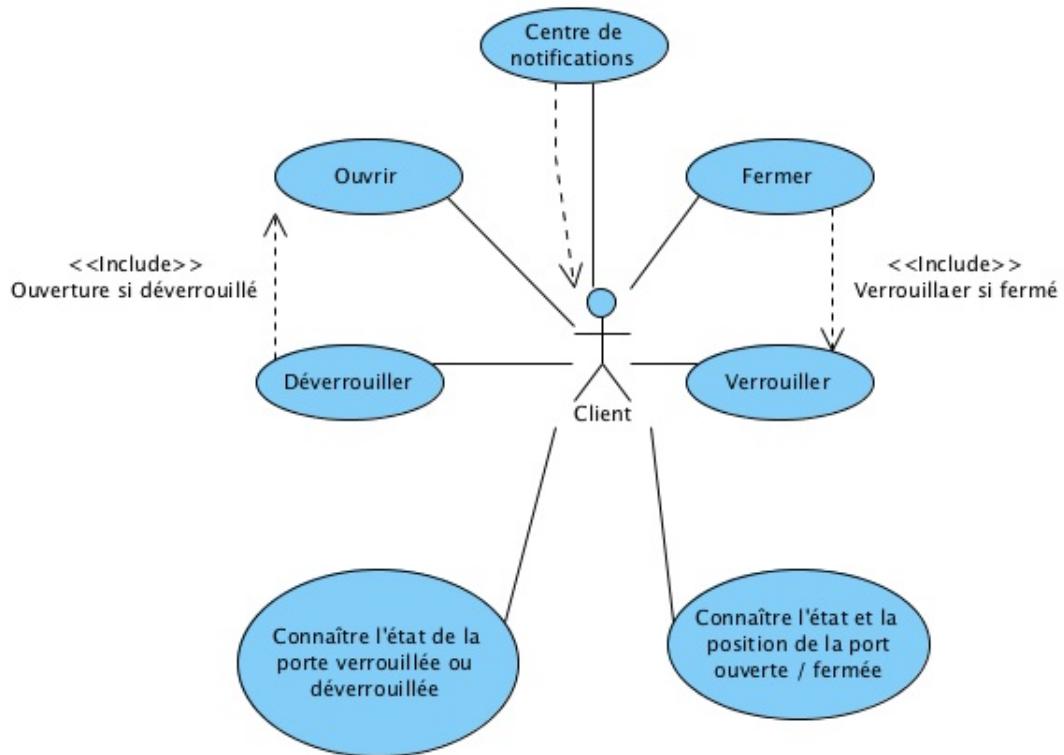


Figure 3.2. – Diagramme simplifié des cas d'utilisation de la porte

3.2. Aquaponie

L'aquaponie est un système de permaculture, permettant de cultiver des poissons et des végétaux dans un espace restreint avec théoriquement une utilisation quasi nulle d'eau et d'énergie, ainsi que sans apport extérieur d'engrais ou de nutriments.

3.2.1. L'aquaponie explications

L'aquaponie se base sur le principe que certaines espèces animales et végétales sont complémentaires les unes des autres et peuvent ainsi fonctionner en cycle fermé, avec un faible apport d'énergie extérieur. Ce système peut être contenu dans un seul container, prenant peu de place et facilement transportable¹. Ainsi, il est tout à fait imaginable que des toits plats d'immeubles ou d'usines accueillent des containers renfermant des systèmes d'aquaponie et fournissant de la nourriture aux occupants. Cette manière de faire s'inscrit dans la tendance de l'Urban Farming, ou maraîchage urbain, qui consiste à recentrer la production agricole dans les villes. Les avantages sont multiples, les ressources sont consommées à proximité du lieu de production, le nombre d'intermédiaires est restreint, tout en développant des espaces verts dans les villes.

Principe

Les plantes sont cultivées de manière hors-sol, en milieu hydroponique. Les poissons sont élevés à proximité des plantes. L'eau, les déchets et les nutriments sont échangés entre les poissons et les plantes. Les uns utilisant les déchets de l'autre comme nutriment, et inversement, le système fonctionne en principe en autarcie.

Afin de garantir une croissance optimale pour les poissons, l'eau doit être maintenue à une température constante. De l'électricité étant nécessaire au fonctionnement de la (des) pompe(s) (échange d'eau entre les plantes et les poissons) et aux différents capteurs, il est usuel d'ajouter des panneaux photovoltaïques, des panneaux thermiques, pour la production d'eau chaude et de récupérer l'eau de pluie.

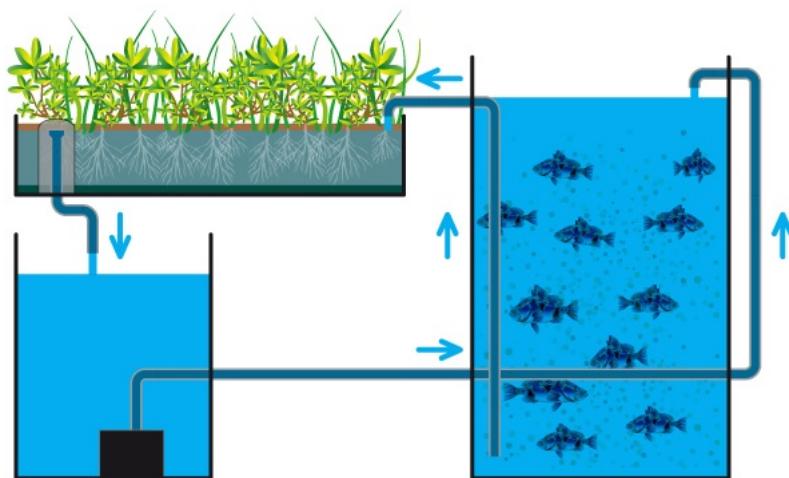


Figure 3.3. – Schéma du principe de l'aquaponie. Depuis [14]

3.2.2. Motivations

L'aquaponie est un système compliqué à gérer. Ainsi, une personne qualifiée pourrait gérer un nombre important de systèmes différents à distance. Grâce à un navigateur web

¹Un projet de la sorte se trouve sur le site de financement participatif fr.ulule.com [13]

et une connexion Internet, n'importe quel utilisateur peut interagir avec le système. Ainsi, le système peut être n'importe où, chez n'importe quelle personne intéressée à y consacrer quelques heures par semaine et être géré par une personne qualifiée qui se trouve également n'importe où dans le monde. Les seules contraintes sont : 1. Suffisamment de place pour un container ; 2. Une connexion Internet ; 3. Un navigateur Web et une connexion Internet possédés par le gestionnaire du système.

Une façon d'utiliser le système serait basée sur le principe de gains mutuels. Une entreprise, un particulier ou une municipalité met à disposition d'un « agriculteur » des terrains inutilisés. C'est-à-dire, toits plats d'immeubles ou d'usines, bordures d'autoroutes, friches industrielles, bâtiments en attente de démolition, etc. En contrepartie, l'« agriculteur » partage la récolte, ou les gains de la récolte avec la personne ayant fourni le terrain nécessaire. Deux types de travail :

1. Maintenir le système et interagir manuellement quand cela n'est pas possible virtuellement. Par exemple, récolter, nettoyer, etc. Ce travail peut être effectué par n'importe quelle personne à proximité des containers, typiquement le concierge du bâtiment sur lequel se trouvent les containers, les employés de la voirie, un étudiant cherchant un travail de quelques heures par semaine, etc.
2. Gérer le système à distance. Cette personne, qualifiée, est capable de gérer un nombre élevé de différents systèmes se trouvant dans différents endroits. Il est même possible d'imaginer une application gérant de manière automatique un nombre élevé d'applications de systèmes d'aquaponie. A ce moment là, la personne effectue un rôle de surveillance et peut s'occuper d'un très grand nombre de systèmes à la fois. Ce qui nous ramène au sujet des Mash-ups.

Une autre façon d'utiliser le système d'aquaponie est de permettre à des personnes citadines, ayant peu de connaissances en agriculture mais habitées par l'esprit de l'Urban Farming de produire légumes et poissons. En effet, une personne possédant un système d'aquaponie peut être assistée, à distance, par une personne experte pour gérer son système. De même que lorsque cette personne doit s'absenter pour une longue période le système peut être géré à distance. Cette façon de faire est peu différente, en quelque sorte, du dépannage et des conseils à distance concernant les problèmes informatiques.

3.2.3. Description du problème

L'échange d'eau et donc de déchets et de nutriments ne se fait pas en continu. En effet, au départ le bac des plantes et à sec, puis la pompe s'enclenche et l'eau arrive dans le bac des poissons depuis le bassin de rétention. Au fur et à mesure que l'eau se déverse dans le bac des poissons, une autre partie de l'eau des poissons est transférée au bac des plantes. Une fois un certain niveau d'eau atteint dans le bac des plantes, l'échange s'arrête. Puis au bout d'un temps donné, l'eau des plantes se vide dans le bassin de rétention et tout recommence. Ainsi, le rythme auquel l'eau est changée permet de contrôler la quantité de nutriments et de déchets dans une certaine quantité d'eau. De même que la vitesse à laquelle l'eau est transférée d'un bassin à l'autre (débit) permet de contrôler à quelle vitesse les nutriments et déchets sont échangés. Ainsi, grâce à ces deux paramètres, il est possible de contrôler la quantité de déchets et de nutriments dans l'eau. Cependant, dans notre cas, nous nous occupons uniquement du taux d'oxygène, bien que d'autres paramètres puissent être pris en compte : taux de nitrates, d'ammonium, de différents sels, etc.

Afin de garantir une température constante de l'eau, nous pouvons à volonté ajouter de l'eau chaude.

Le taux d'oxygène est également mesuré et peut être réglé grâce à une augmentation du débit et de la diminution ou l'augmentation du taux de changement de l'eau.

Sous nos latitudes, pour cultiver des plantes toute l'année, nous devons le faire sous serre. C'est pourquoi dans ce cas, nous devons mesurer la température de l'air et la réguler.

Comme tout fonctionne avec des énergies renouvelables, en plus des paramètres susmentionnés, il est intéressant de connaître :

- La quantité d'eau chaude et sa température
- Le courant électrique produit
- Le courant électrique utilisé
- La charge des batteries
- La quantité d'eau de pluie en réserve

Afin de contrôler la croissance des plantes et des poissons, nous plaçons des caméras aux endroits stratégiques.

Finalement tout le système doit pouvoir tenir dans un seul conteneur. Comme pour le premier cas, chaque système doit avoir sa propre adresse IP et son propre serveur Web, accessibles à travers Internet. La programmation du service Web doit suivre les conventions RESTful et retourner les données sous différents formats. Le système doit également pouvoir envoyer des notifications aux clients à travers le protocole HTTP.

Cette implémentation étant complexe et le sujet mal maîtrisé, dans le cadre de ce travail, nous ne fournissons pas de diagrammes de cas d'utilisation.

3.3. Machine à caramels

Quoi de meilleur que des caramels faits maison ? Malheureusement cela demande l'effort de mélanger du sucre et de la crème brûlante pendant une demi-heure. Fort de ce constat, nous avons imaginé une machine capable de le faire à notre place. Comme nous le verrons ci-après, connecter cette machine au Web décuple les possibilités d'interactions avec l'utilisateur et de fabrication de caramels

Ce cas s'inscrit dans le contexte de la domotique et pourrait faire partie d'un exemple de maison connectée. Maison dans laquelle tous les objets sont reliés entre eux et interrogables à distance.

3.3.1. Motivations

Relier au Web une machine à caramels permet de l'intégrer dans un système plus important encore. Par exemple, une cuisine, puis de là, une maison, composée de dizaines, de centaines ou de milliers d'applications différentes. Ce qui nous amène à nouveau à un Mash-up composé d'applications semblables à celle de la machine à caramels. L'intérêt de créer des Mash-ups, dans le cadre de la domotique, est multiple : pouvoir contrôler depuis une application tous les composants d'une maison, procéder à des recoupements de données, tout en offrant une grande modularité.

Une autre motivation est de pouvoir contrôler la fabrication des caramels à distance depuis n'importe où avec n'importe quel terminal possédant un navigateur Web. Il est facile de lancer le processus de fabrication depuis sa chambre, avant de prendre une douche, ou en rentrant du travail afin de déguster des caramels au moment souhaité. Cependant, le but n'est pas d'automatiser le processus afin d'uniformiser la fabrication de caramels, mais plutôt de permettre à tout un chacun de fabriquer ses propres caramels, en dosant le temps de cuisson, la température et la vitesse de battage. Ainsi, le système permet à n'importe qui de fabriquer ses caramels sans être physiquement présent dans la cuisine ni même dans la maison.

Un troisième point à relever est que grâce au système de notifications décrit dans le xWoT, il est facile de gérer la cuisson et d'attirer l'attention de la personne qui a commandé la fabrication. De plus, comme cet objet respecte les standards du Web, la création d'applications clients en est grandement facilitée. Ainsi, la cuisson de caramels pourrait très rapidement s'apparenter à un jeu pour Smartphone. Nous pouvons aussi imaginer une application pilotant la machine à caramels, depuis le Smartphone, grâce à son API REST. Cette application pourrait tout à fait aller chercher différentes recettes de caramels sur Internet, les proposer à l'utilisateur et piloter la machine à caramels, d'après la recette sélectionnée, afin d'obtenir les caramels souhaités par l'utilisateur. Grâce aux notifications de la machine à caramels, cette application peut savoir à quel moment passer à la suite de la recette.

Imaginons le cas suivant : A sept heures, le réveil du Smartphone sonne. La personne l'attrape, éteint le réveil et en profite pour lancer la fabrication de caramels en sélectionnant une recette parmi d'autres proposées. Puis, cette personne se lève, prend sa douche, prépare ses affaires pour la journée et quand elle arrive dans la cuisine pour déjeuner, des caramels encore tièdes l'attendent.

3.3.2. Description du problème

Le système est composé d'une casserole avec une plaque chauffante et d'un fouet mécanique. Le principe est de battre le mélange de crème et de sucre pendant à peu près une demi-heure jusqu'à ce que le sucre caramélise. Cependant, si la température est trop élevée, le mélange peut facilement déborder, c'est pourquoi, dans notre implémentation, un capteur de contact, situé en-haut de la casserole indique si le mélange risque de déborder. De plus, un moteur à degrés permet de retirer la casserole de la plaque chauffante.

L'utilisateur peut connaître et ajuster la température du mélange afin de mieux maîtriser la cuisson. L'ajustement de la température est possible par deux moyens, l'augmentation ou la diminution de la chaleur de la plaque de cuisson et le fait de pouvoir retirer la casserole de la plaque de cuisson. De plus, la vitesse de battage du fouet peut être connue et réglée, cela permet de gérer, en partie, le fait que mélange ne déborde pas. Finalement, le capteur de lumière situé en bas de la casserole, permet de savoir quand le mélange est prêt. En effet, au départ, le mélange est liquide et recouvre entièrement le capteur, lui cachant la lumière. Quand le mélange se caramélise (passage à l'état solide), il s'agrège en une boule et découvre, du moins partiellement, le capteur de lumière.

Il est possible de recevoir des notifications pour trois changements d'état du système.
1. Quand le mélange va déborder
2. Quand le mélange a atteint l'état solide
3. Quand le mélange atteint une certaine température.

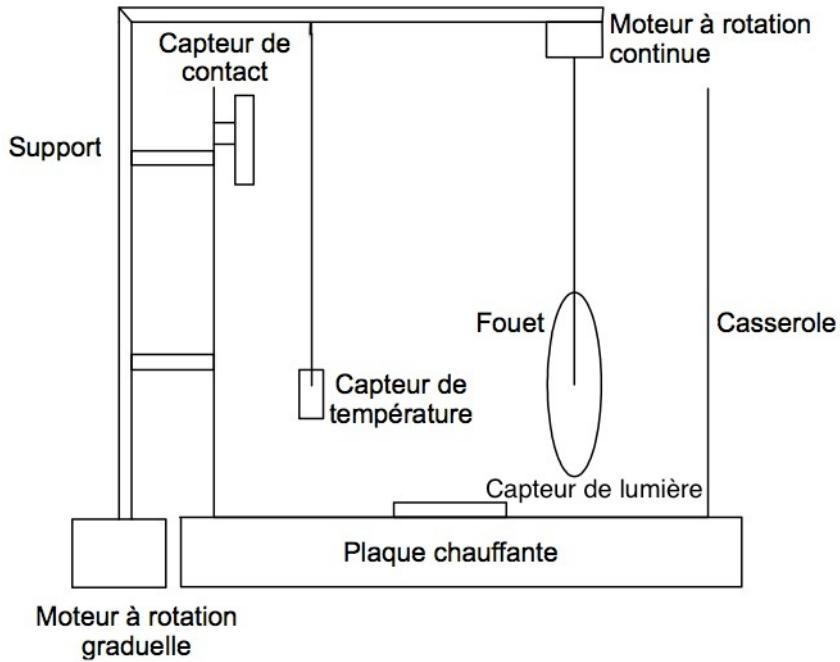


Figure 3.4. – Schéma du système pour la production de caramels

Le diagramme 3.5 décrit les actions que peut effectuer l'utilisateur et la façon de gérer les notifications. Comme indiqué plus haut, il y a trois types de notifications et pour chacune de ces notifications il est possible d'effectuer cinq actions différentes :

- S'enregistrer pour recevoir des notifications
- Connaître les informations de son enregistrement
- Modifier son enregistrement
- Supprimer son enregistrement
- Lister toutes les personnes enregistrées

Puis, le système décide de lui-même, en fonction du type de changements d'état, quand envoyer une notification aux personnes enregistrées. Décider à quel moment envoyer une notification n'est pas toujours trivial, notamment lors de changements continus. Par exemple, la température, est un changement continu qui ne peut pas clairement être défini par la personne développant l'application. À partir de quand considérons-nous qu'il y a un changement ? 1 degré ? 0.1 degré ? 0.01 degré ? Dans d'autres cas, cela devient une évidence, par exemple quand le mélange va déborder, le moment où le mélange déborde est clairement identifiable. Ainsi, dans ce cas-là, nous avons affaire à un changement graduel. Une solution serait de permettre au client de déterminer par lui-même quand recevoir des notifications, mais cela sort malheureusement du cadre de ce travail. Nous nous sommes concentrés sur le comment envoyer une notification plutôt que sur le quand.

Une dernière chose à spécifier est que les notifications sont envoyées grâce au protocole HTTP. Cependant, la manière d'envoyer les notifications doit être bien choisie en fonction des différents changements d'état. Certains requièrent l'envoi de notifications rapprochées dans le temps, d'autres un envoi très rapide de la notification et d'autres encore requièrent que deux notifications soient envoyées dans un grand intervalle de temps entre chacune. Par exemple le débordement de la casserole nécessite que le client reçoive très rapidement cette notification, tandis que la température du mélange, réclame des notifications

envoyées dans un intervalle de temps court. Au contraire, l'envoi de la notification annonçant que les caramels sont prêts est unique et ne survient qu'une fois. Différentes options s'offrent à nous au niveau de l'implémentation :

- Polling
- Long-polling
- Web socket
- Serveur HTTP chez le client prêt à recevoir des notifications par POST(Webhooks)

Comme le système n'a jamais été construit, mais uniquement imaginé, pour les besoins de ce travail, cela ne portera pas à conséquence dans notre implémentation restreinte. Finalement, notons encore que cette façon de gérer les notifications est très générale et convient à la quasi totalité des objets exposés au Web. Notamment pour le cas des notifications du rideau de fer.

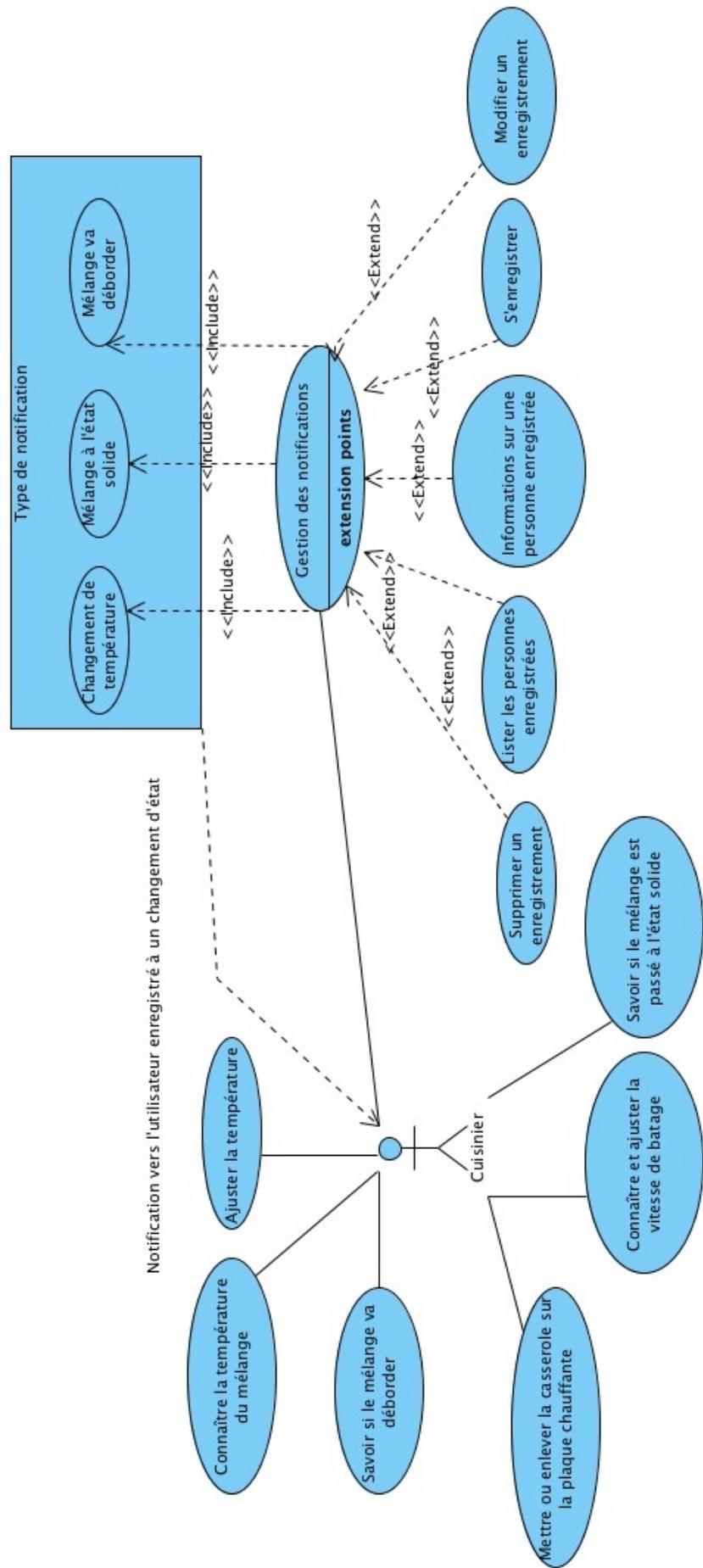


Figure 3.5. – Diagramme des cas d'utilisation de la machine à caramels

4

Implémentation

4.1. Jersey, un framework web	26
4.1.1. Implémenter une ressource avec Jersey	27
4.2. Rideau de fer	30
4.2.1. Modélisation	30
4.2.2. Discussion au sujet du XSD	32
4.2.3. Implémentation de l'Arduino	32
4.2.4. Obtenir et modifier l'état de l'Arduino	34
4.2.5. Implémentation d'un publisher	37
4.2.6. Tests	40
4.2.7. Implémentation de l'interface client	43
4.2.8. Construction physique de la porte	48
4.3. Aquaponie	51
4.3.1. Modélisation	51
4.4. Machine à caramels	53
4.4.1. Modélisation	54
4.4.2. Discussion au sujet du XSD	55
4.4.3. Création du projet	56

Comment implémenter un rideau de fer, une machine à caramels et un système d'aquaponie ? Dans cette partie, vous découvrirez comment nous l'avons réalisé. Pour le rideau de fer, toutes les étapes, de la construction de la porte à l'implémentation du client Web, sont décrites. Mais l'accent a surtout été mis sur l'implémentation de l'Arduino et du serveur Web. Tandis que Les différentes bibliothèques développées dans le cadre de ce travail sont discutées au chapitre 5. Pour l'aquaponie, seule la modélisation a été effectuée grâce au xWoT méta-modèle. Pour la machine à caramels, en plus de la modélisation, la structuration des données a été définie grâce à un fichier XSD et du code exécutable Java a été généré.

4.1. Jersey, un framework web

L'implémentation du service Web a été réalisée grâce au framework Jersey. Il s'agit du framework de référence en Java pour développer des services Web RESTful. Bien que le but de ce travail ne soit pas de connaître Jersey sur le bout des doigts, il est utile d'expliquer quelques principes de base nécessaires à l'implémentation du rideau de fer.

4.1.1. Implémenter une ressource avec Jersey

Jersey est un framework dédié à la création de services Web. Comme nous le verrons ci-après, il facilite grandement la construction d'un service Web qui respecte les principes de l'architecture RESTful. De plus, de par sa structure, il favorise l'utilisation de différents patterns, notamment le pattern MVC(Model View Controller). Le MVC est utilisé dans la plupart des logiciels modernes. Il définit une stricte séparation entre la vue (View), le modèle (Model) et le contrôleur (Controller). La vue est responsable d'encoder, dans un format approprié, les informations fournies par le modèle afin de les transmettre à l'utilisateur. Le modèle est responsable de la façon de structurer, d'enregistrer les données utiles à l'application ainsi que d'y accéder. Finalement, le contrôleur est responsable de gérer les actions de l'utilisateur. À partir de l'action de l'utilisateur, il fait appel à la vue et au modèle approprié.

Le contrôleur

Dans notre façon d'utiliser Jersey, une URI est généralement associée à une classe Java. Cette classe, appelée ressource dans Jersey, fait partie de la section contrôleur du pattern MVC, puisqu'elle répond à différentes actions que l'utilisateur peut effectuer sur cette URI. Cette classe contient au minimum une méthode correspondant à un verbe HTTP lié à l'URI. Cependant, il est très courant d'avoir entre trois et quatre méthodes dans une classe de ressource. Ces trois méthodes correspondent, généralement, aux verbes GET, PUT, DELETE et éventuellement POST. De plus, Jersey met à disposition des annotations définissant la manière d'interroger la ressource. Par exemple, l'annotation `@Path("/le-chemin")` associe l'URI `/le-chemin` à la classe ou méthode correspondante. Notons encore qu'il existe une grande quantité d'annotations possibles pour définir une ressource HTTP avec Jersey. Dans l'exemple qui suit, nous ne parlerons que des plus connues, `@Path`, `@Produces`, `@Consumes`, `@GET`, `@PUT`, etc.

La vue

Chaque ressource est généralement associée à une donnée ou à un groupe de données particulières. Pour représenter les données qui sont envoyées par la ressource au client, ou les données envoyées par le client à la ressource, il est courant d'utiliser une classe JAXB. Cette classe permet de structurer des données et de les convertir en différents formats (XML, JSON, etc). Cette classe JAXB correspond à la partie vue du pattern MVC, puisqu'elle encode des données, dans un format approprié pour l'utilisateur. De plus, suivant la valeur de la clé `accept`, passée dans le header par l'utilisateur, différents formats peuvent être encodés. Par exemple, pour la ressource `GET /livre-d-or/premier` l'utilisateur fournit le header suivant : `accept: application/json`. Avec cette valeur du header, les données sont encodées au format JSON par la classe JAXB correspondante. Tandis que si l'utilisateur fournit le header `accept: application/xml`, les données sont encodées au format XML. Tout ceci est automatisé par Jersey, mais il est toujours possible de définir un nouvel encodeur pour un nouveau format de données. De plus, l'utilisation d'annotations dans les classes JAXB permet de définir la structure des données échangées avec le client. Il est encore intéressant de noter que si ces annotations sont très efficaces pour définir une structure de données au format XML, elles le sont moins pour d'autres formats, le JSON notamment.

Le modèle

Généralement une base de données est utilisée côté serveur afin d'assurer une persistance des données. Avec jersey des classes JPA sont utilisées pour interagir avec cette base de données. Ces classes JPA correspondent à la partie modèle du pattern MVC. Elles permettent d'écrire et d'accéder facilement à des données stockées dans une base de données. Par exemple, pour aller chercher un commentaire dans la table `guest_books`, identifié par son titre, la classe JPA `GuestBook` va effectuer une requête SQL dans la base de donnée de ce type :

```
1 SELECT * FROM guest_books WHERE guest_books.title='premier'
```

Une instance de la classe `GuestBook` est retournée qui contient les valeurs correspondant au commentaire désiré.

Dans le cadre de ce travail, nous n'avons pas utilisé de base de données. Cependant, nous pouvons considérer l'Arduino en lui-même comme l'ensemble des données relatives à l'application puisque nous pouvons obtenir des informations et en envoyer.

Un exemple complet

Comme exemple, imaginons un service Web qui propose comme seule fonctionnalité, la gestion d'un livre d'or. Un livre d'or est composé d'un nombre indéfini de commentaires tous accompagnés d'un titre. Dans notre exemple, chaque titre d'un commentaire doit être unique. Les actions possibles sont :

- entrer un commentaire accompagné d'un titre unique
- modifier un commentaire en fonction de son titre
- trouver un commentaire en fonction de son titre

Ainsi, cette ressource, appelons-la `Guestbook`, comporte une URI avec deux méthodes : 1. GET `/livre-d-or/{title}` 2. PUT `/livre-d-or/{title}` où `{title}` est le titre associé à un commentaire. PUT est ici utilisé pour créer ou mettre à jour un nouveau commentaire. Ainsi, comme spécifié ci-dessus, chaque commentaire possède un titre unique.

Le listing 4.1 montre une façon possible d'implémenter une ressource de type livre d'or. La classe du listing 4.1 utilise une classe JPA, pour aller chercher les informations correspondantes dans une base de données MySQL, c'est la partie modèle du pattern MVC. Afin d'encoder les données, extraites de la base de donnée MySQL, dans le format désiré par l'utilisateur, ici XML ou JSON, une classe JAXB est utilisée. Cette classe JAXB est générée automatiquement grâce à un fichier XSD, qui permet de définir une structure de données.

Il est intéressant de relever l'utilisation de l'annotation `@Produces({"application/xml", "application/json"})` pour définir dans quels formats l'utilisateur peut demander d'afficher les informations. De même que l'annotation `@Consumes({"application/json", "application/xml"})` permet de définir dans quels formats l'utilisateur peut envoyer des données au serveur. Finalement, l'annotation `@Path("guest-book/{title}")`, au-dessus de la définition de la classe, indique que toutes les URI ayant la forme `guest-book/{title}` seront traitées par cette classe. L'annotation `@GET` définit que la méthode `getGuestBook` de l'instance de classe `GuestBookRessource` répond au verbe HTTP GET. De même que l'annotation `@PUT` associe la méthode `putGuestBookResource` au verbe PUT. Ainsi, dans

cette classe, la méthode `getGuestBook`, retourne un commentaire, identifié par son titre, enregistré dans la base de données et la méthode `putGuestBook` ajoute ou modifie un commentaire dans le livre d'or.

Une implémentation possible de la méthode `getGuestBook` est la suivante :

```

1  @Path("/guest-book/{title}")
2  public class GuestBookResource {
3
4      @Produces({"application/xml", "application/json"})
5      @GET
6      public Response getGuestBook(@PathParam("title") String title) {
7          /* do a request to the database
8             * by searching for a comment in function of his title
9             * if the comment required by the user isn't found the
10            * resource return a 404 error
11            * otherwise, datas of the required comment are send to the user
12            * encoded in the desired format (XML or JSON) */
13            return null;
14        }
15
16        @Consumes({"application/xml", "application/json"})
17        public Response putGuestBook(@PathParam("title") String title, GuestBookJaxb
18            jaxbEntity) {
19            /* do a request to the database
20               * to know if a comment, identified by his title, already exist
21               * if it doesn't exist try to save a new comment
22               * return to the user the status code 304 if an error occure durring the save
23               * process of the comment
24               * otherwise return a 204 status code and the URI of the new comment
25               * otherwise if a comment already exist
26               * update the comment
27               * if an error occured during the save process, return the status code 304
28               * otherwise return the updated comment with a status code 200 */
29            return null;
}

```

Listing 4.1 – Code simplifié de l'implémentation du livre d'or

Le code affiché ici a été volontairement simplifié pour des raisons de lisibilité, une version complète, de même que le code de la classe JAXB, JPA et le fichier XSD se trouvent dans l'annexe.

A notre avis, la conversion automatisée est une des fonction les plus puissantes de JAXB. Grâce à des conventions fortes, une classe JAXB peut être convertie vers le format de données JSON ou XML et inversement. Ainsi, la classe `GuestBookJAXB` qui contient un titre et un commentaire est encodée de manière automatique en XML.

Notons encore que, par défaut, l'encodage et le décodage de données avec Jersey se font grâce au framework JAXB. Cependant, il est tout à fait possible d'utiliser un autre framework ou une autre librairie. Par exemple, pour traiter les données au format JSON, la librairie Gson¹, serait tout à fait utilisable.

¹Librairie Open Source de Google pour manipuler le langage JSON, <https://code.google.com/p/google-gson/>

Deux caractéristiques de la méthode putGuestBook sont intéressantes à relever. La première est la notation `@Consumes` qui spécifie sous quels formats les données peuvent être envoyées au serveur. La deuxième est le paramètre de méthode `jaxbEntity`. Ce paramètre correspond aux données envoyées au serveur, par le client, et traduites automatiquement en classe JAXB. Il est de plus intéressant de noter que l'annotation `@PathParam("title")` permet de récupérer le paramètre `title` de l'URI.

4.2. Rideau de fer

Cette partie explique en détail comment l'implémentation du rideau de fer a été réalisée. Cependant, elle n'explique pas comment toutes les dépendances annexes, nécessaires à la réalisation de cette implémentation ont été réalisées. Ceci est discuté dans le chapitre 5.

4.2.1. Modélisation

La première étape pour exposer un objet au WoT consiste à le modéliser grâce au xWoT métamodèle. Cette modélisation est réalisée en deux étapes, tout d'abord la modélisation physique, qui définit les composants de l'objet, puis la modélisation virtuelle, qui définit les ressources accessibles depuis le Web à partir des composants tels que modélisés dans la partie précédente. La modélisation est grandement facilitée grâce à deux plugins Eclipse créés par M. A. Ruppen.

Partie physique

La modélisation de la partie physique du rideau de fer est simple. L'entité (le rideau de fer) est composé d'un device pour ouverture/fermeture et d'un device pour verrouillage/déverrouillage. Les deux devices contiennent eux-mêmes chacun un actuateur de type moteur à rotation continue et un senseur de type potentiomètre linéaire. Les deux devices obtenus, sont eux-mêmes contenus dans un seul device, qui est le device « rideau de fer ». Définir le rideau de fer comme un device donne la possibilité de regrouper tous les devices du type « rideau de fer » afin de créer une application composite (mash-up).

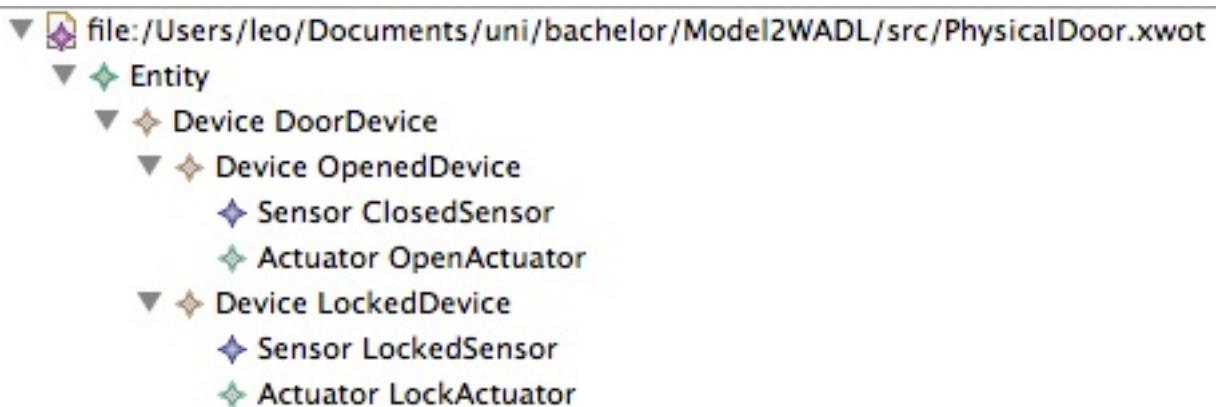


Figure 4.1. – La modélisation physique de la porte sous eclipse

Partie virtuelle

Un script Python est utilisé pour modéliser la partie virtuelle. Ce script génère la partie virtuelle du modèle. Par la suite, seul quelques petits ajustements seront nécessaires.

L'entité possède deux ressources de contexte qui possèdent chacune un publisher. Le premier contexte est celui correspondant à ouvrir/fermer la porte et il s'identifie par l'URI `/door/open`. Le deuxième correspond à verrouiller/déverrouiller la porte et s'identifie par l'URI `/door/lock`.

L'entité possède également une ressource de base identifiée par l'URI `/door` qui liste les autres sous-ressources accessibles, `open` et `lock` dans ce cas.

Une étape importante de la modélisation virtuelle est de choisir si un device, composé de deux sous-devices, doit être modélisé comme un contexte ou comme une ressource composée de deux sous-ressources. Rappelons qu'un contexte est composé de deux devices qui interrogent et modifient la même propriété physique. Dans notre cas, le device « open » est ainsi composé d'un moteur qui permet d'ouvrir/fermer la porte et d'un senseur qui permet de savoir si la porte est ouverte/fermée. Donc, les deux devices sont intrinsèquement liés puisque le moteur va modifier ce qu'observe le potentiomètre. De plus, au niveau contextuel cela fait sens d'imaginer une seule ressource qui permet d'ouvrir la porte et savoir si elle est ouverte. Dans la réalité physique, c'est également le même cas, puisque le fait d'ouvrir la porte à la main nous permet d'observer un changement d'état de la porte.

Le publisher lié au contexte `/door/open` publie les événements liés à l'ouverture/fermeture de la porte. Le publisher lié au contexte `/door/lock` publie les événements de verrouillage/déverrouillage de la porte. Les deux publisher ont les méthodes suivantes :

- GET `/door/[open|lock]/pub` liste tous les clients enregistrés pour recevoir un type d'événements
- PUT `/door/[open|lock]/pub/{url}` permet à un client de s'enregistrer, ou de modifier ses paramètres, afin de recevoir des événements.
- GET `/door/[open|lock]/pub/{url}` permet à un client enregistré de connaître ses paramètres
- DELETE `/door/[open|lock]/pub/{url}` permet à un client de se désenregistrer de l'envoi de notifications.

« [open|lock] » est ici une convention pour indiquer deux URIs différentes, l'URI contenant le mot `open` et l'URI contenant le mot `lock`. « url » est une valeur passée en paramètre de l'URI. Cette valeur indique l'url sur laquelle le client veut recevoir des notifications.

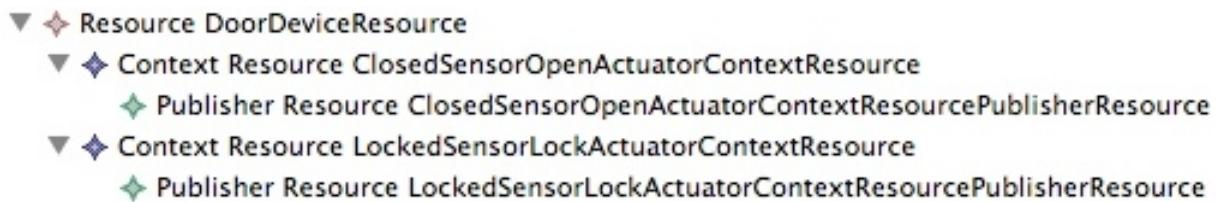


Figure 4.2. – La modélisation virtuelle de la porte sous eclipse

4.2.2. Discussion au sujet du XSD

Le langage XSD est utilisé pour structurer les données échangées avec le client. Concrètement, le fichier XSD est interprété par le plugin Maven `maven-jaxb2-plugin` qui génère une ou plusieurs classes JAXB. Puis, ces classes permettent d'encoder l'information, dans le format JSON ou XML.

Pour le rideau de fer, la structure des données est la suivante : trois éléments principaux sont définis, correspondants aux trois ressources de l'application. Le premier est l'élément « Door », il possède un sous-élément qui liste l'élément « Open » et « Lock ». Ces deux éléments contiennent uniquement un attribut dont la valeur est l'URI à travers laquelle les ressources `open` et `lock` sont accessibles.

Le deuxième élément principal représente la ressource `open`, il contient un attribut indiquant son URI et deux sous-éléments :

- State qui peut avoir quatre états définis. A savoir 1. CLOSING 2. OPENING 3. OPEN 4. CLOSED Ces états correspondent aux différents états d'ouverture/-fermeture de l'entité.
- Position qui est un entier décrivant la position de la porte, où zéro est fermé et cent est ouvert.

Le troisième élément principal représente la ressource `lock`, il contient un attribut indiquant son URI et le sous-élément « State » qui peut avoir deux états définis. A savoir CLOSED et OPEN. L'état OPEN indique que la porte est déverrouillée et l'état CLOSED indique que la porte est verrouillée.

Un dernier point à mentionner sont les « global bindings » qui permettent d'utiliser des classes Java dans le xsd. Ainsi, grâce aux « global bindings » il est possible de modéliser des données qui utiliseront la classe Java URI.

4.2.3. Implémentation de l'Arduino

Comme décrit dans la section 4.2.1, notre rideau de fer se compose de quatre éléments. Deux potentiomètres linéaire et deux moteurs à rotation continue. Après quelques tests, il est très vite apparu que connecter tous ces éléments ensemble, en partant de zéro, requérait de bonnes connaissances en électronique, que nous ne possédons pas. C'est pourquoi notre choix s'est porté sur l'interface TinkerShield qui simplifie grandement le montage de composants électroniques. Grâce à cette interface, il est uniquement nécessaire de brancher les composants sur une prise à trois broches. De plus, une petite bibliothèque est fournie, qui simplifie l'interaction avec les composants.

Communication avec le serveur

Obtenir l'état d'un potentiomètre linéaire, manipuler un servo-moteur ou connaître son état est facile [35]. Par contre, communiquer avec le serveur pose plus de problème. En effet, la communication se fait dans les deux sens, le nombre et le type d'informations changent à chaque fois. De plus, il y a la volonté de réutiliser au maximum le code pour d'autres projets. C'est pourquoi l'idée d'implémenter une façon de communiquer propre au projet a très vite été écartée. Il ne restait plus que deux choix, soit implémenter notre propre protocole de communication binaire, soit utiliser une façon d'encoder les

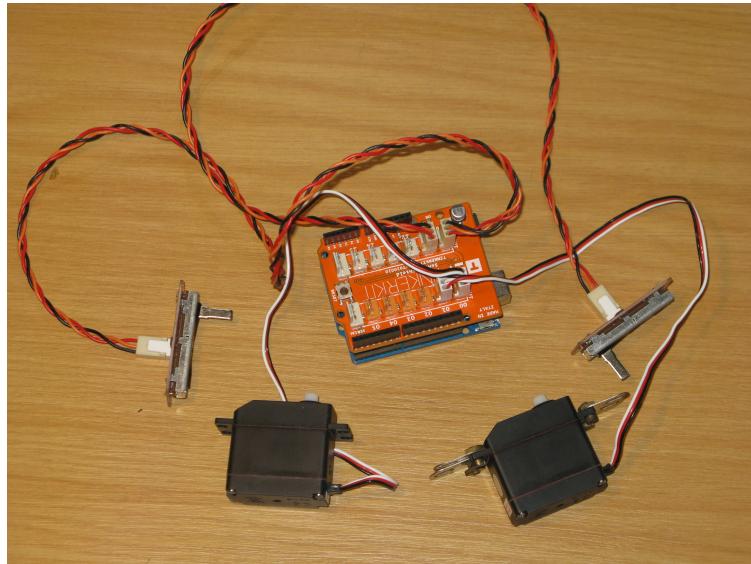


Figure 4.3. – Les deux moteurs et les deux potentiomètre linéaires connectés à l’Arduino grâce à une carte Tinkershield

données déjà existantes. Comme nous ne sommes pas des grands spécialistes du C++ et que nous préférions nous consacrer à d’autres parties du projet, le choix s’est porté sur la bibliothèque aJson[11]. C’est une bibliothèque qui permet d’encoder/décoder des données au format JSON. Très fortement inspirée de la bibliothèque cJSon[12], elle utilise les pointeurs ainsi que les fonctions `malloc()` et `free()` pour manipuler le JSON. Le choix s’est porté sur cette bibliothèque pour trois raisons :

1. Sur la page Web du projet², il y a un nombre de commits et de rapports de bugs relativement importants, ce qui indique que le projet est utilisé et mis à jour.
2. La documentation est claire et bien fournie
3. Elle est déjà utilisée dans d’autres projets avec des retours positifs des utilisateurs.

Malheureusement, une fois le service Web implémenté nous nous sommes aperçus d’un bug important. Après plusieurs ouverture/fermeture ou verrouillage/déverrouillage, généralement deux, du rideau de fer, l’Arduino ne répondait plus et une remise à zéro était nécessaire. Après relecture et debuggage du code, nous avons supposé que le bug venait de l’utilisation de la bibliothèque aJson et était dû à un memory overflow ou à un memory leak. Tout d’abord, nous avons essayé de changer la façon de communiquer en définissant un protocole binaire de communication. Mais vu la relative complexité des données envoyées, la définition et l’implémentation d’un tel protocole n’est pas chose aisée. En effet, un protocole doit tenir compte d’un système de clé-valeur et pouvoir utiliser trois types de données au minimum (entier, nombre à virgule flottante et booléen). Après réflexion, nous avons conclu que le travail à fournir serait trop important pour le résultat à obtenir. C’est à ce moment-là, que nous avons trouvé une bibliothèque, récemment créée (premier commit sur github le 12 janvier 2014), résolvant ce problème et donnant une explication plausible du bug rencontré.

D’après l’auteur de la bibliothèque ArduinoJsonParser[17], le problème est dû à la gestion de la mémoire de l’Arduino[16]. L’appel à la fonction `malloc()` assigne un certain espace

²<https://github.com/interactive-matter/aJson>

de la mémoire disponible, puis, plus tard, un deuxième appel à `malloc()` assigne un autre espace mémoire. Le premier appel à la méthode `free()` libère la mémoire assignée par le premier `malloc()`, laissant à nouveau un espace vide. A nouveau un appel à `malloc()` est fait, la mémoire libérée précédemment sera à nouveau assignée mais pas en totalité, laissant un petit espace non-utilisé. Ainsi, à force d'assignation et de libération de mémoire, cela cause une fragmentation de la mémoire jusqu'à produire un overflow. Bien que nous n'ayons pas trouvé d'autres documentations, il semblerait que l'Arduino ne gère pas le problème de fragmentation de la mémoire, contrairement à des systèmes plus évolués. Pour répondre à cette problématique, l'auteur de la bibliothèque travail uniquement avec des primitives et assigne la mémoire nécessaire à la manipulation du JSON lors de l'initialisation du programme.

Encoder une donnée n'est malheureusement pas possible avec la nouvelle bibliothèque utilisée. Nous avons donc défini une fonction qui permet de le faire simplement. Le principe est simple, chaque clé et valeur passées en paramètre de la fonction sont ajoutées à une chaîne de caractère, avec la syntaxe JSON. Au moment de l'envoi, les crochets extérieurs sont rajoutés.

Grâce à la nouvelle bibliothèque, il est facile de parser une chaîne de caractère JSON et d'en extraire la clé requise. Dans l'utilisation de cette bibliothèque, il est intéressant de noter que la place utilisée pour stocker le JSON parsé est assignée à l'initialisation du programme, évitant ainsi tout risque de fragmentation de la mémoire.

La communication entre l'Arduino et le serveur a été la partie la plus difficile, surtout pour comprendre les problèmes encourus. Une autre partie n'a pas été facile non plus, c'est de gérer à quel moment mettre en marche/arrêter les deux moteurs. En effet, le serveur transmet une valeur indiquant jusqu'à quelle position la porte doit s'ouvrir/fermer, et donc le moteur tourner. A partir de cette valeur et de la position actuelle de la porte, le micro-contrôleur décide quand arrêter le moteur, tout en prenant en compte des marges d'erreurs. De plus, il faut également gérer les cas quand la porte est manipulée à la main et quand l'utilisateur essaie d'ouvrir la porte alors qu'elle est verrouillée. Nous avons choisi d'empêcher le moteur ouverture/fermeture de tourner si la porte est verrouillée afin de ne pas griller les moteurs ou casser la partie mécanique de la porte. Par contre, nous n'avons pas géré le cas de verrouiller quand la porte n'est pas fermée, puisque cela ne porte pas préjudice au rideau de fer et cela découle de la responsabilité de l'utilisateur.

4.2.4. Obtenir et modifier l'état de l'Arduino

Un des premiers défis de cette implémentation était de pouvoir contrôler l'Arduino depuis le serveur. Mais, avant de pouvoir contrôler l'Arduino, il était nécessaire de définir un canal de communication.

Notre choix du canal de communication s'est très vite porté sur le port série, relié par un câble USB. Une autre possibilité aurait été d'utiliser un shield ethernet et de communiquer grâce au protocole TCP/IP. La première solution a été retenue pour sa simplicité d'utilisation et d'implémentation ainsi que pour sa stabilité et sa plus faible consommation de ressources matérielles de l'Arduino.

Une fois le choix du canal de communication défini, il restait à choisir comment implémenter le contrôle de l'Arduino. A ce stade, deux possibilités s'offraient à nous.

La première, utiliser une bibliothèque Java et un sketch Arduino déjà existant. Appelé jArduino[29], ce projet offre l'avantage de pouvoir interagir avec un Arduino depuis du code java. Par exemple, voici comment faire clignoter une LED avec jArduino :

```
1 /**
2 * Licensed under the GNU LESSER GENERAL PUBLIC LICENSE, Version 3, 29 June 2007;
3 * you may not use this file except in compliance with the License.
4 * You may obtain a copy of the License at
5 *
6 *   http://www.gnu.org/licenses/lgpl-3.0.txt
7 *
8 * Unless required by applicable law or agreed to in writing, software
9 * distributed under the License is distributed on an "AS IS" BASIS,
10 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
11 * See the License for the specific language governing permissions and
12 * limitations under the License.
13 *
14 * Authors: Franck Fleurey and Brice Morin
15 * Company: SINTEF IKT, Oslo, Norway
16 * Date: 2011
17 */
18 package org.sintef.jarduino.examples.basic;
19
20 import org.sintef.jarduino.DigitalPin;
21 import org.sintef.jarduino.DigitalState;
22 import org.sintef.jarduino.JArduino;
23 import org.sintef.jarduino.PinMode;
24 import org.sintef.jarduino.comm.Serial4JArduino;
25 /*
26 Blink
27 Turns on an LED on for one second, then off for one second, repeatedly.
28
29 This example code is in the public domain.
30 */
31
32 public class Blink extends JArduino {
33
34     public Blink(String port) {
35         super(port);
36     }
37
38     @Override
39     protected void setup() {
40         // initialize the digital pin as an output.
41         // Pin 13 has an LED connected on most Arduino boards:
42         pinMode(DigitalPin.PIN_12, PinMode.OUTPUT);
43     }
44
45     @Override
46     protected void loop() {
47         // set the LED on
48         digitalWrite(DigitalPin.PIN_12, DigitalState.HIGH);
49         delay(1000); // wait for a second
50         // set the LED off
51         digitalWrite(DigitalPin.PIN_12, DigitalState.LOW);
52         delay(1000); // wait for a second
53     }
54 }
```

```

55  public static void main(String[] args) {
56      String serialPort;
57      if (args.length == 1) {
58          serialPort = args[0];
59      } else {
60          serialPort = Serial4JArduino.selectSerialPort();
61      }
62      JArduino arduino = new Blink(serialPort);
63      arduino.runArduinoProcess();
64  }
65 }
```

Listing 4.2 – Exemple de clignotement d'une Led avec Jarduino. Cf le listing 2.1 pour l'exemple en Arduino

Le désavantage de cette première possibilité, est qu'elle ne permet pas d'utiliser l'Arduino à son plein potentiel. Cela est dû au fait que toutes les librairies Arduino existantes ne sont pas utilisables avec jArduino et que cela aurait demandé un investissement important d'inclure de nouvelles possibilités à la bibliothèque. De plus, jArduino est un projet que nous ne maîtrisons pas et dont l'extension est mal documentée.

La deuxième possibilité était d'implémenter notre propre solution en utilisant la bibliothèque java RxTx. Cette bibliothèque permet l'interaction avec les ports séries en Java. La discussion de l'implémentation de notre bibliothèque se fait à la section 5.1.2. Les avantages de cette solution sont de maîtriser parfaitement le code et d'avoir une bibliothèque totalement adaptée à nos cas d'utilisation actuels et futurs. Les désavantages sont que le code est écrit par une seule personne et n'a pas de maturité, cela demande de plus un investissement en temps non négligeable. Par contre, sur plusieurs projets, cet investissement sera plus faible que l'utilisation de jArduino.

Une chose encore importante à noter est que nous ne demandons pas l'état du système à l'Arduino. Au contraire, c'est l'Arduino qui informe le serveur quand un changement d'état survient, et le serveur stocke cette information. Cette manière de communiquer permet de réduire l'échange de données entre l'Arduino et le serveur, puisqu'il y a un échange uniquement après un changement d'état. Toutefois, ce système est adapté pour les changements d'état discrets, mais plus du tout pour les changements continus. Un changement continu implique soit d'envoyer un flux constant de données, ce qui a pour conséquence un grand échange de données inutiles entre le serveur et l'Arduino, soit que l'Arduino définisse à partir de quelle différence de valeur envoyer des données au serveur, ce qui est une solution beaucoup moins souple. Un utilisateur peut vouloir recevoir une notification avec une différence de 0.1 entre deux valeurs et un autre avec une différence de 0.2. Ce qui n'est pas possible dans ce cas. Cependant, nous proposons une solution à ce problème à la section 5.2.5

Au final, afin de connaître l'état d'un de nos composants, par exemple, un potentiomètre linéaire, un code similaire au listing 4.3 est utilisé. Si uniquement deux lignes de code sont nécessaires pour connaître l'état d'un composant, c'est grâce à la dépendance Maven `ArduinoCommunication` que nous avons développée. La ligne numéro deux initialise une classe dont une des méthodes simplifie l'utilisation de la bibliothèque `JSON`, permettant de lire facilement les données envoyées par l'Arduino. La ligne numéro quatre prend la dernière chaîne de caractères `JSON` envoyée par le serveur, la parse et assigne à la classe `LinearPotentiometer` les valeurs contenues par la chaîne de caractères `JSON`. Tout le code utilisé dans cette exemple a été développé par nos soins.

```

1 // The RxtxUtils class is used to deal with ease with data from arduino
2 RxtxUtils utils = new RxtxUtils();
3 // LinearPotentiometer is filled with the data corresponding of the lock action linear
   potentiometer
4 LinearPotentiometer lp = utils.getComponent(LinearPotentiometer.class,
   ArduinoComponents.LOCK_SENSOR);

```

Listing 4.3 – Comment connaître la valeur d'un composant de l'Arduino en Java grâce à notre bibliothèque

Afin de modifier l'état d'un composant, comme par exemple faire tourner à plein régime un moteur, nous utilisons un code similaire au listing 4.4. A nouveau, tout le code utilisé ici provient de la dépendance ArduinoComunication. La ligne numéro six assigne la valeur de la vitesse maximale à une instance de la classe représentant un composant Arduino en Java. Puis, à la ligne numéro neuf, les valeurs de l'instance précédemment mentionnée sont encodées en JSON et identifiées par le port de l'Arduino auquel est connecté le moteur de déverrouillage du rideau de fer. Finalement, à la ligne numéro onze, les données sont envoyées à l'Arduino, qui, quand il les recevra, fera tourner à plein régime le moteur de déverrouillage de la porte. Si dans un premier temps, nous préparons les données à envoyer puis nous les envoyons, c'est pour la simple raison qu'il est possible de transmettre plusieurs informations à la fois à l'Arduino. Par exemple, il aurait été possible d'également modifier la vitesse du moteur d'ouverture du rideau de fer et d'envoyer toutes les informations en une fois.

```

1 // The RxtxUtils class is used to deal with ease with data from arduino
2 RxtxUtils utils = new RxtxUtils();
3 // The class representing a continuous servo on arduino
4 ContinuousServo cs = new ContinuousServo();
5 // We set the speed to max speed for opening the door
6 cs.setSpeed(ContinuousServo.OPEN_MAX_SPEED);
7 // We add the data of the cs to the datas that will be sent to the arduino
8 // We send to arduino datas for the servo of the lock action
9 utils.addComponent(ArduinoComponents.LOCK_SERVO, cs);
10 // Data are sent to arduino
11 utils.send();

```

Listing 4.4 – Modification de l'état d'un composant Arduino afin de faire tourner un moteur à vitesse maximale pour déverrouiller le rideau de fer

A noter que la classe « LinearPotentiometer » est une classe de la dépendance Maven ArduinoComponents, également développée par nos soins. Cette classe contient deux éléments, un ensemble de variables d'instances privées, correspondantes aux différentes informations envoyées par l'Arduino, ainsi que des méthodes utiles à manipuler les données renvoyées.

4.2.5. Implémentation d'un publisher

Une partie de notre travail, consistait à implémenter un système de Webhooks. C'est-à-dire envoyer des requêtes POST vers un serveur client à chaque changement d'état de l'Arduino. Ceci est appelé un publisher dans notre travail. L'implémentation d'un publisher peut être séparées en deux phases.

Première phase : implémentation des méthodes

Comme décrite dans la section 2.5.1, un publisher possède quatre méthodes. Trois en interaction directe avec le client et une pour lister tous les clients enregistrés pour un certain type d'événement. Afin de retourner une liste de tous les clients enregistrés et sans utiliser une classe JAXB supplémentaire qui alourdirait l'ensemble pour une seule méthode, nous avons développé ce code un peu spécial dans le listing 4.5

```

1  @GET
2  @Produces({"application/xml", "application/json", "text/xml"})
3  public Response getLockClientsResourceXML() {
4      List<Client> clients = new ArrayList<Client>() {
5          };
6      for (Object o : NotificationFactory.getInstance().getLockNotification().
7          getClients()) {
8          clients.add((Client) o);
9      }
10     final GenericEntity<List<Client>> list = new GenericEntity<List<Client>>(clients)
11     {
12     };
13     return Response.ok(list).build();
14 }
```

Listing 4.5 – Retourner tous les clients enregistré pour un certain type d'événement

Nous avons procédé de la sorte, car cela évitait de créer une classe JAXB supplémentaire et simplifiait le code.

Les quatre autres ressources ont deux spécificités intéressantes. La première est que pour ces quatres ressources, l'URL du client est passée en paramètre. Afin de capturer ce paramètre, nous avons défini cette RegEx :

```
1 [a-zA-Z0-9_.:\\\-/]+
```

La deuxième est l'enregistrement et l'accès aux clients d'un publisher. Pour des raisons de rapidité et simplicité, nous ne stockons pas ces clients dans une base de données, mais directement dans la mémoire vive du serveur grâce à une simple liste de clés-valeurs, une classe `HashMap` en Java. Le désavantage est bien évidemment que si le serveur est arrêté, les clients enregistrés sont perdus. Une solution adaptée à nos besoins serait d'utiliser une base de données Redis. Redis est une base de données stockant les informations selon un système de clés-valeurs. Plus exactement, les données sont stockées au format JSON. Une spécificité de Redis est que les données sont stockées dans la mémoire vive et uniquement sauvegardées sur le disque en cas de besoin. Ainsi, ce système serait bien adapté à nos besoins et ne requerrait que peu de changements dans notre code. Afin d'accéder aux informations d'un client spécifique, identifié par son URL, nous utilisons ce code :

```
1 NotificationFactory.getInstance().getLockNotification().getClient(uri);
```

Le listing 4.6 montre comment nous avons procédé afin d'enregistrer ou de mettre à jour l'inscription d'un client pour un certain événement. Les lignes une à cinq n'ont rien de particulier et sont semblables à n'importe quelle méthode d'une classe de ressource Jersey. Par contre, à la ligne numéro sept, nous utilisons une classe implementant le pattern `Singleton` et `Factory` créée par nos soins. Une des méthodes de cette classe (`getLockNotification`) permet de retourner toujours la même instance de la classe `LockNotification` s'occupant d'enregistrer les clients pour les événements de type `lock`.

C'est-à-dire quand le rideau de fer est verrouillé/déverrouillé. La classe `LockNotification` utilise un `HashMap` pour stocker tous les clients enregistrés, identifiés par leur URL, d'où la méthode `addClient()` dont le premier paramètre est la clé et le deuxième la valeur. Procéder de la sorte, et grâce au principe de la classe `HashMap`, comporte l'avantage que la méthode utilisée est la même pour modifier ou enregistrer une inscription. Finalement, la ligne numéro neuf renvoie au client les données qu'il nous a lui-même fournies. Nous procédons de la sorte puisque cela permet au client de valider si son inscription s'est effectuée correctement.

```

1 @PUT
2 @Path(uriPattern)
3 @Consumes({"application/xml", "application/json"})
4 @Produces({"application/xml", "application/json", "text/xml"})
5 public Response putLockClientResourceXML(Client client, @PathParam("uri") String uri)
6 {
7     client.setUri(uri);
8     NotificationFactory.getInstance().getLockNotification().addClient(client
9         .getUri(), client);
10    return Response.ok(client).build();
11 }
```

Listing 4.6 – Enregistrer ou mettre à jour l'inscription d'un client pour un certain type d'événement

A noter, que nous utilisons le pattern Factory associé au pattern Singleton avec eager loading afin d'avoir accès tout au long de l'exécution aux mêmes informations dans tous les publishers. Nous utilisons le eager loading, car, après une phase de debuggage particulièrement longue, il est apparu que le lazy loading posait problème au niveau du multi-threading.

Deuxième phase : Envoi de requêtes POST (notifications) au serveur client.

L'envoi de notifications vers les clients est séparé en deux parties distinctes. Premièrement L'envoi de la notification grâce à un client HTTP. Deuxièmement, décider à quel moment envoyer une notification.

L'envoi de la notification est fait dans une dépendance Maven créée par nos soins et discutée à la section 5.1.2. Déterminer quand l'envoi d'une notification est nécessaire est implémenté partiellement dans la dépendance citée précédemment et doit être complété pour chaque publisher. Afin de faciliter l'implémentation pour chaque publisher, la dépendance utilise le pattern builder. Ainsi, il ne reste plus qu'à créer un sujet pour chaque publisher.

Un sujet est composé de deux méthodes redéfinies. La première détermine quand envoyer la notification. Tandis que la deuxième, « `isEqualTo` », est implémentée dans la classe JAXB de Lock. Elle permet de comparer l'état d'un composant Arduino à un autre et de déterminer s'ils sont équivalents ou non.

Cette deuxième méthode permet en outre de déterminer comment et sous quel format les données doivent être envoyées au client. Dans notre cas, le code est trivial, mais cette méthode a été créée afin de prendre en compte le fait que chaque client peut avoir une manière bien à lui de vouloir recevoir les données.

4.2.6. Tests

Nous nous sommes rapidement aperçus que, sans écrire des tests, la programmation d'une application pour le web des objets n'est pas évidente. Devoir allumer et éteindre l'Arduino, relancer le serveur et tester différentes URIs pour un changement minime est long et fastidieux. De plus, cela nécessite en premier lieu de coder pour la partie Arduino, puis de coder la partie serveur.

Tests d'intégration

Dans notre implémentation, l'interaction avec l'Arduino se fait par le biais des méthodes d'une ressource Jersey et de la gestion des notifications. Ainsi, pour tester une ressource, et de là l'interaction avec l'Arduino, nous écrivons des tests d'intégration. Par exemple, nous voulons vérifier que quand l'Arduino indique que la porte est déverrouillée, le serveur répond au client que l'état de la porte est `OPEN`. Les tests d'intégration signifient que nous lançons le serveur, nous simulons l'envoi de données par l'Arduino vers le serveur et que nous effectuons une requête HTTP vers le serveur. Si la requête renvoie le code de réponse attendu et le bon contenu de réponse, le test est réussi.

Afin de tester au mieux une méthode d'une ressource Jersey, nous avons besoin de simuler un Arduino connecté au serveur par port série. Pour ce faire, nous avons développé un ensemble de classes permettant de simuler une connexion par port série et de là, un Arduino. Ces classes sont intégrées à notre dépendance Maven `ArduinoCommunication` responsable de la communication avec l'Arduino. L'implémentation et l'organisation exacte en sont discutées à la section 5.1.4.

La configuration par défaut des tests d'intégration se situe dans le fichier `pom.xml`, qui est utilisé pour configurer tout projet Maven. Cependant, afin d'inclure la simulation de l'Arduino dans les tests d'intégration, nous avons ajouté les lignes suivantes dans l'élément `configuration` de la phase `pre-integration-test` de la configuration des tests unitaires :

```

1 <configuration>
2   <scanIntervalSeconds>0</scanIntervalSeconds>
3   <daemon>true</daemon>
4   <systemProperties>
5     <systemProperty>
6       <name>diuf.uniffr.xwot.start-serial-simulation</name>
7       <value>true</value>
8     </systemProperty>
9   </systemProperties>
10 </configuration>
```

Listing 4.7 – Configuration du projet Maven pour des tests d'intégration

De ce fait, les lignes ajoutées permettent de créer une propriété système. La ligne numéro six définit le nom de la propriété système et la ligne numéro sept sa valeur. Cette propriété est utilisée par la méthode `onLoaded` de la classe `Listener` du paquet `monitoring.listener`, appelée en fin d'initialisation de l'application. Si cette propriété système est égale à la chaîne de caractères `true`, la connexion par port série est simulée et un Thread est créé afin de stopper la simulation de la connexion lors de l'arrêt de l'application. Si la propriété système a une autre valeur que `true`, le serveur essaie de se connecter normalement à l'Arduino.

Toute classe dont le nom fini par `ITCase` est considérée comme une classe de test d'intégration. Afin de lancer les tests d'intégration, la commande `maven verify` est utilisée. Cette commande démarre un serveur Jetty, effectue les différents tests, puis stoppe le serveur jetty et génère un rapport de test.

Exemple d'un test d'intégration

Le listing 4.8 représente le test d'intégration de la méthode `getLockContextResourceXML` de la classe `LockContextResource`, qui gère l'action GET `/door/lock` du rideau de fer :

```

1 @Test
2 public void testGetLockContextResourceXML() {
3     // We get the factory used to simulate a certain state of the arduino
4     JaxbTestFactory fac = new JaxbTestFactory(ConnectionSimulator.getInstance().
5         getHardwareSpeaker());
6     // We simulate that the door is unlocked and the state of the arduino is
7     // transmitted to the server
8     fac.createLockOpen();
9     // HTTP request on the URI /door/lock
10    WebResource ws = getClient("/lock");
11    // set that we wannt to have the response in the XML format and do the request
12    String result = ws.accept(MediaType.APPLICATION_XML).get(String.class);
13    // we test if something is returned i.e: status code = 200 with a response
14    assertNotNull("response should not be null", result);
15    //private method to test if the server's response is correct
16    assertXmlResponse(result);
17 }
```

Listing 4.8 – Test d'intégration correspondant à l'action GET `/door/lock`

Il est intéressant de noter la grande simplicité de notre test. La simulation d'un état de l'Arduino et la transmission de cet état vers le serveur est réalisé en deux lignes. La ligne numéro quatre initialise une classe, créée spécialement pour cette application, qui contient plusieurs méthodes qui simulent différents états de l'Arduino. La ligne numéro six simule le fait que l'Arduino envoie un nouvel état au serveur. Cet état signifie que la porte est déverrouillée. Ainsi, le serveur reçoit la chaîne de caractère JSON suivante :

```
1 {14: {"position": 0, "old_position": 0}}
```

Puis, quand la requête HTTP est effectuée, la chaîne de caractère reçue sera interprétée afin de retourner la réponse appropriée au client. Le reste du code est un code semblable à n'importe quel test d'intégration. Les lignes numéros huit et dix effectuent la requête HTTP suivante :

```
1 accept: application/xml
2 GET /door/lock
```

Finalement, les lignes numéros douze et quatorze testent si la réponse du serveur est réussie et que le XML renvoyé correspond bien à nos attentes. Afin d'effectuer une requête vers le serveur, nous utilisons la dépendance Maven `jersey-client` qui est un client HTTP adapté pour communiquer avec un service Web Jersey. Dans notre cas, l'utilisation du client HTTP est toujours la même, c'est pourquoi nous avons défini la méthode `getClient` dans la classe `TestHelper`. Ainsi, chaque classe de tests d'intégration peut étendre de la classe `TestHelper` et simplifier le processus pour interroger le serveur. Il en est de même

pour tester la réponse du serveur et vérifier si le XML renvoyé correspond à ce que nous attendons. Pour vérifier le XML, nous étendons la méthode `assertXMLResponse` de la classe `TestHelper` dans chaque classe de tests d'intégration, permettant ainsi de nous adapter à tous les types de réponses.

Nous utilisons le pattern factory pour définir différents états de l'Arduino réutilisables à volonté. Pour ce faire, nous nous sommes inspirés de la bibliothèque Ruby `factory_girl` [19] qui permet de créer facilement des données pour les tests à partir d'instances de classes. Nous avons procédé de la sorte, car cela permet d'écrire des tests de manière tout à fait lisible et favorisant la réutilisation de code. Le principe est que pour chaque application nous définissons une ou plusieurs classes qui étendent une classe de base facilitant la simulation de l'Arduino. Chaque classe définit une ou plusieurs méthodes représentant chacune un état de l'Arduino.

Tests Unitaires

Écrire des tests d'intégration ne suffit pas pour tester une application. Il est également nécessaire de décomposer chaque étape d'une action en tests unitaires. Par exemple, l'action `GET /door/lock` comprend plusieurs étapes pouvant toutes être décomposées en sous-étapes uniques et testées. Dans notre application, une action peut toujours être décomposée selon une suite d'étapes, bien que toutes les étapes mentionnées ci-dessous ne soient pas présentes à chaque action :

- Utiliser une ou des classes de notre dépendance Maven `ArduinoComponents` représentant un composant de l'Arduino
- Utiliser une ou des classes JAXB permettant de sérialiser/désérialiser des données
- Utiliser une classe ou des classes de types `Mapper` permettant de transformer une classe représentant un composant Arduino en classe JAXB
- Assigner des valeurs à une ou des classes représentant un composant de l'Arduino à partir des informations d'une classe JAXB.
- Communiquer avec l'Arduino

Comme la communication avec l'Arduino, la sérialisation/désérialisation par les classes JAXB et les classes représentant un composant Arduino ont déjà été testées, les tests unitaires deviennent simples. Il est uniquement nécessaire de tester le mappage d'une classe de `ArduinoComponents` vers une classe JAXB et inversement, ainsi que les méthodes implémentées dans les classes JAXB. Dans notre cas, uniquement une méthode a été implémentée pour les classes JAXB `Lock` et `Open`. Dans ce cas-là, l'écriture de tests devient facile, avec comme exemple le listing 4.9 qui teste si, en fonction des valeurs envoyées par l'Arduino, la classe JAXB correspondante possède soit l'état `OPEN` soit l'état `CLOSED`. Comme nous pouvons le voir, le code est similaire à n'importe quel test réalisé avec la bibliothèque JUnit.

```

1 @Test
2 public void testMap() {
3     LinearPotentiometer lp = new LinearPotentiometer();
4     lp.setPosition(0);
5     Lock lock;
6     lock = new LockMapper(lp).map();
7     assertEquals("lock should be open", lock.getState(), Lock.State.OPEN);
8     lp.setPosition(1023);

```

```

9     lock = new LockMapper(lp).map();
10    assertEquals("lock should be closed", lock.getState(), Lock.State.CLOSED);
11 }

```

Listing 4.9 – Test de la transformation de la classe LinearPotentiometer en classe JAXB

Cependant, les tests pour l'envoi de notifications vers le client manquent dans notre batterie de tests. Il est bien entendu possible de tester si une notification doit être envoyée, ce qui est simple, mais il est plus ardu de tester si le client a bien reçu la notification qui lui était destinée et que cette notification possède des valeurs correctes. Cela serait encore une amélioration à apporter à notre interface de tests et elle est discutée à la section 5.2.1.

4.2.7. Implémentation de l'interface client

Afin de permettre à tout un chacun d'utiliser notre service web, nous voulons quelque chose de mieux que la figure 4.4

```

leo@wlan-per2-151-48 ~/projects/wot/FirstXwot/FirstXwotServer $ curl -H "Accept: application/xml" 0.0.0.0:9090
~/FirstXwotServer/resources/door/
<?xml version="1.0" encoding="UTF-8" standalone="yes"?><door xmlns="http://jaxb.xwot.first.ch.unifr.diuf"><lis
tOfDevices><open uri="http://0.0.0.0:9090/FirstXwotServer/resources/door/open"><position>0</position></open><l
ock uri="http://0.0.0.0:9090/FirstXwotServer/resources/door/lock"/></listOfDevices></door>leo@wlan-per2-151-48
~/projects/wot/FirstXwot/FirstXwotServer $ 

```

Figure 4.4. – Requête avec le client HTTP curl sur la ressource GET /door et demandant de retourner la réponse XML

C'est pourquoi nous avons décidé d'implémenter une interface client Web. Après quelques recherches, deux possibilités d'implémentation s'offraient à nous. La première en utilisant une solution pure Java de type GWT ou Vaadin. GWT a rapidement été écarté de par sa réputation d'instabilité et de déploiement difficile. Quant à Vaadin, les avantages sont l'écriture uniquement en Java, langage bien maîtrisé et la simplification de l'interaction client-serveur en AJAX. De plus, nous pouvons nous concentrer uniquement sur le codage et pas du tout sur le design. Énormément de composants sont fournis par Vaadin et il est uniquement nécessaire de les assembler grâce à une interface graphique. Le désavantage est la courbe d'apprentissage plutôt raide. La deuxième possibilité était de développer une petite application Ruby on Rails, framework très bien maîtrisé. Les avantages sont une très bonne maîtrise du framework, ne nécessitant pas de temps d'apprentissage, ainsi qu'une grande souplesse dans l'implémentation. Les désavantages sont la nécessité d'investir du temps dans le design pour avoir une application plaisante. De plus, la communication client-serveur peut être plus compliquée que celle de Vaadin.

Finalement, nous avons retenu la deuxième solution, notamment parce que grâce à l'utilisation d'un framework CSS, le design devient extrêmement simple et que grâce à l'utilisation de certaines techniques de Rails, la communication client-serveur se trouve grandement simplifiée. Nous y reviendrons à la section 4.2.7.

Principes de base de Ruby on Rails

Ruby on Rails est un framework orienté MVC, développé en Ruby, avec tous les outils nécessaires à la création d'applications Web. Par défaut, il fournit un ORM pour interagir

avec une base de donnée SQL et une grande quantité d'autres outils pour faciliter le développement. C'est-à-dire, un moteur de template pour encoder des données en différents formats, simplification de la création de requêtes AJAX, génération semi-automatique de code, interface de tests et bien d'autres outils simplifiant la vie d'un développeur. De plus, il encourage vivement la création d'applications RESTful et est basé sur le paradigme « conventions over configurations » (conventions avant configurations).

Comme nous l'avons spécifié plus haut, Rails suit très fortement le pattern MVC. Ainsi, les *modèles* ont comme responsabilité de gérer les données de l'application, le plus souvent par l'ORM **ActiveRecord**, mais ils contiennent également toute la « Buisnens Logic ». Le code spécifique à chaque application, les algorithmes, la gestion des droits, etc. devrait se trouver dans les modèles. Tandis que les contrôleurs ont pour tâche de répondre aux requêtes HTTP du client et de retourner des données selon le format souhaité. Normalement, le code contenu dans les contrôleurs diffère peu d'une application à une autre et est simple à écrire et à lire. Par exemple, pour développer une ressource de type livre d'or, le code des contrôleurs généré automatiquement n'a plus besoin d'être modifié. Finalement, les vues sont un système de template utilisé pour formater les données fournies par le modèle. Pour chaque format de données un système de templates différent est proposé. ERB, fourni par défaut, permet d'encoder à peu près n'importe quel format, mais est surtout utilisé pour le HTML et le Javascript. Tandis que d'autres templates comme JBuilder qui est spécifique pour le JSON, Builder, lui est spécifique pour tous les formats de types XML et HAML est spécifique pour le HTML.

Un aspect important est l'utilisation des conventions à la place de la configuration. Ainsi, en quelques minutes, il est possible de mettre en place une application Web fonctionnelle et de commencer le développement, sans devoir se préoccuper de la configuration de chaque module. Cela nous a été particulièrement utile pour le développement de l'interface client du rideau de fer. En quelques heures, l'interface était implémentée et utilisable.

Notons encore que Rails fournit deux moyens d'indiquer à l'application dans quel format encoder la réponse. La première façon est d'indiquer `.le-nom-du-format` à la fin de l'URI. Par exemple, l'URI `/livre-d-or.json` indiquera à l'application que les données sont souhaitées au format JSON. La deuxième manière est d'indiquer dans le header le format souhaité pour retourner des données. Par exemple, la requête `GET /livre-d-or accept: application/json` indique à l'application que pour l'URI `/livre-d-or` et la méthode GET, le format de données souhaité est du JSON. Il est encore important de savoir que le deuxième cas prime sur le premier si la requête suivante est effectuée : `GET /livre-d-or.xml accept:application/json` les données seront retournées au format JSON. Ces deux possibilités peuvent amener de la confusion au départ, mais sont utiles dans certains cas.

Finalement, dans un certain cas comme le nôtre une bibliothèque Javascript peut être utilisée pour simplifier les requêtes Ajax, ce qui représente une fonctionnalité très utile. Par exemple, en ajoutant l'attribut `data-remote="true"` à un lien, au clique de ce lien une requête AJAX vers le serveur est effectuée. Par défaut, la requête demande au serveur de retourner les données au format Javascript. Puis le Javascript retourné par le serveur sera évalué automatiquement par le client et, en fonction du Javascript, la page se mettra à jour. Le listing 4.10 est une concaténation de plusieurs fichiers et montre comment effectuer une requête AJAX de manière simple.

¹ `#views/guest_books/index.html.erb`
² `<html>`

```

3 ...
4 <h1>Tous les commentaires</h1>
5 <a href="/livre-d-or" data-remote="true">Afficher</a>
6 <div id="show"></div>
7 ...
8 </html>
9
10 #controllers/guest_books_controller.rb
11 class GuestBooksController < ApplicationController
12   ...
13   def index
14     @guest_books = GuestBook.all
15     respond_to do |format|
16       format.js
17       format.html
18     end
19   end
20 end
21
22 #views/guest_books/index.js.erb
23 $('#show').html(<%= escape_javascript(render @guest_books) %>)
24
25 #views/guest_books/_guest_book.js.erb
26 <h3><%= guest_book.title %></h3>
27 <p><%= guest_book.comment %></p>

```

Listing 4.10 – Exemple d'une requête AJAX avec Rails

En moins de vingt-six lignes il est donc possible d'écrire le code pour le scénario suivant :

1. Un utilisateur effectue une requête GET sur l'URI `/livre-d-or` et une page HTML contenant un titre `h1` « Tous les commentaires » et un lien « Afficher » lui est renvoyée
2. L'utilisateur clique sur le lien Afficher et tous les commentaires enregistrés dans la base de données de l'application lui sont renvoyés de manière dynamique (la page n'est pas rechargée) en AJAX.

Les trois points présents aux lignes numéro trois et onze sont là pour indiquer que du code a été omis. Les ligne commençant par `#` indique dans quel fichier se trouve le code qui va suivre. A la ligne numéro quatorze, nous allons chercher tous les commentaires dans la base de données et nous assignons le résultat à la variable d'instance de la classe `GuestBooksController`. Cette variable sera disponible depuis les vues. La classe `GuestBook`, de la ligne numéro treize a également été créée pour ce cas, fait partie des modèles de l'application et utilise l'ORM ActiveRecord. Nous avons omis le code de cette classe pour des raisons de brièveté, de même que le code assignant l'URI `/livre-d-or` à la méthode `index` de `GuestBooksController`.

Description de l'interface

L'interface, se compose de trois parties distinctes. La première permet d'enregistrer une nouvelle porte, de lister toutes les portes déjà enregistrées et de supprimer ou modifier une porte enregistrée. La deuxième permet d'interagir avec la porte sélectionnée. A savoir ouvrir/fermer la porte et verrouiller/déverrouiller la porte. La troisième partie liste tous les états passés de la porte sélectionnée. L'implémentation des notifications n'a pas été

directement réalisée dans l’interface. Le serveur Rails reçoit les notifications du serveur de l’Arduino, les enregistre, mais ne les affiche pas en temps réel. Cependant, cela serait très facile de le faire grâce à un Web socket ou à du Polling, par exemple grâce à la gem websocket-rails. Il n’y a pas non plus de gestion des utilisateurs ni des droits. Tout le monde peut tout faire. Cependant, cela serait très facile à réaliser via des gems telles que Devise et Cancan. Deux bibliothèques gérant parfaitement les utilisateurs et les droits. Nous n’avons pas implémenté ces deux fonctionnalités, parce qu’elles ont été réalisées à de nombreuses reprises et que nous avons préféré nous concentrer sur d’autres aspects plus intéressants.

Description de l’implémentation

Pour cette partie, nous n’allons pas discuter de l’implémentation en détails, nous partons du principe qu’avec une bonne connaissance du framework Rails, le code est compréhensible. Nous allons uniquement nous concentrer sur la fonctionnalité d’enregistrement/sélection du rideau de fer que nous avons implémentée.

Tout d’abord, rappelons qu’une des conventions de Rails veut que toute la « Business Logic » se trouve dans les modèles. De plus, nous voulons que les informations transmises par le serveur du rideau de fer soient enregistrées, dans une base de donnée SQLite dans notre application client. Nous avons donc défini trois modèles en utilisant ActiveRecord, c’est-à-dire, les modèles `Door`, `Lock` et `Open`. Le schéma 4.6 montre les champs de chaque modèle et leurs relations. Retenons uniquement qu’un modèle `Door` peut être relié entre zéro et un nombre infini de fois au modèle `Lock` et `Open`. Ainsi, tous les états d’un rideau de fer sont sauvegardés, de même que quand notre application client reçoit une notification du rideau de fer, l’état du composant d’ouverture/fermeture ou du composant de verrouillage/déverrouillage est sauvegardé.

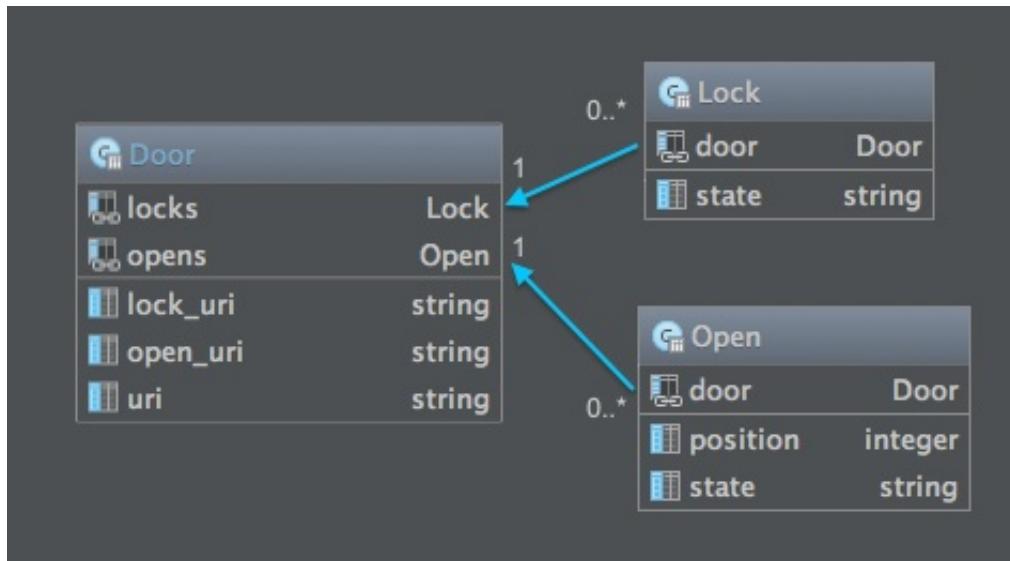


Figure 4.5. – Diagrammes des modèles de l’interface client pour la gestion de rideaux de fer

Pour interroger le serveur, nous avons défini une méthode `fetch` dans chacun des trois modèles. Cette méthode est responsable d’effectuer la requête vers le serveur, grâce à la

librairie Ruby RestClient, de parser le JSON obtenu et finalement de retourner une instance du modèle contenant les valeurs obtenues du serveur. Comme le code pour effectuer une requête vers le serveur et parser le JSON est toujours le même, nous avons défini la méthode `get_params` dans le module `ClassParams`, lui-même compris dans le module `Concerns::DoorMethods` étendu de `ActiveSupport::Concern`. Grâce aux spécificités de Ruby, chaque classe incluant ce module dispose de la méthode `get_params`. De même que pour simplifier l'interaction avec le `Hash` obtenu à partir du JSON, nous avons défini la classe `DoorParams`. Dans le listing 4.11 nous observons comment la méthode `fetch` du modèle `Open` est implémentée.

```

1 def self.fetch(uri)
2   params = get_params(uri)
3   open = Open.new(params.remove(:uri).params)
4   open.state = params[:state]
5   open
6 end

```

Listing 4.11 – Implémentation de la méthode `fetch` pour le modèle `Open`

Pour le modèle `Door`, nous avons également implémenté notre propre validateur (`DoorValidator`) qui effectue deux validations spécifiques à notre cas. La première vérifie que l'URL de la porte fournie n'est pas vide, tandis que la deuxième vérifie que le serveur a retourné une réponse correcte. Procéder ainsi, permet d'afficher des messages d'erreurs adaptés dans le formulaire d'enregistrement de la porte.

A chaque fois qu'une porte est enregistrée, l'état du rideau de fer correspondant est récupéré depuis le serveur et l'inscription aux notifications du rideau de fer est également créée. De même qu'à chaque sélection d'une porte, l'inscription aux notifications est mise à jour. Ainsi, dès qu'une porte est enregistrée les notifications la concernant seront réceptionnées par l'interface client et stockées dans la base de donnée. À la sélection d'une porte, tous les états, et donc aussi les notifications, qui sont également des états du rideau de fer, sont consultables. Tandis que si une porte et tous ses états sont supprimés de l'interface client une requête sera envoyée vers le rideau de fer afin d'annuler l'envoi de notifications.

Afin d'obtenir un visuel agréable à peu de frais, nous avons utilisé le framework HTML /CSS/Javascript Foundation. Ce framework, un peu moins connu que bootstrap, permet d'obtenir des interfaces visuelles propres en assemblant des composants proposés par le framework. Nous avons effectué ce choix pour quatre raisons :

1. Framework bien maîtrisé et ne demande pas de temps d'apprentissage
2. Grand choix de composants et très bien documenté
3. Adaptatif à différentes tailles d'écrans (du smartphone à l'écran de télévision)
4. Très facilement modifiable grâce au langage Sass

Finalement, le listing 4.12 montre la liste de toutes les URIs que nous avons définies pour notre interface client, y compris les deux URIs, aux lignes numéro trois et sept, à travers lesquelles le serveur du rideau de fer peut envoyer des notifications à l'application client.

```

1 GET /
2 PUT /doors/:door_id/locks
3 POST /doors/:door_id/locks
4 GET /doors/:door_id/locks
5 GET /doors/:door_id/locks/:id

```

```

6 PUT /doors/:door_id/opens
7 POST /doors/:door_id/opens
8 GET /doors/:door_id/opens
9 GET /doors/:door_id/opens/:id
10 GET /doors
11 POST /doors
12 GET /doors/new
13 GET /doors/:id/edit
14 GET /doors/:id
15 PATCH /doors/:id
16 PUT /doors/:id
17 DELETE /doors/:id

```

Listing 4.12 – URIs de notre interface client

4.2.8. Construction physique de la porte

Finalement, pour compléter l'implémentation, nous avons construit un rideau de fer. Comme décrit à la section 4.2.1 et comme le montre le schéma 4.2, le rideau de fer doit posséder deux fonctionnalités. 1. Monter et s'abaisser 2. Quand le rideau est abaissé, se verrouiller et se déverrouiller De plus, il doit pouvoir être manipulé à la main tout en transmettant ses changements d'états à l'application Web. Après bien des heures d'efforts, nous avons obtenu la construction 4.6.



Figure 4.6. – Notre construction d'un « rideau de fer »

Choix de la technique de construction

Il est très vite apparu que de construire le rideau uniquement avec des matériaux de base demande beaucoup de temps et des compétences techniques élevées. La construction du rideau a été effectuée uniquement dans le but d'une démonstration de notre travail et ne doit répondre à aucun critère esthétique ni pratique. C'est pourquoi, nous avons cherché un système proposant un certain nombre d'éléments (cornières, montants, engrenages, etc) qu'il suffit d'assembler. A notre connaissance, deux systèmes de ce type existent, les legos technique³ et le meccano⁴. Malheureusement, il est quasiment impossible d'acheter des pièces spécifiques pour les legos techniques, mais uniquement des modèles avec un nombre restreint de pièces sont proposés à la vente. Le cas du meccano est similaire, à la différence que les packs proposés permettent de construire plus de modèles différents, sont moins chers et contiennent plus de pièces. De plus, nous possédions déjà un stock important de pièces de meccano et la manière de construire un objet est plus souple qu'avec les legos technique, ce qui nous a définitivement décidé à utiliser le meccano pour construire notre rideau.

Réalisation technique

Le choix du type de moteur et de potentiomètre était imposé. En effet, au départ, nous ne possédions que des moteurs à rotation continue et des potentiomètres linéaires. À partir de cette contrainte, nous avons imaginé un certain type de porte et écrit beaucoup de code. C'est uniquement à la toute fin, une fois que tout le code était écrit, que nous avons construit la porte et réalisé que l'utilisation d'un moteur à degrés pour le verrouillage et de potentiomètres rotatifs aurait simplifié la construction.

Ci-dessous, nous décrivons la réalisation en cinq parties, en suivant l'ordre dans lequel le « rideau de fer » a été construit.

1. Système pour monter et abaisser le rideau. Un bout de sac plastique lesté d'une barre métallique et fixé à deux grandes roues. Ces deux roues sont fixées à une barre métallique horizontale, elle-même fixée à deux montants. Puisque le rayon d'une roue est de 3.8 cm et que le rideau est fixé au bord de la roue, un demi-tour de la roue déplace le rideau d'environ 12 cm à la verticale. $\pi r \cong 11.94 \cong 12$ où r est le rayon est égal à 3.8 cm.
2. Système pour mesurer le déplacement du rideau grâce à un potentiomètre linéaire. Les potentiomètres linéaires à notre disposition mesurent un déplacement maximum de 3.2 cm. Comme la porte peut se déplacer de 12 cm, nous avons utilisé une bielle fixé à la barre métallique reliant les deux roues du rideau. Ainsi, un demi-tour de la barre métallique est converti par la bielle en un mouvement vertical d'environ 3 cm. Dans ce cas-là, nous avons cherché la valeur idéale par essais successifs.
3. Ouverture/fermeture du rideau grâce à un moteur à rotation continue. La spécificité d'un moteur à rotation continue est qu'il est possible de modifier la vitesse de rotation et non pas le nombre de tours, par opposition à un moteur à degrés. Il est donc difficilement possible et pas du tout adapté d'utiliser le moteur pour faire directement pivoter la barre métallique d'un demi-tour. C'est pour cette raison que nous avons utilisé deux engrenages pour démultiplier le nombre de tours. Un premier

³<http://www.lego.com/fr-fr/technic/?domainredir=technic.lego.com>

⁴<http://www.meccano.fr>

engrenage de vingt dents est mû par le moteur et un deuxième de 120 dents est fixé à l'axe du rideau. Ainsi, au maximum, trois tours de moteur sont nécessaires pour faire pivoter l'axe d'un demi-tour. De plus, pour des raisons de précision, il n'est pas possible de fixer directement l'engrenage de vingt dents au moteur et de l'apposer à l'engrenage à 120 dents. Le moteur n'est pas tout à fait compatible aux dimensions du meccano, les vibrations et le poids du moteur déstabilisent l'ensemble. C'est pourquoi, nous avons fixé le moteur le plus bas possible et que nous avons utilisé un élastique pour transmettre la force du moteur à l'engrenage à vingt dents. Cette manière de faire demande des tours supplémentaires au moteur pour déplacer le rideau, puisque le système de l'élastique n'est pas parfait et que quelque fois il tourne à vide. Ce qui, soit dit en passant, arrange parfaitement notre cas, puisque l'utilisation d'un moteur à rotation continue fait maintenant tout à fait sens.

4. Verrouillage/déverrouillage de la porte grâce à un moteur à rotation continue. Le verrouillage/déverrouillage nécessite un peu d'imagination. En effet, pour notre construction, une barre horizontale coulisse en bas de la porte, sans autre conséquence. Il faut imaginer que cette barre est censée bloquer le rideau quand celui-ci est en position fermé et ainsi empêcher son ouverture. Originellement, il était prévu d'utiliser une barre crantée et un ou plusieurs engrenages pour le verrouillage de la porte. Mais par manque de matériel, nous avons imaginé un autre système qui serait en fait plus adapté à un moteur à degrés. Une barre en plastique est fixée au moteur, une autre barre relie la barre en plastique du moteur et la barre horizontale du verrouillage. Par effet de levier, et après plusieurs ajustements successifs, une demi-rotation du moteur déplace la barre horizontale d'environ trois centimètres.
5. Mesurer la position de la barre de verrouillage/déverrouillage. Pour mesurer la position de la barre, nous utilisons à nouveau un potentiomètre linéaire. Cependant, le système est plus simple que pour mesurer la position du rideau. Nous avons simplement fixé le potentiomètre au-dessus de la barre de verrouillage et relié le curseur à cette même barre. Comme notre système déplace la barre d'environ trois centimètres et que la course maximal du potentiomètre linéaire est de 3.2 cm, cela fonctionne à la perfection.

Construire un nouveau rideau de fer

Théoriquement, n'importe quel rideau de fer peut être construit en utilisant le code que nous avons développé, le sketch Arduino, le service Web Jersey et l'interface client Ruby on Rails. Cependant, plusieurs paramètres peuvent être configurés :

- le numéro des pins pour les potentiomètres et les moteurs sont donnés pour l'Arduino entre la ligne dix et treize du fichier `door.ino`. Si les numéros des pins sont changés pour l'Arduino, il doivent également être changés dans le service Web. Pour cela il faut modifier le fichier `diuf.unifr.ch.first.xwot.components`.

ArduinoComponents

- Les potentiomètres ne s'ouvrent et ne se ferment pas complètement et les moteurs ne sont pas exactement bien calibrés, c'est pourquoi nous avons dû définir des valeurs pour les cas suivants :
 - Empêcher l'ouverture du rideau s'il est verrouillé, ligne numéro 98
 - Arrêt du moteur de fermeture du rideau quand le rideau est fermé, les valeurs à modifier sont à la ligne numéro 103

- Arrêt du moteur d'ouverture du rideau quand le rideau est ouvert, ligne numéro 107
- Arrêt du moteur de verrouillage quand la barre de verrouillage est totalement sortie (verrouillée), ligne numéro 110
- Arrêt du moteur de déverrouillage quand la barre de verrouillage est totalement rentrée (déverrouillée), ligne numéro 114
- Une marge d'erreur est utilisée pour calculer quand arrêter les moteurs une fois que les potentiomètres ont atteint une position donnée par l'utilisateur et pour déterminer quand envoyer un changement au service Web. Cette valeur peut être modifiée à la ligne numéro 31.
- Normalement, la valeur nonante indique qu'un moteur est à l'arrêt. Dans notre cas, cette valeur est 93.
- La vitesse maximale d'ouverture, de fermeture et la vitesse nulle des moteurs sont des constantes définies dans la classe `ContinuousServo`.
- La position ouverte, à moitié ouverte, fermée et une marge d'erreur des potentiomètres linéaires sont des constantes définies dans la classe `LinearPotentiometer`.

4.3. Aquaponie

Pour l'implémentation du système d'aquaponie, nous avons uniquement développé la modélisation physique et virtuelle du système. Ce deuxième cas étant plus compliqué, avec plus de ressources et sous-ressources, nous trouvions intéressant de nous focaliser sur l'utilisation du xWoT méta-modèle. De plus, nous n'avions pas les compétences techniques ni les ressources matérielles nécessaires pour réaliser une implémentation aussi complète que la porte.

4.3.1. Modélisation

Contrairement au rideau de fer, la modélisation de la partie physique a déjà demandé du temps. Dans ce cas là, nous avons remarqué que si la modélisation physique est bien effectuée, la modélisation virtuelle, grâce au script Python, est très facile. En effet, telle que nous l'avons définie, plus de treize actuateurs et senseurs sont nécessaires à la réalisation du système d'aquaponie. Les organiser de la bonne manière afin d'obtenir une représentation cohérente de chaque devices et sous-devices n'est pas aisé.

Partie physique

Les actuateurs et senseurs, eux-mêmes des devices, faisant partie du système d'aquaponie peuvent être regroupés dans trois catégories, ou devices principaux : 1. Énergie, devices en rapports avec la gestion de l'énergie (chaleur et électricité). 2. Eau, gestion de l'eau., température, courant, quantité. 3. Pièce, température du local et ouverture/fermeture des fenêtres.

Ainsi, chaque catégorie contient des sous-catégories et des sous-sous-catégories suivant les besoins de l'implémentation. Les devices ont été classés dans les différentes catégories

suivant deux critères. Le premier dépend de la fonctionnalité du device, les devices ayant des fonctionnalités proches sont regroupés. Par exemple, le composant pour mesurer la quantité d'eau chaude est dans la même catégorie que le composant pour mesurer la température de l'eau chaude. Ils partagent la même fonctionnalité qui est de mesurer un aspect de l'eau chaude. Le deuxième critère est celui de comment seront organisées les ressources de l'application Web. Par exemple, nous regroupons un actuateur et un senseur dans la même catégorie, quand l'actuateur modifie le système et le senseur mesure la modification. Ce qui formera un contexte dans la partie virtuelle.

Le choix de comment regrouper les composants dans différentes catégories et quand les regrouper en contextes est totalement arbitraire. Il n'y a pas de règles fixes et plusieurs solutions sont possibles. Dans ce cas, nous avons proposé une solution, mais d'autres peuvent également exister. Par contre, de notre expérience, la chose à ne pas faire, est de tout mettre sur le même plan. Il devient très vite difficile de s'y retrouver et cela complique la modélisation de la partie virtuelle, tout en perdant un sens réel. De plus, le modèle n'est pas conçu de cette manière.

La figure 4.7 montre comment la modélisation physique a été réalisée sous Eclipse.

Partie virtuelle

Pour modéliser la partie virtuelle le principe est aisé. Nous utilisons un script Python qui va poser une série de questions en fonction de la modélisation physique. Typiquement, pour chaque device, il demande si ce device est un contexte ou non. Si le device est bien un contexte, ou un senseur, il demande également si le device possède un publisher. De plus, pour chaque device, le script Python demande de fournir une URI. Au final, la partie virtuelle est générée et le résultat visible sous Eclipse.

Il y a deux points importants à relever avec notre modélisation de l'aquaponie. Premièrement, le nombre de ressources d'actuateurs est très faible, la majorité des ressources sont des ressources de contextes ou de senseurs, sans oublier les publishers. Deuxièmement, le cas de certaines ressources d'actuateurs est encore ouvert à la discussion. Normalement, le méta-modèle prévoit qu'un actuateur possède uniquement une méthode PUT. Cependant, dans certains cas, il peut être intéressant de connaître l'état de l'actuateur avant de le modifier, particulièrement dans un contexte multi-utilisateur. Par exemple, dans notre modélisation, la ressource actuateur responsable de pomper l'eau est un moteur tournant à un régime donné. Quelques fois, il peut être intéressant de connaître les pulsations (le voltage et de là une vitesse) du moteur avant de modifier son régime. Dans ce cas-là, il serait nécessaire que la ressource implémente une méthode GET supplémentaire. Mais ceci serait en contradiction avec ce qui est prévu par le méta-modèle. Cependant, une autre façon de voir les choses, est que si l'on veut connaître l'état d'un actuateur, il est nécessaire d'implémenter une ressource de contexte. Cela est vrai dans le cas où l'état de l'actuateur n'est pas directement observable, ou n'est pas intéressant à connaître, mais où il est requis de connaître l'état de la partie du système modifiée par l'actuateur. Par exemple, l'actuateur pour l'ouverture des fenêtres, un moteur à degrés, n'ouvre pas directement la fenêtre mais actionne un mécanisme qui ouvre la fenêtre. Ainsi, connaître la position du moteur ne nous renseigne pas forcément sur la position de la fenêtre. Dans l'exemple de la pompe, la création d'un contexte fait moins de sens, puisque la pulsation du moteur est directement donnée par l'actuateur et non par un système lié. Par contre, si nous voulions connaître le débit de la pompe, il serait nécessaire d'implémenter un

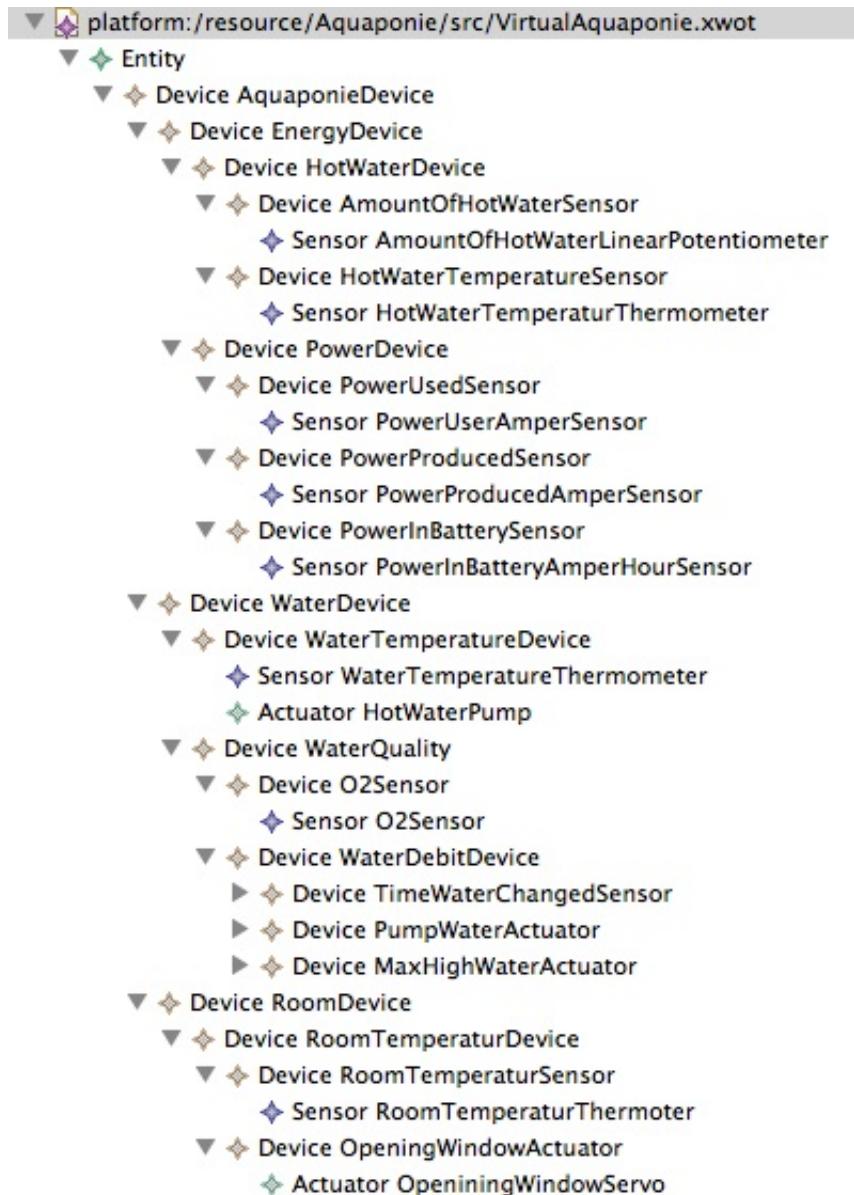


Figure 4.7. – Auquaponie, modélisation de la partie physique sous Eclipse

contexte, la vitesse du moteur n’indiquant pas nécessairement le débit de la pompe. Par exemple, un défaut dans l’alimentation en eau produirait un débit nul quand bien même le moteur tournerait à plein régime. Il peut donc y avoir une contradiction entre le débit estimé, en fonction du régime du moteur, et le débit réel. Tandis que la pulsation du moteur, quoi qu’il arrive, sera toujours mesurable et vérifiable.

4.4. Machine à caramels

L’implémentation de la machine à caramels est plus complète que pour le système d’aquaponie. En plus de décrire la modélisation, la section 4.4.2 expliquera comment nous avons structuré les données échangées avec l’utilisateur au moyen du langage XSD. Bien que la machine à caramels soit un système plus petit que l’aquaponie, le fichier XSD est déjà

plus important et légèrement plus complexe que dans le cas du rideau de fer, rendant la discussion plus intéressante. Finalement, la section 4.4.3 abordera un point pas encore évoqué dans ce travail, comment générer le projet Maven, à partir du fichier XSD et du fichier `application.wadl` (obtenu après la modélisation virtuelle). En effet, grâce à des plugins Maven et un squelette de projet type, il est possible de créer une application Jersey fonctionnelle qui nécessite uniquement, à quelques exceptions près, l'ajout de quelques lignes de code dans chacune des méthodes d'une classe de ressource. Et comme nous l'avons vu dans les listing 4.3 et 4.4 ce code est court, toujours assez semblable et facile à écrire.

4.4.1. Modélisation

Au bout de la troisième fois, la modélisation devient quelque peu répétitive avec peu d'aspects qui n'ont pas encore été discutés dans la modélisation du rideau de fer et dans le système d'aquaponie. C'est pourquoi, pour cette dernière modélisation nous nous focalisons surtout sur comment la réaliser d'un point de vue pratique.

Partie physique

Tout d'abord, la première fois que nous voulons modéliser un objet, il est nécessaire d'installer deux plugins Eclipse. Comme ces plugins doivent être installés manuellement, il est nécessaire de les placer dans un dossier défini d'Eclipse. Une fois cela effectué, Eclipse peut être lancé et un nouveau projet créé. Puis, il suffit de créer un nouveau fichier `Xwot Model` afin de commencer la modélisation physique.

Comme le montre la figure 4.8, nous avons un device principal qui représente une machine à caramel. Ce device contient quatre autres devices représentant les quatre fonctionnalités de notre machine : 1. Débordement du mélange 2. Battage 3. Contrôle de quand le mélange est prêt 4. Température du mélange Les devices `OverflowDevice` et `HeatDevice` contiennent chacun un actuateur et un senseur. Nous les avons regroupés de cette manière pour une raison de sens et en pensant en créer deux contextes dans la partie virtuelle. Tandis que le device `WhipDevice` contient uniquement un actuateur et le device `CheckEndDevice` un senseur.

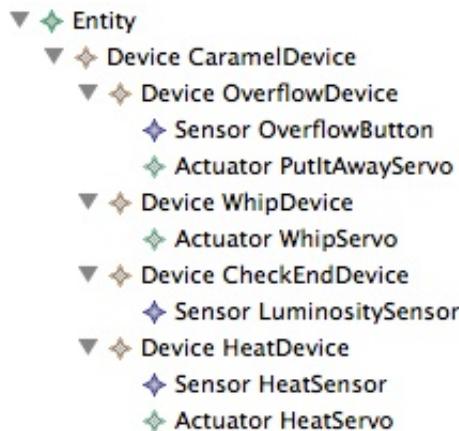


Figure 4.8. – Machine à caramels, modélisation de la partie physique sous eclipse

Partie virtuelle

Afin d'obtenir le fichier modélisant la partie virtuelle, nous utilisons un script Python de la manière suivante :

```
1 ./physical2virtualEntities.py -i PhysicalCaramelMachine.xwot -o VirtualCaramelMachine.xwot
```

`PhysicalCaramelMachine.xwot` est le fichier modélisant la partie physique de la machine à caramels et `VirtualCaramelMachine.xwot` est le nom du fichier qui sera créé une fois le script Python terminé. Ce script Python est contenu dans un dossier `src` lui-même contenu dans un dossier `Model2WADL`, il est important de savoir que le script Python doit être exécuté depuis le répertoire dans lequel il se trouve.

Une fois le script lancé, il nous demande quel est l'URI pour le premier device, `CaramelDevice` dans notre cas. Une fois ceci fait, il nous demande si le device contient des sous-devices pouvant former un contexte, question à laquelle nous répondons non, puis il passe au premier sous-device, `OverflowDevice` dans notre cas. Les mêmes opérations sont répétées jusqu'à ce que tous les devices aient été passés en revue. Au final, nous obtenons le fichier modélisant la machine à caramels d'un point de vue virtuel et physique.

Un point important à relever dans la modélisation virtuelle, est que pour la ressource `WhipeRessource` nous nous trouvons en présence du même cas que discuté à la section 4.3.1. Cette fois, nous avons décidé d'ajouter une méthode GET supplémentaire à la ressource. Deux ressources de contexte ont été créées, une pour gérer le débordement et l'autre pour la température.

Le fichier généré peut être ouvert sous Eclipse et les URI ainsi que les noms de chaque ressources modifiés. Quand nous sommes satisfaits du résultat, nous lançons un deuxième script Python, `model2WADL.py` qui va générer le fichier WADL décrivant toutes les ressources de l'application. Ce même fichier sera utilisé par le plugin Maven `cxf-wadl2java-plugin` afin de générer les classes Java de ressources utilisables dans Jersey.

4.4.2. Discussion au sujet du XSD

Comme pour le rideau de fer, nous avons un élément appelé `factory` qui contient une liste de quatre éléments. Dans la liste, ces quatre éléments contiennent chacun un attribut `uri` qui fait référence à la ressource à laquelle correspond cet élément. Les quatre éléments de la liste de l'élément `factory` représentent les données des quatre ressources définies, à savoir :

- Overflow
- Heat
- Whipe
- CheckEnd

L'élément `overflow` contient deux booléens, un pour indiquer si le mélange déborde ou non et l'autre pour indiquer si la casserole est enlevée du feu ou non. Le deuxième booléen peut également être utilisé par le client afin de donner l'ordre à la machine de retirer la casserole du feu. L'élément `whipe` contient uniquement une énumération avec comme états `NULL`, `FULL` et `TURBO`. C'est trois états correspondent à trois vitesses possibles pour battre le mélange. Nous avons procédé ainsi car nous avons estimé que l'utilisateur

n'est pas intéressé à régler la vitesse de manière plus précise. L'élément `heat` contient une énumération avec également trois états utilisés pour contrôler la chaleur. Le client peut uniquement indiquer à l'application que la chaleur de la plaque doit être `NONE`, `MIDDLE` ou `FULL`. En plus de l'énumération, un élément contenant un entier pour renseigner la température en degré et une énumération indiquant quelle échelle de température est utilisée peuvent être fournis au client. Finalement l'élément `caramel`, correspondant à la ressource `CheckEnd` contient un booléen qui est vrai si les caramels sont prêts.

Il reste à noter que nous utilisons un « Global Bindings » pour afficher l'URI d'une ressource, avec la classe Java `Uri`, en attribut d'un élément. De même que nous utilisons deux « Bindings » pour transformer une énumération XSD en un `Enum` Java.

4.4.3. Création du projet

La création du projet s'effectue en deux phases. 1. La génération du squelette du projet est effectuée par une commande Maven. 2. Les classes de ressources et les classes JAXB sont générées par deux plugins Maven. Nous avons défini notre propre squelette d'application Jersey, afin qu'il soit adapté au contexte du Web des Objets. Pour ce faire, nous sommes partis d'un squelette créé par A. Ruppen et que nous avons modifié. Les modifications apportées au squelette et comment nous avons réalisé ce dernier seront discutés à la section 5.1.6. Toujours est-il, qu'une fois ceci réalisé, nous lançons la commande suivante :

```
1 mvn archetype:generate -DarchetypeCatalog=http://diufpc46.unifr.ch/artifactory/ext-release-local
```

Maven va encore nous demander quelques informations concernant le projet, son nom et le nom de package par défaut notamment. Puis Maven s'occupe de générer l'intégralité du code et nous obtenons une application prête à être utilisée avec une classe JAXB et une classe de ressources comme exemple. Une fois le projet généré, il ne reste plus qu'à positionner le fichier WADL et le fichier XSD dans le dossier `src/main/resources/WADL` du projet. Il est important que le fichier XSD ait bien le nom spécifié dans le fichier WADL. Puis il ne reste plus qu'à exécuter la commande

```
1 mvn clean compile package
```

pour avoir les classes JAXB et les ressources générées.

5

Développements annexes

5.1. Bibliothèques, tests unitaires, et gem	57
5.1.1. Protocole de communication	58
5.1.2. ArduinoCommunication	60
5.1.3. ArduinoComponents	65
5.1.4. Simulation de l'Arduino	66
5.1.5. Icwot	71
5.1.6. Archéotype Maven	74
5.2. Futurs développements	75
5.2.1. Amélioration de la bibliothèque de tests	76
5.2.2. Implémentation du concept des publisher dans un navigateur web	76
5.2.3. Communication entre l'Arduino et l'application Web	78
5.2.4. Modélisation de composants	78
5.2.5. Langage de gestion des événements d'un publisher	79

L'implémentation du rideau de fer a nécessité une création importante de code. Afin de pouvoir réutiliser le code produit au mieux, nous l'avons extrait dans plusieurs bibliothèques. Dans ce chapitre, nous détaillons tous les développements effectués dans le cadre de ce travail, qui sortent de la stricte implémentation des trois cas d'utilisations.

Au fur et à mesure de l'avancement de ce travail, nous avons noté un certain nombre d'aspects en relation avec le Web des Objets qui pourraient encore être améliorés. Malheureusement, comme nous ne réalisons qu'un travail de Bachelor tout ce que nous avons noté n'a pas pu être réalisé. C'est pourquoi nous présentons ces observations dans ce chapitre, donnant ainsi des pistes pour des développements futures.

5.1. Bibliothèques, tests unitaires, et gem

Dans cette section, nous discutons du code qui a été produit durant l'implémentation du rideau de fer, mais extrait dans plusieurs bibliothèques de manière à pouvoir être réutilisable dans d'autres projets. Il s'agit de fonctionnalités pour communiquer avec l'Arduino, d'une suite de tests adaptée à la communication avec un Arduino par port série et de la possibilité de pouvoir simuler un Arduino. Afin de valider le principe des notifications envoyées par une requête HTTP POST au client, nous avons implémenté

un petit serveur qui permet de s'enregistrer pour un type de notification et de stocker les notifications reçues sous forme de log. Ce petit serveur est distribué via une gem (librairie Ruby installable automatiquement) appelée icwot (Inversion of Control for Web Of Things) et s'installe en une commande. Finalement, pour permettre une création plus rapide de futurs projets, nous avons encore créé un archétype Maven. Cet archétype génère un projet Jersey fonctionnel adapté au Web des Objets et à la communication avec un Arduino à partir de quelques informations fournies par l'utilisateur.

Le diagramme 5.1, montre la relation entre les différentes implémentations décrites aux sections 5.1.2 et 5.1.3. Dans toute cette section, nous allons toujours partir du niveau d'abstraction le plus bas pour aller au niveau d'abstraction le plus élevé.

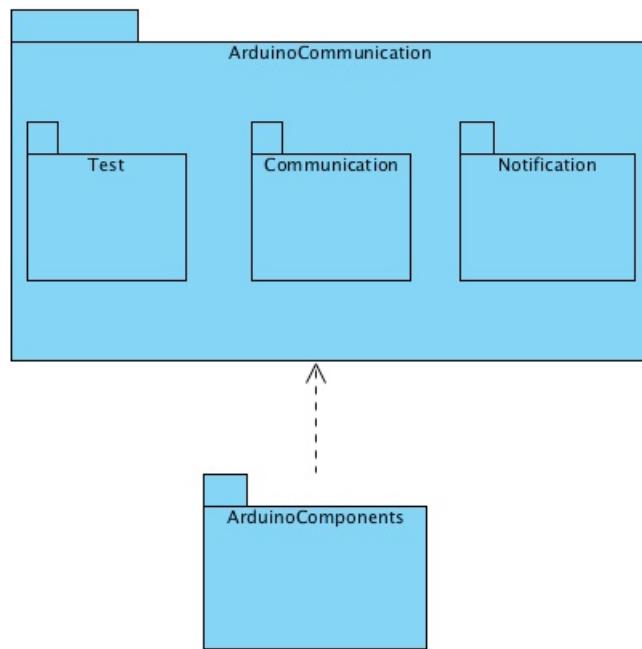


Figure 5.1. – Hiérarchie des différents composants ayant été implémentés

5.1.1. Protocole de communication

Avant tout chose, nous avons défini un format de donnée pour structurer les informations échangées entre l'Arduino et le serveur. A la section 4.2.3, nous expliquons la motivation et les raisons du choix du langage JSON comme manière d'encoder les données et expliquons comment les données échangées sont structurées grâce au JSON.

Nous avons voulu définir un protocole de communication qui soit d'une part utilisable pour n'importe quel cas d'implémentation et qui soit d'autre part facile à implémenter. De plus, il y a également une volonté de limiter la quantité de données échangées.

Motivation

La motivation d'un tel protocole est de permettre la réutilisation du code tant du côté serveur que du côté de l'Arduino. En effet, puisque ce protocole peut être utilisé dans n'importe quel projet, il est possible de créer une bibliothèque côté serveur afin d'en

simplifier l'usage. Une telle bibliothèque est expliquée à la section 5.1.2. Coté Arduino, la grande facilité d'utilisation de ce protocole, à l'aide d'une bibliothèque pour parser le JSON, permet de communiquer de manière efficace en seulement quelques lignes. Ainsi, un protocole adapté et simple d'utilisation permet la réutilisation du code, l'écriture de code facile à comprendre et évite donc des erreurs.

Description

Pour structurer les données échangées, nous nous basons sur une propriété du JSON et une de l'Arduino. Premièrement, en JSON, un certain type de données (Chaîne de caractères, nombre entier ou réel, sous-élément JSON, etc) est identifié par une clé pouvant être composée de chiffres et de lettres. Deuxièmement l'Arduino possède un certain nombre de connecteurs tous identifiés par un nombre. Dans le langage Arduino, ces nombres sont représentés par des constantes. Nous avons défini ainsi le principe suivant : toutes les informations relatives à un composant sont regroupées dans un sous-élément JSON qui est identifié par le numéro du pin auquel est connecté le composant produisant ces informations. Par exemple, pour le composant qui est le moteur du rideau de fer, produisant comme information la vitesse à laquelle il tourne, la sous-chaîne JSON est la suivante :

```
1 {"speed": 90}
```

Comme le moteur est connecté au pin 01, identifié par le nombre dix, la sous-chaîne est identifiée par le nombre dix et est ajoutée à la chaîne principale échangée avec le serveur. Au final, nous obtenons du JSON structuré de la sorte :

```
1 {"10": {"speed": 90}, "14": {"value": 0, "oldValue": 500}, ...}
```

Où ... signifient que d'autres sous-éléments identifiés par le nombre d'un pin peuvent exister. Le deuxième élément, identifié par le nombre 14, représente un autre composant connecté au connecteur I1 (connecteur d'entrée de l'Arduino, sur lequel peut être branché différents types de senseurs). Dans le cas du rideau de fer, cette représentation est pour un potentiomètre linéaire.

L'avantage d'un tel système est que d'une part, il est possible de représenter tous les composants d'un Arduino quelque soit le cas d'implémentation. Puisque chaque composant est forcément relié d'une manière unique à l'Arduino et donc identifiable de manière unique. D'autre part, ce principe permet d'imbriquer à l'infini des informations dans des informations sans limites de complexité. Cependant, les contraintes matérielles de l'Arduino limitent rapidement le nombre de données échangées.

Du côté de l'Arduino, l'implémentation est extrêmement simple et est discutée à la section 4.2.3. De plus, le code du rideau de fer donne un exemple d'implémentation. Coté Java, grâce aux bibliothèques ArduinoCommunication et ArduinoComponents, expliquées à la section 5.1.2 et 5.1.3, il est possible de travailler uniquement avec des objets Java. Les valeurs des variables d'instance de ces objets sont en fonction du JSON à décoder ou à encoder. Cependant, pour encoder et décoder le bon objet en fonction d'un composant, il est nécessaire de connaître le pin sur lequel est branché ce composant. Pour ce faire, nous avons défini un enum, `TinkerShield`, dont le nom de ses variables correspond aux noms des pins (O1..O5 et I1 .. I5) et la valeur de ses variables à la valeur du pin sur l'Arduino. Par exemple, `TinkerShield.o_1` retourne le nombre entier 10.

5.1.2. ArduinoCommunication

Cette bibliothèque comprend toutes les fonctionnalités utiles à l'interaction avec l'Arduino. Cela va de la connexion à un port série jusqu'à la transformation d'objets Java en informations compréhensibles pour l'Arduino, en passant par une suite de tests adaptés à l'interaction avec l'Arduino. De plus, nous avons également inclu des classes simplifiant la gestion des notifications.

Comme nous avons défini que l'Arduino est relié au serveur Web par un port série via un câble USB, il nous fallait pouvoir écrire et lire des informations sur le port série. La bibliothèque RxTx s'occupe de cela, mais est trop généraliste. Il était donc nécessaire de créer notre propre ensemble de classes adaptées à notre cas spécifique. Cela signifie, se connecter facilement au port série, pouvoir y lire et écrire, implémenter notre propre pattern observateur pour savoir quand de nouvelles informations sont disponibles, encoder et décoder des données en JSON et facilement interagir avec les informations des différents composants de l'Arduino. Au final, l'interaction avec l'Arduino doit être tout aussi facile qu'avec une base de données traditionnelle.

La section 5.1.4 explique comment il est possible de simuler un Arduino. Un des premiers usages de la simulation est de pouvoir tester les applications développées sans dépendre de l'Arduino. Afin d'écrire facilement des tests utilisant la simulation de l'Arduino, nous avons défini un ensemble de fonctionnalités.

Finalement, nous avons défini quelques fonctionnalités pour gérer les notifications. C'est à dire, fournir un moyen d'enregistrer les clients auxquels envoyer des notifications, envoyer les notifications aux clients et utiliser le design pattern builder pour déterminer quand et comment envoyer une notification en fonction des événements.

Le diagramme 5.2 synthétise de quelle manière ces fonctionnalités sont structurées dans la bibliothèque ArduinoCommunication

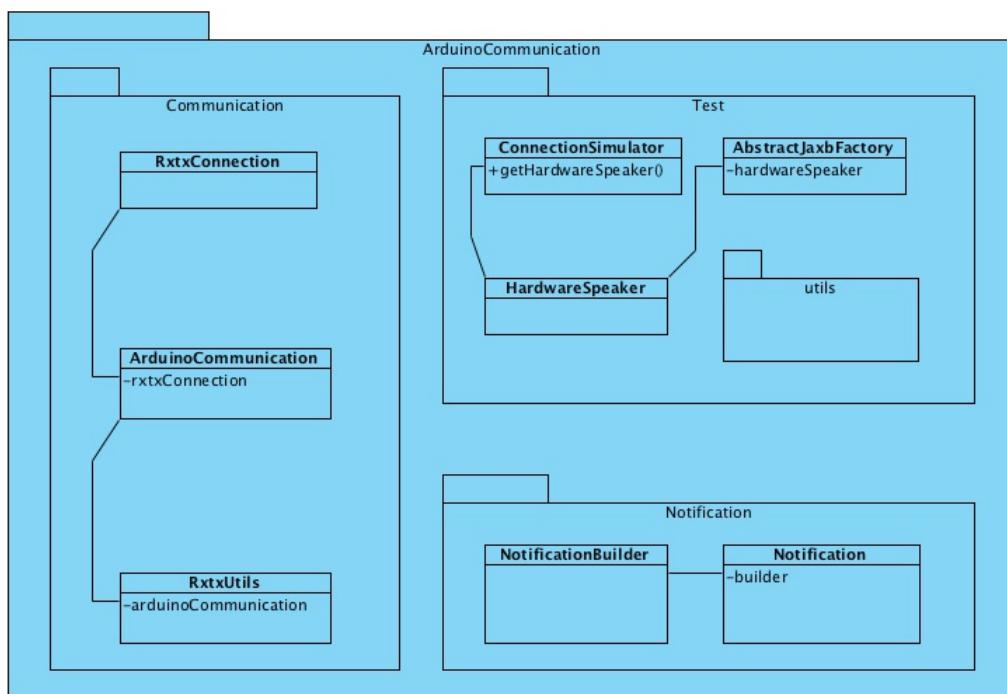


Figure 5.2. – Structuration des différentes fonctionnalités de ArduinoCommunication

Motivation

La première motivation est de créer un ensemble de fonctionnalités, spécifiques à notre cas, en utilisant celles de RxTX, pour interagir avec l'Arduino connecté au port série. Ainsi, nous permettons la réutilisation de code, le code de l'application est beaucoup plus lisible et nous pouvons réduire le nombre d'erreurs potentielles.

La deuxième motivation est de faciliter la création de tests adaptés au cas d'un Arduino connecté à l'application. Les motivations à l'écriture de tests sont une évidence. Grâce à notre système, le code nécessaire pour écrire des tests est moins important tout en évitant des erreurs. Ainsi, notre système encourage et simplifie l'écriture de tests.

Finalement, la dernière motivation est de simplifier la gestion des notifications en stockant les clients enregistrés, en envoyant les notifications et en implémentant le pattern observateur. Ainsi, pour l'implémentation de futures applications du Web des Objets, il ne sera plus nécessaire de réfléchir comment implémenter les notifications, mais uniquement remplir les vides d'une classe type. A nouveau notre système encourage la réutilisation du code. Si, par la suite un système pour stocker les clients dans une base de données est développé, ou l'envoi des notifications est amélioré grâce au multi-threading, toutes les applications utilisant cette bibliothèque en profiteront.

Il y a encore l'avantage relatif à toute bibliothèque qui est que le code est utilisé et testé en situation réelle, réduisant ainsi le risque d'erreurs (par opposition à la même fonctionnalité implémentée par chacun dans son coin).

Réalisation de l'interaction avec l'Arduino

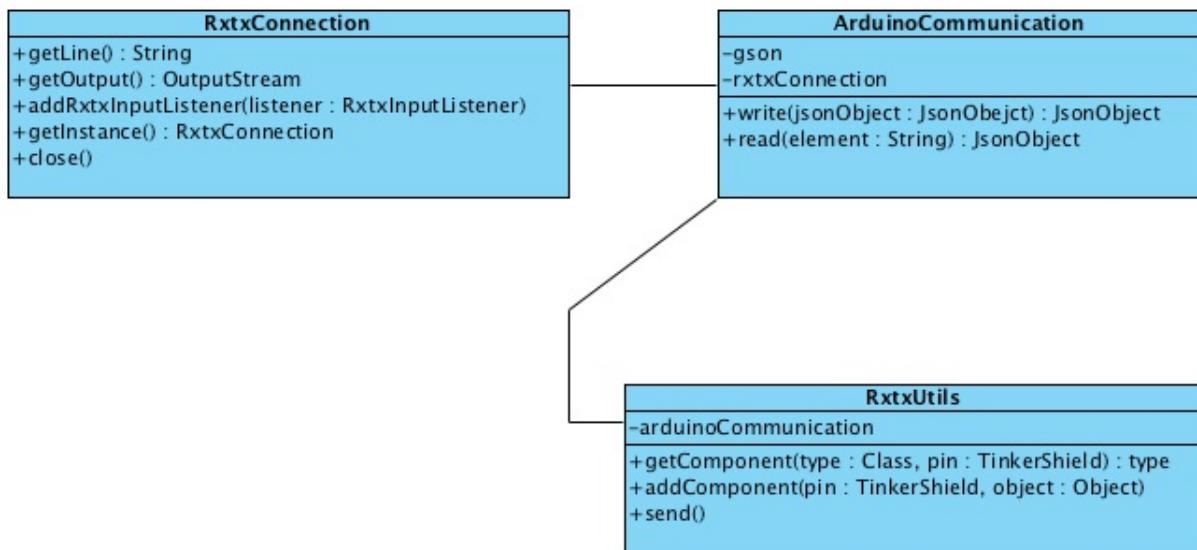


Figure 5.3. – Hiérarchie des classes pour la communication avec l'Arduino

Les fonctionnalités pour interagir avec l'Arduino sont réparties dans trois classes, suivant leur degré d'abstraction. Le diagramme 5.3, ci-avant, montre les relations entre ces classes et leurs méthodes.

La première classe, RxTxConnection, permet la connexion/déconnexion au port série, la lecture et l'écriture d'informations du port série et implémente un observateur pour

la réception d'informations envoyées par l'Arduino. RxTx propose déjà un observateur, mais seul un observant à la fois peut l'utiliser. Notre observateur se base donc sur celui de RxTx, mais permet un nombre illimité d'observants. Un dernier point est que cette classe est un singleton autorisant une unique connexion au port série dans toute l'application. Il est important de noter la manière dont les informations sont lues du port série. Dès qu'une nouvelle information est disponible sur le port série, l'observateur de RxTx lance un événement et nous enregistrons l'information reçue dans une variable d'instance de la classe RxTX. Ce faisant, cela nous permet de ne jamais manquer une information, de toujours disposer d'une information quelconque, à part au lancement de l'application, et d'être simple. Une mauvaise solution aurait été d'aller lire les informations sur le port série à chaque fois qu'une autre classe de l'application a besoin d'informations de l'Arduino. Cette (mauvaise) solution demanderait un code plus compliqué pour obtenir le même résultat. En somme, cette classe est utilisée pour améliorer et simplifier l'usage de RxTx dans notre cas spécifique.

La deuxième classe, `ArduinoCommunication`, utilise la première et permet, grâce à la bibliothèque Gson, d'encoder/décoder en JSON les informations relatives au port série. Cette classe a pour unique but de simplifier l'usage de la bibliothèque Gson pour notre cas.

La troisième et dernière classe, `RxtxUtils`, permet d'encoder/décoder en JSON des objets Java en se basant sur la classe `ArduinoCommunication`. L'encodage et le décodage d'objets Java en JSON est géré grâce à la bibliothèque Gson¹ de Google. Les méthodes `addComponent` et `getComponent`, de `RxtxUtils`, utilisent les types génériques et permettent ainsi une transformation aisée entre des composants Arduino, encodés en JSON, en objets Java représentant ces composants. Un aspect qui mérite d'être relevé est que pour permettre d'envoyer en même temps les informations de plusieurs composants à l'Arduino, il est nécessaire de l'effectuer en deux étapes. Tout d'abord, la méthode `addComponent` permet de stocker dans une variable les différents objets à envoyer. Technique, ces objets Java sont transformés en instance de classe `JsonElement` et ajoutés à une instance de la classe `JsonComponent`, qui toutes deux appartiennent à la bibliothèque Gson. Ensuite, à l'appel de la méthode `send`, les objets stockés sont transformés en JSON et envoyés à l'Arduino.

Bien que les différentes fonctionnalités soient simples à comprendre et encore plus à utiliser, la réalisation n'était pas évidente. Nous avons dû comprendre l'utilisation de deux bibliothèques et imaginer une structure adaptée à nos besoins.

Interagir avec l'Arduino

L'interaction avec l'Arduino peut s'effectuer de plusieurs manières, suivant la classe utilisée, `RxtxConnection`, `ArduinoConnection` ou `RxtxUtils`. Dans cette section, nous détaillons uniquement l'usage de la classe `RxtxUtils`, c'est normalement la seule classe utilisée et la plus facile d'emploi.

Pour obtenir une instance d'une classe Java à partir d'un élément JSON représentant l'état d'un composant Arduino, deux étapes sont nécessaires.

1. Définir une classe possédant des attributs du même nom que les éléments du JSON qui nous intéressent. Les types des variables d'instance doivent être des types simples

¹<https://code.google.com/p/google-gson/>

(int, float, boolean, String, etc). Cette classe a peut-être déjà été définie dans la dépendance Maven ArduinoComponents à la section 5.1.3

2. Appeler la méthode `getComponent` de `RxtxUtils`.

La définition d'une classe est un processus standard et nous ne le commenterons pas ici. La méthode `getComponent` requiert deux arguments, le premier doit être le type de l'objet que l'on cherche à récupérer et le deuxième le pin, identifié par un nombre, sur lequel est connecté le composant Arduino. Pour le nombre du pin, l'enum `TinkerShield`, discuté à la section 5.1.1 est d'une grande aide. Le listing 5.1 est un exemple standard de comment le réaliser et assume que la classe `LinearPotentiometer` fait partie de la dépendance ArduinoComponents.

```
1 RxtxUtils utils = new RxtxUtils();
2 LinearPotentiometer pot = utils.getComponent(LinearPotentiometer.class, TinkerShield.
   i_0);
```

Listing 5.1 – Obtenir des objets Java à partir d'informations de l'Arduino

Si pour une raison ou une autre, il n'est pas possible de retourner une instance de la classe `LinearPotentiometer`, JSON mal formé, pas d'information de l'Arduino, etc, la valeur est nulle. A noter encore que le JSON envoyé par l'Arduino est équivalent à :

```
1 {14: {oldPosition: 0, position: 1023}}
```

Transmettre les valeurs des variables d'instance d'une classe Java n'est pas plus compliqué. En partant du principe qu'une classe Java contenant une ou plusieurs variables d'instance de type simple (int, float, boolean, String, etc) existe, il suffit d'appeler la méthode `addComponent`, puis si l'on désire encoder d'autres objets, appeler cette méthode à volonté. Afin de transmettre tous les objets encodés précédemment, il ne reste plus qu'à appeler la méthode `send`. Le listing 5.2 donne un exemple de la manière d'encoder deux objets et de les envoyer à l'Arduino.

```
1 RxtxUtils utils = new RxtxUtils();
2 ContiniousServo cs1 = new ContiniousServo();
3 ContiniousServo cs2 = new ContiniousServo();
4 cs1.setSpeed(ContiniousServo.OPEN_MAX_SPEED);
5 cs2.setSpeed(ContiniousServo.CLOSE_MAX_SPEED);
6 utils.addComponent(TinkerShield.o_1, cs1);
7 utils.addComponent(TinkerShield.o_2, cs2);
8 utils.send();
```

Listing 5.2 – Encoder des objets et les envoyer à l'Arduino

Le JSON produit sera de cette forme :

```
1 {10: {speed: 180}, 11: {speed: 0}}
```

Aux lignes numéro six et sept du listing 5.2, nous observons que la méthode `addComponent` prend deux arguments. Le premier indique le numéro du port de l'Arduino auquel est branché l'Arduino et le deuxième est l'objet à encoder. La méthode `send` de la ligne numéro huit envoie les objets `cs1` et `cs2` à l'Arduino. C'est uniquement à ce moment-là que des données sont transmises à l'Arduino.

Réalisation de la gestion des notifications

La gestion des notifications se décompose en deux parties. La première permet d'enregistrer les clients désireux de recevoir des notifications dans un `HashMap`. Cette partie est évidente et ne mérite pas de commentaires.

L'autre partie s'occupe de l'envoi des notifications. Le pseudo-code du listing 5.3 montre comment nous l'avons réalisée.

```

1 at initialisation
2   start listening event from RxtxInputListener
3 end initialisation
4 at each events from RxtxInputListener
5   determine with the aim of the assigned builder if the client need to receive a
     notification
6   if true, for each registered clients
7     send a HTTP POST request to the client's URL
8   end each registered clients
9 end each events

```

Listing 5.3 – Pseudo-code pour l'envoi de notifications

Ainsi, l'utilisation du pattern builder permet une politique d'envoi des notifications différente suivant les besoins de l'implémentation. Il suffit d'implémenter un builder différent pour chaque type de notification que l'on souhaite et l'assigner à une instance de la classe `Notification`. La méthode `hasNotification`, qui doit être redéfinie pour chaque builder, renvoie un booléen utilisé pour déterminer s'il faut notifier les clients.

Le principal défaut de notre gestion des notifications est la lenteur potentielle. Comme tout se passe dans un seul thread, les notifications sont envoyées une à une à chaque clients enregistrés. En partant du principe qu'un client unique peut s'enregistrer à plusieurs types de notifications et que la durée de la requête effectuée vers chaque client dépend de la rapidité du client à y répondre, nous risquons d'avoir très rapidement avoir un problème de performance et de notifications perdues. Bien évidemment, la solution serait d'utiliser le multi-threading, une telle implémentation aurait demandé trop de temps et sortait du cadre du travail.

Utilisation de la simulation pour les tests

Comme déjà mentionné à la section 4.2.6, nous utilisons deux types de test, les tests unitaires et les tests d'intégration. Pour les tests unitaires, nous avons défini une classe, `SerialHelpers`, qui peut être étendue par n'importe quelle classe de tests. `SerialHelpers`, créé une instance des classes `ConnectionSimulator` et `HardwareSpeaker` avant que le premier test soit exécuté et les assignent à deux variables d'instances ayant le mot-clé `protected`. Après que le dernier test a été exécuté, la méthode `stop` de l'instance de `ConnectionSimulator` est invoquée, causant l'arrêt de la simulation. Ainsi, dans n'importe quelle classe de tests étendant `SerialHelpers`, il est possible d'accéder à toutes les méthodes des classes `ConnectionSimulator` et `HardwareSpeaker`, sans se préoccuper de quand arrêter et démarrer la simulation. Pour une description détaillée des classes `ConnectionSimulator` et `HardwareSpeaker`, qui permet de simuler une connexion du port série, se référer à la section 5.1.4

Distribuer RxTx

La bibliothèque RxTx comporte des dépendances natives au système d'exploitation écrites en C. Afin de permettre une installation aisée sous n'importe quel système d'exploitation, c'est-à-dire que la commande `mvn clean compile jetty:run` s'occupe de télécharger et installer les dépendances requises, nous avons utilisé la dépendance RxTxRebundled, qui gère cela. Cependant, il y avait toujours une erreur lors de l'installation sous Raspbian, le système d'exploitation du Raspberry PI. Après investigations, nous avons remarqué qu'il manquait les sources compilées pour Linux avec processeur ARM. Nous avons donc cloné la dépendance, compilé les sources manuellement avec le Raspberry PI et les avons ajoutées à la dépendance. Puis, nous avons modifié le fichier `pom.xml` afin qu'il utilise notre dépendance placée sur un serveur de l'Université de Fribourg.

5.1.3. ArduinoComponents

L'idée de cette bibliothèque est simple. Comme nous avons une classe Java par composant Arduino et qu'il existe un nombre limité de composants Arduino, l'idée est de rassembler toutes les classes de composants Arduino. Ainsi, au fur et à mesure que le nombre de projets, suivant la même structure que le nôtre, augmentera, il sera possible d'utiliser n'importe quelle classe Java représentant un composant sans avoir à le recréer à chaque fois.

À chaque fois qu'un composant Arduino n'a pas d'équivalent Java, l'idée est de le créer et de l'ajouter à la bibliothèque. Ainsi, au fur et à mesure des projets, cette bibliothèque s'agrandira. Cependant, c'est une idée expérimentale et des propositions de modifications et d'améliorations sont données à la section 5.2.4.

Motivation

La motivation est de pouvoir gagner du temps durant le processus de développement de l'application en utilisant du code existant. Pour chaque application du Web des Objets, il est possible d'imaginer que les composants Arduino utilisés sont pratiquement les mêmes. De plus, créer à chaque fois les mêmes classes représentant les composants est un travail ennuyeux et inutile. Imaginez une dizaine de projets utilisant tous un composant de type moteur à rotation continue. Sans cette librairie, cela signifie qu'une dizaine de classes quasiment identiques sont créées, représentant toutes un moteur à rotation continue. De plus, si nous partons du principe qu'un développeur crée plusieurs applications et qu'un développeur est de nature flemarde, il aura tôt fait d'effectuer du copier-coller pour éviter de devoir écrire du code. Si le même développeur remarque qu'il manque une fonctionnalité ou un bug dans une des classes de composants, il devra modifier manuellement toutes les classes.

C'est pourquoi, notre solution permet de résoudre le problème posé ci-dessus. Comme tout le monde, dans tous les projets, utilisent les mêmes classes, un bug ou une fonctionnalité manquante dans une des classes peut être corrigé rapidement et être effectif dans tous les projets.

Coupler un logiciel de contrôle de version de type Git et rendre cette bibliothèque publique sur un site tel que Github, améliore encore sa facilité d'utilisation., permettant à chacun d'ajouter et de modifier une classe de composants. La dernière motivation pour cette

bibliothèque est le fait qu'un même code utilisé, testé et relu par un grand nombre de personnes ne peut être que meilleur que le code développé par un seul programmeur.

Utilisation

Comme il est prévu de distribuer cette bibliothèque sous forme de dépendance Maven, il suffit de l'inclure dans le fichier `pom.xml` puis d'utiliser les classes qu'elle contient pour interagir avec l'Arduino.

Pour ajouter ou modifier une classe de composant le processus est simple et commun à beaucoup de projets open-source sur Github :

1. Forker le projet
2. Modifier selon ses désirs le répertoire
3. Soumettre une requête pull

Puis les personnes responsables du répertoire Github, contrôlent les modifications, en discutent avec la personne ayant proposé des modifications et avec la communauté. Si la modification fait sens, la requête pull est accepté et les nouveaux changements sont disponibles pour tous les utilisateurs de la dépendance. La seule nécessité étant d'avoir une équipe de gestion du projet suffisamment grande, disciplinée et réactive pour répondre efficacement aux requêtes pull.

5.1.4. Simulation de l'Arduino

Au début, pouvoir simuler le fait qu'un Arduino soit connecté au serveur, nous semblait relever du domaine des rêves. Nous rêvions de ceci afin de s'affranchir de l'Arduino pour l'implémentation du service Web et de l'interface client du rideau de fer. Simuler un Arduino, nous permet également d'écrire toute une série de tests et une nouvelle façon de développer une application du Web des Objets. Au sujet des tests, la section 4.2.6 explique comment nous avons utilisé la simulation pour écrire des tests et la section 5.2.3 comment cela a été implémenté. Dans cette section, nous avons préféré nous concentrer uniquement sur la simulation de l'Arduino.

Motivation

La simulation de l'Arduino permet plusieurs opérations. Mais, dans notre cas, elle est principalement utilisée pour écrire des tests d'intégration. Cependant nous donnons ci-dessous des motivations plus générales que l'écriture de tests.

La première motivation est que la simulation d'un Arduino peut accroître de manière significative la productivité pour la réalisation complète d'un objet connecté au Web. Tout d'abord, cela permet aux développeurs du service Web d'écrire des tests et de là de choisir la méthode de développement Agile de leur choix. Surtout, cela permet une répartition du travail beaucoup plus efficace puisque l'implémentation du service Web et la programmation de l'Arduino peuvent être réalisées en même temps. Même pour le développement de l'interface client, il n'est pas nécessaire d'attendre que l'objet soit fonctionnel, du côté du service Web, il suffit de simuler la connexion avec l'Arduino pour avoir déjà une interface client pleinement fonctionnelle. Procéder de la sorte permet une

meilleure séparation des rôles et d'effectuer plusieurs tâches de développement en même temps.

La deuxième motivation est plus personnelle et découle de l'expérience directe acquise durant l'implémentation du rideau de fer. Très vite, il est apparu qu'une grande partie de notre temps de développement était accaparé par des manipulations de l'Arduino, des redémarrages du serveur et toujours les mêmes tests effectués manuellement. C'est pourquoi, écrire des tests pour notre application, ou des bibliothèques relatives à notre application, qui utilisent une simulation de l'Arduino représentent un gain de temps appréciable et diminuent le nombre d'erreurs. Pour le développement de notre interface client, il n'est plus nécessaire de connecter l'Arduino au serveur. Il suffit de simuler un Arduino depuis une petite interface graphique, programmée en Java, pour faire croire à l'interface client que le service Web est pleinement opérationnel. Ainsi, il est possible de se passer physiquement de l'Arduino ce qui est un avantage lors d'un déplacement par exemple.

Réalisation

Toute la magie de la simulation d'un Arduino vient du programme socat² qui permet de simuler des ports séries. Le principe est simple, nous créons deux ports séries connectés, appelés **master** et **slave**. Toutes les informations envoyées sur le premier port seront automatiquement disponibles sur le deuxième port et inversement. Normalement, une seule connexion par port série est autorisée. Mais l'utilisation de deux ports séries, dont les mêmes informations sont disponibles en même temps sur les deux ports permet en quelque sorte deux connexions sur un seul port. C'est exactement cette caractéristique qui nous intéresse, puisque nous avons besoin d'une connexion pour simuler un Arduino et d'une connexion utilisée par la bibliothèque ArduinoCommunication. Le diagramme 5.4 met en évidence ce principe.

Nous connectons le port **master** à la classe **HardwareSpeaker** permettant de simuler l'envoi et la lecture d'informations par port série. **HardwareSpeaker** possède deux méthodes importantes, une pour envoyer des informations sur le port **master**, qui seront disponibles sur le port **slave**. L'autre méthode pour permettre de lire les informations sur le port **master**, qui ont été envoyées par la bibliothèque ArduinoCommunication sur le port **slave** et donc disponibles sur le port **master**. Par exemple, quand nous ordonnons à **HardwareSpeaker** d'envoyer des informations sur le port **master**, les informations sont disponibles sur le port **slave**, faisant penser que l'Arduino a envoyé ces informations.

La simulation d'un Arduino est réalisée dans trois classes, qui sont représentées à l'aide du diagramme 5.5. La première, **ConnectionSimulator**, permet d'effectuer plusieurs actions :

1. Lancer le programme socat
2. Vérifier que socat ait démarré correctement
3. Ajouter des propriétés système
4. Factory pour fournir une unique instance de la classe **HardwareSpeaker**. Cette classe est responsable de se connecter à un port, d'envoyer et de recevoir des données de ce port.
5. Arrêter le processus socat

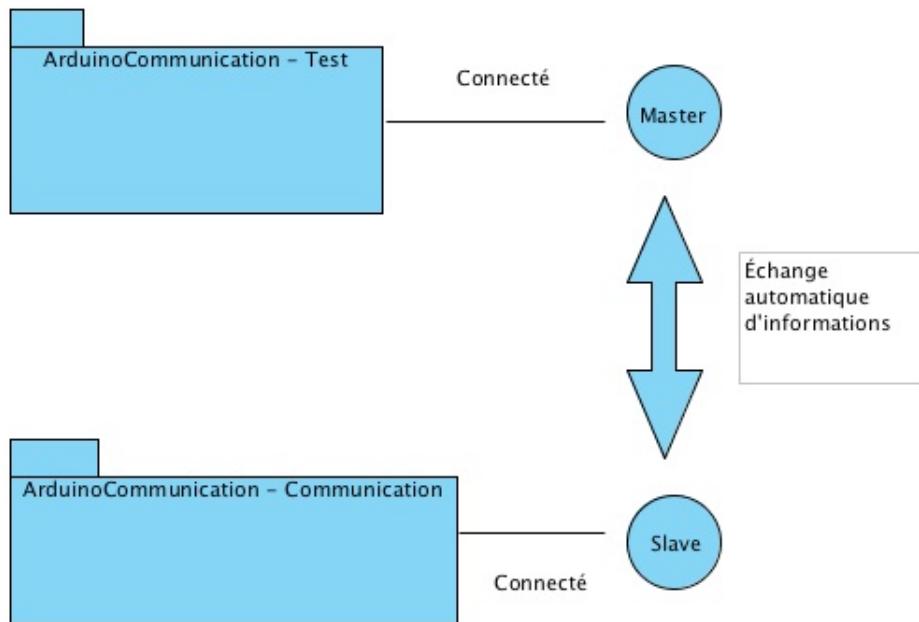


Figure 5.4. – Principe de simulation de connexions sur un port série

`ConnectionSimulator` est utilisé dans un contexte multi-thread, chaque méthode sensible utilise le mot-clé `synchronized` afin d’empêcher que deux threads accèdent en même temps à la même méthode. Socat est un programme disponible pour les systèmes d’exploitation Windows, Linux et Mac OS X. Cependant, son utilisation varie d’un OS à l’autre, c’est pourquoi dans `ConnectionSimulator` nous ne lançons pas socat de la même manière suivant que le code soit exécuté sous Mac OS X ou Linux. Pour Windows, le cas n’a pas été géré, puisque nous ne disposons pas de machine utilisant cet OS. Un défaut de notre implémentation est que socat a besoin d’être installé séparément par l’utilisateur et disponible dans le chemin d’exécution. Afin de vérifier que socat ait bien démarré, nous lisons les informations envoyées par socat sur le flux d’erreurs standard. Si le texte envoyé ne correspond pas à certains critères, ou qu’après cinq secondes aucun texte n’a été envoyé, nous lançons l’erreur `SocatNotStartedError` afin d’avertir l’utilisateur que socat n’a pas pu être démarré correctement. Par défaut, RxTx, la librairie utilisée en Java pour interagir avec les ports séries, cherche à se connecter uniquement à un nombre restreint de ports. La propriété système `gnu.io.rxtx.SerialPorts` permet d’indiquer à RxTx de chercher à se connecter à d’autres ports. La deuxième propriété système que nous donnons sert à indiquer à la classe responsable de la connexion à l’Arduino quel port utiliser. Il est encore utile de noter que la classe `ConnectionSimulator` peut s’utiliser comme Singleton ou non, suivant l’utilisation requise. La deuxième classe créée pour simuler une connexion par port série, `HardwareSpeaker`, offre quatre fonctionnalités :

1. Se connecter au port série dont le nom est passé en paramètre dans l’instance de la classe
2. Envoyer des données au port série
3. Lire des données du port série
4. Se déconnecter du port série précédemment connecté

²<http://www.dest-unreach.org/socat/>

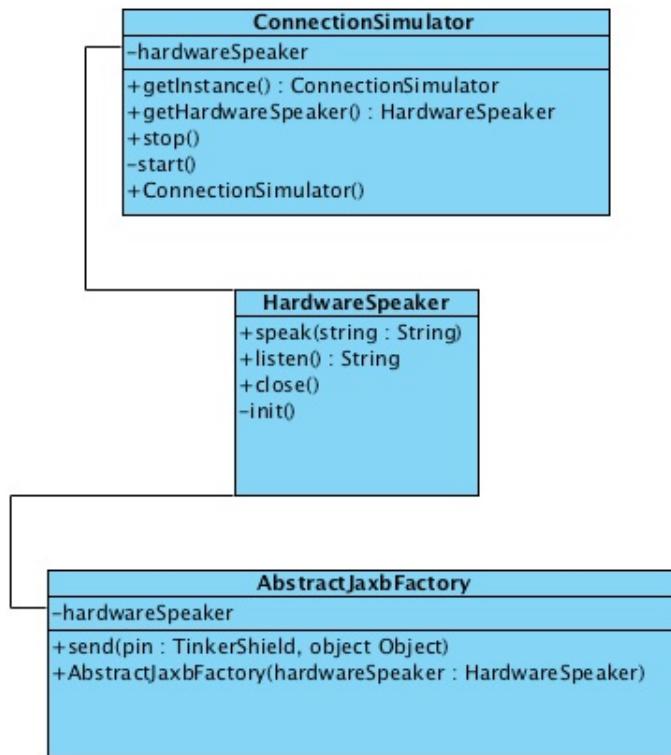


Figure 5.5. – Hiérarchie des classes utilisées pour la simulation

Pour cette classe, un point important est à relever. Durant le développement, nous avons remarqué que si deux messages sont envoyés dans un espace de temps très court au port série, le premier message ne sera jamais reçu et sera écrasé par le second. Pour pallier à ce problème, nous effectuons une boucle, durant au maximum cinq secondes, tant que le message n'a pas été envoyé au port série. Le listing 5.4 explique en pseudo-code comment cela est réalisé.

```

1 loop while the port is null and the message on the port is not equal to the given one
2   increment maxIteration by one
3   if maxIteration is bigger than 5000
4     break the loop
5   end if
6   wait 1ms
7 end loop
  
```

Listing 5.4 – Pseudo-code permettant de vérifier que le message est bien transmis au port

Les deux classes **HardwareSpeaker** et **ConnectionSimulator** sont généralistes et peuvent être utilisées pour simuler n'importe quelle connexion par port série. Cependant, la troisième classe, **AbstractJaxbFactory**, permet de simuler un Arduino. Le principe est simple, à chaque appel de la méthode `send`, l'objet Java passé en deuxième paramètre est encodé en un élément JSON représentant un composant. Comme nous l'avons vu à la section 5.1.1, chaque élément JSON représentant un composant doit être identifié par la valeur du pin auquel il est connecté. Pour ce faire, nous utilisons l'enum **TinkerShield** qui dans ce cas-là, retourne la valeur 9. Dans la méthode `send`, la chaîne de caractères ainsi obtenue est passée en paramètre de la méthode `speak` de **HardwareSpeaker**. Comme le montre le listing 5.5 une utilisation de cette classe peut être faite en simplifiant la création de Factory, qui étend de **AbstractJaxbFactory**, simulant différents états de l'Arduino.

```

1 public class JaxbTestFactory extends AbstractJaxbFactory{
2
3     public JaxbTestFactory(HardwareSpeaker hardware) {
4         super(hardware);
5     }
6
7     // simulate that the door is full unlocked
8     public void createLockOpen() {
9         // It is a class from the ArduinoComponents library
10        // which represent a linear potentiometer
11        LinearPotentiometer lp = new LinearPotentiometer();
12        lp.setPosition(0);
13        lp.setOldPosition(0);
14        // The first parameter is the value of the pin on
15        // which the linear potentiometer is connected
16        // lp will be converted to a string in the send method
17        // Finally, the JSON will be like this:
18        // {"14": {"oldPosition": 0, "position": 0}}
19        super.send(TinkerShield.o_1, lp);
20    }
21 }
```

Listing 5.5 – Factory pour simuler différents états de l’Arduino

Utilisation

L’utilisation est extrêmement simple, deux classes sont à notre disposition `HardwareSpeaker`, en remplacement de l’Arduino et `ConnectionSimulator`, utilisé pour gérer la simulation des deux ports, c’est-à-dire socat.

Tout d’abord, il est nécessaire de lancer la simulation de ports séries. Pour ce faire :

```
1 ConnectionSimulator simulator = new ConnectionSimulator();
```

Si pour une raison ou un autre, utiliser un Singleton est plus adapté :

```
1 ConnectionSimulator.getInstance();
```

Le premier appel à la méthode `getInstance` démarrera socat.

`HardwareSpeaker`, la classe pour simuler une connexion par port série, possède deux méthodes. La première, `speak(java.util.String)`, permet de simuler l’envoi de données sur le port série. Typiquement, elle permet de simuler un Arduino envoyant des données sur le port série. La deuxième méthode, `listen`, permet de lire les informations sur le port série. Dans ce cas, c’est l’inverse, cela peut par exemple simuler le fait qu’un Arduino lise des données qui ont été envoyées sur le port série.

Par exemple, pour envoyer un premier message à la place de l’Arduino :

```
1 simulator.getHardwareSpeaker().speak("Hello world!");
```

Ainsi l’application normalement connectée à l’Arduino et écoutant sur le port série approprié (pour la simulation le port série est indiqué dans la propriété système `xwot.test.port`), recevra `Hello world!`

A son tour, l’application envoie le message `Hello people!` et pour simuler le fait que l’Arduino lit l’information :

```
1 simulator.getHardwareSpeaker().listen() // return Hello people!
```

L'utilisation est simple et peut être adaptée à beaucoup de cas comme le montrent les exemples ci-dessus. Cependant, si la bibliothèque ArduinoComponents ou le principe de la représentation d'un composant Arduino par une classe Java est utilisée, le niveau d'abstraction devient plus élevé. En effet, la classe `JaxbAbstractFactory` permet véritablement de simuler l'envoi de JSON structuré, de la manière décrite à la section 5.1.1, par un Arduino. Cela s'effectue en quatre étapes :

1. Initialiser la classe `JaxbAbstractFactory`
2. Initialiser une classe représentant un composant Arduino. Typiquement une de celles contenues dans la bibliothèque ArduinoComponents
3. Assigner les valeurs souhaitées à l'objet créé au point deux
4. Utiliser la méthode `send` de `JaxbAbstractFactory` qui utilise la méthode `speak` de `HardwareSpeaker` afin de simuler l'envoi de JSON structuré à l'application

```
1 AbstractJaxbFactory fac = new AbstractJaxbFactory(simulator.getHardwareSpeaker());
2 LinearPotentiometer pot = new LinearPotentiometer();
3 pot.setPercentPosition(100);
4 fac.send(TinkerShield.i_0, pot);
```

La ligne numéro trois assigne la position cent (en pourcent), qui est ensuite transposée sur une échelle de 0 à 1023. La ligne numéro quatre encode l'objet `pot` en JSON et envoie la chaîne obtenue sur le port série de l'application grâce à la méthode `speak` de `HardwareSpeaker`. `TinkerShield.i_0` correspond au pin I0 de l'Arduino (valeur quatorze) et donc le JSON produit sera de la forme :

```
1 {14: {position: 1023, oldPosition: 0}}
```

Où 14 est la valeur numérique de I0, côté serveur stockée dans l'enum `TinkerShield` et permet donc d'identifier le composant Arduino avec précision, puisque cela indique sur quel pin de l'Arduino le composant de type potentiomètre linéaire est connecté.

Pour utiliser la classe `AbstractJaxbFactory`, nous recommandons la création d'une Factory qui étend cette classe, comme dans le listing 5.5. Par contre, ce que ne fait pas la classe `AbstractJaxbFactory`, c'est de pouvoir envoyer plusieurs objets dans une même chaîne de caractères JSON. La lecture de messages JSON envoyés par l'application est possible grâce à la méthode `listen` de `HardwareSpeaker`, mais cette méthode renvoie uniquement une chaîne de caractères et ne permet ni de parser le JSON ni de le transformer en instance de classe représentant un composant. Cependant, ceci pourrait être implémenté sans trop de mal en s'inspirant de la bibliothèque `ArduinoCommunication`.

5.1.5. Icwot

Icwot pour Inversion of Control for Web Of Things est un serveur Web basé sur le framework Sinatra³ qui démontre l'inversion de contrôle telle que définie dans le méta-modèle. C'est-à-dire qu'un serveur est mis en place côté client et est prêt à recevoir des requêtes HTTP POST du service Web auprès duquel il s'est enregistré (Webhook). Icwot a été développé grâce au langage Ruby et est distribué grâce au gestionnaire de bibliothèques Ruby, RubyGems⁴.

³<http://www.sinatrarb.com>

⁴<http://en.wikipedia.org/wiki/RubyGems>

Motivation

La motivation de réaliser un tel module était de démontrer avec quelle facilité il est possible d'implémenter l'inversion de contrôle et de le distribuer. Nous voulons également démontrer qu'il est possible d'implémenter avec facilité un serveur Web répondant à nos besoins. Si notre implémentation est extrêmement basique, c'est de par la volonté de permettre à n'importe qui de comprendre le code et de là, de développer une implémentation plus complète. Nous en reviendrons à la section 5.2.2.

Réalisation

Afin de distribuer au mieux notre travail, nous avons décidé de créer une gem, c'est-à-dire une bibliothèque empaquetée par RubyGems et facilement distribuable. La création est simple et très bien documentée. Une fonctionnalité intéressante est que RubyGems permet également la distribution de fichiers exécutables disponibles, après installation, dans le chemin d'exécution de l'utilisateur.

Icwot est structuré comme suit : le dossier `bin/` contient le fichier Ruby exécutable `icwot` qui permet de lancer `icwot` depuis le terminal. Le dossier `lib/` contient quatre fichiers nécessaires à l'implémentation :

1. Le fichier `server_client.rb` contenant la classe `ServerClient` responsable de l'implémentation du serveur Web
2. Le fichier `console.rb` pour l'interaction avec l'utilisateur
3. Le fichier `client.rb` pour enregistrer les informations du client et les encoder en XML ou JSON
4. Le fichier `version.rb` pour maintenir un numéro de version de la gem

Pour s'enregistrer vers le serveur afin de recevoir des notifications, la gem `RestClient` est utilisée. Si le code HTTP de la réponse renvoyée par le serveur est 200, nous estimons que l'enregistrement est réussi et nous démarrons le serveur Sinatra.

L'implémentation du serveur Sinatra est extrêmement simple, le listing 5.7 montre le code nécessaire pour que le serveur puisse recevoir une requête HTTP POST à l'URI / et l'enregistre dans un fichier de log.

```

1 post '/' do
2   # response is returned in text/plain
3   content_type 'text/plain'
4   # save in a special log the body of the received request
5   msg.info request.body.read
6   # write in main log (the console) that we have received a new notification
7   logger.info "message saved to #{self.class.logger_log_file.path}"
8   # return the HTTP code 200 with the text ok
9   'ok'
10 end

```

Listing 5.6 – Le serveur Sinatra reçoit des requêtes POST à l'URI /

Une des difficultés était de créer un logger adapté à nos besoins. Finalement, après avoir compris que Sinatra utilisait le logger de Ruby par défaut, contrairement à Rails, les choses allèrent beaucoup mieux. Nous avons défini que le format du logger pour enregistrer les notifications serait de la forme : date heure:break message, où date et heure sont la

date et l'heure à laquelle la notification est reçue. `break` indique un saut à la ligne et `message` est la notification reçue.

Quand l'application reçoit le signal `EXIT`, elle envoie une requête au serveur pour se désenregistrer de l'envoi de notifications et arrête le serveur Sinatra.

La dernière étape de notre implémentation est de rendre notre application utilisable par tout le monde. Il suffit de lancer la commande `gem build` qui prépare notre gem à être distribuée et la commande `gem push` qui pousse la gem créée sur le serveur rubygems.org et la rend utilisable par n'importe qui possédant Ruby et RubyGems d'installé.

Utilisation

L'usage et l'installation en sont très faciles. Une fois Ruby et RubyGems installés, lancer la commande

```
1 gem install icwot
```

qui va télécharger la gem, l'installer et ajouter l'exécutable `icwot` dans le chemin d'exécution. Puis, depuis n'importe quel répertoire, lancer la commande

```
1 icwot HOST
```

où `HOST` est l'URL de la ressource Web de laquelle nous souhaitons recevoir des notifications. Cette commande va :

1. Enregistrer le client par une requête HTTP POST sur l'URL fournie. Dans la requête POST une URL est transmise afin d'indiquer où recevoir des notifications. Par défaut `ip-du-client:4567/`
2. Lancer le serveur Sinatra avec le port 4567 par défaut. Les notifications reçues vont être enregistrées dans le dossier `log/icwot/` du répertoire de l'utilisateur

Par défaut, le header de la requête POST pour s'enregistrer est `accept:application/json ;content_type:application/json`. Il est possible de le changer en ajoutant les arguments `-c le-content-type` et `-a le-accept`. Où `le-content-type` et `le-accept` peuvent chacun être soit `json` soit `xml`. Donc suivant la valeur de l'argument `le-accept`, le corps de la requête sera encodé de deux manières :

1. En JSON :

```
1 {url:xx.xxx.x.x:4567/}
```

2. En XML :

```
1 <client xmlns="http://jaxb.xwot.first.ch.unifr.diuf">
2   <uri>xx.xxx.x.x:4567/</uri>
3 </client>
```

De même que le serveur encode les données en XML si `le-content-type` est égal à `xml`, sinon en JSON.

Il y a également différentes options pour la ligne de commande :

```
1 Usage : icwot <host>
2           -h print help
3           -l the host is localhost
4           -c the content-type value for the header application/json by default
5           -a the accept value for the header text/plain by default
```

```

6   -p the port where to run the server
7   -t the protocol to use http:// by default
8   -o where to save the log. By default your-home-directory/log/icwot-{port
9       }-msg.log
      host is the URL of the resource to register for the service.

```

Listing 5.7 – Les différentes options de l'exécutable icwot

5.1.6. Archétype Maven

Un archétype est une fonctionnalité de Maven qui permet de générer un projet avec un certain nombre de fichiers et de dossiers déjà existants. Ainsi, pour chaque même type de projet, il n'est plus nécessaire de repartir de zéro. Pour l'application du rideau de fer, nous sommes également partis d'un archétype Maven. Cet archétype génère un projet Jersey adapté au développement d'applications Web RESTful utilisant le framework JPA et JAXB. Comme une application RESTful du Web des Objets est quelque peu différente d'une application classique, nous avons défini notre propre archétype Maven.

Définir un archétype permet deux choses. Premièrement, cela offre un gain de temps appréciable en fournissant une application prête à être utilisée. Aucune dépendance ne doit être ajoutée dans le POM, la configuration est déjà faite et des fichiers d'exemples sont déjà là, qui peuvent être utilisés comme source d'inspiration. Deuxièmement, cela offre une structure adaptée pour le type d'application à réaliser.

Structure de l'archétype

Dans notre cas, l'archétype que nous avons créé propose une structure quelque peu modifiée. En plus des classes JAXB et des classes de ressources que l'on trouve dans presque toute application Jersey, nous avons ajouté :

- le package notifications dans lequel se situent les classes faisant partie du pattern builder des notifications de la bibliothèque `ArduinoCommunication`. Une classe faisant office de Factory pour les différents types de notification est déjà créée.
- le package mapper est censé accueillir toutes les classes effectuant la transformation entre une classe représentant un composant Arduino en une classe JAXB.
- la classe `Listener` du paquet `monitoring.listener` qui gère la connexion à l'Arduino, ou la simulation de l'Arduino. Cette classe implémente un observateur appelé juste avant que le serveur ait fini de démarrer.
- Le fichier `pom.xml` contient toutes les dépendances et plugins Maven nécessaires à l'application. De même que toute la configuration standard est déjà effectuée.
- Pour chaque élément de l'application (classe de ressources, JAXB, notifications, etc) un exemple est donné.

Nous n'avons pas inclus de package pour les classes représentant un composant de l'Arduino, car nous partons du principe que ces classes devraient se trouver dans la dépendance `ArduinoComponents`.

Utilisation

La création d'un projet Maven avec cet archétype et la génération des classes de ressources et des classes JAXB se fait en cinq étapes :

1. Lancer la commande pour créer un nouveau projet avec un des archétype développé :

```
1 mvn archetype:generate -DarchetypeCatalog=http://diufpc46.unifr.ch/artifactory/  
ext-release-local
```

Attention ! Il est important de ne pas ajouter un slash à la fin de l'URL et il est requis d'utiliser le VPN ou d'être connecté au réseau de l'université de Fribourg.

2. Deux archétypes sont proposés, choisir le premier (RESTArduino-archetype).
3. Fournir les informations demandées. Uniquement `groupId` (le nom de base des packages) et `artifactId` (le nom du projet) sont obligatoires.
4. Dans le dossier `nom-du-projet/src/main/resources/WADL` remplacer le fichier `application.wadl` par celui obtenu grâce à la modélisation et son propre fichier XSD.
5. Lancer la commande :

```
1 mvn clean compile package
```

6. Copier les classes de ressources et les classes JAXB depuis `nom-du-projet/target/generated-sources` respectivement vers les paquets finissant par `ressources` et `jaxb` situé dans le dossier `nom-du-projet/src`

Et voilà, il ne reste plus qu'à développer l'application !

Typiquement, le développement d'une ressource d'une application se déroule comme suit :

1. Implémenter une ou plusieurs classes dans le package mapper pour transformer une classe de ArduinoComponents vers une classe JAXB et inversement.
2. Implémenter les différentes méthodes de la ressource. Cela consiste généralement à interagir avec l'Arduino et est souvent très similaire à l'exemple fourni. De plus, il est recommandé d'écrire des tests d'intégration utilisant la simulation de l'Arduino.
3. Implémenter les méthodes du publisher, à nouveau il est possible de s'inspirer de l'exemple correspondant et il est recommandé d'écrire des tests d'intégration.
4. Implémenter les méthodes `hasNotification` et `jaxbToStringEntity` d'une nouvelle classe étendant de NotificationBuilder afin de créer une nouvelle notification.
5. Ajouter une méthode à la classe `Factory` du package `notifications`, afin de pouvoir utiliser toujours la même instance de la classe précédemment créée.

Voilà. Une nouvelle ressource avec publisher et envoi de notifications est créée !

5.2. *Futurs développements*

Comme tout travail, le notre n'est jamais totalement achevé. Chaque résultat, obtenu donnant de nouvelles idées à développer. Malheureusement, comme notre travail nécessite bien une fin, nous n'avons pas pu développer toutes les idées qui nous sont apparues durant ces neuf mois. C'est pourquoi, dans les sections suivantes, nous développons brièvement ce qui pourrait encore être fait.

5.2.1. Amélioration de la bibliothèque de tests

Durant l'implémentation des notifications du rideau de fer, nous avons rencontré quelques bugs pour l'envoi de notifications dont la résolution fut difficile. De plus, il manque actuellement des tests pour l'envoi et la rapidité d'envoi des notifications pour implémenter un envoi des notifications plus rapide et multi-thread.

À notre sens, ce qui manque, c'est un serveur HTTP du même type que icwot, mais utilisable pour l'écriture de tests. Ainsi, pour l'écriture de tests des notifications nous utiliserions trois modules. 1. Simulation de l'Arduino 2. Serveur Jetty pour interroger le service Web, le même que pour les tests d'intégration 3. Serveur client du type icwot pour tester si le service envoie bien des notifications.

Les fonctionnalités d'un serveur du type icwot, appelons-le serveur de notifications, comporteraient divers aspects :

- S'enregistrer auprès d'une application pour recevoir des notifications
- Recevoir des requêtes HTTP POST
- Stocker ces données et fournir une API pour les manipuler. Obtenir une notification envoyée selon divers critères (timestamp, type, format, etc), obtenir la dernière notification, obtenir divers informations d'une notification et sûrement d'autres fonctionnalités.

Dans l'idéal, il serait possible de tester l'envoi de notifications à un nombre élevé de personnes et de mesurer le temps et les ressources nécessaires à l'envoi des notifications. Le listing 5.8 donne un exemple de comment des tests d'envoi de notifications pourraient être écrits. Ceci n'a pas été implémenté.

```

1  @Test
2  public void testLockNotification() {
3      NotificationServer notificationServer = new NotificationServer();
4      notificationServer.registerFor('/door/lock/pub');
5      JaxbTestFactory fac = new JaxbTestFactory(ConnectionSimulator.getInstance().
6          getHardwareSpeaker());
6      fac.createLockOpen();
7      assertEquals(notificationServer.getLastNotification().getBody());
8 }
```

Listing 5.8 – Ce à quoi pourraient ressembler les tests d'envoi de notifications.

5.2.2. Implémentation du concept des publisher dans un navigateur web

Toujours dans le domaine des notifications, une implémentation intéressante pourrait être d'étendre les fonctionnalités des navigateurs Web. L'idée est de permettre l'utilisation des publishers dans une page Internet. C'est-à-dire, permettre qu'un serveur Web puisse modifier le contenu d'une page Web sans que le client n'ait effectué de requêtes. L'exemple type est l'implémentation d'un système de chat. Dans ce cas-là, il est souhaitable que l'utilisateur voit directement la réponse de son correspondant sans qu'il ne doive recharger la page. Actuellement, deux possibilités d'implémentation s'offrent au développeur. La première consiste à utiliser un langage du type Flash ou Java s'exécutant du côté client. Dans ce cas-là, un système de socket est généralement implémenté, ce qui équivaut

au websocket en Javascript. L'autre solution consiste à implémenter en Javascript un système de poling, long-poling ou websocket. Cependant ces deux façons de faire comportent des désavantages. L'utilisation de socket n'est pas adaptée pour un grand temps d'attente entre deux envois de notifications et le polling ou long-polling génèrent une grande quantité de requêtes vers le serveur.

Notre proposition se base sur le fait que les navigateurs Firefox, Safari et Chromium (Chrome) possède tous un système de plugins permettant d'étendre leurs possibilités et de créer des applications semblant être natives. Par exemple, pour Chrome, comme l'illustre l'image 5.6, Sunrise Calendar⁵ est une application utilisant Chrome mais semblant être native.

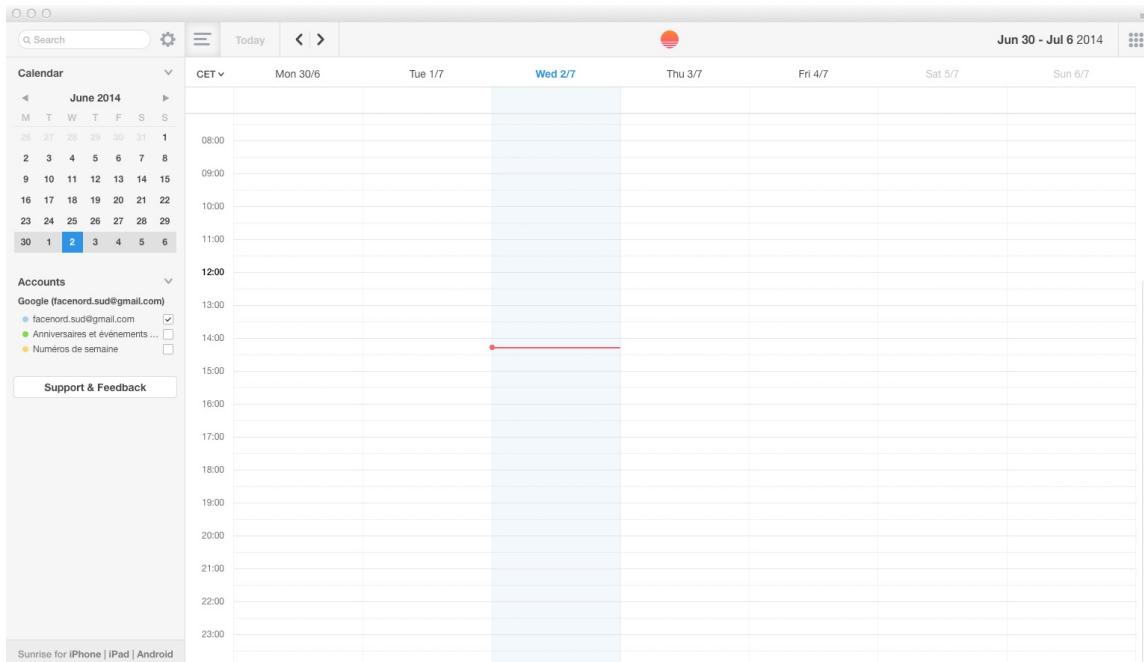


Figure 5.6. – L'application Sunrise Calendar utilisant Chrome

Ainsi, il serait possible d'implémenter une application dédiée au Web de Objets qui soit en fait un petit serveur Web du même type que icwot. Ce serveur pourrait gérer de différentes manières les notifications reçues suivant le format dans lequel elles sont envoyées. Par exemple les notifications reçues en un langage du type JSON, XML ou HTML seraient rendues accessibles à d'autres programmes Javascript et un événement serait lancé à la réception de chaque notification. Ceci permettrait de développer un script Javascript qui écouterait sur l'événement « réception de notification » et, quand un événement se produit, utiliserait les données de la notification pour modifier le contenu de la page Web. Au niveau de la réalisation, il serait tout à fait possible d'imaginer une solution avec node.js⁶. De même que le client devrait installer une première fois l'application (en général un ou deux clics de souris) et ensuite il ne devrait plus rien faire pour la gérer.

A noter qu'il existe déjà les notifications HTML5 [21] qui utilisent le même principe, mais pour le moment la gestion de ces notifications est extrêmement basique.

⁵<https://chrome.google.com/webstore/detail/sunrise-calendar/mojepfklcankkmikonjlnidioanmpbb>

⁶<http://nodejs.org>

5.2.3. Communication entre l'Arduino et l'application Web

La façon de communiquer entre le serveur et l'Arduino que nous avons développée est adaptée à notre cas d'utilisation mais pourrait être améliorée dans le cas d'événements continus. Actuellement, dans le cas d'une mesure continue, l'Arduino n'a pas d'autres choix que d'envoyer tous les X temps la dernière mesure effectuée. Ceci a pour conséquence l'envoi d'une grande quantité d'informations envoyées depuis l'Arduino vers l'application sans pour autant garantir que toutes les informations envoyées soient utilisées.

La solution proposée est adaptée pour les cas de mesures discrètes et continues et réduit de beaucoup le nombre d'échanges. L'idée est que l'application puisse ordonner à l'Arduino l'envoi ou non de messages. Par défaut, l'Arduino envoie des messages à l'application tant qu'il n'a pas reçu un message lui ordonnant d'arrêter. Une fois qu'il a reçu l'ordre d'arrêter, il n'enverra plus aucun message tant qu'il n'a pas reçu l'ordre d'envoyer des messages. Du côté de l'application, elle ordonne à l'Arduino l'arrêt d'envoi des messages uniquement si, une fois un message reçu, l'avant-dernier n'a pas encore été lu. De même que l'application ordonne à l'Arduino d'envoyer à nouveau des messages seulement si le dernier message reçu a été lu. Par exemple, nous obtenons le scénario suivant :

1. Du côté de l'Arduino, dès la première donnée disponible devant être envoyée, l'Arduino envoie le message requis à l'application.
2. L'application reçoit l'information et la stocke en attente d'être lue.
3. L'Arduino dispose de nouvelles données, envoie un message à l'application et l'application lui répond que le premier message n'a pas encore été lu.
4. l'Arduino possède de nouvelles données mais n'envoie rien car le message précédent n'a pas été lu.
5. Le message est lu côté application. A ce moment-là, l'application envoie un message à l'Arduino lui notifiant que le message a été lu.
6. L'Arduino dispose de nouvelles données et les envoie à l'application.
7. L'information est lue côté application.
8. L'Arduino dispose de nouvelles données et les envoie à l'application.
9. La nouvelle information envoyée par l'Arduino est lue côté application.
10. et ainsi de suite.

Notre façon de procéder est beaucoup plus économique en informations envoyées et peut tout aussi bien être utilisée pour des mesures continues. Pour autant que du côté de l'application une façon de quand envoyer des événements, à partir d'une mesure discrète, soit définie. Par exemple, grâce à un langage de gestion des événements comme décrit à la section 5.2.5. Si le service Web est très fréquenté, chaque message envoyé par l'Arduino est lu et donc l'échange d'informations est optimal (un message envoyé pour une information devant être transmise). Le pire cas possible, qui est assez improbable qu'il se produise au long terme, nécessite l'envoi de cinq messages pour trois informations à envoyer. Le cas des mesures continues est le même que le cas des mesures discrètes puisque les informations sont envoyées uniquement en cas de besoin.

5.2.4. Modélisation de composants

Nous avons observé durant le développement du rideau de fer, que le code pour représenter un composant Arduino est simple et est probablement identique dans beaucoup de projets,

d'où l'idée de la bibliothèque ArduinoComponents. Cependant, une meilleur solution existe. En s'inspirant des langages comme le WADL ou le XSD utilisés pour générer des classes Java, il devrait être possible de développer un langage du même genre pour modéliser chaque composant de l'objet, puis, à partir de là, générer des classes Java et du code Arduino les représentant.

De cette manière, une étape supplémentaire serait introduite dans le processus de modélisation. En plus de modéliser un objet du point de vue de ses devices, il serait possible de modéliser chaque device de l'objet. Par la suite, il serait également possible de modifier le code de la bibliothèque ArduinoCommunication afin de permettre la modélisation de devices imbriqués. C'est-à-dire que le premier device contient un senseur de type simple, ce device, avec d'autres, est contenues dans un device principal et ainsi de suite sans limite du niveau d'imbrication.

Procéder de cette manière, permet de structurer de manière beaucoup plus efficace l'information échangée entre un Arduino et une application Web, particulièrement dans le cas de systèmes importants. Nous pourrions même imaginer que plusieurs Arduino soient connectés à l'application sans que cela ne pose de problème. De plus, grâce à une génération automatique de code, le développeur peut se concentrer sur trois aspects essentiels de l'application. 1. La modélisation 2. Le développement des ressources 3. Le développement d'algorithmes pour l'Arduino. Par exemple, pour la machine à caramel, développer un algorithme qui détermine quand le mélange est prêt.

Ainsi, nous pourrions pousser encore plus loin la modélisation d'une application du WoT et permettre au développeur de se concentrer sur des tâches réellement importantes, sans devoir maîtriser un nombre de technologies importantes ni tout le processus de A à Z.

5.2.5. Langage de gestion des événements d'un publisher

Dans la gestion des notifications, un point important à améliorer est de donner la possibilité au client de décider de lui-même quand recevoir des notifications. En effet, il est aisés d'imaginer des cas où chaque client enregistré veut recevoir des notifications pour un certain type d'événements. Par exemple, pour la mesure d'une température, un client peut vouloir recevoir une notification pour chaque changement de degré, tandis qu'un autre peut vouloir recevoir une notification uniquement quand la température atteint un certain seuil. Dans un cas plus concret et plus proche de ce que nous avons implémenté, nous pouvons imaginer l'employé d'un magasin qui veut recevoir une notification quand le rideau de fer est descendu avant l'heure de fermeture, tandis que le gardien de nuit veut recevoir une notification quand le rideau est ouvert durant les heures de fermeture du magasin.

Une façon de résoudre le problème posé, serait d'implémenter un langage simple que le client pourrait utiliser pour décider quand recevoir une notification. Puis le code du client serait utilisé par le serveur pour déterminer si ce client doit recevoir une notification pour un événement donné.

6

Conclusion

Du point de vue du projet en lui-même, il me semble évident que les objets connectés seront dans les cinq à dix prochaines années une réalité. De plus, de grandes compagnies informatiques tendent à proposer leurs propres solutions. Google dans le domaine de la domotique et avec les Google Glass ; Apple dans le domaine de la domotique et de la santé par exemple. C'est pourquoi, chercher à présenter une solution open-source, elle-même basée sur des technologies open-source et des standards reconnus me semble important. De ce point de vue, le Web des Objets me paraît être une bonne solution.

Concernant le xWoT méta-modèle, il me semble qu'il est parfaitement adapté à développer des applications du Web des Objets. En effet, comme nous l'avons démontré dans les différents cas d'utilisation et bibliothèques, il offre de belles possibilités de développer des solutions pour connecter facilement un objet au Web. Basé sur ce méta-modèle, la création de frameworks, de bibliothèques et de différents outils est tout à fait possible et même facile. À notre avis, c'est une belle possibilité de développement pour proposer des outils en relation avec le WoT.

Ce travail a été un vrai enrichissement personnel. J'ai pu passer de la programmation d'un Arduino à celle d'un service Web, tout en construisant une porte, en imaginant une machine à caramels, en apprenant plus sur les systèmes d'aquaponie, en implémentant une simulation d'un Arduino, en réfléchissant à protocole binaire de communication et en imaginant encore une foule de choses qui resteraient à inventer. Comme vous le constatez, la liste est longue et vous êtes bien courageux si vous êtes arrivés au bout de la liste et de ce travail. Tout cela pour exprimer la grande liberté que j'ai eu en réalisant ce travail et le nombre de choses qu'il m'a été donné d'apprendre.

Dans ce travail, le point le plus important pour moi aura été tout ce que j'ai appris. Non seulement en programmation et en informatique, mais également dans d'autres domaines telles que la construction d'une porte et la gestion d'un projet de plus grande envergure. La plus grosse difficulté dans ce travail aura été pour moi l'écriture. D'une part parce que c'est bien plus difficile d'expliquer ce que l'on fait que de le réaliser et d'autre part, parce qu'il a fallu m'astreindre à avancer régulièrement. Le dernier point à relever est que même si l'on fait plein de choses, cela n'est qu'à moitié réalisé tant que ce n'est pas été posé par écrit !

A

Common Acronyms

AJAX	Asynchronous JavaScript And XML
ANSI	American National Standards Institute
CERN	European Organization for Nuclear Research
CSS	Cascading Style Sheet
CGI	Common Gateway Interface
cURL	Client for URL
DBMS	Database Management System
DOM	Document Object Model
ERM	Entity Relationship Model
HL7	Health Level 7
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
JAXB	Java Architecture for XML Binding
JAX-RS	Java API for RESTful Web Services
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JSP	Java Server Pages
JSTL	Java Server Tag Library
NCSA	National Center for Supercomputing Applications
PHP	Hypertext Processor
POJO	Plain Old Java Object
POM	Project Object Model
QR	Quick Response
RFID	Radio Frequency Identification
REST	Representational State Transfer
ROA	Resource Oriented Architecture
SAX	Simple API for XML
SOA	Service Oriented Architecture
SQL	Structured Query Language

SPDY	SPeeDY, an open networking protocol
TCP	Transmission Control Protocol
URI	Unified Resource Identifier
URL	Uniform Resource Locator
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WSDL	Web Service Description Language
WoT	Web of Things
XML	eXtensible Markup Language
XSD	XML Schema Definition

B

License of the Documentation

Copyright (c) 2014 Numa de Montmollin.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation ; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from [20].

C

CD-Rom

Structure du contenu

```
/  
  arduino  
  bachelor.pdf  
    door  
      door.ino  
  clients  
    DoorClient  
    icwot  
  door-construction  
  modelisation  
    Aquaponie  
    CaramelMachine  
    Door  
  presentation  
  request.json  
  server  
    apps  
      CaramelMachine  
      FirstXwot  
    archetype  
    depedencies  
      ArduinoCommunication  
      ArduinoComponents  
      RxTx-Rebundled
```

C.1. Description rapide

arduino/door/door.ino

Est le fichier nécessaire à la programmation de l'Arduino pour le rideau de fer.

bachelor.pdf

Est le fichier PDF que vous lisez actuellement.

DoorClient

Contient le code nécessaire au fonctionnement du client Web.

icwot

Contient le code pour le fonctionnement d'icwot (Inversion of Control for Web of Things).

door-construction

Contient des photos du rideau de fer construit en meccano.

modelisation

Contient les différents fichiers utilisés pour modéliser les trois cas d'utilisation.

presentation

Contient un Powerpoint et un PDF de la présentation orale de ce travail,

request.json

Est un fichier pouvant être importé dans DHC (Dev HTTP Client), un plugin de Chrome.
Il contient les différentes requêtes possibles pour le service Web du rideau de fer.

CaramelMachine

Contient le code généré automatiquement à partir de la modélisation de la machine à caramels.

FirstXwot

Contient le code nécessaire au fonctionnement du service Web du rideau de fer.

archetype

Contient l'archétype Maven que nous avons défini.

depencies

Contient les trois dépendances Maven que nous avons créées. Pour RxTx-Rebundled, nous y avons légèrement participé.

Bibliographie

- [1] R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1*. 1999. RFC 2616.
- [2] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 8
- [3] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
- [4] Marc Hadley. Web application description language (wadl). Technical Report TR-2006-153, Sun Microsystems, April 2006.
- [5] Ian Jacobs. Uris, addressability, and the use of http get and post. World Wide Web Consortium, TAG Finding, March 2004.
- [6] Andreas Meier. *Relationale und postrelationale Datenbanken*. eXamen.press. Springer, Berlin [u.a.], 6., überarb. und erw. edition, 2007.
- [7] Goestenmeyer Ralph and Rehn-Goestenmeier Gudrun. *Das Einsteigerseminar Linux*. Verlag moderne industrie Buch AG & Co. KG, Landsberg, Königswinterer Str. 418, 35227 Bonn, Deutschland, 4., überarb. edition, 2003.
- [8] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.
- [9] Andreas Ruppen and Jacques Pasquier-Rocha. A Meta-Model for the Web of Things. Internal Working Paper 13-03, Department of Informatics, University of Fribourg, Switzerland, June 2013.
- [10] Alexander Schatten. *Best Practice Software-Engineering*. Springer DE, 2010.

Sites Web

- [11] aJson - handle JSON with arduino. <http://interactive-matter.eu/blog/2010/08/14/json-handle-with-arduino/> (dernière consultation le Mai 26, 2014). 33
- [12] cJSON. <http://sourceforge.net/projects/cjson/> (dernière consultation le Mai 26, 2014). 33
- [13] Présentation du système d'aquaponie en container. <http://fr.ulule.com/aquaponie/> (dernière consultation le Juin 09, 2014). 19
- [14] Schéma d'un système d'aquaponie. <http://aquaponie-pratique.com/wp-content/uploads/2014/02/aquaponie-avec-siphon-et-reservoir.jpg> (dernière consultation le Mai 14, 2014). vi, 19
- [15] Arduino Homepage. <http://arduino.cc> (dernière consultation le April 30, 2014). 2, 10
- [16] [Arduino] Parse Json efficiently. <http://blog.benoitblanchon.fr/arduino-json-parser/> (dernière consultation le Mai 26, 2014). 33
- [17] ArduinoJsonParser. <https://github.com/bblanchon/ArduinoJsonParser> (dernière consultation le Mai 26, 2014). 33
- [18] The Internet of Things How the Next Evolution of the Internet Is Changing Everything. http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf (dernière consultation le April 04, 2014). 2
- [19] Factory_girl. https://github.com/thoughtbot/factory_girl (dernière consultation le Mai 26, 2014). 42
- [20] Free Documentation Licence (GNU FDL). <http://www.gnu.org/licenses/fdl.txt> (dernière consultation le July 30, 2005).
- [21] Démonstrations des notifications en HTML5. https://developer.cdn.mozilla.net/media/uploads/demos/e/1/elfoxero/c17223c414d8ddafb7808972b5617d9e/html5-notifications_1400214081_demo_package/index.html (dernière consultation le Juillet 02, 2014). 77
- [22] RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1. <http://tools.ietf.org/html/rfc2616> (dernière consultation le April 30, 2014). 6, 7
- [23] The HTTP Protocol As Implemented in W3c. <http://www.w3.org/Protocols/HTTP/AsImplemented.html> (dernière consultation le April 30, 2014). 7

- [24] RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0. <http://tools.ietf.org/html/rfc1945> (dernière consultation le April 30, 2014). 7
- [25] Communicating With Things – An Energy Consumption Analysis. <http://hal.archives-ouvertes.fr/docs/00/80/04/71/PDF/Poster.pdf> (dernière consultation le April 30, 2014). 6
- [26] Codes de réponse HTTP. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html> (dernière consultation le May 07, 2014). 9
- [27] HTTP/1.1 :Method Definitions. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html> (dernière consultation le April 30, 2014). 9
- [28] That 'Internet of Things' Thing - RFID Journal. <http://www.rfidjournal.com/articles/view?4986> (dernière consultation le April 04, 2014). 5
- [29] Jarduino. <https://github.com/SINTEF-9012/JArduino> (dernière consultation le Mai 26, 2014). 35
- [30] Loi de Moore - Wikipedia. http://fr.wikipedia.org/wiki/Loi_de_Moore (dernière consultation le April 04, 2014). 4
- [31] Le mouvement global d'exode rural fait entrer la faim dans les villes. http://www.lemonde.fr/planete/article/2009/03/16/le-mouvement-global-d-exode-rural-fait-entrer-la-faim-dans-les-villes_1168424_3244.html (dernière consultation le Juin 09, 2014).
- [32] Raspberry Pi. <http://www.raspberrypi.org> (dernière consultation le April 30, 2014). 2
- [33] Image d'un Raspberry PI. <http://commons.wikimedia.org/wiki/File:RaspberryPi.jpg> (dernière consultation le April 30, 2014). vi, 12
- [34] The Computer for the 21st Century. <http://www.ics.uci.edu/~corps/phaseii/Weiser-Computer21stCentury-SciAm.pdf> (dernière consultation le April 04, 2014). 5
- [35] Exemple d'utilisation d'un moteur à rotation continue. <http://www.tinkerkit.com/intro-to-servos/> (dernière consultation le Mai 26, 2014). 32
- [36] Website of the Sun Microsystems. <http://www.sun.com/software/solutions/rfid/> (dernière consultation le July 28, 2005).
- [37] Architecture of the World Wide Web, Volume One. <http://www.w3.org/TR/2004/REC-webarch-20041215/> (dernière consultation le April 30, 2014). 6
- [38] World Wide Web - Wikipedia, the free encyclopdia. http://en.wikipedia.org/wiki/World_Wide_Web (dernière consultation le April 30, 2014). 6