heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Travail de diplôme

Remote monitoring using iCelsius Wireless

Département : TIC

Orientation : IE

Auteur : **Tinguely Joël**

Professeur : **Robert Stephan**

Date : **01 août 2013**

1 Table des matières

2 Avant-propos

Ce document est le rapport d'un travail de Bachelor effectué par Joël Tinguely entre le 05 mars 2013 et le 01 août 2013. Ce travail est effectué pour Aginova sous la direction du Dr. Stephan Robert, professeur à la Haute Ecole d'Ingénierie et de Gestion du canton de Vaud (HEIG-VD).

Le travail sur ce projet s'est déroulé en deux étapes différentes. La première partie a eue lieu au court du deuxième semestre de troisième année d'étude à la HEIG-VD, à Yverdon-les-Bains (CH). Lors de ce semestre, deux jours par semaine étaient consacrés au travail de Bachelor dont un jour par semaine dans les locaux d'Aginova, au parc scientifique de l'EPFL. Cette étape s'est déroulée du 05 mars 2013 au 21 juin 2013.

La deuxième partie s'est effectuée en collaboration avec une université californienne qui accueil des étudiants pour ce type de projet, San José State University (SJSU). Cette étape de six semaines s'est déroulée dans la ville de San José en Californie (USA), sur le campus de SJSU. Un mois et demi était alors consacré à plein temps au travail de Bachelor. Cette étape a eu lieu du 22 juin 2013 au 01 août 2013.

Une défense orale du travail sera ensuite présentée en Suisse, dans le courant du mois de septembre 2013.

3 Résumé

Le but du travail est de fournir un système fonctionnel permettant la visualisation de valeurs provenant de sensors sur un appareil mobile fonctionnant avec Android. Pour ce faire, le projet comportera trois parties distinctes. La première est l'implémentation d'un serveur permettant d'avoir une interface web et comportant des bases de données. La deuxième partie est la communication entre le serveur web et les sensors. Enfin, la troisième partie de ce projet est la conception d'une application Android permettant à l'utilisateur de voir ses données depuis son smartphone ou autre device Android.

La première partie de ce travail utilise Google App Engine afin de gérer plus facilement le serveur web et les bases de données. Ce service est une solution de Cloud computing qui nous permet d'ajouter une couche d'abstraction au projet. Ceci nous apporte des facilités telles que la mise en place et la gestion automatique des ressources ou un grand nombre d'APIs fournies par Google. Le rôle de ce serveur est de récolter les informations envoyées par les capteurs et de les stocker. Ces données seront ensuite récupérées en temps voulu par l'utilisateur. De plus, cette application propose une interface web sur laquelle les utilisateurs pourront gérer leurs sensors et observer les données reçues en directe. Un point important de cette partie du travail est l'implémentation des bases de données. Effectivement ce sont elles qui seront la mémoire de notre application. Si elles sont mal pensées lors de leur conception, toute l'application en sera pénalisée.

La deuxième partie du projet consiste à implémenter le bloc communication sur le sensor lui-même. Cette étape est beaucoup plus légère que les deux autres. Toute la gestion de capture des données est déjà mise en place. Il ne reste alors qu'à envoyer ces données sur notre serveur web. Cette communication se fait avec le protocole HTTP afin de ne pas avoir de problème de firewall.

La dernière partie du projet est la conception de l'application Android permettant aux utilisateurs de visualiser les données provenant des sensors directement sur leurs smartphones. Il est nécessaire de mettre en place la communication entre l'appareil mobile et le serveur web. De plus, une partie graphique est nécessaire afin que l'application soit agréable d'utilisation. Le développement sur Android nécessite de comprendre le fonctionnement du système d'exploitation lui-même. Une grande partie du travail consiste alors à apprendre le fonctionnement d'Android.

4 Description du projet

4.1 Présentation du produit iCelsius Wireless

Le but de ce projet est de mettre en place un système de communication à distance avec iCelsius Wireless. Le produit iCelsius Wireless est un dispositif développé par Aginova permettant la mesure de différents types de valeurs via diverses probes et l'envoi de cette information à un utilisateur distant. Ce produit peut fonctionner selon trois modes différents, décrits dans les paragraphes qui suivent.

Le premier mode mis à disposition par iCelsius Wireless est le mode direct. Dans cette configuration, le sensor agit comme point d'accès wifi. L'utilisateur peut directement se connecter sur iCelsius Wireless avec son smartphone ou son PC afin d'acquérir les informations récoltées par la probe. Ce mode ne nécessite pas de matériel autre que le produit iCelsius Wireless et un appareil mobile avec une carte wifi. La Figure 1 ci-dessous représente le produit iCelsius Wireless (*Sensor*) transmettant des informations à l'utilisateur (*Mobile device*).

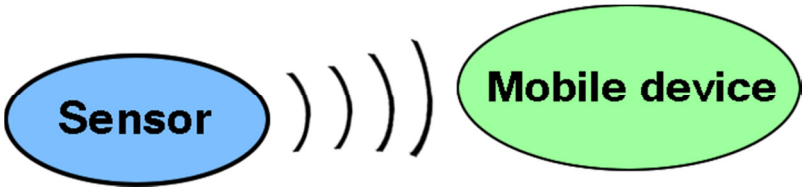


Figure 1: iCelsius Wireless, mode 1

Le deuxième mode possible avec iCelsius Wireless est le mode infrastructure locale. Lorsque le sensor est configuré dans ce mode, il agit comme un client et se connecte à un point d'accès wifi sélectionné dans les paramètres. L'utilisateur doit alors se connecter à ce point d'accès afin de pouvoir communiquer avec le sensor. Dans ce mode, le point d'accès wifi doit être existant. L'utilisateur peut observer les données de la probe depuis un appareil connecté au point d'accès, sans obligation d'y être connecté en wifi. La Figure 2 illustre ce mode de fonctionnement. On remarque qu'un nouvel élément apparait par rapport à la Figure 1, le point d'accès wifi (*AP*).

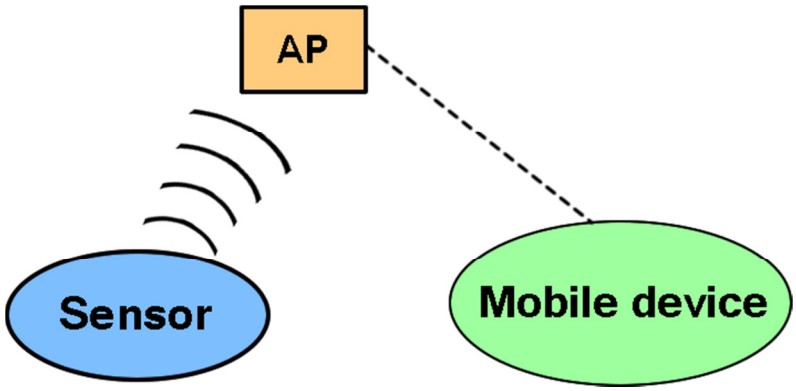


Figure 2: iCelsius Wireless, mode 2

Le troisième mode possible est le mode infrastructure distant. Dans cette configuration, iCelsius Wireless agit comme un client et se connecte sur un point d'accès wifi existant afin d'envoyer ses données à un serveur distant. Ce serveur peut alors stocker les données et les traiter de différentes manières, comme à des fins statistiques par exemple. L'utilisateur peut alors se connecter à ce

serveur via internet depuis n'importe où afin de voir les données récoltées par ses différentes probes. La Figure 3 représente ce dernier mode de fonctionnement. Nous pouvons voir que les données envoyées par le sensor transitent par le point d'accès wifi avant d'aller sur internet. Sur internet, les données sont redirigées vers notre application serveur tournant sur les serveurs de Google. L'utilisateur doit alors se connecter à internet pour avoir accès aux données présentes sur les serveurs de Google.

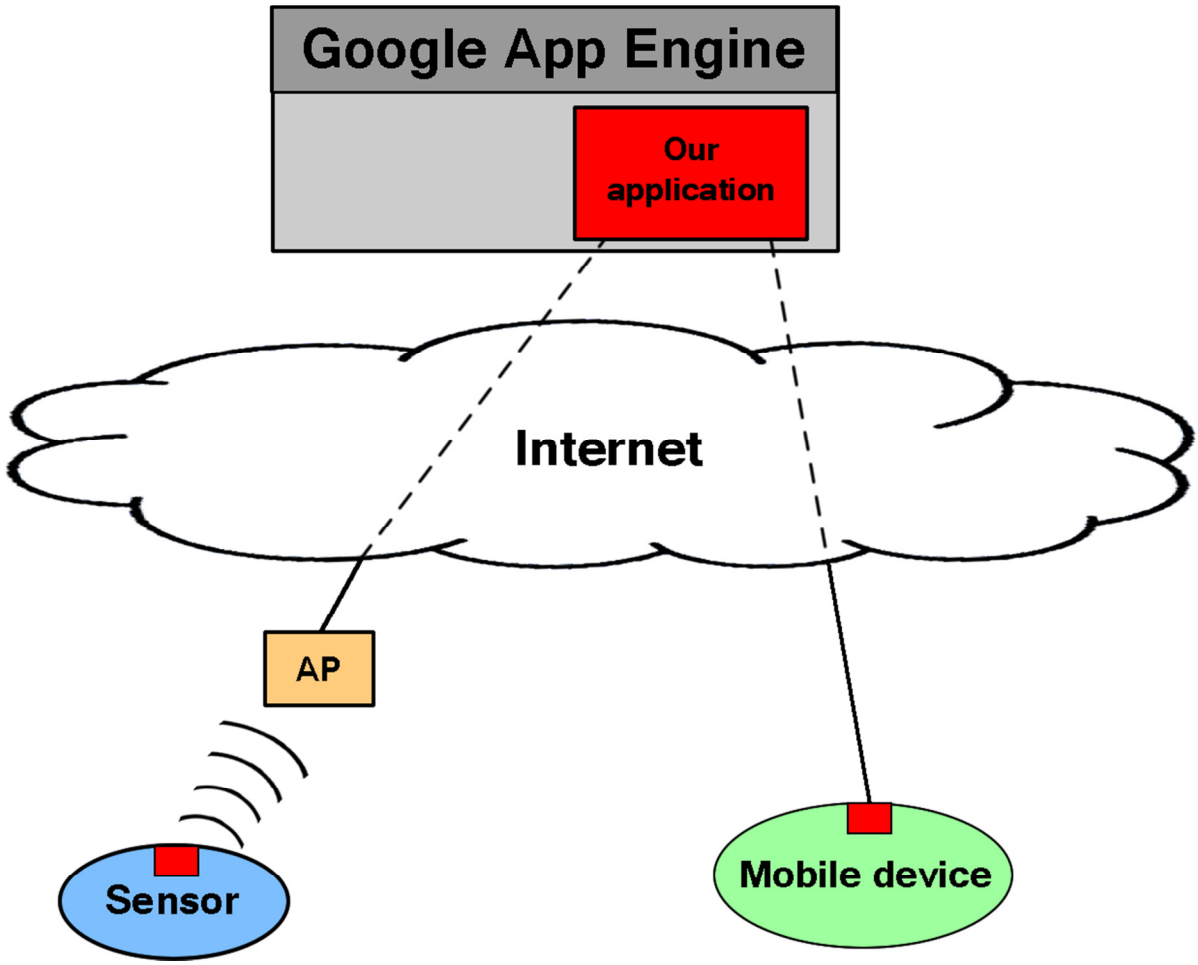


Figure 3: iCelsius Wireless, mode 3

Les deux premiers modes de fonctionnement ont l'avantage de ne pas nécessiter internet mais présentent un défaut majeur; Il faut que l'utilisateur se trouve à proximité du sensor afin d'en recevoir les données. Le troisième mode possible d'iCelsius Wireless remédie à ce problème tout en ajoutant un service en plus, le stockage des données. L'accès à internet est alors obligatoire pour avoir accès aux données stockées.

4.2 Travail à effectuer

Ce projet a pour but l'implémentation du troisième mode de fonctionnement d'iCelsius Wireless. Dans la Figure 3, toutes les parties en rouge sont les fonctionnalités à réaliser dans ce projet. Le premier point, le plus imposant, est la mise en place d'une application web permettant de recevoir, stocker et redistribuer correctement les données provenant de différents sensors. L'idée initiale est d'implémenter et héberger l'application web sur la plateforme Google App Engine. Cette application comprendra aussi une interface utilisateur simple afin que les données soient aisément accessibles

via n'importe quel navigateur internet. Le deuxième point à implémenter est la partie communication avec le sensor. La communication entre le sensor et le serveur se fera à travers le protocole HTTP afin d'éviter tout problème de firewall. Le dernier point à mettre en place est une application Android que l'utilisateur installera sur son smartphone ou autre appareil Android afin d'avoir un accès rapide à ses données.

L'application web devra tenir compte de contraintes supplémentaires. Premièrement, il existera deux modes pour l'utilisateur, le mode standard et le mode premium qui sera soumis à une taxe. L'utilisateur pourra choisir pour chacun de ses sensors s'il souhaite que son sensor soit standard ou premium. Dans le mode standard, l'utilisateur n'a accès qu'aux dernières valeurs fournies par son sensor. Dans le mode premium, les données des différentes probes sont stockées et l'utilisateur peut les consulter à tout moment. Une amélioration à faire si le temps le permet est d'introduire des statistiques sur les valeurs des différentes probes.

Une autre contrainte importante est que plusieurs utilisateurs peuvent avoir accès aux données du même sensor. Il est en effet très probable que les données du sensor doivent être accessibles par plusieurs personnes. C'est le cas par exemple si le sensor appartient à une entreprise et que plusieurs collaborateurs doivent avoir accès aux valeurs mesurées.

Le cahier des charges tel qu'il a été reçu est disponible dans les annexes, voir le chapitre *18.2 Cahier des charges*.

4.3 Vocabulaire

Tout au long de ce rapport, une convention des termes est utilisée. Voici les termes utilisés ainsi que leur signification:

Sensor (iCelsius Wireless)	Le module wifi avec la probe connectée
Probe	Le capteur, que ça soit un capteur de température, d'humidité, de CO_2, de pH, etc...
Sensor ID	Un identifiant unique de chaque sensor
User ID	Un identifiant unique pour l'utilisateur final
User device	Smartphone ou tablette Android avec l'application iCelsius Wireless pour Android
Server, GAE, App Engine	Les serveurs de Google App Engine avec notre application s'exécutant dessus
Android App	L'application Android pour iCelsius Wireless

5 Google App Engine

5.1 Introduction

Google App Engine, appelé GAE ou simplement App Engine, est un framework middleware qui vient concurrencer les services d'Amazon, Amazon Web Services, et de Microsoft, Windows Azure. App Engine est une solution de PaaS, basée sur le Cloud computing, permettant la conception et l'hébergement d'applications web. Platform as a Service (PaaS) est un type de Cloud computing permettant aux entreprises de développer et maintenir des applications web sur les serveurs d'un fournisseur Cloud. Le rôle du fournisseur Cloud est de mettre à disposition tout le nécessaire pour que le client puisse utiliser sa plateforme d'exécution de manière simple et efficace.

Une des motivations pour la mise en place d'une solution de ce genre est d'éviter au développeur de l'application web tout le travail inhérent à la gestion du serveur. Effectivement, le développement de l'infrastructure web utilisée n'est pas nécessaire car l'hébergeur nous met à disposition ses propres serveurs. Le fournisseur de Cloud nous met alors une API à disposition afin que nous puissions interagir avec l'interface proposée.

Les applications créées pour Google App Engine seront malheureusement difficilement portables vers d'autres plateformes. Cela est dû aux fonctionnalités mises en place par Google afin de faciliter le développement d'applications. Effectivement, Google nous met à disposition certains de ces services qui nous seront très utiles pour notre projet.

GAE a donc l'avantage de fournir une infrastructure complète permettant de développer des applications web plus aisément que ses concurrents. De plus, Google s'occupe de tout ce qui est hébergement et exécution des applications et nous fournis un certain nombre de services qui s'avèrent très pratiques et simple d'utilisation. C'est pour ces raisons que nous avons choisi d'utiliser cette plateforme.

5.2 Présentation

Google App Engine est un moyen de développer des applications destinées au web sans pour autant devoir implémenter et gérer la partie serveur web du projet. Beaucoup de facilités nous sont mises à disposition afin d'assurer un développement et une gestion rapide et efficace. L'utilisation de App Engine nous permet de ne pas nous soucier des problèmes de maintenance tels que l'évolution des besoins, la mise en place de l'administration système ou même l'achat de nouveau matériel hardware. L'hébergement sur Google App Engine nous apporte donc une couche d'abstraction du matériel qui s'avère très confortable pour un développement plus rapide. De plus, nous utilisons du matériel virtuel ce qui implique qu'il devient possible d'adapter nos ressources physiques selon nos besoins en quelques clics seulement. Les ressources réellement utilisées sont gérées automatiquement par Google selon les besoins du moment.

Figure 4 Logo de Google App Engine

Lors du développement d'une application, Google App Engine nous met à disposition gratuitement un serveur local ainsi qu'un hébergement en ligne. Ce n'est que lorsque notre application atteint certains quotas de tailles ou de trafics que le service devient payant. L'hébergement en ligne se fait gratuitement via un nom de domaine spécifique à GAE que nous pouvons changer par la suite, moyennant payement. Le domaine fourni gratuitement par Google est *appspot.com*, l'URL de chaque application est alors *<nomApplication>.appspot.com*.

L'utilisation de cette plateforme nous permet d'utiliser certains services de Google tels que les comptes utilisateur ou le datastore. Toutes les applications qui sont hébergées sur App Engine bénéficient de la sécurité de Google tant au niveau de la fiabilité des infrastructures qu'au niveau de la sécurité des données. Effectivement, Google possèdent plus de 900'000 serveurs à travers le monde et son réseau est très fiable. Au niveau sécurité, les applications hébergées sur App Engine sous soumises aux mêmes règles concernant la sécurité et la confidentialité que les applications officielles de Google. Le code ainsi que les données des applications sont automatiquement sécurisés.

Le développement d'applications pour App Engine peut se faire en plusieurs langages de programmation. Actuellement, seul Java et Python sont pleinement supportés, d'autres langages de programmation sont en cours de préparation comme par exemple "Go" qui vient de faire son apparition en mode expérimental dans GAE. Il n'y a rien de spécial concernant ces deux langages car App Engine utilise les technologies Java et Python standard.

Le développement se fait en plusieurs étapes. Il est d'abord nécessaire d'écrire le code dans le langage de programmation choisi. Nous pouvons ensuite le compiler puis le tester sur le serveur local. L'étape suivante est le déploiement de l'application créée sur les serveurs de Google, en ligne. A ce moment, le développeur a accès à une console d'administration qui lui permet de gérer toute son application. Lorsque l'application est accessible pour des utilisateurs, ces derniers envoient des requêtes à l'application qui les traitera et renverra des réponses.

La mise en place d'applications pour App Engine est possible à l'aide d'un éditeur de texte et d'une console. Il est possible de gérer toute l'application jusqu'à son déploiement grâce à des lignes de commandes. Cependant, Google nous met à disposition un plug-in pour Eclipse qui simplifie toutes ces manipulations.

5.3 Fonctionnement

Les applications hébergées sur App Engine sont accessibles par les clients via des sites internet dynamiques. Les pages dynamiques sont reconstruites à la volée par le serveur avant d'être renvoyées au client. Le serveur peut alors construire une page personnalisée pour l'utilisateur courant selon certains paramètres tels que la langue ou les pages déjà visitées.

Contrairement aux pages statiques qui n'utilisent que des langages de description des données comme le HTML, les pages dynamiques nécessitent en plus l'utilisation de langages de programmation tels que Java et JSP. C'est grâce à ces langages qu'il est devenu possible d'exécuter des instructions conditionnelles, des boucles ou d'autres traitements très complexes sur la requête du client.

Un client standard n'étant capable d'afficher que des pages en HTML, les langages de programmation nécessaires pour la création de pages dynamiques ne remplacent pas le HTML mais en créent à la volée. La page renvoyée au client n'est alors composée que de HTML. Cette manière de faire permet de s'assurer que tous les clients seront capables de lire correctement la page renvoyée car le HTML est un standard pour tous les navigateurs internet. De plus, tous les traitements sont effectués par le serveur de manière totalement transparente pour le client. Il n'y a alors aucun problème de compatibilité avec les différents clients qui peuvent supporter ou non certains langages spécifiques.

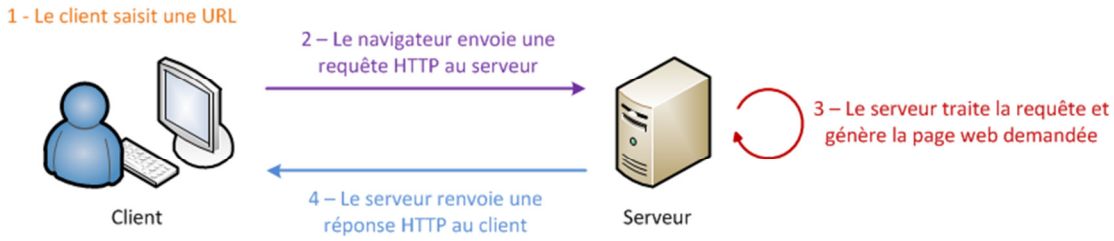


Figure 5 Fonctionnement général de GAE

Contrairement à HTML associé au CSS qui forment un couple incontournable pour le web, le traitement sur le serveur peut se faire avec plusieurs langages de programmation tels que le PHP et le .NET pour ne citer que les plus connus. Comme notre application sera créée en Java, les langages de programmation utilisés pour la génération des pages dynamiques seront le couple Java JSP. Un chapitre est consacré à l'utilisation de ces langages.

5.4 Sandbox

Les applications présentes sur Google App Engine s'exécutent dans un bac à sable, une sandbox en anglais. La sandbox isole les applications du matériel et permet à Google de mieux gérer ses différents serveurs. La couche d'abstraction supplémentaire apportée par le bac à sable permet à App Engine de distribuer les requêtes des applications sur différents serveurs HTTP. Il est alors possible d'allumer et d'éteindre des serveurs selon la demande de trafic.

La sandbox nous fournit un environnement fiable et sécurisé mais limite l'accès à la machine sur laquelle s'exécute notre application. De par ce mécanisme, le système d'exploitation n'est pas accessible de notre application, il n'est alors pas possible d'écrire directement dans le système de fichier de l'OS, de créer des sockets ou des threads. Certaines de ces contraintes sont des avantages au niveau sécurité, comme par exemple l'impossibilité d'ouvrir une console via la méthode système *console().* Google nous met des services à disposition afin de combler les lacunes dues au bac à sable.

La sandbox apporte des aspects sécuritaires mais elle impose certaines contraintes d'utilisation. A cause de la sandbox, les connexions avec d'autre utilisateur doivent se faire à travers un service spécialement mis en place par Google. De plus, un client peut uniquement se connecter à l'application avec le protocole HTTP ou HTTPS sur le port par défaut. Il n'est actuellement pas possible de créer des sockets pour communiquer, mais Google prépare un service qui viendra combler ce manque.

5.5 Environnement Java

Google App Engine supporte complètement le développement avec Java en tant que langage de programmation. Le JDK (Java Development Kit) actuellement recommandé utilise Java 7. Le SDK (Software Development Kit) supporté pour le développement utilise Java 6 et 7.

Le développement d'application web peut se faire avec les outils courants et les API standards. L'application Java pourra être en interaction avec l'environnement via l'utilisation de Servlet Java. Le formatage de l'affichage quant à lui peut se faire avec des fichiers JSP, Java Server Page. Actuellement, seule la version Servlet 2.5 est disponible pour Google App Engine.

La machine virtuelle Java est l'élément qui implémente les limitations imposées par le bac à sable. Toutes les fonctionnalités présentes dans le bytecode et dans les librairies peuvent être utilisées pour autant que l'on respecte les limitations de la sandbox. Si on essaie de dépasser ces limitations, Java lève une exception d'exécution.

Le SDK pour Java comporte toutes les API disponibles dans App Engine et fournit une application de serveur web qui simule les services de App Engine sur un ordinateur local. Le serveur local simule tout l'environnement d'App Engine, y compris les limitations imposées par la sandbox.

De plus, le SDK fournit un utilitaire permettant de transférer une application sur App Engine. Lors du transfert sur App Engine, il est possible de conserver la version présente sur les serveurs et de transférer l'application sous un autre numéro de version. Les utilisateurs peuvent alors continuer à utiliser l'ancienne version jusqu'au moment où l'administrateur décide de basculer vers la nouvelle version. Ceci peut permettre d'effectuer des tests de la nouvelle version de l'application.

5.6 Services

Google nous met à disposition un nombre croissant de services. Ces services nous permettent de contourner les limitations imposées par la sandbox mais aussi d'utiliser des fonctionnalités additionnelles comme l'utilisation des comptes de Google, les Google Accounts. Dans ce document, seul les services qui seront utilisés dans notre application sont présentés. Il s'agit des services nommé *Datastore*, *Users, MemCache* et *Mail*.

5.6.1 Datastore

Google nous met à disposition un service de stockages des données qui s'adapte au volume des données qui y sont stockées. Google assure la cohérence et l'intégrité des données stockées. La mise à jour d'entités supporte les accès concurrents. Si un processus tente de mettre à jour une entité alors que celle-ci est déjà utilisée, la transaction est répétée.

Un point important à relever sur le magasin de données proposé par Google App Engine est que ce n'est pas une base de données relationnelle. Le datastore agit plutôt comme un HashMap persistant avec un kind, identifiant le type d'entrée, ainsi qu'un ensemble de propriétés qui lui sont associées. Les requêtes effectuées sur le datastore concerneront alors les données d'un seul kind à la fois. Comme avec les bases de données traditionnelles, les requêtes peuvent être filtrées selon certains critères. Le type de données stockées est totalement défini par les besoins de l'application.

Le SDK Java intègre des implémentations pour supporter les interfaces Java Data Object (JDO) et Java Persistence API (JPA). Ces interfaces permettent la modélisation et gèrent la persistance des

données. JDO est un standard basé sur Java permettant d'accéder simplement aux données de la base de données de manière transparente. L'utilisation de JDO est facilitée grâce à l'utilisation d'un POJO (Plain Old Java Object) et au fait que le développeur ne doit pas se soucier de la persistance. Un POJO est une simple classe qui n'implémente pas d'interface spécifique à un framework afin d'être accessible dans une base de données. La persistance permet au programme de se terminer sans que les données soient perdues, ceci grâce à des mécanismes de sauvegarde et de restauration des données.

Google met à disposition un long tutoriel[1] sur l'utilisation de son magasin de données. Les explications fournies par Google ne sont pas très claires et l'utilisation du JDO et du JPA ne s'avère pas très intuitive. C'est pourquoi nous avons décidé d'utiliser une interface plus conviviale, recommandée par la documentation de Google App Engine, Objectify. Un chapitre est dédié à l'utilisation du datastore avec Objectify, voir chapitre *5.7 Objectify* page 17.

5.6.2 Users

L'authentification des utilisateurs sur différentes applications s'avère très utile, voire indispensable pour l'implémentation de certaines applications. Grâce à l'authentification, il est possible de restreindre l'accès à un groupe de pages accessibles uniquement aux utilisateurs identifiés ou encore à un administrateur.

App Engine propose trois manières permettant l'authentification des utilisateurs: via OpenID, via des comptes sur notre propre domaine Google App, ou par le biais de Google Account. Nous avons choisi d'utiliser Google Account car la gestion de l'interface de demande de connexion est entièrement gérée par Google. De plus Google Account est un service plus connu que OpenID ou encore notre propre système que nous aurions mis en place. L'utilisateur aura alors plus confiance lors de la demande de connexion avec notre application. Le choix de l'option d'identification est effectué lors de la création de l'application.

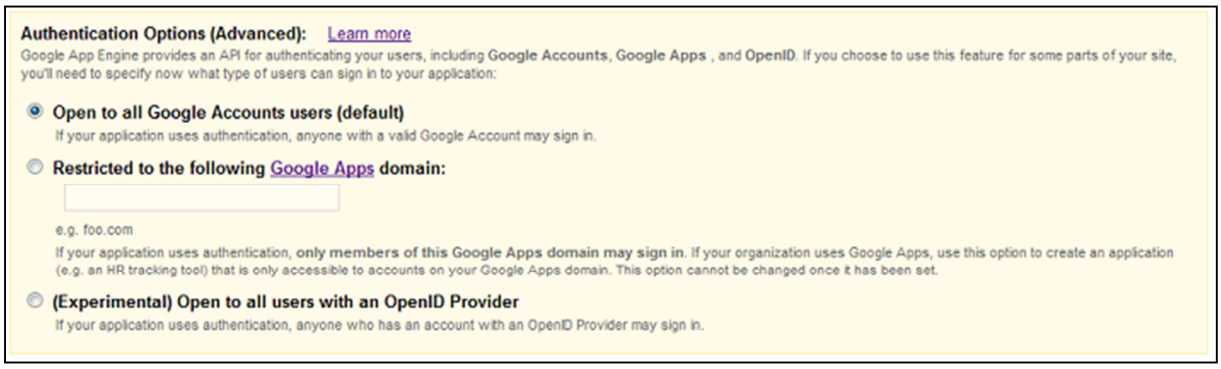


Figure 6 Choix de type d'authentification

Quel que soit le type d'authentification choisi, les APIs de Java en permettent une gestion très aisée. Les liens de connexion et de déconnexion de l'utilisateur nous sont fournis par l'API du service *Users*. Si l'utilisateur n'est pas connecté, il est alors facile de le rediriger correctement vers la page d'authentification correspondante au type d'authentification choisi.

[1] https://developers.google.com/appengine/docs/java/datastore/?hl=fr

Lors de la connexion avec Google Account, l'utilisateur est redirigé vers la page standard de connexion de Google (Figure 7). Lors de la première connexion avec l'application, Google demande une autorisation supplémentaire à l'utilisateur (Figure 8). Cette étape n'est nécessaire uniquement lors de la première connexion, puis tous les 30 jours. Lors de la déconnexion, aucune page spéciale n'est affichée.

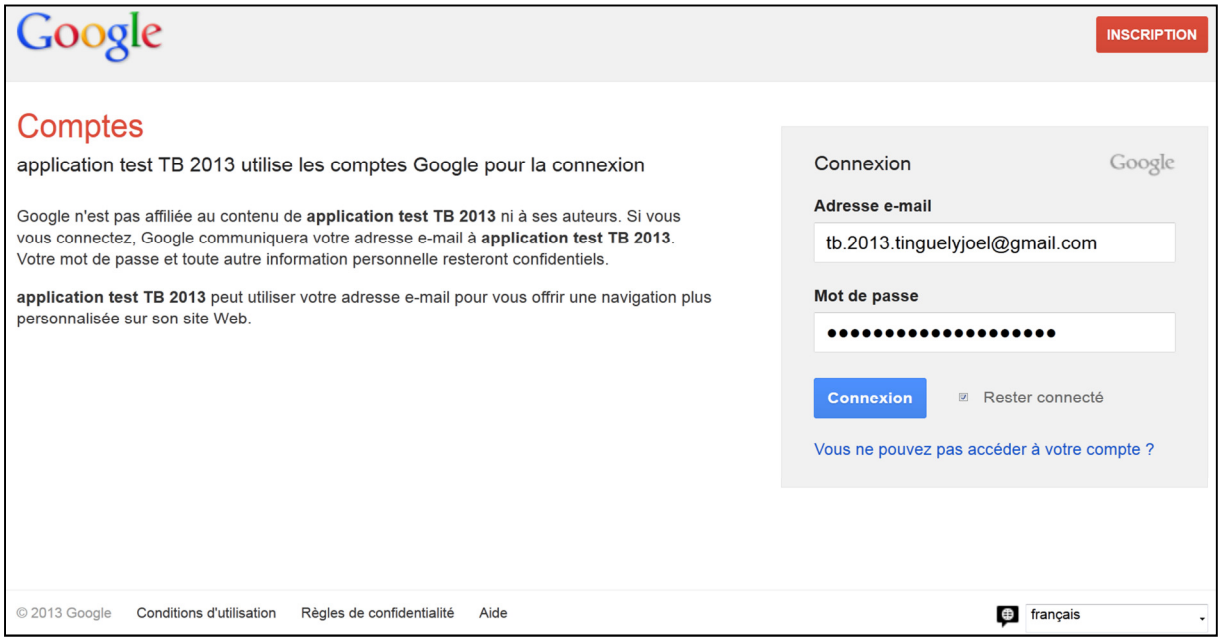


Figure 7 Page de connexion

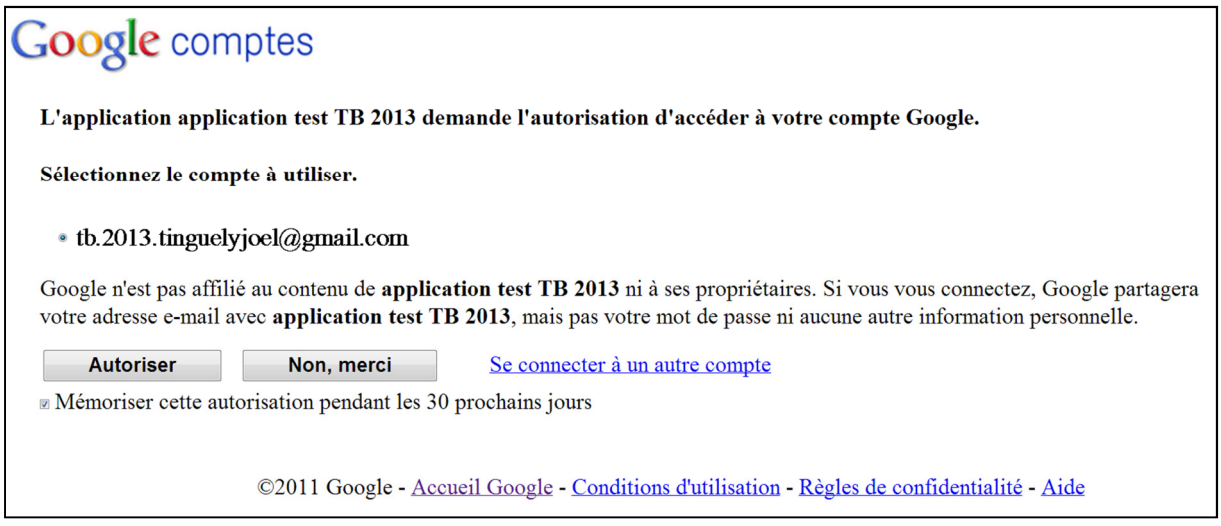


Figure 8 Demande d'autorisation lors de l'authentification

Lorsqu'un utilisateur est connecté à une application, celle-ci a accès à l'email et au nickname de l'utilisateur. L'application a aussi accès à un numéro identifiant l'utilisateur de manière unique, même si ce dernier change d'adresse email ou de nickname. Cette dernière valeur est très utile pour identifier un utilisateur dans le datastore.

5.6.3 Memcache

Le service Memcache fourni par Google nous permet de stocker temporairement des valeurs sans faire appel au datastore. Le memcache fournit une interface de type Map afin de pouvoir accéder aux valeurs enregistrées. Chaque élément enregistré est alors accessible via une clé. Cette clé ainsi que l'élément à stocker peuvent être de n'importe quel type.

Le memcache est généralement utilisé lors de l'utilisation du datastore. Les valeurs fournies suite aux requêtes effectuées sur le datastore sont stockées dans le memcache. Les prochaines requêtes identiques seront alors plus rapides et moins coûteuses car les valeurs proviendront directement du Memcache, sans passer par le datastore. Ce mécanisme nous permet d'économiser des requêtes sur le datastore. Lors de requêtes sur le datastore, il est alors nécessaire de vérifier si la valeur recherchée est déjà présente dans le memcache. Une requête sera effectuée sur le datastore uniquement si la valeur recherchée n'est pas présente dans le cache.

Les données enregistrées sur la mémoire cache sont conservées le plus longtemps possible. Si le memcache est plein et qu'une nouvelle valeur doit y être sauvegardée, la donnée la moins récemment utilisée sera supprimée du memcache. Lors de problème serveur, il est possible que les données présentes dans la mémoire cache soient supprimées. Il ne faut alors pas compter sur la disponibilité des valeurs dans le memcache.

Il est possible de sauvegarder des valeurs temporaires dans la mémoire cache. Celle-ci n'est pas réservée au datastore. Cependant, les valeurs présentes dans la mémoire cache peuvent disparaitre à tout moment. Il est alors nécessaire de prévoir un mécanisme de traitement si la donnée sauvegardée n'est plus présente dans le cache.

5.6.4 Mail

App Engine nous met à disposition un service permettant la gestion de l'envoi et la réception de courriers électroniques. Cette fonctionnalité s'avère très pratique lorsque l'utilisateur souhaite poser des questions à l'administrateur ou que le système souhaite signaler une erreur.

L'envoi de messages se fait grâce à l'interface JavaMail qui est contenu dans le SDK d'App Engine. L'envoi d'un message ne nécessite pas de configurer le serveur SMTP car App Engine utilise son service Mail. L'email peut être adressé à plusieurs destinataires, contenir un objet, un corps et des fichiers en pièces jointes. Le service Mail de App Engine s'occupe de contacter les serveurs de messagerie des destinataires, de transmettre le message et de répéter cette action si une erreur survient. Si suite à une erreur l'email ne peut pas être envoyé, un message d'erreur est envoyé à l'adresse email de l'expéditeur. L'application n'a aucun moyen de savoir si le message est arrivé correctement.

La réception d'email dans une application se fait via des requêtes HTTP contenant des informations MIME. Ces données sont ensuite transmises au handler de l'application en tant que contenu d'une requête POST. Le handler peut être choisi dans le descripteur de déploiement de l'application. Par défaut, la réception d'email dans notre application est désactivée. Pour activer le service, le descripteur de déploiement doit explicitement indiquer qu'il accepte les emails entrant.

5.7 Objectify

Objectify est une interface fournissant une API Java permettant l'accès aux données du datastore de Google Apps Engine. Cette interface est plus transparente que JDO ou JPA et est beaucoup plus commode que l'API bas niveau proposée par Google. Objectify propose un wiki[2] très complet permettant de se familiariser avec son utilisation.

Il est important de rappeler que le datastore de App Engine agit comme un HashMap persistant sur lequel il est possible d'effectuer des recherches. Les objets stockés dans le datastore sont appelés des entités et ils correspondent à une classe POJO définie par l'application. Pour rappel, un POJO est une simple classe Java qui n'implémente pas d'interface spécifique à un framework.

Toutes les entités doivent posséder un identifiant unique. Cette valeur peut être de type long, Long ou String. Contrairement à ce que l'on pourrait penser, l'identifiant d'une entité ne permet pas de l'identifier univoquement. Dans le datastore, les entités sont identifiées par l'identifiant, un kind et le parent. Le kind correspond à la classe de l'objet stocké. La notion de parent permet de regrouper nos entités au même endroit physique. Les transactions à travers plusieurs entités sont alors plus rapides. Il ne faut pourtant pas abuser de cette méthode car Google limite le nombre de requêtes effectuées chaque seconde sur un groupe d'entité. Ces trois valeurs qui permettent d'individualiser l'entité dans le datastore sont regroupées dans une classe générique *Key*. A noter que la valeur du parent est très souvent fixée à *null* indiquant que l'entité est une entité racine. Dans ce cas, il n'est alors plus nécessaire d'indiquer le parent lors des transactions.

Afin d'effectuer des recherches sur les données de notre datastore, nous devons gérer des index. Un index est un fichier permettant au système de retrouver rapidement les informations qui lui sont demandées. Le fichier d'index du datastore, *datastore-indexes.xml*, est géré automatiquement par App Engine. Dans le datastore, chaque propriété d'une entité peut être indexée ou non-indexée. Par défaut, Objectify index toutes les propriétés. Il est alors nécessaire de spécifier si l'on ne souhaite pas indexer une valeur avec l'annotation *@Unindexed*. A noter que les champs *static*, *final* ainsi que les *String* de plus de 500 caractères ne sont jamais indexés. La non-indexation permet d'économiser du temps processeur en réduisant la taille du fichier d'index. Les requêtes effectuées sur une entité ne pourront pas rechercher l'entité via des valeurs non-indexées. L'indexation est alors nécessaire pour effectuer des requêtes mais elle est coûteuse en niveau temps processeur et place de stockage. Ces deux valeurs sont des services qui deviennent payant au-delà d'une certaine limite. Il est alors important de bien définir quelle propriété d'une entité doit être indexée. Avec Objectify, il est possible d'indexer des champs uniquement si certaines conditions sont remplies, comme par exemple si le champ n'a pas sa valeur par défaut ou que la valeur ne soit pas *null*.

Une des propriétés d'une entité doit être déclarée comme identifiant avec l'annotation *@Id*. Cet identifiant peut être de type long, Long ou String. Si on définit l'identifiant en tant que Long, la valeur de ce champ pourra être automatiquement initialisée par Objectify. Toutes les entités qui sont stockées doivent posséder un constructeur sans paramètre quel que soit le niveau de protection attribué (*private*, *protected*, *public*).

[2] https://code.google.com/p/objectify-appengine/wiki/Concepts?tm=6

Le temps maximal entre une requête et sa réponse est borné par App Engine à 30 secondes maximum. Si une recherche sur le datastore prend plus de 30 secondes, nous devons utiliser un curseur. Les curseurs nous permettent de fixer un checkpoint dans une requête afin de reprendre la recherche suivante en partant de ce point. Les requêtes sont effectuées sur le datastore de manière concurrente. L'accès aux données fournies par ces requêtes peut par contre être bloquant, attendant que la requête en cours d'exécution fournisse un élément ou se termine.

Objectify nous met à disposition des annotations très utiles lorsque nous devons effectuer des changements sur notre modélisation. Il est possible d'ajouter, supprimer ou modifier des propriétés d'une entité en ajoutant des annotations et des éventuelles fonctions permettant la transformation du champ. Les modifications des valeurs dans le datastore sont effectuées naturellement lors des opération *get()* et *put()*.

5.8 Serveur de développement local

Le serveur de développement, contenu dans le SDK, est un serveur local permettant de simuler l'environnement web final afin de tester notre application en local. Tous les outils et services de Google sont présents dans le serveur local. Nous pouvons notamment y tester la gestion des utilisateurs avec Google Account ainsi que le magasin de données.

L'exécution du serveur de développement peut se faire via Eclipse ou par des lignes de commandes. Dans ce document, seule l'utilisation d'Eclipse est présentée. Pour démarrer le serveur local depuis Eclipse, nous devons sélectionner notre projet et le débugger ( ) en tant qu'application web. Le serveur est ensuite accessible à l'adresse suivante: http://localhost:8888. Pour plus d'information, voir le chapitre dédié au plug-in pour Eclipse, *5.11 Plug-in Eclipse*. Le serveur local peut fonctionner avec ou sans valeur entre les éléments *<application>* et *</application>* du fichier *appengine-web.xml.*

Le serveur de développement gère le datastore en créant un fichier de données, *local_db.bin*. Ce fichier est situé dans le répertoire *war/WEB-INF/appengine-generated*. Il permet de conserver les données persistantes entre deux exécutions du serveur local. La suppression des données de ce fichier peut se faire par la console d'administration local ou simplement en supprimant le fichier.

Le serveur local simule le système d'authentification de Google Account avec un page factice qui accepte tous les utilisateurs sans demander de mots de passe. Sur cette page, nous pouvons choisir de se connecter en tant qu'administrateur afin de tester certaines parties de l'application réservées aux administrateurs. La Figure 9 illustre la page de connexion. Tout comme sur le vrai serveur, lors de la déconnexion, aucune page spéciale n'est affichée, l'utilisateur est simplement redirigé selon les paramètres de l'application.

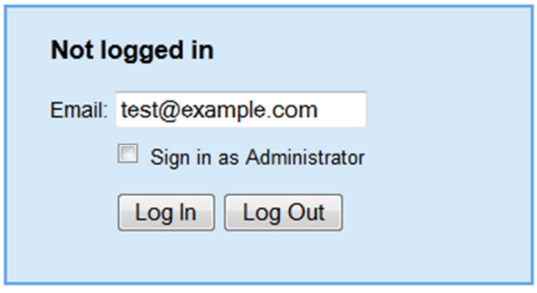


Figure 9 Google Account du le serveur de développement

Le service Mail de Google n'envoie pas de message lors de l'utilisation du serveur de développement. Lorsque ce service est utilisé, les messages envoyés sont consignés dans le journal. Il est alors possible de visualiser les informations sur le message envoyé dans la console d'Eclipse.

La gestion de l'application fonctionnant sur le serveur de développement peut être effectuée par la console d'administration locale. Celle-ci se trouve par défaut à l'adresse suivante: *http://localhost:8888/_ah_admin*. Depuis cette adresse, il est possible de parcourir et gérer le datastore. Cette console est une version simplifiée de la console d'administration de App Engine. La Figure 10 nous permet de visualiser l'interface que présente la console d'administration locale. Pour plus d'informations sur la console d'administration, voir le chapitre en page 20 qui lui est dédié.

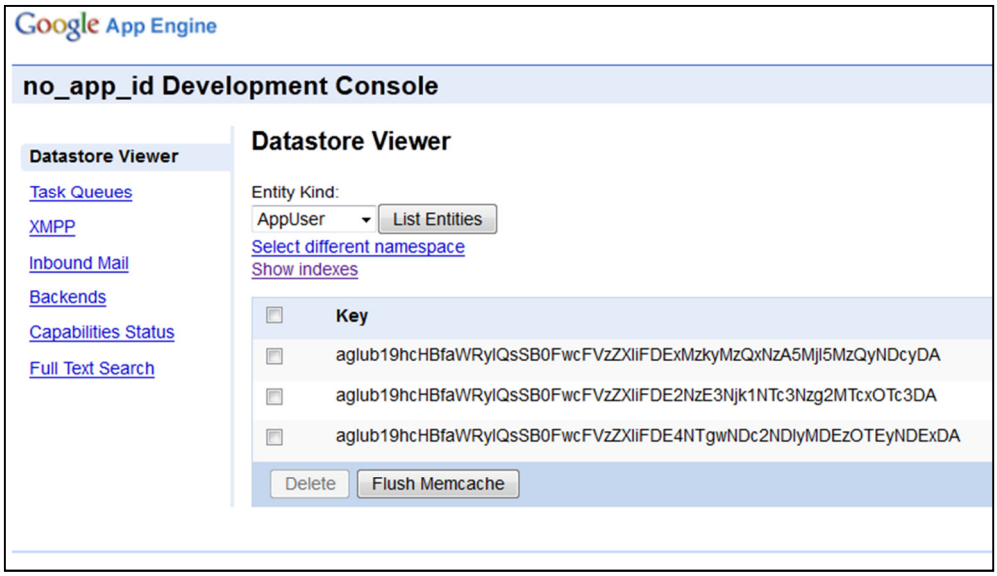


Figure 10 Console d'administration locale

Le serveur local fonctionnant avec Eclipse, il peut être utile de connaitre le chemin pour obtenir les fichiers de log d'Eclipse : *.metadata/.plugins/org.eclipse.ui.workbench/log*.

Une modification nécessaire pour le développement local est d'autoriser l'accès au serveur local depuis le réseau. Ceci nous est très utile pour que les sensors puissent envoyer des données sur le serveur local. Pour effectuer cette modification, nous devons modifier les arguments passés lors du lancement du programme, *debug configuration -> arguments*, et ajouter l'argument "*--address=0.0.0.0*".

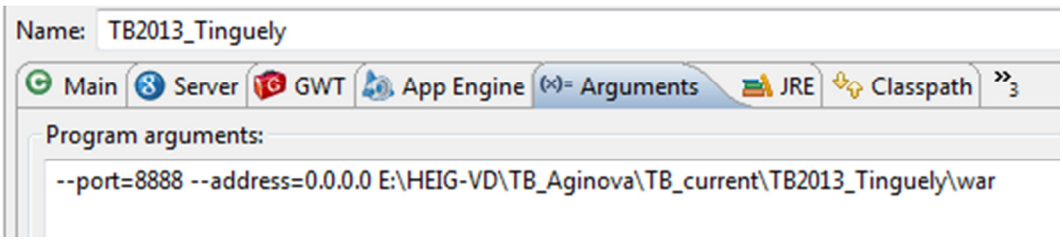


Figure 11 Modification des arguments du serveur local

5.9 Console d'administration

Une fois qu'une application a été développée est déployée sur le serveur de App Engine, il est possible de la gérer via une console d'administration. L'accès à cette console se fait par le lien suivant: *https://appengine.google.com*.

A partir de ce lien, il est possible de créer une nouvelle application. A noter qu'il est obligatoire de passer par cette étape lors de la création d'une application, il n'est pas possible de le faire directement depuis Eclipse.

Après avoir créé l'application, la console d'administration permet d'avoir une vue globale de toutes les informations relatives à l'application. Ci-dessous la description des sections les plus importantes de la console d'administration.

- D*ashboard* : nous permet d'observer des graphiques ainsi que l'état actuel des ressources utilisées.
- *Logs* : Visualisation des différents messages de notre application.
- *Versions* : Permet de voir les différentes versions de l'application et de définir quelle est la version accessible par les utilisateurs. Une option supplémentaire est le *traffic splitting* qui permet de tester une version sur un certain pourcentage d'utilisateur.
- *Quota Details* : Différents pourcentages de tous les quotas en vigueur.
- *Data Viewer* : Cette page nous permet de visualiser le contenu de tous les datastores de notre application. Il est possible de créer, modifier ou supprimer des entrées. Il est également possible d'effectuer des recherches avec des requêtes GQL (Google Query Langage) afin de filtrer les informations affichées. Ce langage est basé sur la même syntaxe que le SLQ.
- *Datastore Admin* : Depuis cette page, il est possible d'effectuer des backups ou restores de nos datastores, de les copier vers d'autres applications ou de supprimer les données.
- *Application Settings* : Ici se trouvent tous les paramètres principaux de l'application. Nous pouvons par exemple modifier le nom ou l'URL de l'application, le type d'authentification, la durée de vie des cookies, et des activations de service ou même de l'application.
- *Permission* : Cette page permet de visualiser tous les collaborateurs de l'application. Nous pouvons ainsi inviter des nouvelles personnes et leur attribuer un rôle. Ce rôle peut être développeur, propriétaire ou simple visiteur.
- *Admin logs* : Cette section permet d'avoir une vue sur toutes les actions effectuées par les administrateurs de l'application.
- *Billing Settings* : Ici se trouvent tous les paramètres concernant les payements des services.
- *Billing History* : Permet d'avoir un rapport journalier de l'utilisation des ressources ainsi que leurs facturations.

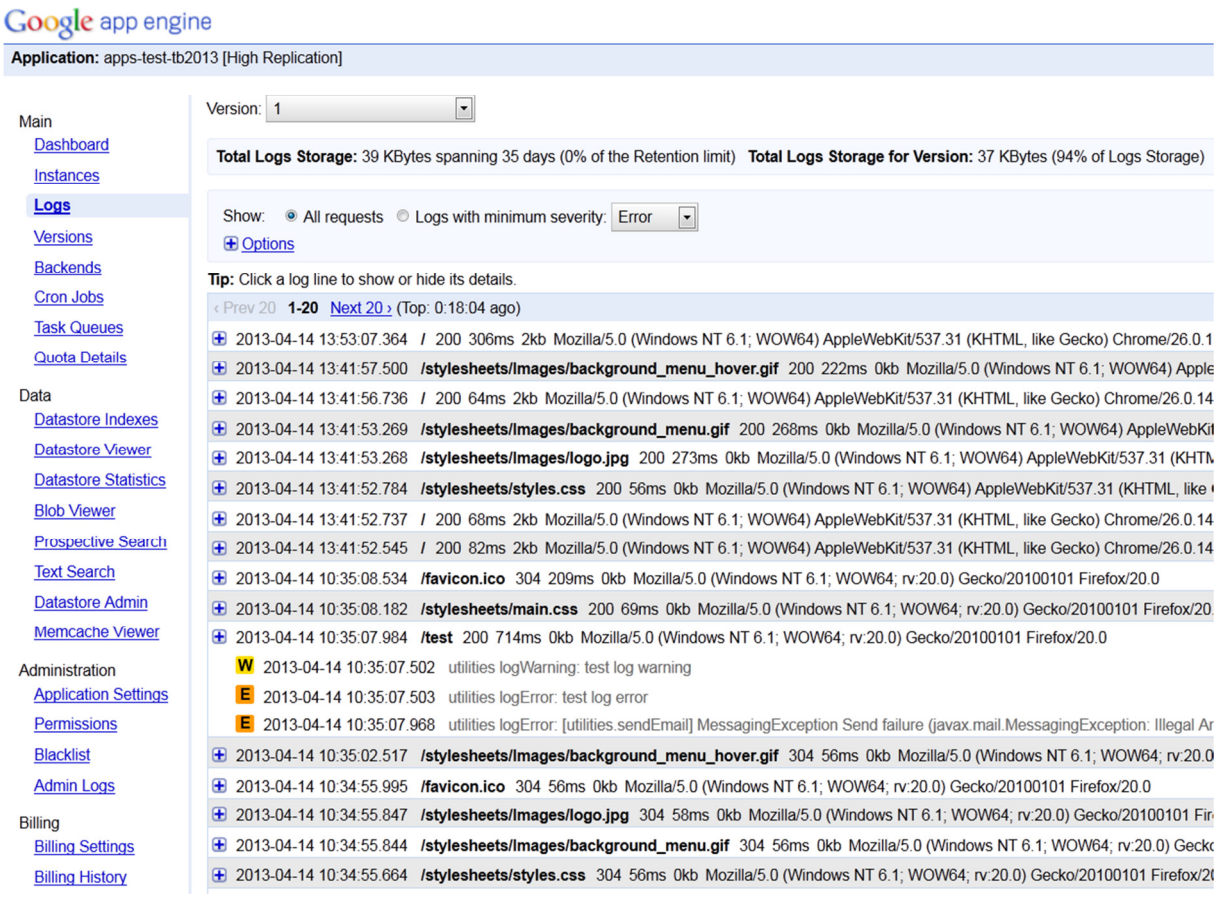


Figure 12 Console d'administration

5.10 Quotas

L'utilisation de App Engine est gratuite jusqu'à certains quotas, au-delà desquels les services sont facturés. La facturation des services concerne uniquement ce que l'application consomme en termes de ressources mais ne se compose d'aucun frais de gestion ou autre frais récurant. On paie donc uniquement ce que l'on consomme. Il est possible de fixer des seuils limites à ne pas dépasser afin de contrôler notre budget.

Le nombre maximum d'applications enregistrées par développeur est de 10, que les ressources soient facturées ou non. Le temps maximal entre une requête et sa réponse est au maximum de 30 secondes.

Les ressources allouées aux applications sont automatiquement définies par Google selon le trafic courant. Cet automatisme fonctionne uniquement pour les applications dont le temps de latence est de moins d'une seconde. Les applications avec une plus grande latence sont limitées en termes de ressources. Il est toutefois possible d'obtenir une dérogation pour ce type d'application afin de ne pas être limité. Toutes les ressources possèdent des limitations de base pour la version gratuite. Dans ces ressources, on trouve notamment le temps CPU, le nombre de requêtes sortantes et entrantes ainsi que la place prise par notre datastore.

App Engine relève toutes les ressources utilisées par les applications afin de les utiliser pour le système de quotas. Les quotas sont mis à jour toutes les 60 secondes et les valeurs relevées sont

remises à zéro chaque jour à minuit, Pacific time. Seules les ressources utilisées pour le stockage sur le datastore ne sont pas remises à zéro.

Lorsqu'une application atteint la limite d'un de ses quotas, la ressource épuisée ne peut plus être utilisée. Si la ressource épuisée concerne les requêtes, l'accès à l'application provoquera l'erreur HTTP *403 Forbidden*. Si on tente d'accéder à une autre ressource alors qu'elle est épuisée, l'application provoque une exception, *com.google.apphosting.api.ApiProxy.OverQuotaException*.

5.11 Plug-in Eclipse

Google nous met à disposition un plug-in Eclipse pour le développement d'applications Java destinées à Google App Engine. Ce plug-in comprend tous les outils nécessaires au développement d'application en Java, comme le SDK. Grâce à ce plug-in, le développement, les tests sur le serveur local ainsi que le déploiement de l'application sur le web sont simplifiés.

5.11.1 Installation de l'environnement de développement

1. Télécharger et installer la dernière version de Java JDK et JRE[3]
2. Télécharger, extraire et lancer l'IDE Eclipse Juno[4]
   2.1. Si la fenêtre de la Figure 14 s'affiche, il y a une erreur de variable d'environnement. Modifier les variables d'environnement du système comme indiqué ci-dessous.
   PATH => C:\Program Files\Java\jdk<version>\bin
   CLASSPATH => C:\Program Files\Java\jdk<version>\\lib\tools.jar
   JAVA_HOME => C:\Program Files\Java\jdk<version>\
   2.2. Si la fenêtre de la Figure 15 s'affiche, il y a une erreur entre Eclipse et Java. Vérifier que la version de Java corresponde à la version d'Eclipse au niveau du nombre de bit du système (x86, x64)
3. Installation du SDK et du plug-in App Engine pour Eclipse. Ces éléments sont disponibles à l'aide de la fonction de mise à jour logiciel. Dans Eclipse: *Help -> Install new Software*. Entrer *http://dl.google.com/eclipse/plugin/4.2* dans le champ *work with* et appuyer sur le bouton *add*, puis *ok* dans la nouvelle fenêtre. Installer les composants *Google Plugin for Eclipse* et *SDKs*.
4. Sélection des JDK au lieu des JRE afin de pouvoir gérer tous les fichiers, notamment les JSP. *Windows -> Preferences -> Java -> Installed JREs*. Sélectionner uniquement le JDK installé au point 1.
5. Sélection de la machine virtuelle Java sur laquelle s'exécute Eclipse afin d'éviter des problèmes lors du déploiement de l'application sur App Engine. Ouvrir le fichier *eclipse.ini* dans le dossier d'installation d'Eclipse. Ajouter les deux lignes de la Figure 13 juste avant les paramètres de la machine virtuelle (*-vmargs*). Avec ces lignes, on indique le chemin de la machine virtuelle à utiliser. Indiquer alors le dossier *bin* du JDK installé au point 1.
6. Pour pouvoir éditer correctement les fichiers JSP, il faut ajouter le plug-in Java EE dans Eclipse. Pour se faire, aller dans le menu *Help -> Install new Software* et indiquer *http://download.eclipse.org/releases/juno* dans *Work with*. Sélectionner et installer *Web, XML, Java EE and OSGi Enterprise Development* ou spécifier les paquets concernant le développement web et Java EE.

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[4] http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/junosr2

```
-vm
C:\Program Files\Java\jdk1.7.0_17\bin
```

Figure 13 : Sélection de la machine virtuelle Java

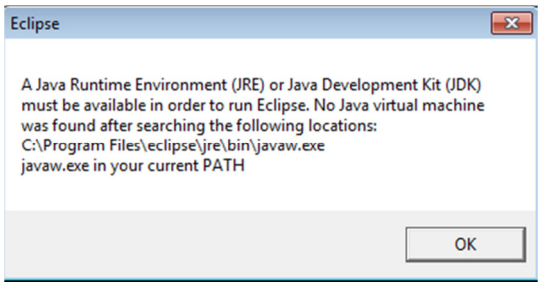


Figure 14 : Erreur de variable d'environnement

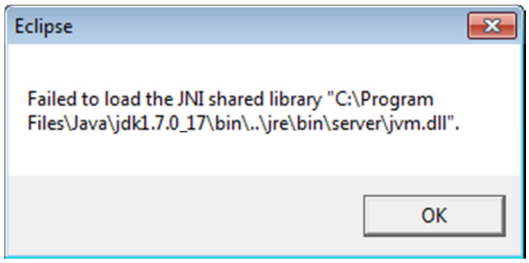


Figure 15: Erreur de version (x86, x64)

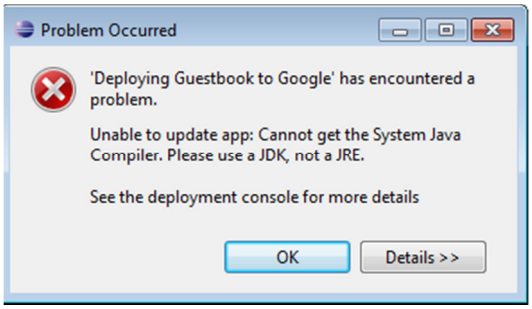


Figure 16 : Erreur: Eclipse oubli de modification de eclipse.ini

5.11.2 Création d'un projet

Pour la création d'un nouveau projet App Engine, cliquer sur le bouton Google ( ) puis *New Web Application Project...*.

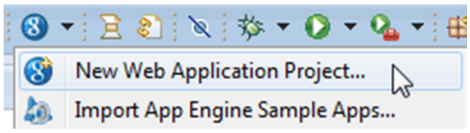


Figure 17 Création d'un nouveau projet sous Eclipse

L'assistant *Create a Web Application Project* s'ouvre. Il faut alors spécifier le nom du projet ainsi que le paquetage de base. Vérifier ensuite que la case *Use Google App Engine* soit cochée et cliquer sur terminer. L'assistant de création nous crée alors une arborescence complète pour notre projet.

```
Guestbook/
  src/
    guestbook/
      server/
        GuestbookServlet.java
    META-INF/
      jdoconfig.xml
    log4j.properties
    logging.properties
  war/
    WEB-INF/
      lib/
        ...App Engine JARs...
      appengine-web.xml
      web.xml
    index.html
```

Figure 18 Arborescence du projet après création

La structure du répertoire de travail utilisé par Eclipse est bien définie. Le dossier racine porte le nom de l'application et se compose de deux sous-dossiers, *src* et *war*. Le dossier *src* contient le code source Java et un dossier *META-INF* qui contient des fichiers de configuration. Le code source de Java est classé en différent paquetage selon la volonté du concepteur.

Le dossier *war* contient l'intégralité de l'application au format WAR. Le format WAR, Web Application Archive, correspondrait à un fichier JAR qui contient des pages JSP, des classes Java, des fichiers XML et des pages statiques (HTML). C'est sous ce dossier que nous allons retrouver les classes compilées, les librairies, des fichiers de configuration ainsi que des fichiers statiques. Ce sont ces fichiers qui seront alors utilisés sur le serveur de développement ou sur le serveur de App Engine.

5.11.3 Exécution sur le serveur de développement

Pour exécuter notre application sur le serveur local depuis Eclipse, nous devons sélectionner notre projet et le débugger ( ) en tant qu'application web. Nous voyons alors le lancement du serveur et de notre application dans la console d'Eclipse. Dès que le message *INFO: Dev App Server is now running* apparait, nous pouvons utiliser notre application sur le serveur local. Si l'application effectue des affichages sur les sorties standard Java, ceux-ci seront affichés dans la console d'Eclipse.

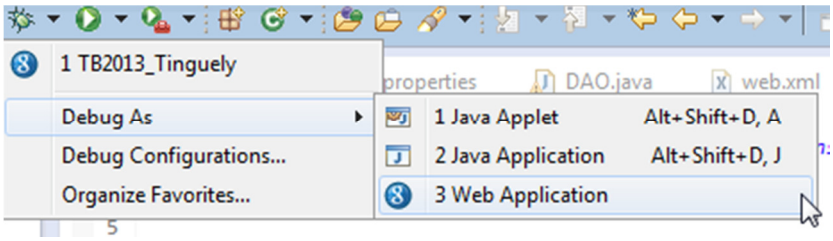


Figure 19 Exécution de notre application sur le serveur local

L'adresse du serveur par défaut est http://localhost:8888. Il est possible de modifier les paramètres du serveur local, comme le numéro de port par exemple, en créant une nouvelle configuration de débogages de type *Web Application*.

Les changements apportés à l'application lorsque celle-ci est exécutée sur le serveur local sont pris en compte en directe. Il n'est alors pas nécessaire de redémarrer le serveur à chaque modification, il suffit de recharger la page sur le navigateur. Néanmoins, le serveur doit être redémarré si le descripteur de déploiement, *web.xml*, est modifié. Pour arrêter le serveur, cliquer simplement sur le bouton stop ( ) dans Eclipse.

5.11.4 Déploiement de l'application

Lors du premier déploiement, il est nécessaire de passer par la console d'administration afin de créer un nouveau identifiant de projet. Cet identifiant doit être placé entre les balises *<application>...</application>* du fichier *appengine-web.xml*. Si le développeur oublie de spécifier cet identifiant dans le fichier de configuration, le message de la Figure 20 sera affiché dans la fenêtre de déploiement.

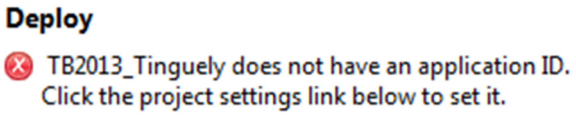


Figure 20 Message d'oubli de l'identifiant de l'application

Lors de l'installation du plug-in d'Eclipse, celui-ci ajoute des boutons dans la barre d'outils de notre IDE. Pour déployer une application sur les serveurs de App Engine, cliquer sur le bouton Google ( ) puis *Deploy to App Engine...*. La fenêtre *Deploy* s'ouvre et demande de s'identifier avec le login de Google. Le déploiement se fait ensuite simplement en cliquant sur le bouton *Deploy*. L'avancement du déploiement est visible dans la console d'Eclipse. L'application déployée est disponible à l'adresse : *http://<applicationID>.appspot.com*.

En cas d'erreur, essayer de se déconnecter du compte Google puis se reconnecter.

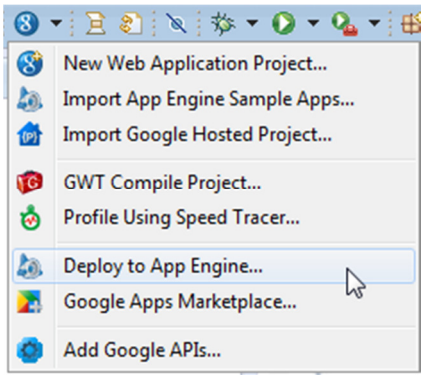


Figure 21 Déploiement d'application depuis Eclipse

5.11.5 Débogage

Le plug-in fourni par Google App Engine nous permet aussi d'utiliser la fonction de débogage d'Eclipse. Cette fonction n'est bien entendu valide que lors de l'utilisation du serveur de développement. Afin que cette fonctionnalité soit disponible, il est nécessaire d'exécuter l'application via l'outil de debug ( ) comme indiqué dans la section *Exécution sur le serveur de développement*.

Pour accéder à l'environnement de debug, il est nécessaire d'insérer un breakpoint dans la partie du code à inspecter puis d'exécuter l'application sur le serveur local. L'insertion d'un breakpoint se fait en double cliquant dans la marge à droite de la ligne de code souhaitée. Il est aussi possible de faire clique-droite dans la marge, puis *Toogle Breakpoint*. Les breakpoints se repèrent grâce aux points bleus présents dans la marge de droite ( ).

Dès que l'utilisateur naviguera sur la page faisant exécuter le breakpoint, le message de la Figure 22 sera affiché dans Eclipse. Il s'agit d'une requête demandant de confirmer que nous souhaitons changer de perspective.

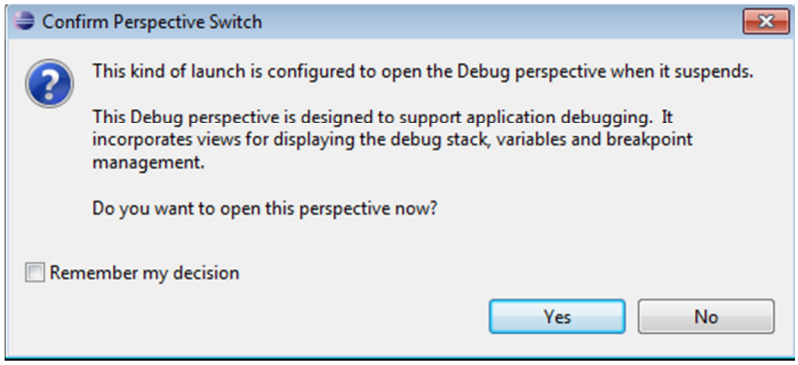


Figure 22 Message de confirmation du changement de perspective, Eclipse

Dès lors que le bouton *Yes* est appuyé, Eclipse passe dans la perspective *Debug* et affiche le code sur lequel le breakpoint avait été inséré. Depuis cette perspective, il est possible de visualiser l'état de toutes les variables au moment de l'arrêt. Les boutons *Step into* ( ) et *Step over* ( ) permettent d'avancer dans le code alors que le bouton *Resume* ( ) reprend l'exécution automatique du code.

Lors du debug, l'utilisateur verra son navigateur rechercher la page tant que l'exécution du code n'aura pas repris.

6 Protocole HTTP

Les communications entre le client et le serveur sont effectuées via le protocole HTTP. HTTP, HyperText Transfer Protocol, est un protocole de communication de la couche application du modèle OSI. Il est utilisé avec le protocole TCP de la couche de transport qui apporte une connexion fiable entre le client et le serveur. Le terme client fait référence au navigateur web d'un utilisateur. Lorsqu'on parle du serveur dans HTTP, il s'agit d'un serveur HTTP qui est conçu pour traiter les requêtes HTTP.

L'utilisation de ce protocole a pour but de permettre l'échange de données entre un client et un serveur. Ces données seront généralement au format HTML et seront accessibles grâce à une chaine de caractères spécifique appelée URL. Une URL, Uniform Resource Locator, est une chaine de caractères ASCII permettant de désigner une ressource sur internet. Il est possible de transférer des données de différents formats grâce au standard MIME. MIME, Multipurpose Internet Mail Extensions, décrit le type du contenu transmis ainsi que l'encodage utilisé dans des en-têtes spéciales.

Dans les paragraphes précédents, il est annoncé que HTTP utilise une connexion fiable grâce au protocole TCP. Cela veut dire que les informations transmises entre le client et le serveur sont garanties correctes. Il ne peut pas y avoir de pertes, de modifications ou d'autres erreurs lors de la transmission des données. Les informations reçues sont alors exactement identiques aux informations envoyées.

Pour obtenir une ressource, un utilisateur va entrer une URL dans son navigateur web qui envoie une requête HTTP sur le serveur correspondant à l'URL. Le serveur reçoit cette requête et la traite en interne de différentes manières. Le traitement de la requête ne fait pas partie du protocole HTTP. Après traitement, le serveur renvoi une réponse HTTP au client. Cette réponse contient la ressource demandée ou un message d'erreur si la ressource n'est pas disponible.

Il est possible de sécuriser les transmissions effectuées avec HTTP en y ajoutant une couche de sécurité apportée par le protocole SSL ou TLS. Le nom de HTTP avec cette couche supplémentaire devient alors HTTPS pour *HyperText Transfert Protocol Secure*. Lorsque cette option est activée, la confidentialité ainsi que l'intégrité de la communication est assurée. L'identité du serveur est aussi assurée par un certificat émis par une entité de confiance.

Le protocole TCP utilisé par HTTP utilise des numéros de port afin de démultiplexer les paquets et les diriger vers le bon service de la couche application. Les numéros de port sont indiqués dans les en-têtes TCP, le protocole HTTP n'a alors pas à se soucier de ce problème. Il existe des numéros de port, dit "port bien connu", qui sont attribué de manière permanente à certains services importants. Pour le protocole HTTP, le port défini est le port 80 alors qu'il s'agit du port 443 pour HTTPS.

L'analyse des communications HTTP entre le client et le serveur peut être effectuée avec le logiciel Wireshark ou l'outil pour Firefox, Firebug. L'utilisation de Firebug à l'avantage de n'afficher que les informations relatives à l'application.

6.1 Requêtes

La requête HTTP est envoyée du client vers le serveur. Elle est composée d'une ligne de requête, des champs d'en-tête et du corps de la requête.

La ligne de requête indique la méthode, l'URL et la version du protocole que le client utilise. Cette ligne est la seule qui est obligatoire pour chaque requête. Toutes les requêtes sont effectuées à travers des méthodes définies dans le protocole HTTP. Les méthodes les plus utilisées sont présentées dans le chapitre suivant, *Méthodes*.

Les champs d'en-tête représentent des informations supplémentaires sur la requête ou sur le client. Chaque ligne d'en-tête est composée d'un identifiant de l'en-tête suivi de deux points (:) puis de la valeur qui lui est associée. Il existe un grand nombre d'en-tête qui peuvent être utilisées par le protocole HTTP, le chapitre *En-têtes* présente les en-têtes les plus fréquentes.

Le corps de la requête doit être séparé des autres parties par une ligne vide. Le corps est un ensemble de lignes qui permettent l'envoi de données. Lorsque l'utilisateur envoie des données avec la méthode POST, les données sont présentent dans le corps de la requête.

6.1.1 Méthodes

Les communications sont effectuées à travers un certain nombre de méthodes proposées par le protocole HTTP. Voici une brève présentation des méthodes les plus utilisées.

GET Cette méthode permet au client de demander une ressource au serveur. Elle est par conséquent très fréquemment utilisée. A la réception de cette méthode, le travail du serveur est d'envoyer une page en réponse. GET ne doit pas avoir d'effet durable sur la ressource, plusieurs GET doivent pouvoir être exécutés sans avoir d'effet sur la ressource. La Figure 23 est une capture d'une demande de ressource effectuée par la méthode GET. On y retrouve les différents éléments évoqués dans les paragraphes précédents.

Figure 23 Capture Wireshark d'une trame GET

POST Cette méthode est utilisée lorsque le client souhaite envoyer des données au serveur. Un exemple d'utilisation de la méthode POST et un formulaire en ligne que le client rempli et renvoie au serveur. Le serveur récupère alors les informations et crée ou modifie des ressources en fonction des valeurs reçues du client. La Figure 24 est une capture de trame d'une communication HTTP avec la méthode POST. On y voit alors les paramètres passés par cette méthode dans la section *Line-based text data*. Chaque paramètre est composé du nom du paramètre, un signe égal, et de sa valeur. Les différents paramètres sont séparés par des esperluettes (&). La taille du corps de la requête POST est illimitée.

```
Hypertext Transfer Protocol
  POST /AddUserSensor HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): POST /AddUserSensor HTTP/1.1\r\n]
      Request Method: POST
      Request URI: /AddUserSensor
      Request Version: HTTP/1.1
    Host: apps-test-tb2013.appspot.com\r\n
    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:20.0) Gecko/20100101 Firefox/20.0\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3\r\n
    Accept-Encoding: gzip, deflate\r\n
    Referer: http://apps-test-tb2013.appspot.com/newUserSensor.jsp\r\n
    [truncated] Cookie: ACSID=AJKiYcE2kUOLtRzoGyBLyJpED2gzF1ljbDbyvub1cqFmYPgsGDVS0TYwad-BaH6
    Connection: keep-alive\r\n
    Content-Type: application/x-www-form-urlencoded\r\n
    Content-Length: 68\r\n
    \r\n
    [Full request URI: http://apps-test-tb2013.appspot.com/AddUserSensor]
Line-based text data: application/x-www-form-urlencoded
    sensorID=mySensorID&sensorKey=mySensorKey&sensorNote=myPersonnalNote
```

Figure 24 Capture Wireshark d'une trame POST

6.1.2 En-têtes

Les en-têtes permettent de spécifier des caractéristiques supplémentaires à la trame envoyée via le protocole HTTP. Voici une présentation non-exhaustive des différentes en-têtes les plus utilisées.

Host	Cette en-tête est vraiment importante. Elle permet d'indiquer quel est le site web adressé par la requête. Ceci est nécessaire si un serveur héberge plusieurs sites sur la même adresse IP.
Accept	Indique les types MIME de contenu que le client peut accepter.
Content-Type	Cette en-tête fournit le type de contenu présent dans le corps de la requête.
User-Agent	Cet élément indique quel est le navigateur internet utilisé par l'utilisateur effectuant la requête.
Referer	Permet d'indiquer l'URL qui a fourni au client le lien pour effectuer cette requête. Il est ainsi possible de savoir où était l'utilisateur lorsqu'il a effectué la requête.

6.1.3 Paramètres

Il est possible de transmettre des valeurs depuis le client vers le serveur grâce à l'utilisation de paramètres. Le passage de paramètres consiste à ajouter une chaine de caractères dans l'URL. Cette chaine de caractères représentera les différents paramètres qui sont passés au serveur. Elle est séparée de l'URL normale grâce au caractère "?". Vient ensuite le nom du paramètre, un signe égal, et enfin sa valeur. Si plusieurs paramètres sont transmis, ils seront séparés les uns des autres par des esperluettes (&). Avec cette méthode, les paramètres sont obligatoirement de type string.

Si un utilisateur souhaite transmettre des informations au serveur lorsqu'il effectue une requête avec la méthode GET, il ne pourra utiliser que des paramètres. Il s'agit effectivement de la seule méthode possible implémentée par HTML pour transmettre des paramètres avec la méthode GET. La taille de l'URL étant limitée en nombre de caractères par les navigateurs internet, le nombre et la longueur des paramètres passés au serveur seront limités.

Les paramètres sont passés en claire dans les URLs et les utilisateurs y ont donc accès. Ils peuvent alors facilement ajouter, supprimer ou modifier des paramètres qui seront utilisés par le serveur. Ceci est un problème de sécurité pour le serveur. Comme pour toutes les saisies utilisateurs, il est alors nécessaire de vérifier chaque valeur reçue du client par cette méthode de passage des paramètres.

6.2 Réponse

Les réponses HTTP sont envoyées au client suite à une requête qu'il a effectuée auprès du serveur. La composition d'une réponse ressemble à celle d'une requête. Elle est composée d'une ligne de statut, des champs d'en-tête ainsi que du corps de la réponse. Les réponses HTTP ne sont pas transmises avec des méthodes comme c'était le cas pour les requêtes. Elles répondent à des requêtes et donc correspondent à la valeur de retour de la méthode.

La ligne de statut indique la version du protocole utilisé, le code de statut ainsi que sa signification. Cette ligne permet alors d'obtenir l'état du traitement via un numéro, le code, ainsi que la signification textuelle de ce code. Les codes renvoyés lors des réponses HTTP sont présentés dans le chapitre *Code de réponse*.

La réponse HTTP, tout comme la requête, peut comporter des champs d'en-tête permettant d'ajouter des informations à la transmission. Ces informations concernent le type de réponse ainsi que certaines caractéristiques de la réponse elle-même ou du serveur.

Le corps de la réponse est l'élément le plus important attendu par le client. Il contient la ressource demandée lors de la requête HTTP. La Figure 25 est une capture Wireshark d'une réponse HTTP à une requête GET. On y voit les différentes parties décrites ci-dessus. De plus, on remarque que le corps de la réponse contient la page web demandée au format HTML.

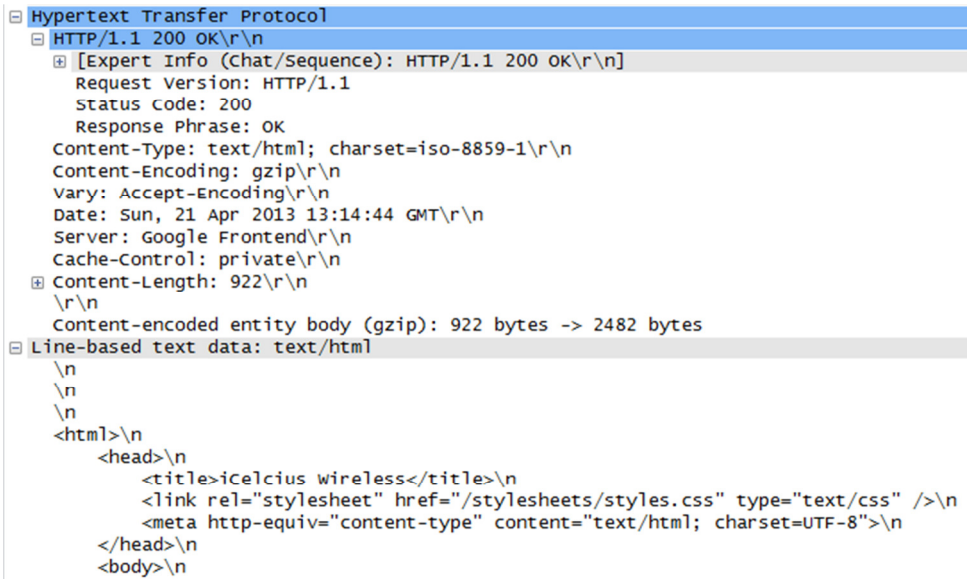


Figure 25 Capture Wireshark d'une réponse à une requête GET

6.2.1 Code de réponse

Le code de réponse présent dans une réponse HTTP permet d'indiquer au client l'état de sa requête. Il est alors possible de savoir si la requête a abouti ou si une erreur s'est produite. En cas d'erreur, le code de réponse permet d'avoir connaissance de l'erreur survenue. Les codes d'erreur sont composés de trois chiffres.

Les codes de réponse commençant par 20X sont des codes de réussite, les 30X sont les codes de redirection, les 40X sont des erreurs provenant du client alors que les 50X sont des erreurs provenant du serveur. Une liste complète des codes d'erreur pouvant être rencontrés est donnée en annexe.

7 Conception application web

7.1 Accès à l'application en ligne

Ce chapitre traite de l'application web créée pour ce projet. Lors de la création du projet dans la console d'administration de App Engine, le mode d'authentification choisi a été *Google Account*. L'identifiant de l'application est *apps-test-tb2013* et l'URL donnant l'accès au site web est *http://apps-test-tb2013.appspot.com/.*

7.2 Langage de programmation

Google App Engine nous met actuellement à disposition deux langages de programmation pleinement supportés par sa plateforme, Java et Python. Ne connaissant pas du tout Python et ayant de bonnes connaissances de Java, le choix du langage de programmation s'est vite porté sur Java.

Notre application serveur se base sur la spécification *Java Enterprise Edition* qui est une plateforme Java facilitant le développement d'applications web exécutées sur un serveur d'applications. App Engine ne supporte pas toutes les fonctionnalités imposées par le standard Java EE. Nous utiliserons *Java Server Pages (JSP)* ainsi que la technologie des Servlets Java afin de créer notre application.

7.3 Fonctionnement

Lorsqu'un utilisateur souhaite accéder à une page web, il utilise son navigateur internet. L'utilisateur va entrer une URL dans son navigateur afin de rechercher la page qu'il souhaite. La communication entre le client, via son navigateur internet, et le serveur se fait grâce au protocole HTTP. Lorsqu'un client souhaite accéder à une page web d'une application hébergée sur App Engine, il effectue une requête HTTP sur les serveurs de Google. Le serveur récupère alors la requête HTTP et la transmet à l'application correspondante. L'application s'occupe de traiter la requête comme elle le souhaite avec les langages de programmation Java et JSP. Le but du trainement est de créer une nouvelle page web dynamique que l'application retournera au serveur en tant que réponse à envoyer au client. Le serveur renvoie ensuite au client la réponse HTTP qui contient une simple page web, constituée avec du HTML et CSS, que le navigateur du client pourra afficher.

Le serveur est décomposé en différentes parties qui ont des rôles différents. L'ensemble qui compose le serveur complet est appelé un serveur d'applications. Comme nous venons de le voir, le client contacte le serveur via une requête HTTP, cette requête est en fait adressée au serveur HTTP de Google App Engine. Le travail du serveur HTTP est d'écouter sur le port réservé pour le protocole HTTP, le port 80, afin de détecter toutes les requêtes entrantes. Les requêtes reçues sont analysées et transmises au conteneur. Le conteneur se compose entre autre d'un load-balancer permettant de répartir les charges sur les différents clusters de Google. La requête est ensuite passée à l'application voulue. Cette dernière traite le contenu de la requête et renvoie une réponse au client. La Figure 26 illustre le mécanisme en place entre le client et le serveur.

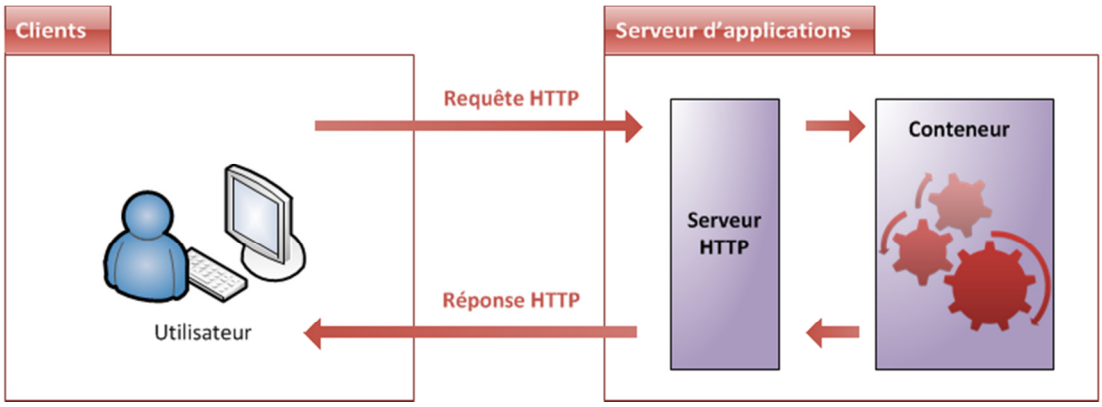


Figure 26 Fonctionnement Client-Serveur

Lorsque le serveur HTTP reçoit une requête, il identifie l'application grâce à l'URL indiqué dans la requête. Cette URL est unique à notre application, ce qui permet au serveur HTTP de rediriger la requête au bon endroit. Lorsque l'application reçoit une requête, le premier fichier analysé est le descripteur de déploiement, *web.xml*. Le descripteur de déploiement redirige la requête vers la bonne Servlet et lui transmet les éléments de requête et de réponse via l'API des Servlets Java. La Servlet pourra alors effectuer le traitement de la requête ou la transférer à d'autres Servlets pour la gestion.

7.4 Fichiers de configuration

Lors de la création du projet comme présenté dans le chapitre sur le plug-in Eclipse, des fichiers sont générés automatiquement dans notre projet. Ces fichiers permettent la configuration de tous les paramètres concernant l'application tels que l'exécution de l'application et la gestion des index du datastore. Pour rappel, le répertoire de travail est décomposé en un sous-répertoire *src* qui contient nos classes Java et un sous-répertoire *war* qui contient l'entièreté de l'application en format WAR. La configuration de notre application peut se faire via un unique fichier YAML ou par deux fichiers XML. Bien que la configuration de l'application via le fichier YAML semble plus aisée, nous allons utiliser le format XML qui permet de porter notre application plus facilement sur d'autre plateforme.

7.4.1 Descripteur de déploiement

Le fichier *web.xml* situé sur le chemin *war/WEB-INF* est le descripteur de déploiement de l'application. Ce fichier est standard pour toutes les applications web fonctionnant avec des Servlets Java, il n'est donc pas spécifique à App Engine. Lorsque le serveur web reçoit une requête, il lira ce fichier afin de savoir quelle classe de Servlet appeler. Ce fichier permet notamment de mapper une URL avec une Servlet.

La déclaration d'une Servlet se fait avec l'élément *<servlet>*, auquel s'ajoute une déclaration de mappage associée à cette Servlet grâce à la balise *<servlet-mapping>*. Entre les éléments *<servlet>*, nous pouvons indiquer le nom utilisé pour identifier la Servlet, la classe Java associée à la Servlet ainsi que des paramètres d'initialisation qui seront passé à la Servlet Java. Entre les éléments gérant le mapping, on retrouve le nom de la Servlet précédemment déclaré ainsi que l'URL correspondante à la Servlet. Lorsque le serveur recevra cette URL, il appellera la classe associée à la Servlet. La Figure 27 map l'URL */admin/AddUserSensor* avec la Servlet *AddUserSensor* du package *userSensor*.

```
<!-- add user-sensor-->
<servlet>
    <servlet-name>addUserSensor</servlet-name>
    <servlet-class>userSensor.AddUserSensor</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>addUserSensor</servlet-name>
    <url-pattern>/admin/AddUserSensor</url-pattern>
</servlet-mapping>
```

Figure 27 Mappage d'une Servlet dans web.xml

Si l'application utilise des comptes utilisateurs basé sur Google Account, il est possible de restreindre l'accès à certaines URLs via le descripteur de déploiement. L'élément *<security-constraint>* permet de spécifier le type de connexion que doit avoir l'utilisateur pour qu'il puisse accéder à une certaine URL. Si un utilisateur essaie d'accéder à une page sécurisée et qu'il ne répond pas à la contrainte de sécurité, il est redirigé vers la page de connexion de Google. Une fois que l'utilisateur se sera connecté et qu'il répondra à la contrainte imposée, il sera redirigé directement sur la page qu'il souhaitait. La Figure 28 indique que toutes les URL commençant par */admin/* ne seront accessibles que par un administrateur.

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

Figure 28 Contrôle du statut de l'utilisateur dans web.xml

Le descripteur de déploiement permet également de forcer l'utilisation d'URLs sécurisées via le protocole HTTPS. Ce protocole est une amélioration de HTTP avec l'ajout du protocole SSL ou TLS afin de gérer la sécurité de la transmission. Le domaine gratuit qui nous est fourni, appspot, supporte les URLs sécurisées. La déclaration des URLs qui doivent être sécurisées est contenue dans l'élément *<security-constraint>* avec la propriété de l'élément *<transport-guarantee>* valant *CONFIDENTIAL*. Les requêtes HTTP effectuées sur une URL dont le protocole de transport est forcé à HTTPS est automatiquement redirigé vers l'URL comprenant HTTPS. La Figure 29 indique que toutes les URLs utilisées (*) doivent utiliser HTTPS.

Le serveur de développement ne tient pas compte du protocole de transport. Des requêtes HTTP ne seront alors pas redirigées vers une URL HTTPS. Lors du développement de l'application, il peut être utile de désactiver cette option. En effet lorsque HTTPS est activé, il n'est plus possible d'observer les trames échangées entre le client et le serveur via Wireshark.

```
<security-constraint>
    <web-resource-collection>
        <url-pattern>*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Figure 29 Activation de HTTPS dans web.xml

Grâce au descripteur de déploiement, il est possible d'appliquer des filtres aux requêtes sur certaines URLs. Un filtre entrant est un objet Java positionné avant nos Servlets traitant la requête. Il est aussi possible de définir des filtres sortant permettant de modifier la réponse envoyée à l'utilisateur. Contrairement aux Servlets, les filtres ne créent pas de réponse mais effectuent des contrôles ou des modifications sur les requêtes et réponses de l'utilisateur. La mise en place de filtres permet alors d'effectuer certains tests sur des requêtes et de rediriger l'utilisateur selon certains critères. Un filtre permet donc la poursuite de la requête sur un autre filtre ou sur la Servlet mappée à l'URL. La déclaration d'un filtre se fait avec l'élément *<filter>* qui comporte un nom et une classe Java implémentant l'interface *Filter*. Comme pour le mapping d'une Servlet, l'élément *<filter-mapping>* associe un nom de filtre existant avec une URL. La classe Java associée doit implémenter l'interface *Filter* et la méthode *doFilter*. C'est dans cette méthode que les tests de filtrage seront effectués.

Le fichier *web.xml* met en place quatre filtres qui seront utilisés pour contrôler l'accès des utilisateurs aux pages de l'application. Les liens dont le chemin commence par */connect/* seront accessibles uniquement pour les utilisateurs connectés à leur compte Google. Les liens dont le chemin commence par */user/admin/* seront accessibles uniquement pour les utilisateurs défini comme administrateur de l'application. Ces deux filtres sont mis en place directement par Google, aucun code de contrôle de doit donc être fourni.

Les liens dont le chemin commence par */sensor/* seront accessibles uniquement pour les sensors enregistrés dans l'application. Ce contrôle est effectué par le fichier *sensorData/CheckSensorConnection_filter.java*. Ce fichier vérifie que les paramètres *sensorID* et *sensorKey* sont passés par la requête et qu'ils correspondent à un sensor de l'application.

Les liens dont le chemin commence par */user/* seront accessibles uniquement pour les utilisateurs connectés à leur compte Google et étant déjà enregistrés dans l'application. Ce contrôle est effectué grâce au filtre *appUser/CheckUserConnection_filter.java*. La Figure 30 montre le mapping du filtre *CheckUserConnection_filter* dans le package *appUser* avec toutes les URLs commençant par */user/*. La Figure 31 représente la classe Java implémentant l'interface *Filtre* qui vérifie si l'utilisateur fait partie des utilisateurs enregistrés dans le datastore. Si l'utilisateur est existant, la requête continue normalement. Sinon, un log est inscrit dans le système et l'utilisateur est renvoyé sur la page d'accueil de notre application.

```
<filter>
    <filter-name>userConnectionFilter</filter-name>
    <filter-class>AppUser.CheckUserConnection</filter-class>
</filter>
<filter-mapping>
    <filter-name>userConnectionFilter</filter-name>
    <url-pattern>/user/*</url-pattern>
</filter-mapping>
```

Figure 30 Filtre dans web.xml

```java
public class CheckUserConnection_filter implements Filter {

    private FilterConfig filterConfig;

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)
        throws IOException, ServletException {

        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();

        /** DAO class to access easily to the datastore */
        DAO_AppUser daoUser = new DAO_AppUser();

        // Check if user is already registered
        if(user != null && daoUser.exists(user.getUserId())){
            filterChain.doFilter(request, response);
        }
        else{
            // User not connected
            Log.logInfo("User connected but not registered, redirect to welcome page");
            ((HttpServletResponse)response).sendRedirect("/");
        }
    }
```

Figure 31 Classe Java associée au filtre de web.xml

Web.xml permet aussi la gestion d'erreur. Il est possible de définir la page envoyée à l'utilisateur en cas d'exception Java ou de code d'erreur de HTTP. L'élément *<error-page>* permet de définir le comportement en cas d'erreur. Dans cet élément, un code d'erreur HTTP ou un type d'exception Java est précisé. Le chemin de l'URL de la page d'erreur est aussi précisé. Actuellement, les erreurs HTTP 403, 404 et 500 ne sont pas prises en compte par cette configuration. L'erreur 404 indique que l'URL reçue n'a pas de Servlet qui lui est mappée. L'erreur 403 indique que le serveur a bien compris la requête mais ne l'effectue pas. Ceci peut être dû au fait que les quotas de l'application sont dépassés ou que l'utilisateur n'a pas les droits pour accéder à l'URL demandée. Si l'erreur 403 survient lorsque l'utilisateur tente d'accéder à une page réservée à un administrateur, la gestion de l'erreur fonctionne. Lorsque l'erreur 403 est due aux quotas, la gestion de cette erreur est possible grâce au fichier de configuration de l'application. L'erreur 500 est quant à elle une erreur interne de App Engine.

```xml
<!-- User tries to access to an admin only page but is not an admin -->
<error-page>
    <error-code>403</error-code>
    <location>/error/notAnAdminError</location>
</error-page>

<!-- Java.io.Exception -->
<error-page>
    <exception-type>java.io.Exception</exception-type>
    <location>/WEB-INF/DefaultPages/unexpectedError.jsp</location>
</error-page>
```

Figure 32 Gestion des erreurs dans web.xml

7.4.2 Configuration de l'application

Le fichier *appengine-web.xml* situé sur le chemin *war/WEB-INF* est un fichier de configuration utilisé par App Engine pour le déploiement et l'exécution d'une application. Ce fichier contient notamment l'ID de l'application ainsi que le numéro de version. Ce fichier ne fait pas partie du standard utilisé avec les Servlets de Java, il est spécifique et indispensable pour les applications App Engine.

Les données minimum requises dans ce fichier sont l'identifiant de l'application ainsi que la version correspondante au code. L'identifiant de l'application est situé entre les éléments *<application>* alors

que la version se trouve entre les éléments *<versions>.* L'identifiant de l'application est l'identifiant enregistré lors de la création de l'application dans la console d'administration de App Engine. La version peut contenir des lettres, chiffres et tirets. Lorsqu'on transfert l'application vers App Engine, si le numéro de version indiqué dans *appengine-web.xml* est déjà présent sur le réseau, la version du réseau est écrasée. Sinon, une nouvelle version apparait dans la console d'administration d'où nous pouvons définir quelle est la version visible pour les utilisateurs. Si nous souhaitons atteindre une version qui n'est pas celle définie par défaut, nous pouvons y accéder via une URL comme suit: *http://<version_ID>.latest.<application_ID>.appspot.com*. Il est important de préciser que lorsqu'on utilise cette URL alors que le fichier *web.xml* force l'utilisation du protocole HTTPS, le navigateur nous informe que le certificat n'est pas signé. Il faut alors accepter le certificat afin d'avoir accès à la page voulue. Depuis avril 2013, l'accès aux versions précédentes d'applications avec l'option HTTPS activé peut se faire avec une URL correspondant au format suivant : https*://<version_ID>*-dot-latest-dot-*<application_ID>*.appspot.com. Cette manière d'accéder à l'application ne devrait plus lever d'erreur de certificat.

```
<application>apps-test-tb2013</application>
<version>1</version>
```

Figure 33 Informations minimales dans appengine-web.xml

Lorsqu'une application utilise des services entrant comme le service *Mail* pour la réception d'email, il est nécessaire d'activer le service explicitement dans ce fichier de configuration. *Appengine-web.xml* permet également de définir des variables d'environnement qui seront accessibles dans toute l'application. Une autre configuration est l'ajout de pages dans la console d'administration ou la gestion de pages d'erreur personnalisées. Les erreurs gérées par ce fichier de configuration sont les erreurs dues aux dépassements de quota, à la configuration contre le déni de service ou à la réception d'un timeout. La Figure 34 illustre la gestion de toutes les erreurs possibles dans le fichier *appengine-web.xml*. On voit qu'il est également possible de ne pas spécifier le code d'erreur afin de définir une page d'erreur par défaut.

```
<static-error-handlers>
  <handler file="/errors/default_error.html"/>
  <handler file="/errors/over_quota.html" error-code="over_quota"/>
  <handler file="/errors/dos.html" error-code="dos_api_denial"/>
  <handler file="/errors/timeout.html" error-code="timeout"/>
</static-error-handlers>
```

Figure 34 Gestion des erreurs dans appengine-web.xml

Notre application App Engine s'exécute sur de multiples serveurs dont le nombre est ajusté selon la quantité de requêtes effectuées sur notre application. Par défaut, les requêtes sont envoyées en série à un serveur web. Nous pouvons modifier ce paramètre afin d'envoyer les requêtes de manières concurrentes. Pour ce faire, nous devons attribuer la valeur *true* entre les éléments *<threadsafe>* de notre fichier *appengine-web.xml*.

```
<!--
  Allows App Engine to send multiple requests to one instance in parallel:
-->
<threadsafe>true</threadsafe>
```

Figure 35 Activation de l'envoi des requêtes en parallèle dans appengine-web.xml

Sur App Engine les fichiers statiques, des pages dont le contenu n'est jamais modifié, ne sont pas stocké sur les mêmes serveurs que les fichiers d'application. Par défaut, App Engine définit tous les fichiers situés dans */war* comme des fichiers statiques, à l'exception des fichiers situés sous */war/WAR-INF* et des fichiers JSP. Les fichiers correspondant à ces deux exceptions sont des ressources utilisées par l'application. Ils sont donc stockés sur les serveurs d'application d'App Engine au même titre que l'application elle-même. Il est possible de définir quels sont les fichiers statiques manuellement, le délai d'expiration du cache ou le répertoire racine des fichiers statiques. Pour notre application, la configuration par défaut nous convient, nous n'allons alors rien ajouter concernant les fichiers statiques au fichier de configuration de l'application.

Appengine-web.xml nous permet également de configurer certains paramètres que nous allons considérer moins utiles pour notre application. Via ce fichier, il est possible d'activer les sessions, d'interdire le protocole HTTPS ou de désactiver la pré-compilation. Nous n'allons pas activer les sessions car cela consomme des ressources du datastore ainsi que de la mémoire cache. Ces paramètres gardent leurs valeurs par défaut.

7.4.3 Configuration des index

Lorsque notre application génère une requête sur le datastore, le magasin de donnée d'App Engine utilise les index afin d'obtenir rapidement une réponse. Ces index doivent être mis à jour à chaque modification d'une entité. Ceci implique que le datastore doit connaitre à l'avance toutes les requêtes pouvant être effectuées par l'application. Le fichier de configuration des index s'appelle *datastore-indexes.xml* et il est placé sur le chemin */war/WEB-INF*.

Il est possible de créer les index manuellement en éditant le fichier d'index. Néanmoins, la procédure de création manuelle ne s'avère pas très aisée et peut être source d'erreurs. Nous avons alors choisi de laisser App Engine créer le fichier d'index à notre place. Pour que GAE gère les index à notre place, nous devons ajouter l'attribut *autoGenerate="true"* à l'élément *<datastore-indexes>* du fichier *datastore-indexes.xml.* Si notre application ne comprend pas de fichier se nommant *datastore-indexes.xml*, la configuration automatique est activée par défaut.

```
<datastore-indexes autoGenerate="true">

</datastore-indexes>
```

Figure 36 Configuration automatique dans datastore-indexes.xml

Lors de la présentation d'Objectify, il a simplement été annoncé que le fichier d'index, *datastore-indexes.xml*, était géré par App Engine. En fait, ce fichier est automatiquement généré par le serveur de développement. Le fichier d'index modifié par le serveur local se nomme *datastore-indexes-auto.xml* et se trouve dans le répertoire */war/WEB-INF/appengine-generated/.*

Lorsque notre application s'exécutant sur le serveur de développement effectue une requête sur le datastore, le serveur recherche dans les deux fichiers d'index. S'il ne trouve pas d'entrée correspondante à la requête, il ajoute la configuration nécessaire dans le fichier d'index automatique. Lors du déploiement de notre application sur les serveurs de App Engine, l'outil de déploiement reprend le fichier d'index ainsi que le fichier créé automatiquement afin de définir les index à importer sur le serveur.

7.5 Pattern MVC

Notre application respectera au mieux le modèle de conception MVC, Model View Controller. Le pattern MVC est un principe de conception d'application qui sépare le modèle, la vue et le contrôleur. Le but d'un tel modèle est de faciliter la maintenance et l'évolutivité d'une application. En effet, le code respectant le modèle MVC suit des règles préétablies qui sont connues par la pluparts des développeurs. La création d'une application respectant ce principe demande par contre l'écriture de plus de code, impliquant un temps de développement plus long.

Une application est donc divisée en trois parties distinctes, le modèle, la vue et le contrôleur. Le modèle gère tous ce qui concerne les données relatives à l'application. Plus concrètement, le modèle contiendra nos classes Java représentant les sensors et les utilisateurs par exemple. Ces classes peuvent contenir des attributs, les informations sur nos objets, et des méthodes permettant la gestion des attributs. De plus, le système de stockage de nos données sera aussi contenu dans le modèle.

La vue s'occupe de l'interaction avec l'utilisateur, la mise en forme et l'affichage des données. Dans notre application, la vue sera implémentée par les pages JSP. Ces pages permettent de générer du HTML sur le serveur avant de le renvoyer au client. Ces pages doivent se contenter d'afficher des informations à l'utilisateur et rien d'autre.

Le contrôleur a le rôle de traiter toutes les actions provenant de l'utilisateur. Il reçoit une requête de l'utilisateur et peut éventuellement faire appel à des traitements du modèle. Le contrôleur peut ensuite personnaliser les réponses envoyées au client via la vue. Le contrôleur de notre application sera constitué de Servlet Java.

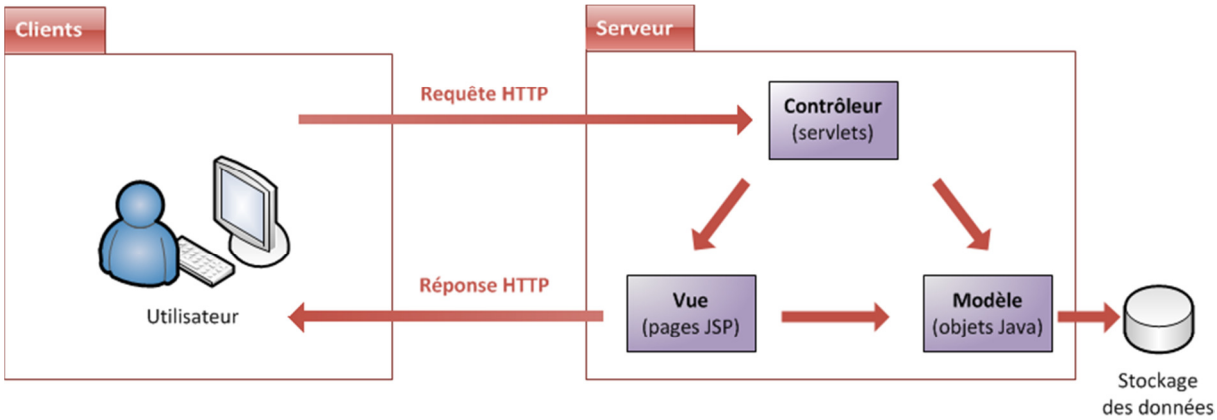


Figure 37 Représentation du pattern MVC

7.6 Contrôleur : Servlets Java

Les Servlets Java auront le rôle de contrôleurs de notre application. C'est donc ces éléments qui s'occuperont de gérer les différents évènements arrivant sur notre application.

Une Servlet Java est une simple classe Java qui est capable de traiter des requêtes et de générer des réponses personnalisées. Les requêtes sont très souvent du type HTTP mais elles pourraient être d'un autre type. Les Servlets seront les points d'entrée de notre application. Il n'existe pas de fichier comportant une méthode *main* qui joue le rôle de point d'entrée prédéfini. C'est donc grâce au

descripteur de déploiement, *web.xml*, que la requête sera dirigée vers le point d'entrée de l'application, une Servlet.

Le paquetage *javax.servlet* contient un grand nombre de classes et d'interfaces dont l'interface *Servlet*. Cette interface est une base que toutes les applications souhaitant utiliser les Servlets doivent implémenter. Cette interface impose l'implémentation de méthodes permettant la gestion de la Servlet. Il faut notamment implémenter des méthodes pour la création, l'utilisation et la suppression de la Servlet. Il est alors possible d'implémenter cette interface afin de créer nos Servlets Java.

Cependant, une classe abstraite se nommant *HttpServlet* implémente déjà cette interface et ajoute des fonctionnalités spécifiques pour les Servlets HTTP. La gestion de la Servlet est alors déjà implémentée par cette classe abstraite. Toutes nos Servlet Java hériteront alors de cette classe abstraite. La seule contrainte imposée est la surcharge d'au moins une des méthodes de la classe abstraite *HttpServlet*.

Pour notre application, seules les méthodes *doGet* et *doPost* seront éventuellement surchargées. Ces deux méthodes prennent en paramètre une requête HTTP de type *HttpServletRequest* et une réponse HTTP de type *HttpServletResponse*. Ces deux paramètres sont très utilisés dans le code des Servlets. Ils possèdent des méthodes permettant la gestion de toutes les valeurs relatives à la requête et à la réponse HTTP. Ces deux éléments nous sont fournis par les classes parentes de *HttpServlet*, contenues dans le conteneur de Servlets. Ce sont ces mêmes classes qui ont le rôle d'associer une requête à la méthode *doXXX* correspondante. Le serveur renverra toujours une réponse au client quel que soit le résultat du traitement de la requête.

Si une requête est redirigée sur une Servlet qui n'implémente par la bonne méthode *doGet* ou *doPost*, un message d'erreur provenant de la classe *HttpServlet* est généré avec le code d'erreur 405. Lorsque tout ce passe bien, la réponse retournée au client contient le code HTTP 200 associé au message *OK*. Ce code est prédéfini par le conteneur de Servlet à la valeur de 200 lors de la génération des paramètres *request* et *response*. Ce code est ensuite modifié ou non par la Servlet selon les événements ayant lieu au court du traitement de la requête.

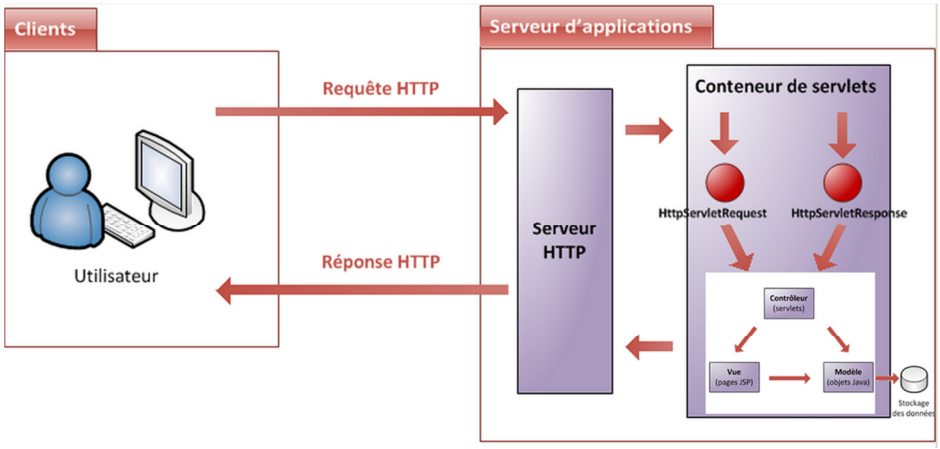


Figure 38 Représentation conceptuelle du fonctionnement avec conteneur de Servlets

La méthode *doGet* est la méthode qui est appelée lors ce que le serveur reçoit une requête GET. Cette requête est utilisée par le client lorsqu'il souhaite récupérer une ressource du serveur web via

une URL. C'est par exemple cette méthode qui est utilisée lorsque l'utilisateur se connecte à notre application via l'URL *http://apps-test-tb2013.appspot.com/*.

La méthode *doPost* correspond à la méthode appelée lors ce que le serveur reçoit une requête POST. Cette requête permet au client de fournir des informations au serveur. C'est cette méthode qui est utilisée lorsque l'utilisateur rempli un formulaire en ligne et le soumet à l'application. Une deuxième utilisation importante de cette méthode est la communication entre le sensor et Google App Engine. Lorsque le sensor aura des nouvelles données à transmettre à l'application, il utilisera la méthode POST.

7.6.1 Gestion des vues

Le rôle de la Servlet est de contrôler les différentes actions des utilisateurs et du serveur. Lorsque la gestion d'une requête HTTP d'un utilisateur est finie, le serveur doit lui renvoyer une réponse. Cette réponse est souvent une page web que l'utilisateur affichera dans son navigateur internet. Cette page ne doit alors pas être générée par le contrôleur mais par la vue du modèle MVC. Cette vue est créée grâce aux pages JSP dont le chapitre *Vue: Pages JSP* est dédié. La Servlet du contrôleur doit alors passer la main à cette vue. Cette action se fait grâce à des méthodes prédéfinies dans le conteneur de Servlets et par l'API Servlet de Java. La Figure 39 montre le code Java qu'il faudra écrire afin de passer la main à la vue. Ce code est étudié dans le paragraphe suivant.

```
// Forward user to the view
request.setAttribute("contentPage", "/errors/errors.jsp");
request.setAttribute("errorMsg", errorMsg);
this.getServletContext().getRequestDispatcher("/WEB-INF/DefaultPages/template.jsp").forward(request, response);
```

Figure 39 Redirection de l'utilisateur vers la vue depuis une Servlet

Comme le code de la Figure 39 provient d'une Servlet Java, le mot clé *this* fait référence à l'instance de la Servlet. La méthode *getServletContext* est ensuite appelée et nous retourne un objet *ServletContext*. Cet objet nous met à disposition des fonctions permettant de communiquer avec le conteneur de Servlet. La méthode que nous utilisons ensuite, *getRequestDispatcher*, nous permet de manipuler une ressource. Grâce à la valeur de retour de cette fonction, un objet *RequestDispatcher*, il nous est possible de faire suivre les paramètres de la Servlet à la vue. Pour rappel, les deux paramètres de la Servlet sont de type *HttpServletRequest* et *HttpServletResponse*. La méthode *getRequestDispatcher* prend comme unique paramètre le chemin absolu vers la page JSP sur laquelle nous voulons être redirigés. Le premier caractère de ce paramètre est donc toujours "/". Dans l'exemple de la Figure 39, la page JSP *template.jsp* est située dans un sous-dossier, *DefaultPages*, qui lui-même est situé dans le dossier *WEB-INF*. La fonction *forward* est ensuite appliquée sur l'objet afin d'être effectivement redirigé vers la page JSP.

7.6.2 Passage d'attributs à la vue

La Figure 39 met aussi en évidence le passage d'attributs entre le contrôleur et la vue. Pour que la vue soit dynamique tout en restant qu'une simple vue, donc non dotée d'intelligence, il est nécessaire de passer des attributs à cette vue.

Les attributs ne sont pas liés au protocole HTTP mais au langage de programmation utilisé par le serveur. Dans notre cas, ce langage est Java. Les attributs ne sont donc pas contenus dans les requêtes HTTP mais dans les objets Java qui portent ces requêtes, les objets de type *HttpServletRequest*. Les attributs sont utilisés pour la transmission de valeurs au sein même du

serveur, entre plusieurs entités Java. Ce sont donc des valeurs créées par le serveur et pour le serveur, le client n'y aura pas accès.

Comme le montre la Figure 39, le paramètre de type *HttpServletRequest* de notre Servlet possède une méthode appelée *setAttribute* qui permet d'enregistrer un attribut supplémentaire à la requête. Cette méthode prend deux paramètres, le premier est le nom donné au nouvel attribut alors que le deuxième est la valeur de cet attribut. Sur l'exemple de la Figure 39, deux attributs sont ajoutés à la requête. Le premier attribut se nomme *contentPage* et sa valeur est */errors/errors.jsp*. Le type d'objet passé par les attributs peut être quelconque.

7.6.3 Spécificités d'App Engine

Les communications entre le client et le serveur sont effectuées via le protocole HTTP. Lorsque le serveur reçoit une requête HTTP, il la transmet à notre Servlet selon le fichier *web.xml*. A la place de transmettre la requête telle quelle à notre Servlet, App Engine effectue certaines opérations dessus. Les en-têtes concernant le transport des données sont retirées par App Engine. La partie transport des communications étant géré par GAE, ces en-têtes ne nous sont pas utiles. La compression des données est par exemple gérée entièrement par App Engine si le client indique qu'il souhaite obtenir des données compressées dans les en-têtes de sa requête. De plus App Engine ajoute des données dans les en-têtes que nous pouvons utiliser dans notre application. Ces en-têtes sont *X-AppEngine-Country*, *X-AppEngine-Region*, *X-AppEngine-City* et *X-AppEngine-CityLatLong* qui nous permettent d'obtenir des informations sur la géolocalisation de l'utilisateur.

Après les modifications des en-têtes de la requête, App Engine appel notre Servlet en lui passant en paramètre un objet requête ainsi qu'un objet réponse. L'objet requête correspond à la requête reçue et traitée par App Engine alors que l'objet réponse est la réponse qui sera envoyé au client après traitement de la requête. App Engine attend ensuite que la Servlet se finisse avant d'envoyer la réponse au client.

App Engine ne prend pas en charge la dernière version des Servlets Java, il utilise actuellement la version 2.5. GAE ne supporte pas le mécanisme de *streaming* qui permet d'envoyer plusieurs réponses à un client suite à une seule requête. De plus, les réponses sont limitées à 32MB. Si cette valeur est dépassée, le client recevra l'erreur 500 indiquant une erreur interne du serveur.

Comme pour la requête, certaines en-têtes de la réponse sont modifiées par Google App Engine avant d'être envoyées au client. Ces en-têtes sont des informations d'encodage ainsi que des informations sur le serveur. Si la réponse est destinée à un administrateur, App Engine ajoute deux en-têtes, *X-AppEngine-Estimated-CPM-US-Dollars* et *X-AppEngine-Ressource-Usage.* La première en-tête fournit une estimation du coût estimé en dollar pour l'exécution de mille requêtes identiques à celle effectuée. L'en-tête *X-AppEngine-Ressource-Usage* indique les ressources utilisées par la requête en termes de millisecondes ainsi que le temps CPU en millisecondes. Ces valeurs ne sont disponibles que sur les serveurs de App Engine, les en-têtes ne sont pas présentes sur le serveur de développement.

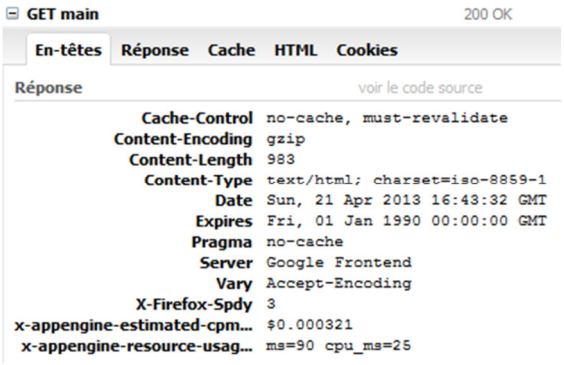


Figure 40 Analyse d'une réponse suite à un GET avec Firebug

Il est important de relever ici que le temps maximal entre la réception d'une requête et l'émission de sa réponse est fixé. Si ce délai est dépassé, App Engine lève une exception, *com.google.apphosting.api.DeadlineExceededException*.

7.7 Vue: Pages JSP

Les pages JSP correspondent à la vue du modèle MVC. C'est donc via ces pages que l'interface utilisateur sera effectuée. Cet affichage comprend le texte à afficher ainsi que sa mise en forme. Ces pages doivent alors être indépendantes du code métier afin de respecter la séparation imposée par le modèle MVC. Les pages JSP seront appelées par les Servlets afin d'entrer en action.

Les fichiers JSP, Java Server Page, sont des fichiers portant l'extension *.jsp. Ces fichiers se présentent au format texte et sont composés de balises spécifiques à la technologie JSP. Ces balises font appel à du code Java de manière totalement transparente pour le programmeur. L'utilisation de fichiers JSP permet la création de pages web dynamiques. Effectivement, les JSP sont exécutées sur le serveur afin de générer une page HTML qui sera envoyée au client. L'utilisation des balises HTML est par ailleurs prise en charge par le format JSP. Il est alors possible de personnaliser les pages pour le client vers lequel la réponse est destinée. Cette personnalisation pourra se faire via des balises spéciales ou directement en ajoutant du code Java dans la page.

Le langage JSP a l'avantage d'être dédié au formatage dynamique de l'affichage. Dans un fichier JSP, il est possible d'accéder aux objets et méthodes de Java tout en permettant l'utilisation de balises pour l'affichage.

Lors de son utilisation par le serveur, la page JSP est traduite en classe Java par le conteneur de Servlet. Nos pages JSP deviennent alors des Servlets Java que le serveur pourra exécuter afin de fournir la sortie de la Servlet à l'utilisateur. Les pages JSP permettent donc d'écrire du code Java sans pour autant écrire en Java. Ceci explique la possibilité d'utiliser du code Java directement dans le fichier JSP.

La page JSP est donc traduite en Servlet Java lors du chargement de l'application ou lors de sa première utilisation. Cette classe est ensuite utilisée lors de chaque requête HTTP destinée à la page JSP. Si le contenu d'une page JSP vient à être modifié, le serveur devra alors recompiler le fichier JSP. Lorsqu'aucune modification n'est apportée au contenu de la page, le serveur réutilise la classe Java créée précédemment, il n'y a alors pas le processus de compilation à chaque chargement de la page

JSP. La Figure 41 montre le traitement d'une Page JSP lors d'une utilisation avec ou sans changement de contenu. Le serveur se base sur les timestamps des pages JSP et des classes Java correspondantes afin de savoir si le fichier JSP a été modifié.

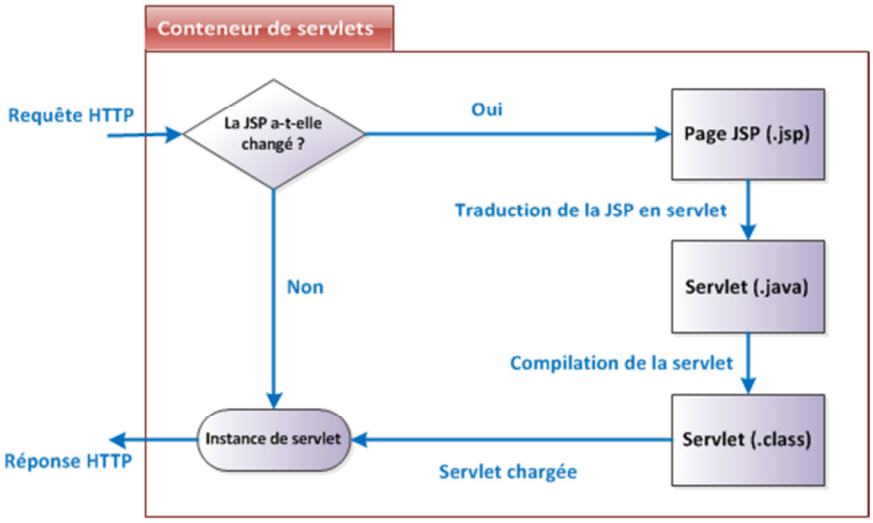


Figure 41 Traitement d'une page JSP

Le serveur traduit nos pages JSP en Servlets, le rôle de ces Servlets reste d'implémenter l'interface utilisateur. Bien que notre contrôleur du modèle MVC soit constitué de Servlets Java, tous les Servlets ne sont pas pour autant des contrôleurs. Les Servlets créées par les pages JSP correspondront donc bien à la vue et non au contrôleur.

Les fichiers présents dans le répertoire *war* de notre application sont accessibles de l'utilisateur via une URL. Afin de s'assurer que l'utilisateur de notre application ne puisse pas accéder directement à une page JSP, nous devons placer ces pages dans le dossier *war/WEB-INF/* de notre répertoire. Ce dossier permet de rendre les fichiers qu'il contient inaccessibles par un lien URL. En faisant ainsi, l'utilisateur est obligé de passer par une Servlet du contrôleur afin d'avoir accès à la page. Nous respectons alors un principe important du modèle MVC en forçant l'utilisation d'une Servlet contrôleur comme point d'entrée de l'application. Toutes les pages JSP seront associées à une Servlet du contrôleur afin d'être accessible par les utilisateurs.

7.7.1 Portée des objets

Le dynamisme de la page est créé grâce à des valeurs passées entre le contrôleur et la vue. Ces valeurs contenues dans des variables peuvent être accessibles depuis différents endroits selon leur portée. La portée des objets, scope en anglais, permet de définir d'où seront visibles les objets créés. Une application comprend quatre portées différentes:

Page	Tous les objets déclarés avec ce scope sont accessibles uniquement dans la page JSP dans laquelle ils ont été déclarés.
Requête	Les objets déclarés avec cette portée seront accessibles durant toute l'existence de la requête courante.

Session — Les objets créés avec la visibilité session seront visibles pendant toute la durée de validité de la session.

Application — Tant que l'application sera exécutée, les objets créés avec ce scope seront accessibles. Tous les clients peuvent avoir accès à cet objet.

Les objets ayant une portée de requête, session ou application peuvent être déclarés depuis une page JSP ou depuis un Servlet Java. Les objets avec un scope page seront uniquement créés par des pages JSP. La Figure 42 représente graphiquement ces explications.

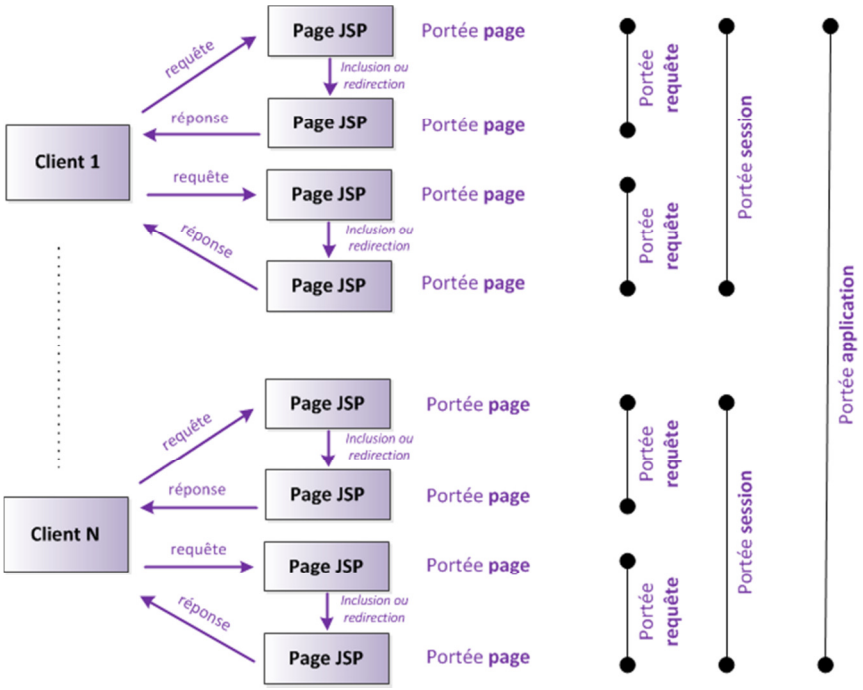


Figure 42 Portée des objets

7.7.2 Récupération de paramètres

Comme nous l'avons vu dans le chapitre *Contrôleur : Servlets Java,* la Servlet du contrôleur peut nous passer des attributs lors de la redirection de l'utilisateur vers la vue. Il est donc nécessaire que la vue puisse récupérer ces attributs. Pour ce faire, nous pouvons passer par du code Java identifié par les balises <% et %>. Ces balises et le code Java qu'elles entourent s'appellent un scriptlet.

Cependant, l'utilisation de code Java dans les pages JSP est à éviter. Les balises qui nous sont mises à disposition permettent d'effectuer les mêmes actions que du code Java mais en nécessitant moins de lignes de code. Il est alors moins probable d'introduire des erreurs dans un fichier plus concis. De plus, en cas d'erreur un scriptlet retournera une page blanche alors que la page JSP peut gérer l'erreur. La maintenance d'un fichier comportant des balises et du code Java est moins aisée que si le fichier ne contient qu'un type de langage. Il est aussi plus facile de différentier le traitement de la vue des autres niveaux du modèle MVC grâce à l'utilisation de balises. Afin d'éviter l'utilisation de code Java dans nos pages JSP, il existe des librairies additionnelles permettant une gestion efficace des fonctionnalités web. L'utilisation de scriptlet Java dans les pages JSP est donc très fortement déconseillée, notamment par Oracle dans ses conventions de codage.

Donc pour récupérer les paramètres qui sont passés à notre page JSP, nous n'allons pas utiliser de scriptlet Java mais les expressions langage. Comme expliqué dans le chapitre *Expression langage*, les EL permettent de récupérer les attributs tout en vérifiant que l'attribut soit bien existant.

7.7.3 Balises et directives JSP

Les fichiers JSP utilisent la technologie JSP qui possède des balises qui lui sont spécifiques. Le tableau ci-dessous, *Tableau 1*, présente quelque unes des balises JSP.

<%-- ... --%>	Commentaires. Ces commentaires sont visibles uniquement du développeur
<%! ... %>	Balise de déclaration. Permet de déclarer des variables
<% ... %>	Balise de scriptlet. Permet d'insérer du code Java dans la page JSP
<%= ... %>	Balise d'expression. Permet d'effectuer un affichage du contenu
<jsp:include ... />	Balise d'inclusion. Permet d'inclure le contenu d'un autre fichier dans le fichier JSP courant.
<jsp:forward … />	Balise de transfère. Permet de rediriger l'utilisateur vers une autre page.

Tableau 1 Tableau des balises JSP

En plus des balises, JSP nous met à disposition des directives qui permettent de contrôler la gestion du fichier JSP. Les directives sont toujours placées entre les balises *<%@* et *%>*. La directive *taglib* permet d'inclure des bibliothèques à notre fichier JSP. La directive *page* permet de définir des informations à propos de la page JSP courante. Nous pouvons notamment définir une page prédéfinie en cas d'erreur dans la page courante. La directive *include* permet d'inclure le contenu d'un fichier dans le fichier courant.

Ce dernier principe est très utile pour le découpage de nos pages en plusieurs sections comme expliqué dans le chapitre *Utilisation d'un template*. Le défaut de cette directive est que le fichier inclus doit être définit avant la compilation. L'utilisation d'un template n'est alors plus possible car le contenu ne pourra jamais varier. La balise *<jsp:include ... />* permet de modifier le fichier chargé lors de l'exécution. Par contre, les dépendances telles que les imports des fichiers inclus ne sont pas pris en charge. Il faudrait alors que le fichier de template importe toutes les dépendances des pages JSP qu'il inclura. Cette solution ne permet donc pas d'utiliser de template pour tous les cas. La solution pour l'implémentation d'un template est l'utilisation de la librairie JSTL comme expliqué dans le chapitre *Librairie JSTL*.

7.7.4 Expression langage

La technologie JSP nous met à disposition un outil puissant, les expressions langage. Les expressions langages, abrégé EL, sont propres au langage JSP. Elles permettent de manipuler les objets et d'effectuer des tests sur ceux-ci tout en épurant la syntaxe utilisée dans nos pages JSP. Ce dernier point est très intéressant pour la maintenance du code. Les expressions langage permettent de s'affranchir de l'utilisation de code Java dans les pages JSP. Le Java est effectivement vivement déconseillé lors de l'écriture de pages JSP. Une EL est précédée d'un symbole dollar ($) puis est placée entre accolades (*${monEL}*). Le conteneur va alors interpréter l'élément se situant entre les accolades.

Le point le plus intéressant des EL est la gestion des JavaBeans. Un JavaBean est une classe Java qui respecte certaines règles standards. Pour plus d'informations sur le JavaBean, voir le chapitre qui lui

est dédié, *JavaBean*. Il est alors possible d'utiliser les différentes méthodes mises à disposition par les classes Java. Le point que nous allons le plus utiliser est l'utilisation implicite des getters de nos JavaBeans. Comme nous pouvons le voir dans la Figure 43, nous pouvons accéder aux champs privés de nos classes Java simplement en donnant le nom de l'objet, point, le nom du champ. Dans notre exemple, la classe *userSensor* possède un champ privé nommé *sensorID*.

```
${userSensor.sensorID}
```

Figure 43 Utilisation des expressions langage dans un fichier JSP

Comme nous pouvons le voir sur la Figure 44, cette expression langage est traduite code Java sensiblement plus long et moins lisible. Première constatation, l'objet utilisé doit être passé à notre page JSP en tant qu'attribut. C'est donc grâce aux EL que nous allons récupérer nos attributs provenant des Servlets du contrôleur. De plus, la validité du JavaBean récupéré est vérifiée. Si le nom de l'attribut ne correspond à aucune valeur reçue, le code ne fait rien. Sinon, le getter du champ demandé est appelé. Encore une fois, la validité de la valeur retournée est vérifiée. Si cette valeur n'existe pas ou qu'elle est settée à null, rien ne se passe.

```
UserSensor bean = (UserSensor) pageContext.findAttribute( "userSensor" );
if ( bean != null ) {
    String sensorID = bean.getSensorID();
    if ( sensorID != null ) {
        out.print( sensorID );
    }
}
```

Figure 44 Traduction en Java d'une expression langage simple

Il est possible d'effectuer les opérations arithmétiques et logiques dans les EL. Nous pouvons donc comparer le champ d'un attribut reçu avec une autre valeur directement dans une expression langage. De plus, nous avons la possibilité d'appeler les autres méthodes de notre classe Java directement dans l'EL. La syntaxe d'un tel appel est quasiment la même que pour obtenir un champ privé. Nous devons préciser le nom de l'objet, point, le nom de la méthode souhaitée avec ses deux parenthèses. L'utilisation de méthodes dans les EL n'est supportée que depuis la version des Servlets Java 3. Actuellement, App Engine supporte la version 2.5 des Servlets Java, il n'est donc pas possible d'appeler des méthodes depuis des expressions langage.

La Figure 45 montre comment nous devons alors utiliser une méthode d'un attribut. Ce code est nettement moins clair et compréhensible que la version avec EL donnée dans le commentaire.

```
<%--
    This code was written to work with Servlet 2.5 technology.
    When Servlet 3.0 or over must be used, please remove java scriptlet and use
    instead : "${userSensor.getUsers()}"
--%>
<td>
    <c:forEach items='<%= ((UserSensor) request.getAttribute("selectedSensor")).getUsers()%>' var="users">
```

Figure 45 Utilisation des méthodes des attributs dans des fichiers JSP

En plus de supporter les classes JavaBeans, les EL permettent de parcourir les collections. Si un attribut fourni par le contrôleur est une collection, nous pouvons alors accéder à chacune de ses valeurs en utilisant des crochets. La Figure 46 illustre l'utilisation de collection dans les expressions langage. A noter que l'indice de l'élément peut être placé entre guillemets, entre apostrophes ou dans rien de spécial comme dans la Figure 46.

```
${listeUserSensor[2]}
```

Figure 46 Utilisation des collections avec les expressions langage

Afin de s'assurer que notre application pourra utiliser correctement les expressions langages, nous devons définir les attributs de la Figure 47 dans notre fichier *web.xml*.

```
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="2.5"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

Figure 47 Attribut de la balise web-app dans web.xml

7.7.5 Objets implicites

Le conteneur de Servlet nous met à disposition des objets implicites que nous pouvons utiliser avec les EL. Ces objets nous permettent d'utiliser pleinement l'environnement qui nous est mis à disposition. Ci-dessous, une liste non exhaustive des différents objets implicites qui peuvent être utilisés:

pageContext, est un objet contenant des informations sur l'environnement du serveur sur lequel notre application est exécutée.
pageScope, requestScope, sessionScope, applicationScope sont des Map qui permettent d'associer les noms aux valeurs des attributs ayant pour portées respectivement la page, la requête, la session et l'application.
param et *initParam* sont des Map qui associent les noms avec les valeurs des paramètres de la requête pour *param* et pour les paramètres initiaux donnés par le fichier *web.xml* pour *initParam*.

Il est important que les paramètres et attributs que nous utilisons dans nos pages JSP ne portent pas de noms identiques aux objets implicites. Auquel cas, le comportement de l'application serait incertaine.

Lorsque nous écrivons une EL, il est possible de ne pas spécifier la portée d'un objet comme nous l'avions fait à la Figure 43. Dans un tel cas, le conteneur parcours chacune des portées dans l'ordre *page*, *request*, *session* et *application*. Dès qu'il trouve un objet qui correspond au nom donné, il s'arrête. Afin de bien savoir ce que nous faisons, il est recommandé de toujours indiquer la portée de l'attribut utilisé comme illustré dans la Figure 48.

```
${requestScope.contentPage}
```

Figure 48 Expression langage avec objet implicite pour définir la portée

7.7.6 Librairie JSTL

La librairie JSTL, Java Standard Tag Library, nous fournit un grand nombre de balises qui nous seront utiles afin de nous dispenser totalement de code Java dans nos pages JSP. Ces balises nous permettent notamment d'effectuer des boucles, des tests conditionnels ou encore le formatage des données. La JSTL nous permet de simplifier l'écriture de notre code, ce qui augmente la lisibilité, et de diminuer le nombre de lignes à écrire.

La librairie JSTL est fournie par App Engine par défaut, aucune installation supplémentaire n'est donc nécessaire pour l'utilisation de ces balises. Toutefois, afin d'indiquer à Eclipse où les tags sont définis,

nous devons inclure la librairie dans nos pages JSP grâce à la directive JSP, *taglib*. La Figure 49 montre la ligne à ajouter dans toutes nos pages JSP utilisant la librairie JSTL. Le paramètre *prefix* permet d'indiquer l'alias que nous utiliserons dans la page JSP afin de faire appel aux balises JSTL.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Figure 49 Inclusion de la librairie JSTL dans une page JSP

Dans notre application, chaque page utilise la JSTL et a donc besoin de cette ligne d'inclusion. Afin d'éviter de devoir recopier cette ligne de code dans toutes les pages JSP, nous pouvons utiliser une propriété supplémentaire du fichier *web.xml*. La balise *jsp-property-group* permet d'ajouter du contenu au début et à la fin de fichiers correspondant à un chemin. Dans notre application, le chemin spécifié par *url-pattern* sélectionne tous les fichiers JSP. La balise *include-prelude* permet ensuite de préciser le fichier dans lequel le contenu à ajouter est présent. Selon notre configuration représentée par la Figure 50, nous avons alors eu à ajouter un fichier, *taglibs.jsp*, dans notre arborescence sous *war/WEB-INF/.* Ce fichier contient uniquement une ligne qui est la ligne d'inclusion de la Figure 49.

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <include-prelude>/WEB-INF/taglibs.jsp</include-prelude>
    </jsp-property-group>
</jsp-config>
```

Figure 50 Inclusion automatique de JSTL dans toutes les pages JSP

7.7.6.1 Balises

Ce chapitre présente quelques balises JSTL utilisées dans l'application. Les attributs sont suivis d'un symbole égal puis de la valeur souhaitée mise entre guillemets. Les variables mises à disposition par certaines de ces balises sont utilisables via les expressions langage.

Affichage

<c:out />	value	Seul attribut obligatoire pour le tag *out*. Affiche la valeur de cet attribut.
	default	Valeur à afficher si le contenu de *value* est vide.
	escapeXml [true/false]	Cette option est activée par défaut. Elle remplace les caractères de script par leur équivalant en HTML.

Il est très intéressant d'utiliser le tag *out* lorsque nous devons afficher des saisies provenant de l'utilisateur. Effectivement, le paramètre *escapeXml* est fixé à *true* par défaut ce qui permet de remplacer les caractères de script par des caractères qui seront affiché. Ceci empêche l'interprétation des saisies utilisateurs comme étant du code de script et nous protège donc contre des attaques XSS ou par injection de code.

Test conditionnel

<c:choose>	-	La balise *choose* ne peut contenir aucun attribut.
<c:when>	test	Attribut obligatoire de la balise *when*. Représente le test effectué qui retournera *true* ou *false*.
</c:when>		
<c:otherwise>	-	La balise *otherwise* ne peut contenir aucun attribut.
</c:otherwise>		
</c:choose>		

La balise *choose* suivie de plusieurs *when* et éventuellement d'un *otherwise* permet d'effectuer un test conditionnel. Les conditions sont contenues dans les attributs *test* des balises *when* qui doivent être mutuellement exclusives. Si la condition retourne *true*, le contenu situé entre les balises *when* est exécuté. Cette suite de balises JSTL permet d'effectuer les mêmes actions qu'un *if/else* ou un *switch* en Java.

Boucle classique

<c:forEach>	var	Variable de boucle non obligatoire qui indique la valeur du compteur de la boucle.
</c: forEach >		
	varStatus	Variable contenant des informations sur la boucle
	begin	Valeur initiale du compteur
	end	Valeur finale du compteur avec un test inférieur ou égal (<=)
	step	Valeur de l'incrément utilisé par le compteur

La balise *forEach* permet d'effectuer une boucle un certain nombre de fois. Ce nombre est donné par les attributs *begin*, *end* et *step*. Le fonctionnement de cette boucle est identique à celui de la boucle *for* de Java:

1. la boucle initialise une valeur de compteur à la valeur de *begin*
2. Compare la valeur courante du compteur avec la valeur de l'attribut *end*
3. effectue le code situé entre les deux balises *forEach*
4. incrémente le compteur selon la valeur de *step* puis retourne au point 2

Une différence importe est le test effectué pour détecter la fin de la boucle. La JSTL compare si la valeur du compteur est plus petite ou égale (<=) à la valeur de *end*. Il est possible d'ajouter l'attribut *var* afin d'avoir accès à la valeur du compteur dans la boucle.

L'attribut *varStatus* permet de stocker un objet *LoopTagStatus* qui nous met à disposition les propriétés suivantes:

begin	valeur de l'attribut *begin*
end	valeur de l'attribut *end*

step	valeur de l'attribut *step*
first	valeur booléenne indiquant si l'itération courante est la première de la boucle
last	valeur booléenne indiquant si l'itération courante est la dernière de la boucle
count	compteur d'itérations commençant à 1
index	compteur d'itérations commençant à 0
current	élément courant lors du parcourt d'une itération

Boucle sur une collection

<c:forEach>	item	Identifie la collection
</c: forEach >	var	Permet d'obtenir la valeur actuelle de la collection dans la boucle
	varStatus	Variable contenant des informations sur la boucle
	begin	Valeur initiale du compteur
	end	Valeur finale du compteur avec un test inférieur ou égal (<=)

La balise *forEach* permet en plus d'effectuer une boucle classique d'effectuer une boucle sur une collection. La collection est spécifiée avec l'attribut *item*. L'objet courant de la collection est utilisable dans la boucle grâce au nom donné à l'attribut *var*. Il est possible de limiter les objets sur lesquels nous effectuons la boucle grâce aux attributs *begin* et *end*. Si la collection est plus petite que ce que souhaite les attributs *begin* et *end*, la boucle s'arrêtera d'après la taille de la collection.

Redirection

<c:redirect />	url	URL sur laquelle le client sera redirigé

La balise *redirect* permet à une page JSP de rediriger le client vers une autre URL. Un message de redirection HTTP est envoyé au client qui doit alors effectuer une nouvelle requête. L'URL affichée dans le navigateur de l'utilisateur sera alors modifiée.

JSP nous met à disposition une balise, *forward*, qui effectue un forwarding du client sur une autre page. La grande différence par rapport à la balise *redirect* de JSTL est que le forwarding est effectué côté serveur. Il n'y a alors pas de message de redirection envoyé au client. Ce dernier n'est même pas informé que la page qu'il consulte n'est pas directement celle qui correspond à l'URL affichée dans son navigateur. Par contre cette méthode limite l'accès aux pages existantes sur le serveur alors qu'avec une redirection, il est possible de fournir une URL d'un autre serveur.

Import

<c:import />	url	URL du fichier à importer

La balise *import* permet à une page JSP d'effectuer une importation d'un autre fichier. Cette balise est utile à notre application lors de l'utilisation d'un template. Pour rappel, les balises fournies par JSP ne sont pas suffisantes pour importer un fichier correctement.

7.7.7 Utilisation d'un template

Le fichier HTML final qui sera transmis à l'utilisateur sera constitué d'un header, d'un menu, d'une zone de contenu et d'un footer. Cette construction est très standard dans les pages web actuelles. Certaines de ces parties seront toujours identiques quel que soit la page sur laquelle l'utilisateur navigue. Il est alors intéressant de découper notre page web en plusieurs blocs. Chacun de ces fragments sera alors contenus dans un fichier JSP indépendant. Il devient ainsi possible de constituer une page contenant toujours le même header, menu et footer et de changer uniquement le contenu. En faisant ainsi, il suffit de modifier un seul fichier JSP, le header par exemple, afin que toutes les pages vues par l'utilisateur soient mises à jour.

Il est alors nécessaire de créer une nouvelle page JSP qui a le rôle d'assembler tous les fragments afin de constituer la page finale qui sera envoyée à l'utilisateur. Cette page est créée grâce au fichier *war/WEB-INF/DefaultPages/template.jsp* de l'arborescence du projet. Comme le montre la Figure 51, l'importation des différents fichiers constituant la page finale est faite grâce à l'utilisation de la balise *import* du JSTL. Si le fichier voulu n'est pas présent dans les arguments passés par la Servlet du contrôleur, des fichiers par défaut sont importés.

Pour l'utilisation normale de l'application, seul un fichier de contenu doit être passé par les arguments. Les logBar header, menu, timeBar et footer par défaut conviennent dans tous les cas. Cependant, le fichier de template est créé ainsi afin de favoriser l'évolution future de l'application.

Un autre point intéressant est la mise en place d'une page d'erreur par défaut en cas d'exception d'exécution. Cette page, *unexpectedError.jsp* contient donc un message d'erreur qui sera affiché lorsqu'une exception survient lors de l'exécution de l'application.

```
<!DOCTYPE html>
<%@ page errorPage="/WEB-INF/DefaultPages/unexpectedError.jsp" %>
<%@ page pageEncoding="UTF-8" %>

<html>
    <head>
        <title>iCelcius Wireless</title>
        <link rel="stylesheet" href="/stylesheets/styles.css" type="text/css" />
        <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <div id="logBar">
            <c:import url="logBar.jsp"/>
        </div>
        <div id="header">
            <c:choose>
                <c:when test="${not empty headerPage}">
                    <c:import url="${headerPage}"/>
                </c:when>
                <c:otherwise>
                    <c:import url="header.jsp"/>
                </c:otherwise>
            </c:choose>
        </div>
        <div id="menu">
            <c:choose>
                <c:when test="${not empty menuPage}">
                    <c:import url="${menuPage}"/>
                </c:when>
                <c:otherwise>
                    <c:import url="menu.jsp"/>
                </c:otherwise>
            </c:choose>
        </div>
        <div id="timeBar">
            <c:if test="${not empty lastUpdate}">
                <c:import url="timeBar.jsp"/>
            </c:if>
        </div>
        <div id="content">
            <c:choose>
                <c:when test="${not empty contentPage}">
                    <c:import url="${contentPage}"/>
                </c:when>
                <c:otherwise>
                    <c:import url="noContent.jsp"/>
                </c:otherwise>
            </c:choose>
        </div>
        <div id="footer">
            <c:choose>
                <c:when test="${not empty footerPage}">
                    <c:import url="${footerPage}"/>
                </c:when>
                <c:otherwise>
                    <c:import url="footer.jsp"/>
                </c:otherwise>
            </c:choose>
        </div>
    </body>
</html>
```

Figure 51 Fichier JSP de template

7.7.8 Formatage de l'affichage

Lorsque le client effectue une requête qui est redirigée vers une page JSP, la réponse qui lui est fournie est un fichier HTML créé par le serveur. Le langage CSS permet de formater les fichiers HTML. Notre application contient alors un fichier CSS qui s'occupe du formatage de tout l'affichage qui sera vu par l'utilisateur.

Ce fichier configure la largeur de l'affichage en pleine largeur. C'est-à-dire que la page prendra toute la largeur de l'écran. Ceci peut fournir un affichage peu adapté pour l'utilisation de notre application sur un poste de travail avec un écran de haute résolution. Cependant, l'utilisation de l'application sur un appareil mobile possédant un plus petit écran est améliorée.

7.8 Modèle: Classes Java

Des classes Java implémenteront la partie modèle de l'application. C'est grâce à elles que sera faite la modélisation du système et le stockage des données dans le datastore. Toutes les classes Java du modèle se trouvent dans le répertoire */src* du projet. Il est intéressant de rappeler que ces classes seront ensuite compilées est placées dans le répertoire */war* de l'application afin d'être transférées sur les serveurs de Google.

7.8.1 JavaBean

Comme présenté dans le chapitre sur les pages JSP, *7.7 Vue: Pages JSP*, les expressions langage nécessitent l'utilisation de JavaBean. Un JavaBean, souvent appelé simplement Bean, est une simple classe Java qui respecte certaines contraintes. La construction d'un Bean doit être simple, ce qui permet à l'application de réduire les duplications de codes. Nos classes implémentant le modèle seront alors des classes répondant aux contraintes pour être un JavaBean.

Les caractéristiques imposées à la création d'un Bean ne sont la plupart du temps pas vraiment contraignantes. La classe doit posséder un constructeur public sans paramètre, ne doit pas avoir de champs publics et ses champs privés doivent tous être accessibles via des getter et setter. Au vu de ces caractéristiques, presque toutes les classes Java standard sont des Beans.

7.8.2 AppUser

AppUser est une classe Java implémentant le modèle pour les utilisateurs de notre application. Cette classe permet de créer des utilisateurs qui possèderont certains paramètres qui leur sont propres. De plus, cette classe permet d'augmenter l'évolutivité du code au cas où nous voudrions changer le système d'authentification des utilisateurs. Cette classe est contenue dans un package, *appUser*, qui contient un certain nombre de classes permettant la gestion des utilisateurs. La classe AppUser contient les différents attributs utilisés pour représenter un utilisateur dans notre application. Ces attributs sont présentés par la Figure 52.

```
/** Primary key of this datastore. This value identify this user only */
@Id
private String userID;

/** Email address of this user */
private String email;

/** Nickname of this user */
private String nickname;

/** Date pattern to print date. Set by the user */
private String datePattern;

/**Unit temperature setted by the user */
private String temperatureUnit;
```

Figure 52 Attributs de la classe AppUser

Le *UserID* est une valeur unique attribuée à un utilisateur qui permet de le reconnaitre de manière univoque dans toute l'application. Cette valeur sera donc essentielle pour l'application car elle sera

utilisée dès qu'un utilisateur devra être reconnu. Ce champ est automatiquement attribué lors de la création de l'AppUser. La valeur attribuée est le Google ID de l'utilisateur. Cette valeur est utilisée par Google afin de reconnaitre de manière univoque ses utilisateurs. L'utilisateur n'y a donc pas accès et ne peut pas le modifier. Il n'est pas possible de reconnaitre un utilisateur via son nickname ou son adresse email. Le nickname peut être changé à tout moment par l'utilisateur et deux personnes peuvent choisir un nom identique. Idem pour l'adresse email qui peut être modifiée par l'utilisateur.

L'email est une valeur saisie par l'utilisateur. Elle représente l'email par lequel il souhaite être contacté si un contact est nécessaire. La valeur de ce champ est attribuée par l'utilisateur lui-même. Lors de la création du compte, la valeur proposée automatiquement est l'adresse email associée au compte Google de l'utilisateur. L'utilisateur peut modifier cette valeur lors de la création du compte ou par la suite via le menu *MyAccount*. Cette valeur n'est alors pas sûre. Le seul contrôle effectué est que la valeur du champ corresponde au format d'une adresse email. Aucune vérification n'est faite quant à la validité de l'email.

Le *nickname* est le nom de l'utilisateur qui sera affiché dans l'application. Ce nom est choisi par l'utilisateur au moment de la création du compte. Le nickname du compte Google de l'utilisateur est automatiquement proposé lors de la création du compte. L'utilisateur peut modifier ce nom lors de la création du compte ou via le menu *MyAccount*. Ce nom est visible pour les autres utilisateurs qui sont associés à un même sensor.

Le champ *datePattern* représente le format de l'affichage des dates. Ce format est choisi parmi une liste déroulante lors de la création du compte. Il est aussi possible de modifier ce format via le menu *MyAccount*. Tous les formats disponibles sont présents dans le fichier *Config.java* du package *utilities*. La Figure 53 représente les formats actuellement supportés. Les symboles sont les suivants:

- MM Le mois sur deux chiffres
- dd Le jour sur deux chiffres
- yyyy L'année sur quatre chiffres
- HH L'heure au format 24h
- KK L'heur au format 12h
- mm Les minutes sur deux chiffres
- ss Les secondes sur deux chiffres
- a L'indication AM/PM pour le format 12h

```
/**
 * Date pattern
 */
public static final String US_PATTERN          = "MM/dd/yyyy KK:mm:ss a";
public static final String EU_PATTERN          = "dd/MM/yyyy HH:mm:ss";
```

Figure 53 Format des dates

Le champ *temperatureUnit* permet à l'utilisateur de définir dans quelle unité doivent être affichées les informations de température. Cette valeur est définie lors de la création du compte et peut être modifiée par l'onglet *MyAccount*. Toutes les unités de température supportées sont présentes dans le fichier *Config.java* du package *utilities*. Actuellement les températures en degrés Celsius et en degrés Fahrenheit sont supportées. La Figure 54 présente les unités de température déclarées dans le fichier *Config.java*.

```
/**
 * Temperature unit
 */
public static final String DEGREE       = "Degree";
public static final String FAHRENHEIT   = "Fahrenheit";
```

Figure 54 Unités de température

Les sensors fournissent une température en degrés Celsius. Il est donc nécessaire de calculer la valeur de la température dans la bonne unité lorsque l'utilisateur choisit une autre unité que le degré Celsius. La Figure 55 illustre la ligne de code Java nécessaire pour convertir une valeur de température du degré Celsius vers degré Fahrenheit. Il faut donc multiplier les Celsius par 1.8 et y ajouter 32. La fonction *round* présente dans le package *utilities* permet ensuite d'arrondir la valeur calculée à une décimale.

```
sd.setValue(utilities.RequestUtilities.round(sd.getValue()*1.8f+32f));
```

Figure 55 Conversion degré Celsius vers Fahrenheit

7.8.3 Sensor

Sensor est une classe Java implémentant le modèle pour les sensors de l'application. Cette classe est contenue dans un package, *sensor*, qui contient un certain nombre de classes permettant la gestion des sensors. La classe Sensor contient les différents attributs utilisés pour représenter un sensor dans notre application. Ces attributs sont présentés par la Figure 56. Chaque sensor physique possède un numéro de sensor et une clé qui lui est propre.

```
/** Unique key of the sensor */
@Id
private String sensorID;

/** Hash value to identify the sensor */
private String hash;

/** Version of the sensor */
private String version;

/** Date when the sensor was producted, UNIX time in second */
private long productDate;
```

Figure 56 Attributs de la classe Sensor

Le *SensorID* est une valeur unique attribuée à un sensor qui permet de le reconnaitre de manière univoque dans toute l'application. Cette valeur sera essentielle pour l'application car elle sera utilisée dès qu'un sensor devra être reconnu. Cette valeur est existante sur le sensor physique lui-même. L'utilisateur a besoin de cette valeur afin de s'associer au sensor. Le champ *SensorID* doit être attribué lors de la création du sensor dans le datastore. Cette création n'est possible que pour les utilisateurs définis en tant qu'administrateur de l'application. Ceci est important car une mauvaise saisie de ce champ pourrait nuire au fonctionnement de l'application. Effectivement, si un sensor est enregistré avec le même *sensorID* qu'un autre sensor, ce dernier est tout simplement écrasé. Si le *hash* est différent, le sensor écrasé ne pourra alors plus communiquer avec l'application.

Le champ *hash* est une valeur permettant de s'assurer de l'identité du sensor. Cette valeur est utilisée lorsque le sensor souhaite communiquer avec l'application ou lorsqu'un utilisateur souhaite s'associer avec le sensor. Chaque sensor possède une clé qui lui est propre. La clé est définie dans l'application lors de la création du sensor. Cette clé doit être donnée en claire. Elle sera ensuite hachée avec l'algorithme MD5 avant d'être stockée dans le champ *hash*. Afin de s'associer avec le

sensor, un utilisateur doit entrer le *sensorID* et le *hash*. Ces deux valeurs doivent être existantes sur le sensor physique. Le hash est fourni en clair et sera ensuite haché avec l'algorithme MD5 avant d'être comparé à la valeur de *hash* du sensor. Si les deux hash correspondent, l'utilisateur sera alors associé au sensor. Un utilisateur malveillant pourrait récupérer le *sensorID* et le *hash* en claire du sensor lors d'une communication wifi. Il lui est alors possible d'envoyer des données en se faisant passer pour le sensor. Cependant, avec ce système de hash, un utilisateur pourrait mettre la main sur le datastore des sensors sans pour autant pouvoir se faire passer pour un sensor. Effectivement, le datastore regroupant les sensors ne stocke que les hash. Or il est impossible de retrouver la clé originale d'une clé hachée avec MD5. L'utilisateur malveillant ne pourra alors pas tirer profit du fichier du datastore.

L'attribut *version* indique quelle est la version actuelle du sensor. Cette valeur sera utile lorsque plusieurs versions de software ou de hardware seront exploitées.

Le champ *productDate* indique la date de production du sensor. Cette valeur sera utile lorsque plusieurs versions de hardware seront existantes. Cette valeur est enregistrée au format UNIX. Il s'agit donc du nombre de secondes écoulées depuis le 1er janvier 1970.

7.8.4 SensorData

SensorData est une classe Java implémentant le modèle pour les données des sensors de notre application. Cette classe est contenue dans un package, *sensorData*, qui contient un certain nombre de classes permettant la gestion des données des sensors. La classe *SensorData* contient les différents attributs utilisés pour représenter une donnée dans notre application. Ces attributs sont présentés par la Figure 57.

```
/** Unique key of the sensor */
@Id
private Long key;

/** Sensor ID to identify the sensor */
private String sensorID;

/** Date of the given data, UNIX format */
private long date;

/** Type of the sensor */
private String type;

/** value of the data */
private float value;
```

Figure 57 Attributs de la classe SensorData

L'attribut *Key* est une valeur unique attribuée à une donnée qui permet de la reconnaitre de manière univoque dans toute l'application. Dans une base de données traditionnelle, la clé de chaque objet *SensorData* serait composée de la clé étrangère *sensorID* et de l'attribut *date*. Dans le datastore, le plus simple et efficace est de créer une nouvelle valeur pour définir la clé. Cette valeur est de type Long et est générée automatiquement par Objectify lors de la création de la donnée.

Le champ *sensorID* permet d'associer la donnée courante à un sensor. La valeur de ce champ est alors un *sensorID* de la classe Sensor. Cette valeur représente un seul et unique sensor, qui est celui d'où provient la donnée.

L'attribut *date* nous informe sur la l'heure et la date de la prise de mesure de la donnée. Cette valeur est fournie en format UNIX. Il s'agit donc du nombre de secondes écoulées depuis le 1er janvier 1970. Cette valeur provient du sensor lui-même. Elle est transmise en même temps que les autres informations.

Le champ *type* nous permet de définir le type de donnée. Plusieurs probes de différents types peuvent être connectées sur un même sensor. Il est alors nécessaire de définir quel est le type de la donnée reçue lors de chaque réception de donnée. Tous les types de données supportés sont présents dans le fichier *Config.java* du package *utilities*. Actuellement les données de type température, CO2 et humidité sont supportées. Un dernier type de donnée, nommé *no data*, est le type qui sera affiché lorsque le sensor n'aura aucune donnée associée. La Figure 58 présente les types de données déclarés dans le fichier *Config.java*.

```
/**
 * Type of probe
 */
public static final String TEMPERATURE  = "temperature";
public static final String CO2          = "CO2";
public static final String HUMIDITY     = "humidity";
public static final String NO_DATA      = "no data";
```

Figure 58 Type de donnée

L'attribut *value* représente la valeur de la donnée courante. Cette valeur est sauvegardée en float afin d'avoir le plus de flexibilité possible selon le type de donnée qui est fourni.

7.8.5 UserSensor

UserSensor est une classe Java permettant de créer une association entre un sensor et les différents utilisateurs qui y sont associés. Chacun de ces utilisateurs peut donner un nom au sensor lors de son association. Cette classe est contenue dans un package, *userSensor*, qui contient un certain nombre de classes permettant la gestion des objets de la classe *UserSensor*. Dans l'application, il y aura un objet *UserSensor* par *Sensor* ayant des utilisateurs associés. La classe *UserSensor* contient les différents attributs utilisés pour associer un sensor et les utilisateurs dans notre application. Ces attributs sont présentés par la Figure 59.

```
/** Primary key of this datastore. This is the same sensor ID in the Sensor datastore */
@Id
private String sensorID;

/** List of unique ID of users
 *
 *  Warning: List of userID and list of Note are connected. User id with index number X get the
 *  corresponding note from note list with index number X
 *
 * */
public List<String> userID;

/** Personal note for user, this note identifies the sensor in the application */
private List<String> note;

/** Indicates whether this sensor is a premium sensor */
private boolean isPremium = false;

/** RSSI of the sensor */
private int RSSI;

/** Battery of the sensor, in mV */
private float battery;

/** Code version on the sensor */
private String codeVersion;
```

Figure 59 Attributs de la classe UserSensor

L'attribut *SensorID* est l'identifiant du sensor pour lequel l'objet *UserSensor* a été créé. Cette identifiant représente un objet *Sensor* de manière univoque. Chaque sensor ayant des utilisateurs associés aura un seul objet de type *UserSensor* le représentant.

La liste *userID* est une liste de String représentant les identifiants de tous les utilisateurs qui sont associés au sensor correspondant au *sensorID*. Cette liste peut contenir un nombre quelconque d'utilisateur.

La liste *note* est une liste de String représentant les noms donnés au sensor par les différents utilisateurs. L'utilisateur doit effectivement donner un nom à ses sensors afin de pouvoir les reconnaitre dans l'application. Grâce à cette liste, chaque utilisateur peut donner un nom différent au même sensor.

La liste de *note* et de *userID* sont logiquement dépendantes. C'est-à-dire que l'utilisateur présent dans la liste *userID* à l'indice 4 se verra correspondre le nom du sensor présent à l'indice 4 de la liste *note*. Il est donc important d'assurer cette cohérence lors de l'ajout de la suppression ou de la modification d'une de ces valeurs. Cette cohérence est assurée par les méthodes de getter et de setter mises à disposition. Ainsi lorsqu'un utilisateur appel simplement *monUserSensor.getNote(),* le code devra définir quelle est la bonne valeur correspondante à l'utilisateur faisant cet appel. Cette implémentation est alors transparente pour l'utilisation de la classe via les getter et setter. Cette manière de faire n'est pas optimale. Il aurait été préférable de créer une classe interne comprenant un user ID et un nom de sensor afin de stocker une liste d'objet de cette classe. Cependant cette solution pose des problèmes de stockage lors de l'utilisation d'Objectify.

La valeur booléenne *isPremium* indique si le sensor associé aux utilisateurs est défini comme un sensor Premium. Les sensors Premium possèderont certains avantages qui restent encore à définir. Cependant ces avantages seront payants pour les utilisateurs.

L'attribut *RSSI* nous donne une indication sur l'intensité du signal wifi reçu. Cette valeur provient du sensor. Elle est donc mise à jour dans un objet *UserSensor* lorsque le sensor envoie une de ces valeurs.

La valeur du champ *Battery* nous informe sur le dernier état de batterie connu du sensor. Cet état est fourni en même temps que les valeurs des mesures.

L'attribut *codeVersion* est le dernier état connu du *codeVersion* du sensor. Cette valeur est mise à jour lorsque le sensor envoi une nouvelle information.

7.9 Datastore

Les différentes classes présentées dans les chapitres précédant doivent être stockées afin que l'application ait accès aux valeurs même après l'extinction du système. Ce stockage est effectué grâce au Datastore de Google et à Objectify qui est présenté dans le chapitre *5.7 Objectify*. Pour rappel, Objectify nous fournit une API pour utiliser le datastore de Google. Ce datastore n'agit pas comme une base de données relationnelle mais comme un HashMap. La Figure 60 représente les différentes relations existantes entre les modèles de l'application.

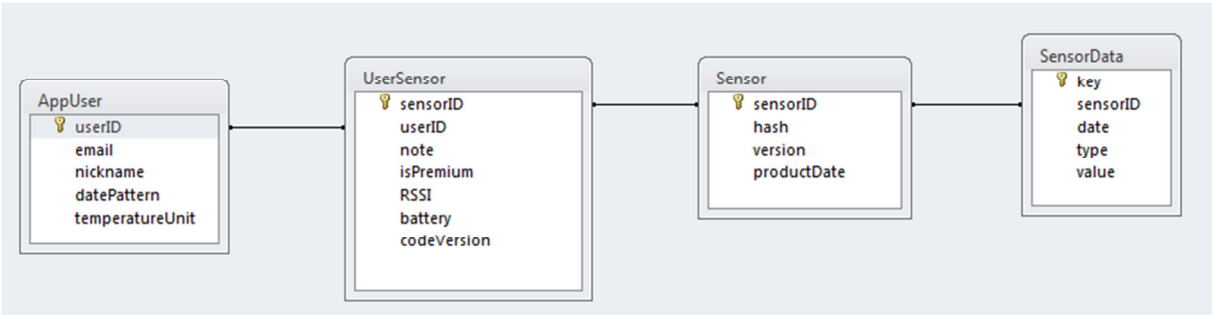


Figure 60 Schéma des entités modèles

Comme mentionné à plusieurs reprises dans ce document, Objectify possède certaines limites qui sont très contraignantes pour l'implémentation de l'application. Cependant, un grand nombre de facilités sont mises en place grâce à Objectify. Les deux paragraphes suivants introduisent la gestion de la mémoire cache ainsi que la création automatique des clés.

Grâce à l'annotation *@cached* placée avant la déclaration de l'entité, Objectify utilise le MemCache de Google afin de minimiser le nombre de lecture sur le datastore. La gestion manuelle du MemCache serait plus fastidieuse. A noter que le MemCache est aussi un service payant de Google, il ne faut alors pas en abuser.

```
@Cached
public class UserSensor {
```

Figure 61 Gestion de la mémoire cache avec Objectify

Objectify nous permet aussi de générer des clés automatiquement. Pour les entités dont la clé ne représente pas une valeur que nous utilisons, nous pouvons alors laisser le choix de la clé au système. C'est grâce à ce mécanisme que les clés des objets *SensorData* sont créées. Pour utiliser cette option, nous devons déclarer notre clé, identifiée avec l'annotation *@Id*, de type String, long ou Long. Si la valeur de la clé n'est pas initialisée dans le constructeur, elle sera attribuée automatiquement.

7.9.1 Modèle DAO

Le modèle DAO est un design pattern permettant d'isoler les objets métiers du système de stockage. Ceci permet de rendre le code Java indépendant du système de stockage sur lequel les données seront sauvegardées. Il est alors possible de changer le système de stockage de manière plus aisée.

Figure 62 Représentation du modèle DAO

Chaque objet du modèle sera alors mis en relation avec une classe Java implémentant le design pattern DAO. Les objets métiers appelleront les fonctions contenues dans cette classe. C'est ces fonctions qui devront travailler directement sur le système de stockage. Dans l'application créée, le système de stockage est le datastore proposé par Google. Chacune de nos entités aura alors une classe Java associée qui lui servira de DAO.

7.9.2 Index

Un index est une structure de données utilisée par un système de gestion de base de données (SGBD). L'index est utilisé afin de rendre plus rapide la recherche et le tri de base de données. Cependant, la mise à jour de l'entité et de son index sera plus lente et plus coûteuse.

Une opération de *put()* dans le datastore coûte 48ms CPU. Si on indexe la valeur, cela prendra 17ms supplémentaires. Cette opération ne change rien du point de vue de l'utilisateur car l'indexation est effectuée en parallèle. Par contre, le temps CPU est facturé. De plus, l'indexation prend beaucoup de place de stockage, qui est aussi facturé.

Il est donc nécessaire de n'indexer que les attributs sur lesquels nous effectuons des recherches. Pour rappel, les fichiers d'index sont créés automatiquement par le serveur de développement. Cependant, pour avoir plus de contrôle sur les index, nous pouvons ajouter l'annotation *@unindexed* aux champs que nous ne souhaitons pas indexer.

7.9.3 Nombre d'accès au datastore

Lors de l'utilisation de l'application, la ressource la plus utilisée sera le datastore. Effectivement, le datastore sera sollicité lorsque chaque sensor envoie une nouvelle information. De plus, lorsque les utilisateurs se connectent depuis leur smartphone ou depuis l'application web, des lectures seront effectuées sur le datastore.

Durant le mois de juin 2013, des tests effectués avec un sensor ont révélé un problème concernant le datastore. Le sensor envoyait des données toutes les 20 secondes. Le pourcentage de write sur le datastore était alors assez grand, mais restait dans la limite des quotas gratuits. Le nombre de write pour ajouter une nouvelle valeur est calculé à partir de la Figure 63. Il y a 16 opérations d'écriture pour chaque nouvelle valeur provenant d'un sensor. Lorsque plusieurs sensors seront mis en service, le nombre d'écriture sur le datastore sera alors important. Il est donc nécessaire de limiter la fréquence de mesure des valeurs afin de limiter les coûts engendrés par le datastore. Le nombre

d'écriture et de lecture sur le datastore peut facilement être visualisé grâce à l'application AppStat. Cette application a été ajoutée dans la page d'administration de l'application.

API Call	Datastore Operations
Entity Get (per entity)	1 read
New Entity Put (per entity, regardless of entity size)	2 writes + 2 writes per indexed property value + 1 write per composite index value
Existing Entity Put (per entity)	1 write + 4 writes per modified indexed property value + 2 writes per modified composite index value
Entity Delete (per entity)	2 writes + 2 writes per indexed property value + 1 write per composite index value

Figure 63 Coûts des opérations sur le datastore

Lors de ces tests, le grand problème venait de la page *My sensors* de l'application. Celle-ci devait obtenir toutes les données relatives à un sensor afin de créer la table de valeur et le graphique. De plus, le refresh automatique n'était pas optimal et rechargeait la page complète. Ceci avait pour effet d'effectuer à nouveau une requête sur toutes les valeurs du sensor. Le nombre d'opérations de lectures sur le datastore était alors très grand et fût rapidement hors des quotas gratuits. La première solution a été de modifier le mode de rafraîchissement de la page. Un script JavaScript associé à une requête JQuery a été mis en place afin d'effectuer des requêtes uniquement sur les nouvelles valeurs à afficher. Ceci diminue grandement le nombre de lectures sur le datastore lorsque la page est mise à jour. Cependant, lors du premier chargement de la page, toutes les données doivent toujours être lues depuis le datastore. Pour l'instant aucune solution définitive n'a été trouvée. Le programme limite actuellement le nombre de valeurs chargées à 100.

Pour diminuer ce problème, il serait possible que les utilisateurs ne possédant pas de compte *premium* n'aient pas accès au datatore. Les données provenant de leurs sensors seraient alors uniquement sauvegardées en mémoire cache. Comme indiqué dans le chapitre *5.6.3 Memcache*, les données sauvegardées dans la mémoire cache peuvent être supprimées à tout moment. Un utilisateur souhaitant consulter la dernière valeur de son sensor pourra alors être surpris en voyant que cette valeur n'existe plus. Le service qui serait offert aux utilisateurs qui n'ont pas de compte premium serait alors du best effort. Aucune garantie ne peut être donnée quant à la présence des données de leur sensor sur le serveur. L'utilisation de la mémoire cache sur Google App Engine est facilitée car l'API JCache est supportée.

Il est difficile de prédire les prix des différentes solutions. Appengine nous met à disposition des tables contenant tous les prix de chaque ressource[5]. Cependant, pour estimer le coût réel de l'utilisation des quotas, le meilleur moyen est d'effectuer des tests. Actuellement, seul des tests avec l'utilisation du datastore ont été réalisés. La modification du code de l'application pour l'utilisation de la memCache peut être rapidement effectuée. Il faudrait alors exécuter des tests en utilisant la memCahe pour définir si la variation de coût vaut vraiment la différence de qualité pour l'utilisateur.

[5] https://developers.google.com/appengine/docs/billing?hl=fr#cost_resource

7.10 Présentation de l'application

7.10.1 Structure du projet

Le code Java créé est séparé en plusieurs packages situés dans le dossier *src* de notre arborescence. Les paquetages *appUser, sensor, sensorData* et *userSensor* sont associés à des modèles. Chacun de ces paquetages contient des fichiers faisant partie du modèle et d'autres faisant partie du contrôleur. Concernant le modèle, les paquetages comprennent tous une classe Java contenant le modèle, une classe pour la gestion du datastore avec DAO et une classe traitant les données entrées par l'utilisateur. Cette dernière classe comporte un nom se composant de la manière suivante : *<nomModèle_management.java>.*

Les fichiers correspondant au contrôleur sont simplement suivis de *_controller* afin de les identifier rapidement. Il s'agit des Servlets utilisées pour gérer des actions effectuées sur les objets correspondant au modèle de chaque paquetage. Les méthodes *doGet* de ces Servlets ne font alors que de renvoyer l'utilisateur sur un formulaire lui permettant de saisir les valeurs nécessaires à l'action sur l'objet. Les méthodes *doPost* de ces Servlets permettent de contrôler la réception des données entrées.

Les paquetages *appUser* et *sensorData* contiennent des fichiers dont le nom se termine par *_filter*. Ces fichiers sont les filtres utilisés par le descripteur de déploiement afin de s'assurer de certains critères permettant à l'utilisateur d'accéder à des groupes de pages.

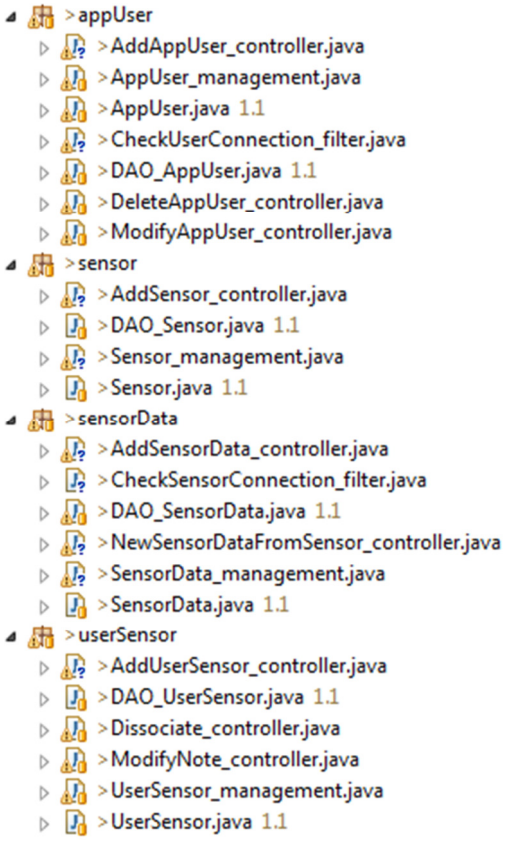


Figure 64 Paquetages Java associés à des modèles

7.10.1.1 Package appUser

Le package *appUser* contient toutes les classes gérant les utilisateurs de l'application. Le modèle présenté dans le chapitre *7.8.2 AppUser* est présent dans le fichier *AppUser.java*. Ce fichier contient aussi des getter et setter pour les champs privés. Seul le champ *userID* ne possède pas de setter. Effectivement, ce champ est la clé de l'entité dans le datastore. Il n'est alors pas possible de le modifier. La Figure 65 représente la Javadoc créée pour les méthodes de la classe *AppUser*.

Modifier and Type	Method and Description
java.lang.String	**getDatePattern**() Getter for the date pattern chosen by the user.
java.lang.String	**getEmail**() Getter of the email of the current user
java.lang.String	**getNickname**() Getter of the nickname of the user
java.lang.String	**getTemperatureUnit**() Getter for temperature unit of all temperature data.
java.lang.String	**getUserID**() Getter of User ID
void	**setDatePattern**(java.lang.String datePattern) Setter of the date pattern.
void	**setEmail**(java.lang.String email) Setter for email value.
void	**setNickname**(java.lang.String nickname) Setter for nickname value.
void	**setTemperatureUnit**(java.lang.String temperatureUnit) Setter for the attribute temperatureUnit.

Figure 65 Méthodes de la classe AppUser

Le fichier *DAO_AppUser.java* est la classe s'occupant de la gestion du stockage des objets *AppUser* dans le datastore. Cette classe met des méthodes à disposition afin de gérer le stockage d'*AppUser* sans avoir connaissance de la plateforme de stockage. Sur la Figure 66 provenant de la Javadoc de la classe *DAO_AppUser*, on voit que toutes les méthodes de recherche prennent un *userID* en paramètre.

Modifier and Type	Method and Description
void	**delete**(java.lang.String userID) Delete entry of AppUser matches with the user ID given in parameter If no entry matches with the userID, nothing append.
boolean	**exists**(java.lang.String userID) Test if the given userID is existing in the datastore.
AppUser	**get**(java.lang.String userID) Get the entry matches with userID given in parameter.
void	**put**(AppUser user) Put the new AppUser given in parameter in the datastore.

Figure 66 Méthodes de la classe DAO_AppUser

Le fichier *CheckUserConnection_filter.java* est le filtre utilisé pour toutes les pages ayant pour chemin */user/**. Ce filtre permet de s'assurer que les pages ne seront consultées que par des utilisateurs connectés et existant dans l'application. Il est mis en place dans le fichier *web.xml*. Si un utilisateur est non connecté ou qu'il ne s'est pas encore enregistré dans l'application, il est redirigé vers la page d'accueil par défaut. Si l'utilisateur est bien connecté et reconnu par l'application, il a accès à la page qu'il souhaitait. Les méthodes *destroy*, *getFilterConfig* et *init* sont des méthodes héritées de la classe *Filter*. Seule la méthode *doFilter* est redéfinie dans la classe *CheckUserConnection_filter*.

Modifier and Type	Method and Description
void	**destroy**()
void	**doFilter**(javax.servlet.ServletRequest request, javax.servlet.ServletResponse response, javax.servlet.FilterChain filterChain) This method filters user.
javax.servlet.FilterConfig	**getFilterConfig**()
void	**init**(javax.servlet.FilterConfig filterConfig)

Figure 67 Méthodes de la classe ChechUserConnection_filter

Avant de donner l'accès à la page à un utilisateur connecté, le filtre ajouter l'objet *AppUser* correspondant à l'utilisateur dans les attributs de la requête. Ceci permet de minimiser le nombre d'accès au datastore. Effectivement, chaque page consultée aura alors la possibilité de récupérer l'*AppUser* directement dans les attributs de la requête à la place d'effectuer une requête sur le datastore. La Figure 68 illustre la mise de l'objet *AppUser* en attribut dans le filtre alors que la Figure 69 illustre la récupération de cet objet dans un contrôleur.

```
// Put user attributes in the request
// Increase CPU time but decease number of database request
request.setAttribute("appUser", appUser);
```

Figure 68 Ajout de l'AppUser dans les attributs de la requête

```
// Get the current user
AppUser appUser = (AppUser) request.getAttribute("appUser");
```

Figure 69 Récupération de l'AppUser dans les attributs de la requête

Le fichier *AppUser_management.java* permet de contrôler les objets *AppUser*. Cette classe met des méthodes à disposition afin de créer, modifier ou supprimer les objets de type *AppUser*. L'attribut *errors*, accessible via le getter *getErrors*, est une Map faisant correspondre le nom du champ de la vue avec un message d'erreur. L'attribut *result*, accessible via le getter *getResult*, permet d'obtenir l'état final de l'opération demandée.

La méthode *tryCreateAppUser* récupère les paramètres nécessaires à la création d'un *AppUser* dans la requête et essaie de créer un nouvel objet *AppUser*. Ces paramètres proviennent de la vue correspondante à la création d'un nouvel *AppUser*, *newAppUser.jsp*. Si les paramètres ne sont pas présents ou ne sont pas corrects, des messages d'erreur sont fournis par les attributs *result* et *errors*. Si tous les paramètres sont corrects, un nouvel objet *AppUser* est créé. Cet objet est ensuite passé au contrôleur pour son traitement.

La méthode *modifyInformation* permet de modifier les informations concernant un *AppUser*. Les nouvelles informations à enregistrer sont contenues dans la requête passée en paramètre. Ces informations proviennent de la vue *modifyAppUser.jsp*. Si tous les paramètres sont présents et corrects, l'objet *AppUser* est mis à jour dans le datastore.

La méthode *deleteAppUser* a pour but de supprimer l'objet *AppUser* de l'application. Ceci est effectué lorsqu'un utilisateur souhaite fermer son compte. La vue correspondante à cette action, *deleteAppUser.jsp*, met un paramètre nommé *deleteConfirmation* dans la requête. Ce paramètre permet de s'assurer que la requête de l'utilisateur provient bien de la page de suppression. La valeur de ce paramètre doit être "*delete*" pour que la suppression ait bien lieu. Lors de la suppression de l'*AppUser*, ce dernier est dissocié de tous les sensors avec lesquels il s'était associé. Si un sensor n'a

plus d'utilisateur associé, l'objet *UserSensor* ainsi que toutes les données de ce sensor sont supprimées.

Modifier and Type	Method and Description
void	**deleteAppUser**(javax.servlet.http.HttpServletRequest request) This method delete the current appUser account.
java.util.Map<java.lang.String,java.lang.String>	**getErrors**() Getter of the error map.
java.lang.String	**getResult**() Getter of result value.
void	**modifyInformation**(javax.servlet.http.HttpServletRequest request) This method modify information of the current user's account.
AppUser	**tryCreateAppUser**(javax.servlet.http.HttpServletRequest request) Try to create a new AppUser from data in the request parameter If request is missing or data in request are not correct, this function return null.

Figure 70 Méthodes de la classe AppUser_management

Le fichier *AddAppUser_controller.java* est la Servlet contrôlant l'ajout d'un utilisateur dans le datastore. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à l'ajout d'un utilisateur, *newAppUser.jsp*. Cette vue est un formulaire que doit remplir l'utilisateur afin de compléter les informations concernant le nouveau compte à créer. Lorsque l'utilisateur envoie le formulaire via le bouton adéquat, la requête arrive dans la méthode *doPost* de la classe *AddAppUser_controller*. Cette méthode essaie de créer un nouvel *AppUser* via la classe *AppUser_management*. Si le nouvel objet a pu être créé, il est sauvegardé dans le datastore et l'utilisateur est redirigé vers le contrôleur de la page principale des utilisateurs. Sinon, l'utilisateur est redirigé vers le même formulaire avec des messages d'erreurs.

Le fichier *DeleteAppUser_controller.java* est la Servlet contrôlant la suppression d'un utilisateur dans le datastore. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à la suppression d'un utilisateur, *deleteAppUser.jsp*. Cette vue est une page de confirmation demandant à l'utilisateur de confirmer son choix de suppression. S'il confirme la suppression, la requête arrive dans la méthode *doPost* de la classe *DeleteAppUser_controller*. Cette méthode fait appel à la classe *AppUser_management* afin de supprimer l'utilisateur. Ce dernier est ensuite redirigé vers la page d'accueil de l'application.

Le fichier *ModifyAppUser_controller.java* est la Servlet contrôlant la modification d'un utilisateur dans le datastore. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à la modification d'un utilisateur, *modifyAppUser.jsp*. Cette vue est un formulaire permettant à l'utilisateur de saisir les nouvelles informations sur son compte. Lorsqu'il finit la saisie et click sur le bouton adéquat, la requête arrive dans méthode *doPost* de la classe *ModifyAppUser_controller*. Cette méthode fait appel à la classe *AppUser_management* afin de modifier l'utilisateur. Ce dernier est ensuite redirigé vers la vue avec un message d'erreur ou de réussite.

7.10.1.2 Package sensor

Le package *sensor* contient toutes les classes gérant les sensors de l'application. Le modèle présenté dans le chapitre *7.8.3 Sensor* est présent dans le fichier *Sensor.java*. Ce fichier contient aussi des getter et setter pour les champs privés. Seuls les champs *sensorID* et *hash* ne possèdent pas de setter. Effectivement, le champ *sensorID* est la clé de l'entité dans le datastore. Il n'est alors pas possible de le modifier. La valeur du *hash* est quant à elle définie lors de la création du sensor. Comme est doit correspondre à une valeur qui sera réellement affichée sur le sensor, elle ne peut pas être modifiée. La Figure 71 représente la Javadoc créée pour les méthodes de la classe *Sensor*.

Modifier and Type	Method and Description
java.lang.String	**getHash**() Getter of the hash of the sensor
long	**getProductDate**() Getter of the date value, when the sensor was produced
java.lang.String	**getSensorID**() Getter of the sensor unique identify
java.lang.String	**getVersion**() Getter of the version of the sensor
void	**setProductDate**(long date) Setter of the date value, the date when the sensor was produced
void	**setVersion**(java.lang.String version) Setter of the version of the sensor

Figure 71 Méthodes de la classe Sensor

Le fichier *DAO_Sensor.java* est la classe s'occupant de la gestion du stockage des objets *Sensor* dans le datastore. Cette classe met des méthodes à disposition afin de gérer le stockage de *Sensor* sans avoir connaissance de la plateforme de stockage. Sur la Figure 72 provenant de la Javadoc de la classe *DAO_Sensor*, on voit que toutes les méthodes de recherche prennent un *sensorID* en paramètre.

Modifier and Type	Method and Description
boolean	**exists**(java.lang.String sensorID) Test if the given sensorID is existing in the datastore.
boolean	**exists**(java.lang.String sensorID, java.lang.String hash) Test if the couple of given sensorID and sensorKey (hash) is existing in the datastore.
Sensor	**get**(java.lang.String sensorID) Get the entry matches with sensorID given in parameter.
java.util.List<java.lang.String>	**getAllSensorID**() Get the sensorID of all sensor in the datastore.
void	**put**(**Sensor** sensor) Put the new Sensor given in parameter in the datastore.

Figure 72 Méthodes de la classe DAO_Sensor

Le fichier *Sensor_management.java* permet de contrôler les objets *Sensor*. Cette classe met une méthode à disposition afin de créer les objets de type *Sensor*. La modification ou la suppression des objets *Sensor* peuvent être effectuées à partie de la console d'administration. L'attribut *errors*, accessible via le getter *getErrors*, est une Map faisant correspondre le nom du champ de la vue avec un message d'erreur. L'attribut *result*, accessible via le getter *getResult*, permet d'obtenir l'état final de l'opération demandée.

La méthode *tryCreateSensor* récupère les paramètres nécessaires à la création d'un *Sensor* dans la requête et essaie de créer un nouvel objet *Sensor*. Ces paramètres proviennent de la vue correspondant à la création d'un nouveau *Sensor*, *newSensor.jsp*. Si les paramètres ne sont pas présents ou ne sont pas correctes, des messages d'erreur sont fournis par les attributs *result* et *errors*. Si tous les paramètres sont corrects, un nouvel objet *Sensor* est créé. Cet objet est ensuite passé au contrôleur pour son traitement.

Modifier and Type	Method and Description
java.util.Map<java.lang.String,java.lang.String>	getErrors() Getter of the error map.
java.lang.String	getResult() Getter of result value.
Sensor	tryCreateSensor(javax.servlet.http.HttpServletRequest request) Try to create a new Sensor from data in the request parameter If request is missing or data in request are not correct, this function return null.

Figure 73 Méthodes de la classe Sensor_management

Le hash passé dans les paramètres de la requête doit être le texte en claire. Il est ensuite haché avec l'algorithme MD5 avant d'être sauvegardé dans le datastore.

Le format de la date de création du sensor doit correspondre au format demandé par la vue. Lors de la réception du paramètre, la date est reçue en tant que String. Il est alors nécessaire de convertir cette valeur en date grâce à un *DateTimeFormatter*. Cette conversion peut lever des erreurs qui doivent être traitées par la méthode *tryCreateSensor*.

```
// Format of productDate to add a new sensor with Sensor_management.java
public static final String dateFormatString = "dd/MM/yyyy";
public static final DateTimeFormatter dateFormatter = DateTimeFormat.forPattern(dateFormatString);
```

Figure 74 Format de la date d'ajout du sensor

Le fichier *AddSensor_controller.java* est la Servlet contrôlant l'ajout d'un sensor dans le datastore. Cette opération ne peut être effectuée que par un administrateur de l'application. Cette limitation est mise en place par le fichier *web.xml*. La méthode *doGet* de cette classe redirige l'administrateur vers la vue correspondant à l'ajout d'un sensor, *newSensor.jsp*. Cette vue est un formulaire que doit remplir l'administrateur afin de compléter les informations concernant le nouveau sensor à créer. Lorsque l'administrateur envoie le formulaire via le bouton adéquat, la requête arrive dans la méthode *doPost* de la classe *AddSensor_controller*. Cette méthode essaie de créer un nouveau sensor via la classe *Sensor_management*. Si le nouvel objet a pu être créé et qu'il n'est pas déjà existant dans le datastore, il est sauvegardé dans le datastore et l'administrateur est redirigé vers le formulaire avec un message de réussite. Si le nouveau sensor n'a pas pu être créé ou qu'il est déjà présent dans le datastore, l'administrateur est redirigé vers la vue avec des messages d'erreur.

7.10.1.3 Package sensorData

Le package *sensorData* contient toutes les classes gérant les données des sensors de l'application. Le modèle présenté dans le chapitre *7.8.4 SensorData* est présent dans le fichier *SensorData.java*. Ce fichier contient aussi des getter pour les champs privés. Seul le champ *value* possède un setter. Effectivement, ce champ est le seul champ susceptible d'être modifié. Cette modification peut survenir lorsque l'utilisateur souhaite obtenir des données de température en une autre unité que les degrés Celsius. La Figure 75 représente la Javadoc créée pour les méthodes de la classe *SensorData*.

Modifier and Type	Method and Description
long	getDate() Getter of the date of the data.
java.lang.String	getSensorID() Getter of the sensor unique identifier
java.lang.String	getType() Getter of the type of data
float	getValue() Getter of the value of the data
void	setValue(float newValue) Setter of the value of the data

Figure 75 Méthodes de la class SensorData

Le fichier *DAO_SensorData.java* est la classe s'occupant de la gestion du stockage des objets *SensorData* dans le datastore. Cette classe met des méthodes à disposition afin de gérer le stockage de *SensorData* sans avoir connaissance de la plateforme de stockage. La Figure 76 provenant de la Javadoc de la classe *DAO_SensorData*, présente les différentes méthodes fournies par cette classe.

Modifier and Type	Method and Description
void	delete(java.lang.String sensorID) Delete all entries of sensorData matches with the sensor ID given in parameter If no entry matches with the sensorID, nothing append.
java.util.List<SensorData>	get(java.lang.String sensorID) Get the data matches with sensorID given in parameter.
java.util.List<SensorData>	get(java.lang.String sensorID, java.lang.String type) Get the data matches with sensorID and type given in parameter.
java.util.List<SensorData>	get(java.lang.String sensorID, java.lang.String type, long timestamp) Get the data matches with sensorID and type given in parameter.
SensorData	getLast(java.lang.String sensorID) Get the last data captured by the sensor matches with sensorID given in parameter.
void	put(SensorData sensorData) Put the new SensorData given in parameter.

Figure 76 Méthodes de la classe DAO_SensorData

Le fichier *CheckSensorConnection_filter.java* est le filtre utilisé pour toutes les pages ayant pour chemin */sensor/**. Ce filtre permet de s'assurer que les informations envoyées au serveur proviendront d'un sensor existant dans l'application. Il est mis en place dans le fichier *web.xml*. Les requêtes pour les pages dont le chemin commence par */sensor/** doivent comporter les paramètres *sensorID* et *sensorKey*. Ces deux paramètres permettent à l'application de définir quel est le sensor actuellement connecté à l'application. Si ces paramètres ne sont pas présent ou qu'ils ne correspondent à aucun sensor du datastore, la requête est redirigée vers la page principale de l'application. Sinon, la requête est redirigée vers la ressource demandée.

Les méthodes *destroy*, *getFilterConfig* et *init* sont des méthodes héritées de la classe *Filter*. Seule la méthode *doFilter* est redéfinie dans la classe *CheckSensorConnection_filter*.

Modifier and Type	Method and Description
void	destroy()
void	doFilter(javax.servlet.ServletRequest request, javax.servlet.ServletResponse response, javax.servlet.FilterChain filterChain) This method filters sensor.
javax.servlet.FilterConfig	getFilterConfig()
void	init(javax.servlet.FilterConfig filterConfig)

Figure 77 Méthodes de la classe CheckSensorConenction_filter

Le fichier *Sensor_management.java* permet de contrôler les objets *SensorData*. Cette classe met des méthodes à disposition afin de créer les objets de type *SensorData*. L'attribut *errors*, accessible via le getter *getErrors*, est une Map faisant correspondre le nom du champ de la vue avec un message d'erreur. L'attribut *result*, accessible via le getter *getResult*, permet d'obtenir l'état final de l'opération demandée.

La méthode *tryCreateSensorData* récupère les paramètres nécessaires à la création d'un *SensorData* dans la requête et essaie de créer un nouvel objet *SensorData*. Ces paramètres proviennent de la vue correspondante à la création d'un nouveau *SensorData*, *addSensorData.jsp*. Les valeurs sont donc entrées manuellement dans le système. Si les paramètres ne sont pas présents ou ne sont pas correctes, des messages d'erreur sont fournis par les attributs *result* et *errors*. Si tous les paramètres sont corrects, un nouvel objet *SensorData* est créé. Cet objet est ensuite passé au contrôleur pour son traitement.

La valeur de la mesure est récupérée dans les paramètres. Elle est donc en format String. Il est alors nécessaire de convertir cette valeur en float afin de créer le *SensorData*. Cette conversion se fait via la méthode *parseFloat* de la classe *Float*. Cette méthode peut lever des exceptions qui doivent être gérées par la méthode *tryCreateSensorData*.

Le format de la date de création de la donnée du sensor doit correspondre au format demandé. Lors de la réception du paramètre, la date est reçue en tant que String. Il est alors nécessaire de convertir cette valeur en date grâce à un *DateTimeFormatter*. Cette conversion peut lever des erreurs qui doivent être traitées par la méthode *tryCreateSensorData*. Le format attendu est indiqué dans la Figure 78.

```
// Format of date to add a new sensorData with SensorData_management.java
public static final String dateHourFormatString = "dd/MM/yyyy HH:mm:ss";
public static final DateTimeFormatter dateHourFormatter = DateTimeFormat.forPattern(dateHourFormatString);
```

Figure 78 Format de la date d'ajout d'une donnée

La méthode *addDataFromSensor* récupère les paramètres nécessaires à la création d'un *SensorData* dans la requête et essaie de créer un nouvel objet *SensorData*. Ces paramètres proviennent directement du sensor, aucune vue n'est associée pour la saisie des valeurs. C'est donc au sensor lui-même d'ajouter ces valeurs dans la requête. Si les paramètres ne sont pas présents la création du nouvel objet échoue et la méthode retourne *null*. Si tous les paramètres sont corrects, un nouvel objet *SensorData* est créé. Cet objet est ensuite passé au contrôleur pour son traitement.

Tout comme pour la méthode *tryCreateSensorData*, il est nécessaire de convertir les Strings de valeur et de timestamp en valeur numérique. Il est donc nécessaire de gérer les exceptions potentiellement levées par ces conversions.

Modifier and Type	Method and Description
SensorData	addDataFromSensor(javax.servlet.http.HttpServletRequest request) Try to create a new sensorData from data in the request parameter from a sensor.
java.util.Map<java.lang.String,java.lang.String>	getErrors() Getter of the error map.
java.lang.String	getResult() Getter of result value.
SensorData	tryCreateSensorData(javax.servlet.http.HttpServletRequest request) Try to create a new sensorData from data in the request parameter.

Figure 79 Méthodes de la classe Sensor_management

Le fichier *AddSensorData_controller.java* est la Servlet contrôlant l'ajout manuel d'une donnée dans le datastore. Cet ajout est effectué par la page *addSensorData.jsp* qui n'est accessible que pour les administrateurs de l'application. La méthode *doGet* de cette classe redirige l'administrateur vers la vue correspondant à l'ajout d'une donnée, *addSensorData.jsp*. Cette vue est un formulaire que doit remplir l'administrateur afin de compléter les informations concernant la nouvelle donnée à créer. Lorsque l'administrateur envoi le formulaire via le bouton adéquat, la requête arrive dans la méthode *doPost* de la classe *AddSensorData_controller*. Cette méthode essaie de créer un nouveau *SensorData* via la classe *SensorData_management*. Si le nouvel objet a pu être créé, il est sauvegardé dans le datastore et l'administrateur est redirigé vers la vue avec un message de réussite. Sinon, l'administrateur est redirigé vers le formulaire avec des messages d'erreurs.

Le fichier *NewSensorDataFromSensor_controller.java* est la Servlet contrôlant l'ajout d'une donnée provenant d'un sensor dans le datastore. Ce contrôleur n'a pas de vue associée. Effectivement, les sensors envoient directement leurs informations sur cette Servlet. C'est donc aux sensors eux-mêmes de mettre les informations nécessaires dans la requête. La méthode *doGet* n'existe pas dans cette classe. La seule action possible est l'envoi de données par le sensor vers le serveur. Cet envoi se fait par une requête POST qui sera dirigée vers la méthode *doPost* de la classe *NewSensorDataFromSensor_controller*. Par la suite, il est possible d'implémenter la méthode *doGet* afin que les sensors puissent demander des informations au serveur.

Lorsqu'un sensor envoie une donnée sur le serveur, la requête arrive dans la méthode *doPost* de la Servlet. Les données envoyées peuvent être de deux types. Le premier est l'envoi de données de mesures. Il s'agit donc de nouvelles valeurs capturées par le sensor. Le deuxième type de données est les données de méta-data. Ces données fournissent des informations sur le sensor lui-même. La Figure 80 donnes les attributs nécessaires afin de pouvoir envoyer les données de chaque type.

```
* The sensor can send two types of messages. Here are the attributes of these
* messages:
*
* 1) New data information
*  - sensorID
*  - sensorKey (no hash)
*  - type
*  - sensorValue
*  - timestamp
*
* 2) Information about sensor itself
*  - sensorID
*  - sensorKey (no hash)
*  - battery
*  - RSSI
*  - codeVersion
```

Figure 80 Type de données provenant du sensor

Le champ *sensorValue* est la valeur mesurée par le sensor. Cette valeur est une mesure qu'il faut adapter avant de la stocker. En se basant sur le wiki d'Aginova, nous voyons que l'adaptation se fait avec la relation mathématique utilisée à la Figure 81.

```
floatValue = RequestUtilities.round(intValue/100.0f - 250f);
```

Figure 81 Adaptation de la valeur mesurée

Lorsque le contrôleur reçoit une requête, il essaie de créer une nouvelle data via la classe *SensorData_management*. Il essaie ensuite de mettre à jour le *UserSensor* associé au sensor ayant

envoyé les données grâce à la classe *UserSensor_management*. Si l'une ou l'autre des opérations réussit, le contrôleur stocke les nouvelles valeurs dans le datastore. La Servlet vérifie ensuite si le timestamp reçu du sensor est correct. Le timestamp est dit correct si la différence entre sa valeur est la valeur courant du timestamp du système est plus grande qu'une constante. Cette constante est définie dans le fichier *Config.java* du package *utilities*. Si le timestamp du sensor n'est pas correct, la valeur du timestamp du système est mise en attribut dans la requête. Cette valeur pourra alors être récupérée par le sensor pour qu'il puisse se synchroniser. De plus, le temps sauvegardé avec la mesure sera alors le moment de la réception du sensor et non le timestamp reçu par le sensor.

La requête du sensor est ensuite renvoyée sur la page *newSensorData.jsp* qui permet de transmettre des messages au sensor. Actuellement, seule une donnée de timestamp est présente sur cette page si le timestamp reçu du sensor n'est pas correct. Il est aussi possible de passer par cette page pour signaler des éventuelles erreurs.

7.10.1.4 Package userSensor

Le package *userSensor* contient toutes les classes gérant l'association entre les sensors et les utilisateurs de l'application. Le modèle présenté dans le chapitre *7.8.5 UserSensor* est présent dans le fichier *UserSensor.java*. Ce fichier contient aussi des getter et setter pour les champs privés. Seul le champ *sensorID* ne possède pas de setter. Effectivement, ce champ est la clé de l'entité dans le datastore. Il n'est alors pas possible de le modifier. La Figure 82 représente la Javadoc créée pour les méthodes de la classe *UserSensor*. Pour rappel, la gestion des deux listes de *userID* et de note est complétement transparente pour l'utilisation de la classe en passant par les getter et setter.

Modifier and Type	Method and Description
void	**addUserID**(java.lang.String id) This method add a new user to the userID List.
void	**addUserID**(java.lang.String id, java.lang.String note) This method add a new user to the userID List and add a corresponding note.
void	**deleteUserID**(java.lang.String id) This method delete the given user to the userID List.
float	**getBattery**() Getter of the last battery level received from the sensor
java.lang.String	**getCodeVersion**() Getter of the code version installed on the sensor
boolean	**getIsPremium**() Getter for isPremium value.
java.lang.String	**getNote**() Getter for note value.
java.lang.String	**getNote**(java.lang.String userID) Getter for note value.
java.util.List<java.lang.String>	**getNotes**() Getter for note value.
int	**getRSSI**() Getter of the last RSSI of the sensor
java.lang.String	**getSensorID**() Getter for sensorID value
java.util.List<java.lang.String>	**getUserID**() Getter of userID, the list of string representing Users ID
java.util.List<**AppUser**>	**getUsers**() Getter a list of users registered as owner of the sensor
boolean	**isPremium**() Getter for isPremium value.
void	**setBattery**(float battery) Setter for the battery value of the sensor
void	**setCodeVersion**(java.lang.String codeVersion) Setter for code version of the sensor
void	**setNote**(java.lang.String note) Setter for note value.
void	**setNote**(java.lang.String userID, java.lang.String note) Setter for note value.
void	**setPremium**(boolean isPremium) Setter for isPremium value
void	**setRSSI**(int RSSI) Setter for RSSI value

Figure 82 Méthodes de la classe UserSensor

Le fichier *DAO_UserSensor.java* est la classe s'occupant de la gestion du stockage des objets *UserSensor* dans le datastore. Cette classe met des méthodes à disposition afin de gérer le stockage de *UserSensor* sans avoir connaissance de la plateforme de stockage. Sur la Figure 83 provenant de la Javadoc de la classe *DAO_UserSensor*, on voit que toutes les méthodes de recherche prennent un *sensorID* en paramètre, qui est la clé de l'entité.

Modifier and Type	Method and Description
void	**delete**(java.lang.String sensorID) Delete the entry of UserSensor from the datastore.
boolean	**exists**(java.lang.String sensorID) Test if the given sensorID is existing in the datastore.
UserSensor	**get**(java.lang.String sensorID) Get the entry matches with sensorID given in parameter.
java.util.List<**UserSensor**>	**getSensors**(java.lang.String userID) Get the entries matches with userID given in parameter.
boolean	**isOwner**(java.lang.String sensorID, java.lang.String userID) Verify if the user is one of the owner of the sensor Return true if the user is one of the owner, false otherwise
void	**put**(**UserSensor** userSensor) Put the UserSensor given in parameter in the datastore If the UserSensor already exists on the datastore, existing entity will be overwritten

Figure 83 Méthodes de la classe DAO_UserSensor

Le fichier *UserSensor_management.java* permet de contrôler les objets *UserSensor*. Cette classe met des méthodes à disposition afin de créer, modifier ou supprimer les objets de type *UserSensor*.

L'attribut *errors*, accessible via le getter *getErrors*, est une Map faisant correspondre le nom du champ de la vue avec un message d'erreur. L'attribut *result*, accessible via le getter *getResult*, permet d'obtenir l'état final de l'opération demandée.

La méthode *modifyNote* est utilisée lorsqu'un utilisateur souhaite modifier le nom qu'il a donné à un sensor. La méthode récupère les paramètres nécessaires à la modification du nom dans la requête et essaie de modifier l'objet *UserSensor* concerné. Ces paramètres proviennent de la vue correspondante à la modification du nom du sensor, *ModifyNote.jsp*. Si les paramètres ne sont pas présents ou ne sont pas correctes, des messages d'erreur sont fournis par les attributs *result* et *errors*. Si tous les paramètres sont corrects, la méthode met à jour le nom du sensor associé à l'utilisateur courant et stocke ce nouveau nom dans le datastore.

La méthode *dissociateUser* permet de supprimer le lien créé entre un utilisateur est un sensor. Cette méthode vérifie que le paramètre *deleteConfirmation* soit présent dans la requête et que sa valeur soit "*delete*". Cette vérification permet de s'assurer que l'utilisateur a confirmé la dissociation de son compte et du sensor. Si tous les paramètres sont corrects, l'utilisateur est dissocié du sensor. Si l'utilisateur était le dernier propriétaire du sensor, l'objet *AppUser* ainsi que toutes les données *SensorData* relatives au sensor sont supprimées du datastore.

La méthode *addMetadataFromSensor* récupère les paramètres nécessaires à la mise à jour des données du *SensorUser*. Ces paramètres proviennent directement du sensor, aucune vue n'est associée pour la saisie des valeurs. C'est donc au sensor lui-même d'ajouter ces valeurs dans la requête. Si les paramètres ne sont pas présents, la mise à jour n'a pas lieu et la méthode retourne *null*. Si tous les paramètres sont corrects, l'objet *UserSensor* correspondant au sensor transmettant les données est mis à jour. Cet objet est ensuite passé au contrôleur pour son traitement.

Modifier and Type	Method and Description
UserSensor	addMetadataFromSensor(javax.servlet.http.HttpServletRequest request) Try to update userSensor from data in the request parameter from a sensor.
void	dissociateUser(javax.servlet.http.HttpServletRequest request) Try to dissociate the current user with the selected sensor present in the parameters.
java.util.Map<java.lang.String,java.lang.String>	getErrors() Getter of the error map.
java.lang.String	getResult() Getter of result value.
void	modifyNote(javax.servlet.http.HttpServletRequest request) Try to modify a note of a sensor from data in the request parameter If the given parameters are correct, the value of the note for the selected sensor will be update.
UserSensor	tryCreateUserSensor(javax.servlet.http.HttpServletRequest request) Try to create a new userSensor from data in the request parameter If request is missing or data in request are not correct, this function return null.

Figure 84 Méthodes de la classe UserSensor_management

Le fichier *AddUserSensor_controller.java* est la Servlet contrôlant l'ajout d'une relation entre un sensor et un utilisateur dans le datastore. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à l'ajout d'un *UserSensor*, *newUserSensor.jsp*. Cette vue est un formulaire que doit remplir l'utilisateur afin de compléter les informations concernant la nouvelle association à créer. Lorsque l'utilisateur envoie le formulaire via le bouton adéquat, la requête arrive dans la méthode *doPost* de la classe *AddUserSensor_controller*. Cette méthode essaie de créer un nouvel *UserSensor* via la classe *UserSensor_management*. Si le nouvel objet a pu être créé, il est sauvegardé dans le datastore et l'utilisateur est redirigé vers la vue avec un message de réussite. Sinon, l'utilisateur est redirigé vers le formulaire avec des messages d'erreurs.

Le fichier *Dissociate_controller.java* est la Servlet contrôlant la suppression d'une relation entre un sensor et un utilisateur dans le datastore. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à la suppression d'un *UserSensor*, *dissociate.jsp*. Cette vue est une page de confirmation demandant à l'utilisateur de confirmer son choix de suppression. S'il confirme la suppression, la requête arrive dans la méthode *doPost* de la classe *Dissociate_controller*. Cette méthode fait appel à la classe *UserSensor_management* afin de supprimer le lien entre le sensor et l'utilisateur. Ce dernier est ensuite redirigé vers vue avec un message de réussite ou des messages d'erreurs.

Le fichier *ModifyNote_controller.java* est la Servlet contrôlant la modification du nom qu'un utilisateur a défini pour une sensor. La méthode *doGet* de cette classe redirige l'utilisateur vers la vue correspondant à la modification d'un nom de sensor, *modifyNote.jsp*. Cette vue est un formulaire permettant à l'utilisateur de saisir un nouveau nom pour un de ses sensors. Lorsqu'il finit la saisie et click sur le bouton adéquat, la requête arrive dans méthode *doPost* de la classe *ModifyNote_controller*. Cette méthode fait appel à la classe *UserSensor_management* afin de modifier le nom du sensor de l'utilisateur. Ce dernier est ensuite redirigé vers la vue avec un message d'erreur ou de réussite.

7.10.1.5 Package applicationGeneralServlet

Dans les paquetages précédant, un petit nombre de Servlets était présent. L'application créée utilise plus de Servlets, qui ne sont pas nécessairement associés à un modèle Java. Toutes ces Servlets sont regroupées sont un paquetage nommé *applicationGeneralServlet*.

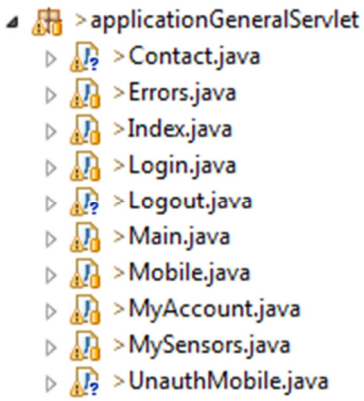


Figure 85 Paquetage de Servlets

Le fichier *Contact.java* est le contrôleur permettant à l'utilisateur de contacter un administrateur de l'application. Lorsque cette classe reçoit une requête GET, elle renvoie l'utilisateur sur la page *contact.jsp* qui est un formulaire de contact. Lorsque l'utilisateur rempli ce formulaire et appuie sur le bouton pour l'envoyer, une requête POST est générée. Elle arrive alors dans la méthode *doPost* de la classe *Contact* qui a pour rôle de gérer la requête. Cette méthode récupère les valeurs saisies par l'utilisateur. Si toutes les valeurs sont existantes, le programme essaie d'envoyer le message grâce au service Mail de Google via la classe *Mail* du package *utilities*. Un message de réussite ou d'erreur est ensuite affiché à l'utilisateur.

Le fichier *errors.java* est une Servlet permettant la gestion des erreurs en créant des logs des évènements reçus. La Servlet redirige ensuite l'utilisateur vers la page *error.jsp* contenant un message d'erreur personnalisé selon l'erreur actuelle. Cette Servlet est fonctionnelle quelle que soit

la méthode utilisée pour la requête. Une requête GET aura alors exactement le même comportement qu'une requête POST.

Le fichier *index.java* est le point d'entrée de l'application. Le descripteur de déploiement redirige toutes les connexions vers ce fichier par défaut. Le rôle de cette Servlet est de définir si un utilisateur est connecté à son compte Google et s'il est déjà existant dans le datastore en tant qu'utilisateur de l'application. Si l'utilisateur n'est pas connecté à son compte Google, il est redirigé vers le fichier *index.jsp* qui lui présente l'application et l'invite à se connecter. Si l'utilisateur est connecté à son compte Google mais ne possède pas encore de compte sur l'application, il est redirigé vers la page *newAppUser.jsp*. Cette page présente l'application et permet à l'utilisateur de s'enregistrer en tant qu'utilisateur de l'application. Si l'utilisateur est déjà enregistré dans l'application, il est redirigé vers la page *main.jsp* qui lui présente les dernières données de ses sensors.

Le fichier *Login.java* permet de rediriger l'utilisateur si celui-ci n'est pas connecté à son compte Google. S'il n'est pas connecté, l'utilisateur sera redirigé vers la page de connexion de Google. Dès que la connexion est effectuée, il est ensuite redirigé vers la page d'accueil de l'application.

Le fichier *Logout.java* est une Servlet permettant à un utilisateur connecté à son compte Google de se déconnecter. Après la déconnexion, l'utilisateur est redirigé sur la page d'accueil du site. Cette redirection après déconnexion est la seule fonctionnalité de la Servlet.

Le fichier *Main.java* est la Servlet correspondante à la page principale sur laquelle les utilisateurs de l'application navigueront. Le but de cette Servlet est de fournir à la vue une liste de tous les sensors associés à l'utilisateur courant. Pour chacun de ces sensors, la Servlet fournit aussi la dernière valeur reçue. De plus, la transformation d'unité des valeurs de température est effectuée si nécessaire avant la transmission des valeurs à la vue. Si les données sont modifiées pour être affichées en degrés Fahrenheit, l'attribut *isFarhenheit* est ajouté à la requête avec la valeur *true*. La vue saura alors comment afficher l'unité correctement. La date de la dernière valeur des sensors est aussi adaptée au format choisi par l'utilisateur.

Le fichier *Mobile.java* permet de transmettre les informations des sensors à un utilisateur utilisant un appareil mobile avec l'application Android. La méthode *doGet* de cette Servlet fournit à la vue les informations nécessaire pour que celle-ci affiche les sensors de l'utilisateur et leur dernière valeur.
La méthode *doGet* fournit à la vue une liste de tous les sensors associés à l'utilisateur courant. Pour chacun de ces sensors, la Servlet fournit aussi la dernière valeur reçue. De plus, la transformation d'unité des valeurs de température est effectuée si nécessaire avant la transmission des valeurs à la vue. Si les données sont modifiées pour être affichées en degrés Fahrenheit, l'attribut *isFarhenheit* est ajouté à la requête avec la valeur *true*. La vue saura alors comment afficher l'unité correctement. La date de la dernière valeur des sensors est donnée à la vue en format UNIX. Il est alors aussi nécessaire de lui transmettre le date pattern définit par l'utilisateur. La différence par rapport au fichier *Main.java* et que la vue n'est pas une interface graphique mais une simple page donnant les informations utiles à l'application en format JSON.
La méthode *doPost* de cette Servlet permet à une application mobile de demander des informations sur un sensor spécifique. Lorsque la Servlet reçoit une requête POST, elle récupère les paramètres de requête nommés *selectedSensor* et *selectedType*. Si ces paramètres sont existants et que le sensor sélectionné est associé à l'utilisateur courant, la méthode fournit à la vue une liste de données du

type sélectionné pour le sensor sélectionné. Si ces paramètres ne sont pas corrects, l'utilisateur est redirigé vers la page d'accueil de l'application web.

Le fichier *MyAccount.java* est la Servlet permettant à l'utilisateur de gérer son compte et ses sensors. Cette Servlet ne possède qu'une méthode *doGet* qui redirige les utilisateurs vers la vue *myAccount.jsp*. Cette vue est alors composée de différents choix permettant de modifier les informations sur le compte ou sur les sensors.

Le fichier *MySensors.java* est la Servlet permettant à l'utilisateur de visualiser des informations spécifiques à un sensor. L'utilisateur peut choisir quelles informations afficher. De plus, il a la possibilité d'activer une option lui permettant de rafraichir automatiquement la page pour voir apparaitre les nouvelles données. Lorsque qu'une requête GET arrive, cette Servlet doit fournir une liste de tous les sensors associés à l'utilisateur ainsi que la liste des types de valeurs possibles. L'utilisateur peut alors sélectionner le sensor et le type de valeurs souhaité grâce à la vue. Les valeurs sélectionnées sont envoyées à la Servlet via une requête POST qui arrive dans la méthode *doPost*. La Servlet récupère alors les informations de la vue et test le paramètre *refreshMode*. Ce paramètre permet d'indiquer au contrôleur si la requête est une demande de rafraichissement ou une demande complète. Lorsqu'il s'agit d'une demande standard, le paramètre *refreshMode* n'existe pas, il vaut donc *null*.

Dans le mode normale, la Servlet récupère toutes les valeurs du datastore concernant le sensor sélectionné et le type sélectionné. Il fournit ensuite une liste de dates au format sélectionné par l'utilisateur. Si les données sont des informations de température et que l'utilisateur ne souhaite pas voir afficher ses données en degrés Celsius, la modification d'unité est effectuée. Si les données sont modifiées pour être affichées en degrés Fahrenheit, l'attribut *isFarhenheit* est ajouté à la requête avec la valeur *true*. La vue saura alors comment afficher l'unité correctement.

Dans le mode refresh, le contrôleur ne doit pas passer toutes les valeurs à la vue. Une requête de type refresh provient d'une page *mySensors.jsp* qui affiche déjà des informations. Cette page ajoute le paramètre *refreshMode* avec la valeur *true* à la requête. La vue n'a alors besoin que des informations plus récentes que celles qu'elle possède déjà. Pour ce faire, la vue fournit un paramètre nommé *lastTimestamp* qui contient le timestamp de la dernière valeur déjà affichée. La vue envoie ensuite une requête POST comprenant ces deux paramètres sur le serveur grâce à JQuery. Dès que la méthode *doPost* de la classe *MySensor.java* reçoit une requête POST avec ces deux paramètres, elle envoie une réponse de rafraichissement. Cette réponse est composée des données plus récentes que le timestamp fourni par la vue. Les dates sont aussi passées en String selon le format demandé par l'utilisateur. Ces données sont formatées en JSON avant d'être renvoyées à la vue.

Le fichier *UnauthMobile.java* permet à des utilisateurs non connecté d'accéder à des informations sur les sensors. Cette Servlet est existante uniquement pendant la période de développement. Elle a été créée pour que l'application Android existante puisse accéder aux informations sans nécessiter la connexion avec un compte Google Account. Les méthodes *doGet* et *doPost* de cette Servlet ont le même rôle que pour la Servlet du fichier *Mobile.java*. Cependant de légères différences apparaissent du fait que l'utilisateur n'a pas besoin d'être connecté. Dans la méthode *doGet*, le code prend comme utilisateur par défaut le compte *TB.2013.TinguelyJoel*. Ce compte a été créé spécialement pour ce travail. Tous les sensors associés à ce compte seront alors fournis à la vue. La méthode *doPost* ne

contrôle pas que le *sensorID* passé dans les paramètres soit bien associé à l'utilisateur. Ce point est la grande différence entre cette méthode et celle définie dans le fichier *Mobile.java*.

7.10.1.6 Package utilities

Le paquetage *utilities* regroupe des fichiers Java qui ne sont pas des Servlets et ne sont pas associés à un modèle particulier. Il s'agit de fichiers fournissant des constantes ou des méthodes utiles à toute l'application.

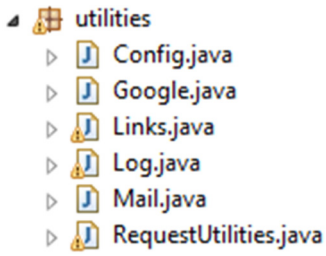


Figure 86 Paquetage utilities

Dans le fichier *Config.java* sont déclarées les différentes constantes utilisées dans l'application. Ces constantes concernent les types de probe, les dates patterns, les unités de température possibles et le délai maximal de désynchronisation d'un sensor. Les types de probe correspondent aux différents types de valeurs pouvant être reçues d'un sensor. Ces valeurs dépendent de la probe connectée au sensor. A chaque donnée reçue d'un sensor, le type de valeur est fourni par le sensor. Les dates patterns permettent de définir le format de la date qui sera affichée à l'utilisateur. Grâce à ces patterns, l'utilisateur peut choisir entre le format de date Européen ou Américain. Les constantes d'unités de température permettent à l'utilisateur de choisir dans quelle unité seront affichées les données de températures. Ces trois types de constantes sont regroupés dans trois listes afin que le programme puisse les parcourir aisément. La liste contenant les types de sensors est définie dans un ordre bien précis. Cet ordre est utilisé dans les vues pour définir quel est le type de la mesure à afficher. Le délai de désynchronisation est le temps maximal entre la capture de la mesure sur le sensor et la réception de celle-ci sur le serveur. Si ce délai est dépassé, le temps sauvegardé avec la mesure sera le moment de la réception de la donnée du sensor.

Le fichier *Google.java* nous met à disposition des méthodes fournissant des valeurs provenant de services de Google. Ces méthodes permettent de simplifier l'écriture du code mais aussi de simplifier le portage de l'application s'il devait avoir lieu. Cette classe contient plusieurs méthodes dont trois nous permettant d'obtenir le *userID, nickname* et *email* associé au compte Google de l'utilisateur. Deux méthodes nous fournissent les liens URL pour la connexion et la déconnexion des utilisateurs. La méthode *isUserAdmin* nous permet de définir si l'utilisateur actuel est connecté en tant qu'administrateur de l'application.

Modifier and Type	Method and Description
static java.lang.String	**getGoogleUserEmail**() Return Google email of the current user
static java.lang.String	**getGoogleUserId**() Return user ID of the current user
static java.lang.String	**getGoogleUserLoginURL**(javax.servlet.http.HttpServletRequest request) Return the login URL for the current user
static java.lang.String	**getGoogleUserLogoutURL**(javax.servlet.http.HttpServletRequest request) Return the logout URL for the current user
static java.lang.String	**getGoogleUserName**() Return Google nickname of the current user
static boolean	**isUserAdmin**() Return true if the current user is an administrator, false otherwise

Figure 87 Méthodes de la classe Google

Le fichier *Links.java* contient des constantes String représentant différents chemins d'accès. Les constantes représentent les chemins d'accès absolus pour les fichiers JSP. De plus les chemins utilisés pour l'accès des Servlets sont aussi définis par des constantes. Il est alors possible d'utiliser ces valeurs dans toutes les classes Java très facilement. Ces constantes permettent à l'application d'être évolutive le plus possible. Il n'est pas possible d'utiliser ces constantes dans le fichier *web.xml*, fichier par lequel passent toutes les requêtes et qu'il faudra donc obligatoirement modifier. Dans l'application créée, l'utilisation des constantes dans les vues est faite avec des scriptlets Java. Dès la version 3.0 des expressions langage, l'utilisation de constantes Java dans les fichiers JSP est grandement simplifiée. Il faudra alors modifier les fichiers JSP de l'application pour éviter l'utilisation de scriptlet et remplacer toutes les occurrences de scriptlet par les expressions langages correspondantes.

Le fichier *Log.java* fournit des méthodes permettant de loguer des informations ayant un niveau *info, warning* ou *severe.* Ce fichier a été créé afin de supporter tout changement du système de log. Effectivement, si un changement survenait, seul ce fichier serait à modifier.

Modifier and Type	Method and Description
static void	**logError**(java.lang.String content) This function write a new log with "content" information The log is a severe level
static void	**logInfo**(java.lang.String content) This function write a new log with "content" information The log is a info level
static void	**logWarning**(java.lang.String content) This function write a new log with "content" information The log is a warning level

Figure 88 Méthodes de la classe Log

Le fichier *Mail.java* fournit une méthode permettant l'envoi d'email à une adresse mail prédéfinie. Cette fonctionnalité est utile pour l'envoi de message d'erreur sur une adresse mail régulièrement consultée. De plus, les utilisateurs ont la possibilité d'envoyer des mails s'ils ont des questions ou remarques concernant l'application. L'envoi d'email utilise le service *Mail* fourni par Google.

Modifier and Type	Method and Description
static int	**sendEmail**(java.lang.String subject, java.lang.String body, **AppUser** user) This function send email to an administrator of the website

Figure 89 Méthode de la classe Mail

Le fichier *RequestUtilities.java* fournit des méthodes qui seront utiles pour les informations transitant par l'objet de requête. Les méthodes *checkEmail* et *checkString* gèrent le contrôle des valeurs fournies dans les formulaires. Ces méthodes permettent de contrôler si le format de la saisie correspond à un email ou à un string valide. La méthode *getRequestValue* permet de récupérer un paramètre présent dans la requête. Si ce paramètre n'est pas trouvé, cette méthode retourne *null*. La méthode *round* permet d'arrondir une valeur float à une décimale après la virgule. Cette méthode est utilisée pour tous les calculs effectués sur des valeurs devant être affichées. La méthode la plus utilisée de ce fichier est *setDefaultAttributes* qui ajoute trois attributs à la requête. Ces attributs sont la page de contenu passée en paramètre, la date de dernière mise à jour de la page et une valeur indiquant si l'utilisateur courant est un administrateur. Ces trois valeurs sont utiles pour toutes les pages consultées sur l'application. Chaque contrôleur appel alors cette méthode afin d'ajouter ces valeurs à la requête avant de la transférer sur la vue.

Modifier and Type	Method and Description
static void	**checkEmail**(java.lang.String email) Check if the given email address is possible
static void	**checkString**(java.lang.String value, int nbCar) Check if the given String is longer than nbCar characters
static java.lang.String	**getRequestValue**(javax.servlet.http.HttpServletRequest request, java.lang.String fieldName) This method return null if the parameter is empty.
static float	**round**(float value) This method get a float value and return this value round with 1 decimal
static void	**setDefaultAttributes**(javax.servlet.http.HttpServletRequest request, java.lang.String content) This method set default attributes to the request given in parameter.

Figure 90 Méthodes de la classe RequestUtilities

7.10.1.7 Répertoire war

Tous comme les fichiers Java présents dans le répertoire *src*, les fichiers présents dans le répertoire *war* utilisés par l'application sont contenus dans plusieurs dossiers. Cette classification permet notamment de faciliter la gestion de l'accès aux pages grâce aux filtres mis en places.

Le répertoire *war* contient les dossiers *errors*, *stylesheets* et *WEB-INF* ainsi qu'un fichier se nommant *favicon.ico.* Ce dernier fichier permet de définir l'image qui sera présente dans la barre d'adresse lorsqu'un utilisateur navigue sur l'application. De plus, cette image sera présente dans la barre de favori si l'utilisateur enregistre l'application web dans ses favoris. La Figure 92 représente le favicon mis en place pour l'application.

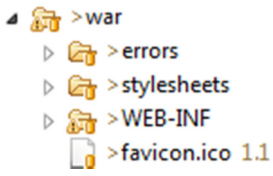


Figure 91 Répertoire war

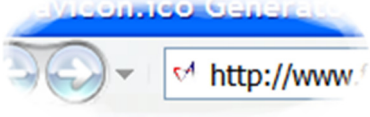


Figure 92 Favicon de l'application

Le dossier *errors* contient des fichiers HTML qui seront utilisés en cas d'erreurs dans l'application. Ces fichiers statiques sont utilisés dans le fichier de configuration de l'application, *appengine-web.xml*. Ils

permettent d'afficher des messages d'erreurs personnalisés en cas de timeout, de dépassement des quotas ou de clients bloqués à cause de la protection DoS. Un dernier fichier HTML est utilisé par le fichier de configuration lors d'une erreur non définie. Chacun de ces fichiers affiche un message d'erreur à l'utilisateur.

Le fichier *exceptionError.jsp* permet de gérer les erreurs dues aux exceptions. Ces erreurs surviennent lors de l'exécution de l'application. Lorsqu'une exception se produit, le fichier *web.xml* et le fichier *template.jsp* sont paramétrés pour envoyer l'utilisateur sur cette page. Actuellement, cette page affiche uniquement l'exception sous forme d'un tableau. Aucun message d'erreur explicite n'est donc affiché pour un utilisateur normal.

Le fichier *errors.jsp* est utilisé par le descripteur de déploiement, *web.xml*, pour traiter les erreurs. Cette vue est associée à la Servlet *Error* contenue dans le package *applicationGeneralServlet*. Cette Servlet fournit un message d'erreur qui sera affiché par la vue. Ce fichier sera utilisé lors d'une erreur interne du serveur, lorsque l'utilisateur tente d'accéder à une page réservée à un administrateur ou lors d'une erreur inconnue.

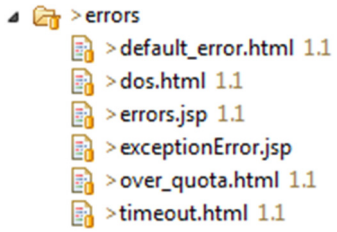


Figure 93 Répertoire war/errors

Le dossier *stylesheets* contient le fichier CSS *styles.css* qui est en charge du formatage de l'affichage de l'application. Ce fichier est importé par le template et est donc utilisé dans toutes les pages de l'application. Dans le dossier *stylesheets/Images*, se situent toutes les images utilisées dans l'application.

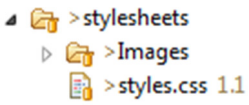


Figure 94 Répertoire war/stylesheets

Le dossier *WEB-INF* contient toutes les pages JSP et les fichiers de configuration de l'application. Pour rappel, les pages JSP présentes dans ce dossier ne sont pas accessibles directement de l'utilisateur. Ce dernier est alors obligé de passer par des Servlets, ce qui respecte le découpage du modèle MVC. Ce répertoire contient les fichiers de configuration suivants:

- *web.xml,* le descripteur de déploiement présenté dans au chapitre *7.4.1 Descripteur de déploiement*
- *appengine-web.xml,* le fichier de configuration de l'application présenté dans le chapitre *7.4.2 Configuration de l'application*.
- *datastore-indexes.xml*, le fichier d'index du datastore local présenté dans le chapitre *7.4.3 Configuration des index.*
- *logging.properties*, le fichier de configuration utilisé pour les logs, présenté au chapitre *7.10.4 Système de log*.

- *taglibs.jsp*, le fichier JSP ajoutant la librairie JSTL à toutes nos pages, présenté au chapitre *7.7.6 Librairie JSTL*.

De plus, le fichier *index.jsp* est présent dans le répertoire *WEB-INF*. Ce fichier représente la page d'accueil de l'application pour un utilisateur non connecté à son compte. Aucun filtre n'est appliqué aux requêtes pour cette page. Cette page présente le site et invite l'utilisateur à se connecter à son compte.

Les fichiers *unauthMobile.jsp* et *unauthMobileSelectedSensor.jsp* sont les vues associées au Servlet du fichier *UnauthMobile.java*. Ces vues on le même rôle que les vues *mobile.jsp* et *mobileSelectedSensor.jsp* présentes dans le dossier *user*. La seule différence par rapport à ces fichiers est le nom du sensor. Comme l'utilisateur provenant de la Servlet *UnauthMobile.java* n'est pas authentifié sur le serveur, il n'est pas possible de récupérer le nom qu'il a défini pour le sensor. Un nom par défaut est alors donné. Il est constitué de la manière suivante: *userName_<sensorID>.*

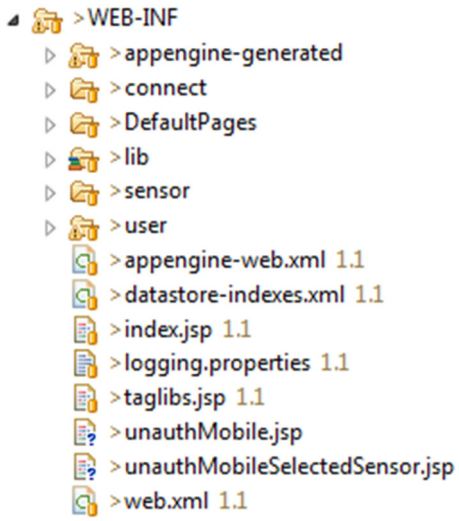


Figure 95 Répertoire war/WEB-INF

Le dossier *connect* contient les pages JSP qui sont accessibles uniquement pour les utilisateurs connectés à leur compte Google. Seule la page d'inscription à l'application, *newAppUser.jsp*, est présente dans ce dossier. Cette page est un formulaire comprenant le nickname, l'email, le format de la date et l'unité de température souhaitée par l'utilisateur.

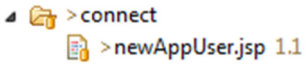


Figure 96 Répertoire war/WEB-INF/connect

Le répertoire *sensor* contient les pages JSP qui sont accessibles uniquement par les sensors. Cette restriction est mise en place grâce à un filtre qui test les paramètres *sensorID* et *sensorKey* de la requête. Seul le fichier *newSensorData.jsp* est présent dans ce répertoire. Ce fichier permet aux sensors de transmettre de nouvelles données au serveur.

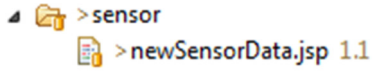


Figure 97 Répertoire war/WEB-INF/sensor

Le répertoire *user* contient les pages JSP qui sont accessibles uniquement par les utilisateurs enregistrés dans l'application. Cette restriction est mise en place grâce au filtre présenté dans la Figure 31 de la page 35. Ce répertoire contient la plupart des fichiers JSP de l'application.

contact.jsp	Met un formulaire à disposition afin de pouvoir envoyer un email.
deleteAppUser.jsp	Page de confirmation qui demande de confirmer la suppression du compte *AppUser.*
dissociate.jsp	Page de confirmation qui demande de sélectionner un sensor et de confirmer la dissociation avec celui-ci.
main.jsp	Page principale qui affiche les dernières valeurs reçues de tous les sensors associés à l'utilisateur courant. L'affichage de ces valeurs se fait avec un tableau écrit en JavaScript. Ce tableau teste quel est le type de valeur afin d'afficher correctement les unités. Ce test est effectué selon l'ordre des types présents dans la liste du fichier *Config.java*. Il aurait été plus juste d'effectuer un test directement avec les valeurs du fichier *Config.java* mais cela ne fonctionnait pas.
mobile.jsp	Page qui transmet les informations concernant les dernières valeurs des sensors associés à un utilisateur connecté via un appareil mobile. Cette page récupère les données provenant de la Servlet du fichier *Mobile.java* et crée une donnée JSON comprenant toutes les informations. Le format du JSON créé est expliqué dans le chapitre *9.8.4 Communication avec le serveur.*
mobileSelectedSensor.jsp	Récupère les informations de la Servlet du fichier *Mobile.java*. Ces informations représentent les données d'un sensor spécifique. Elles sont mises en page de manière à former un string format JSON. Le format du JSON créé est expliqué dans le chapitre *9.8.4 Communication avec le serveur.*
modifyAppUser.jsp	Met un formulaire à disposition afin que l'utilisateur puisse mettre à jour les données sur son compte *AppUser.*
modifyNote.jsp	Met à disposition une liste de tous les sensors associés à l'utilisateur et un champ permettant d'entrer un nouveau nom à un sensor.
myAccount.jsp	Affiche un menu comprenant plusieurs options concernant le compte de l'utilisateur.
mySensor.jsp	Propose de sélectionner un sensor et un type de données à afficher. Si l'utilisateur sélectionne des valeurs dans les champs proposés, la page affiche un script JavaScript avec les informations demandées. Ces informations sont affichées grâce au fichier *tableChart.jsp.*
newUserSensor.jsp	Formulaire demandant d'entrer le *sensorID*, *sensorKey* et le nom d'un nouveau sensor avec lequel l'utilisateur souhaite s'associer.
tableChart.jsp	Ce fichier contient un script JavaScript permettant d'afficher un tableau de valeur et un graphique. Ce fichier est utilisé par *mySensor.jsp* pour afficher les informations demandées par l'utilisateur. Les valeurs reçues sont testées pour déterminer leurs types afin d'afficher correctement les unités. Ce test est effectué selon l'ordre des types présents dans la liste du fichier *Config.java*. Il aurait été plus juste d'effectuer un test directement avec les valeurs du fichier *Config.java* mais cela ne fonctionnait pas. Ce script propose un tableau avec une pagination réglable. Il est donc possible d'afficher plus ou moins de mesure sur une même page.

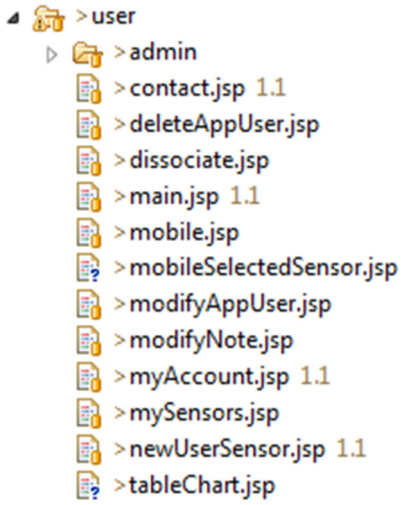


Figure 98 Répertoire war/WEB-INF/user

Le dossier *admin* contient les pages JSP qui sont accessibles uniquement par les utilisateurs enregistrés comme administrateur de l'application. Le répertoire *admin* est un sous-répertoire de *user*, ce qui permet de s'assurer que tous les administrateurs sont enregistrés en tant qu'utilisateur de l'application. Ce répertoire contient le fichier *addSensorData.jsp* qui permet d'ajouter des données à un sensor et le fichier *newSensor.jsp* qui permet d'ajouter des sensors à l'application. Ces deux fichiers sont donc des simples formulaires à remplir par l'administrateur.

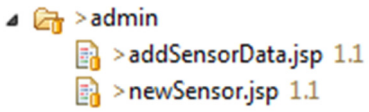


Figure 99 Répertoire war/WEB-INF/user/admin

Le répertoire *lib* contient toutes les librairies utilisées par Eclipse lors de la construction des fichiers exécutables. C'est dans ce dossier qu'il est nécessaire de rajouter les librairies au format JAR si l'application utilise des librairies spécifiques.

Le dernier répertoire présent dans *WEB-INF* est le répertoire *appengine-generated* qui contient le fichier d'index et le datastore pour le serveur de développement.

7.10.2 Formulaires

A plusieurs endroits dans l'application créée, l'utilisateur est confronté à un formulaire. Ceux-ci sont utiles pour la création de nouveaux objets Java appartenant au modèle. Nous pouvons notamment en trouver lorsque l'utilisateur souhaite s'inscrire à l'application ou s'associer à un sensor.

Les formulaires remplis par les utilisateurs sont créés par la vue grâce aux balises HTML standard. Pour la mise en forme des formulaires, les champs de chaque valeur sont contenus dans un tableau. Ceci permet de créer un visuel propre et simple.

Lorsque l'utilisateur soumet les données qu'il vient d'entrer, celles-ci sont transmises au serveur grâce à une requête POST. Le serveur reçoit alors la requête et la redirige sur la méthode *doPost* de la Servlet sélectionnée par le tag *action* du formulaire et le fichier *web.xml.* Cette méthode n'a pas la charge de traiter les données entrées par l'utilisateur. Effectivement, le contrôleur doit uniquement rediriger les requêtes et réponses vers les éléments en charge de leur traitement.

Les Servlets des contrôleurs de chaque formulaire redirigent donc les requêtes contenant les saisies des utilisateurs vers des classes appartenant au modèle qui s'occuperont de la gestion des données. Ces dernières effectuent des contrôles sur les paramètres reçus. Si les données sont correctes, le modèle crée un nouvel objet Java qui sera retourné au contrôleur pour la poursuite du traitement. Si les valeurs saisies par l'utilisateur ne permettent pas de créer un nouvel objet, la méthode gérant les saisies retournera *null*. Chacune des classes de contrôle des données du modèle possède des champs statiques de résultat et d'erreurs. Ces champs peuvent être utilisés par le contrôleur afin de transmettre des messages à la vue en cas d'erreur ou de succès.

En cas d'erreur, l'objet Java du modèle ne peut pas être créé. En plus des messages d'erreurs transmis à la vue, les saisies entrées par l'utilisateur sont réécrites dans les champs du formulaire. Pour ce faire, le contrôleur doit ajouter un attribut à la requête par champs du formulaire. La page JSP gérant la vue doit alors prendre en charge le fait qu'une valeur par défaut peut être donnée en argument.

La Figure 100 illustre le code standard d'un fichier JSP pour un champ de formulaire. On y retrouve la valeur par défaut qui peut être passée en argument ainsi que le message d'erreur associé au champ.

```
<input type="text" id="sensorID" name="sensorID"
       value="<c:out value="${requestScope.sensorID}"/>" size="20" maxlength="20" />
<span class="error">${errors['sensorID']}</span>
```

Figure 100 Champ d'un formulaire, fichier JSP

7.10.3 JavaScript

Les tableaux et graphiques de l'application créée sont des éléments JavaScript. Grâce à ces scripts, l'affichage et la mise à jour des valeurs est plus agréable pour l'utilisateur. Contrairement au HTML et CSS qui sont des langages interprétés par le navigateur, le JavaScript est exécuté sur la machine de l'utilisateur. Ceci peut poser des problèmes d'affichage si l'utilisateur n'est pas capable d'afficher le JavaScript. Ce dernier cas peut aussi se produire sur les navigateurs possédant certain filtre anti-publicité tel que AdBlock pour Google Chrome.

Les tableaux et graphiques sont créés via Google Chart qui nous fournit une API efficace afin de créer rapidement des graphiques esthétiques. Les données passées à la fonction dessinant le graphique sont en format JSON. Il est aussi possible d'ajouter des colonnes ou des lignes manuellement avec des fonctions dédiées. Le format JSON permet de représenter de manière textuelle des données génériques. Il s'agit d'un ensemble de paires nom/valeur. La Figure 101 illustre la syntaxe à respecter pour l'écriture de données en format JSON.

```
{
    "menu": {
        "id": "file",
        "value": "File",
        "popup": {
            "menuitem": [
                { "value": "New", "onclick": "CreateNewDoc()" },
                { "value": "Open", "onclick": "OpenDoc()" },
                { "value": "Close", "onclick": "CloseDoc()" }
            ]
        }
    }
}
```

Figure 101 Exemple du format JSON

Lors de la création des tableaux et graphiques, la Servlet du contrôleur fournit les valeurs à afficher en tant que liste. C'est donc au code JavaScript lui-même de créer un string en format JSON afin de pouvoir dessiner le tableau ou graphique. Lors du rafraichissement automatique de la page *My Sensor,* le contrôleur donne directement les nouvelles données au format JSON. En fournissant les premières données en tant que liste, il est possible de les traiter dans les pages JSP pour d'autres affichages que les tableaux et graphiques. Les données de rafraichissement ne sont utilisées que par les scripts JavaScript, il est donc plus simple de transmettre les données directement en format JSON.

Afin de simplifier l'implémentation du rafraichissement automatique et de certaines fonctions, le code JavaScript utilise jQuery. Cette bibliothèque nous met à disposition certaines fonctions qui nous sont très utiles pour la gestion des requêtes. La Figure 102 montre le code mis en place pour qu'une page affichée sur le client demande les nouvelles valeurs au serveur. Le paramètre *refreshMode* permet à la Servlet du contrôleur de définir si la requête courante concerne le rafraichissement ou le premier chargement de la page. Le paramètre *lastTimestamp* permet à la Servlet du contrôleur de savoir quel est le dernier élément en date connu par le client. La Servlet pourra ainsi retourner que les nouvelles valeurs à ajouter.

```
// Request for new data
$.post("/user/mySensors",
    {refreshMode    : "true",
     lastTimestamp  : data.getValue(0,0).getTime(), // Timestamp of the last data
     selectedSensor : "${selectedSensor.sensorID}",
     selectedType   : "${selectedType}"},
    refreshTable,"json");
```

Figure 102 Requête pour les nouvelles valeurs

7.10.4 Système de log

Lorsque l'application génère une erreur, celle-ci est enregistrée dans un système de log grâce à *java.util.logging.Logger*. Les logs sont créés lorsque l'application lève une exception et que celle-ci est gérée par le code. Les événements loggés sont visibles dans la console d'administration ou via un fichier script shell fourni par Google.

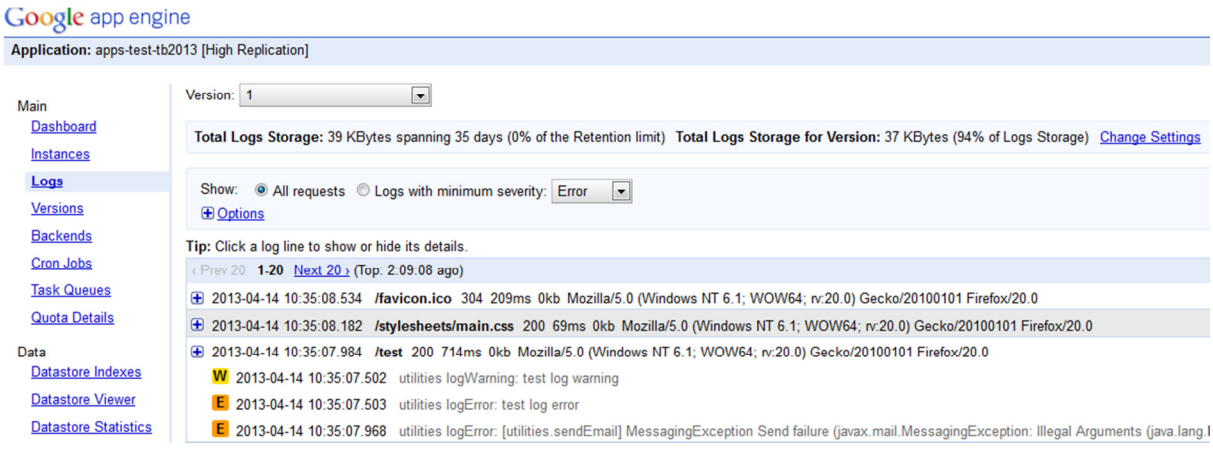


Figure 103 Affichage des logs dans la console d'administration

Lorsque l'application écrit sur la sortie standard *System.out* ou sur la sortie d'erreur, *System.err*, App Engine enregistre automatiquement ces événements dans les logs. Le niveau de log pour la sortie standard est *INFO* alors qu'il est de *WARNING* pour la sortie erreur. L'utilisation de *java.util.logging* permet néanmoins de spécifier plus de niveaux de log.

Pour activer la gestion des logs dans App Engine, nous devons ajouter une propriété système dans notre fichier *appengine-web.xml*. Le nom de cette propriété, *java.util.logging.config.file*, doit être associé avec un fichier de configuration spécial pour les logs. Ce fichier, *logging.properties*, permet de spécifier à partir de quel niveau d'importance les logs doivent être consignés. Il est alors possible de créer des logs utilisés pour le développement et de ne pas les afficher pour l'utilisation finale en modifiant uniquement ce fichier de configuration.

```
<!-- Configure java.util.logging -->
<system-properties>
  <property name="java.util.logging.config.file" value="WEB-INF/logging.properties"/>
</system-properties>
```

Figure 104 Ajout du paramètre système pour les logs dans appengine-web.xml

```
# Set the default logging level for all loggers to WARNING
.level = WARNING
```

Figure 105 Niveau de consignation des logs dans logging.properties

Une classe Java, *Log,* a été créée afin de simplifier l'utilisation des logs de l'application. Cette classe met à disposition des méthodes permettant d'écrire des messages dans le système de log avec différent niveau d'importance.

7.10.5 Gestion du temps

Dans l'application, tous les temps sauvegardés dans le datastore sont au format UNIX. Le format UNIX permet de représenter une date avec un nombre entier, le timestamp. Cette valeur représente le nombre de secondes écoulées depuis le 1er janvier 1970 00:00:00 UTC. La sauvegarde de nombre entier est beaucoup plus simple et portable que la sauvegarde d'objet de type Date ou String.

Pour afficher une date, il est alors nécessaire de convertir le timestamp en String. Cette conversion est effectuée grâce à un objet *SimpleDateFormat* initialisé avec le format souhaité par l'utilisateur. Cette opération est représentée à la Figure 106.

```
SimpleDateFormat sdf = new SimpleDateFormat(appUser.getDatePattern());
for(SensorData sd: newDataList){
    dateList.add(sdf.format(sd.getDate()*1000));
}
```

Figure 106 Conversion d'un timestamp en string

La Figure 106 illustre aussi un problème simple mais qui peut faire perdre beaucoup de temps de développement. Les timestamps UNIX représente le temps en seconde alors que tout le système Java représente le temps en milliseconde. Il est donc nécessaire de multiplier le timestamp UNIX par mille avant de pouvoir l'utiliser dans des fonctions Java.

Pour que les dates soient affichées en heure locale pour tous les utilisateurs à travers le monde, il est nécessaire de gérer les différents décalages horaires. Toutes les dates sont enregistrées dans le datastore au temps UTC. Le programme doit ensuite ajouter la différence de temps dû au décalage horaire selon la position de l'utilisateur. Cette fonctionnalité n'est pas encore implémentée. Tous les temps fournis par le serveur sont alors les dates UTC.

7.10.6 Rendu graphique

Les figures suivantes représentent l'application créée tel que la verront les utilisateurs. En plus de ces figures, les tableaux indiquent le rôle de chaque page, le fichier de Servlet utilisé, le fichier JSP utilisé ainsi que les droits d'accès à la page. Les fichiers JSP indiqués ne sont que ceux utilisés pour le contenu de la page. L'en-tête, menu et les autres parties de la page sont importés automatiquement dans le fichier *template.jsp*.

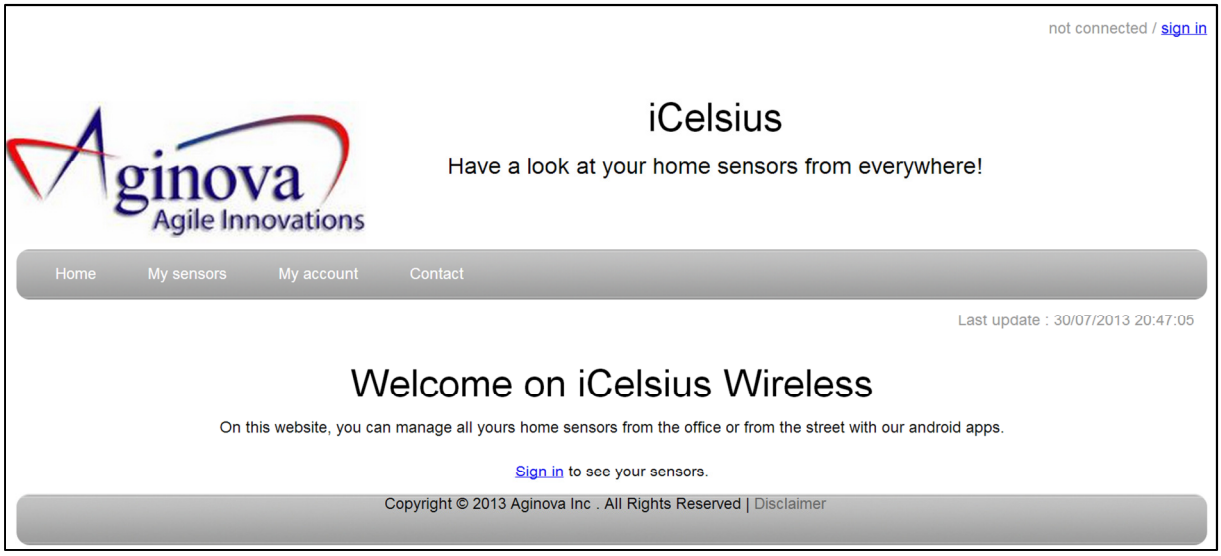


Figure 107 Visualisation de la page d'accueil pour les utilisateurs non connectés

Rôle	Page d'accueil pour les utilisateurs non connectés à leur compte Google.
Servlet	applicationGeneralServlet.Index.java
Fichier JSP	index.jsp
Droit d'accès	Tout le monde

Tableau 2 Information sur la page d'accueil pour les utilisateurs non connectés

Figure 108 Visualisation de la page pour les utilisateurs connectés

Rôle	Page d'accueil pour les utilisateurs connectés à leur compte Google mais ne possédant pas de compte sur l'application.
Servlet	applicationGeneralServlet.AddAppUser_controller.java
Fichier JSP	connect/newAppUser.jsp
Droit d'accès	Utilisateurs connectés à leur compte Google
Remarques	Les champs de saisie sont automatiquement remplis selon les informations du compte Google de l'utilisateur connecté. Les valeurs des champs *Date pattern* et *Température unit* sont des champs par défaut définis dans le fichier *Config.java*.

Tableau 3 Information sur la page pour les utilisateurs connectés

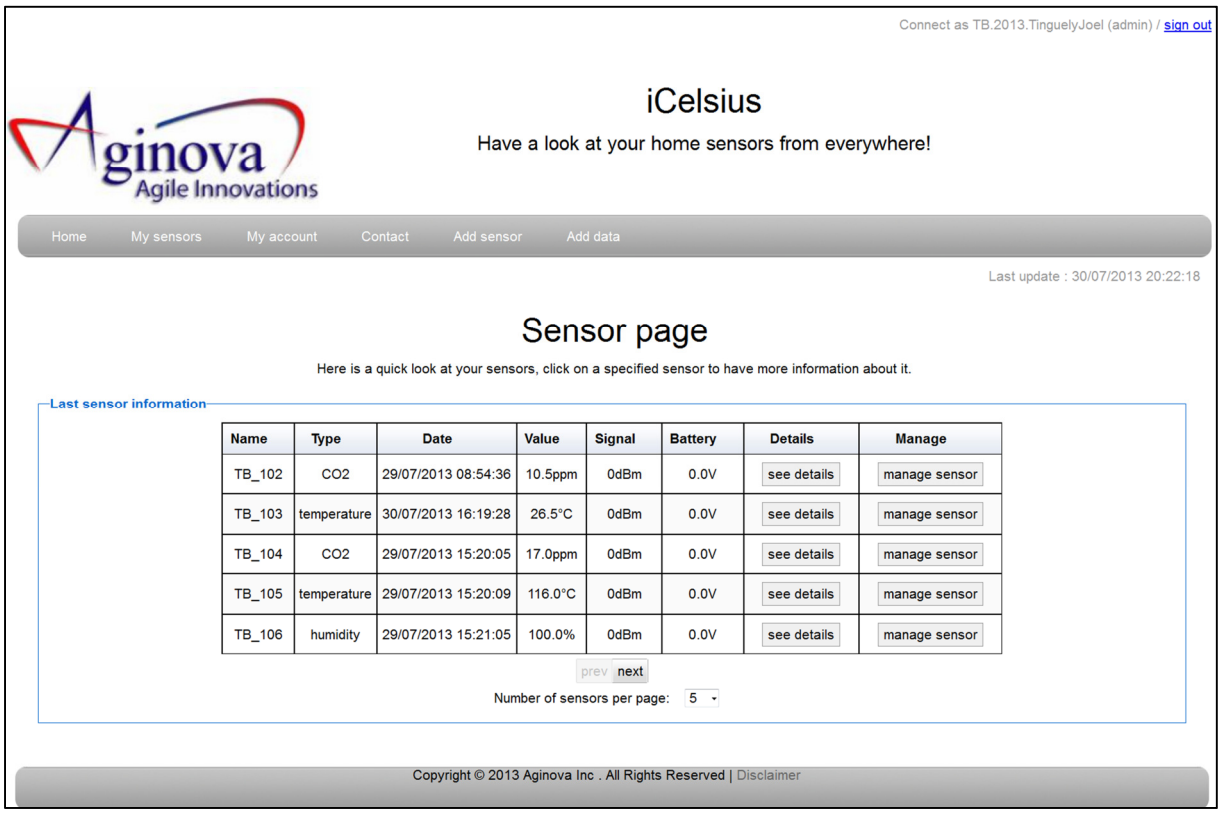


Figure 109 Visualisation de la page Home

Rôle	Page d'accueil pour les utilisateurs connectés et reconnus par l'application.
Servlet	applicationGeneralServlet.Main.java
Fichier JSP	user/main.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 4 Information sur la page Home

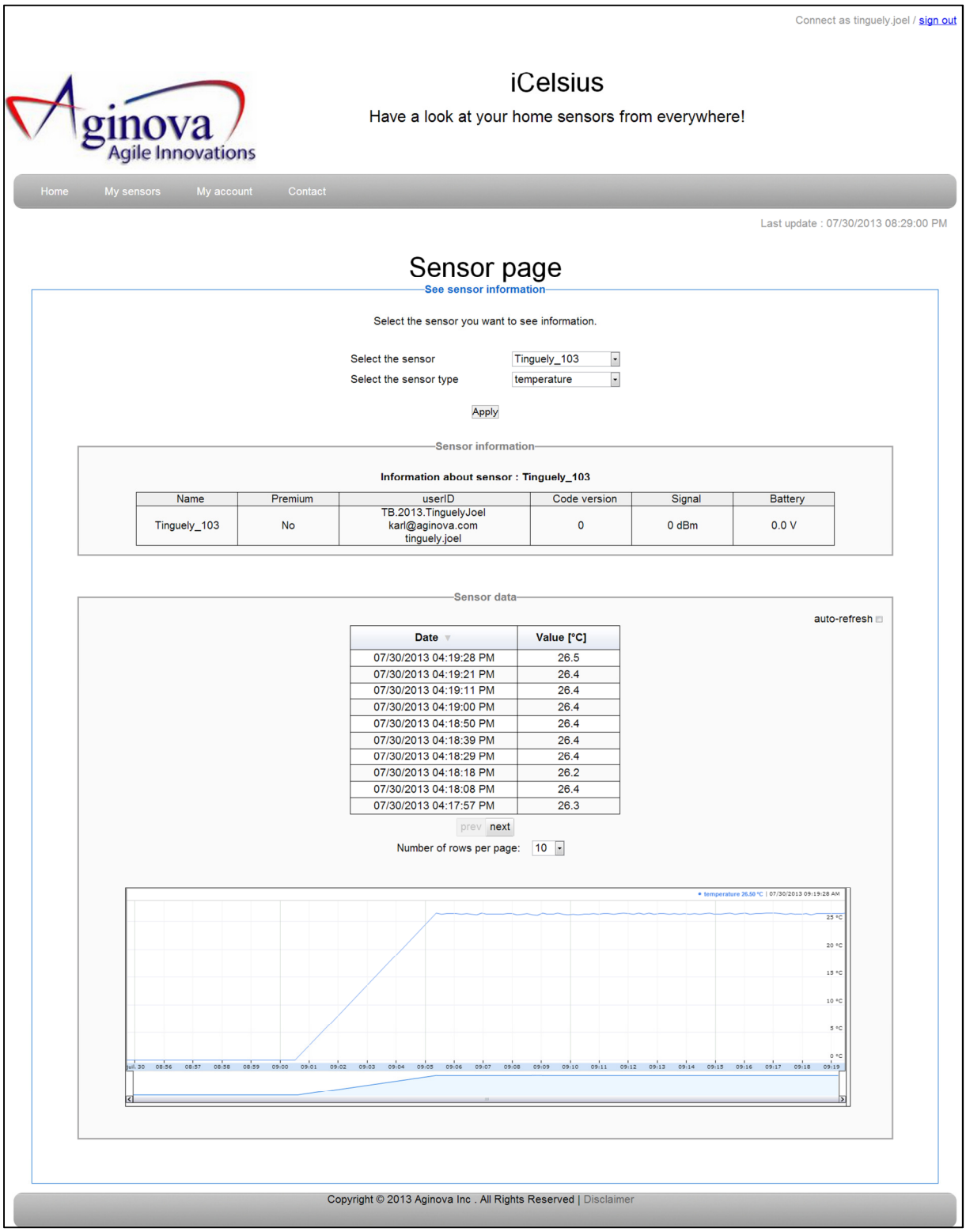


Figure 110 Visualisation de la page My Sensors

Rôle	Page *My Sensors* pour observer les données d'un sensor.
Servlet	applicationGeneralServlet.MySensors.java
Fichier JSP	user/mySensors.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 5 Information sur la page My Sensors

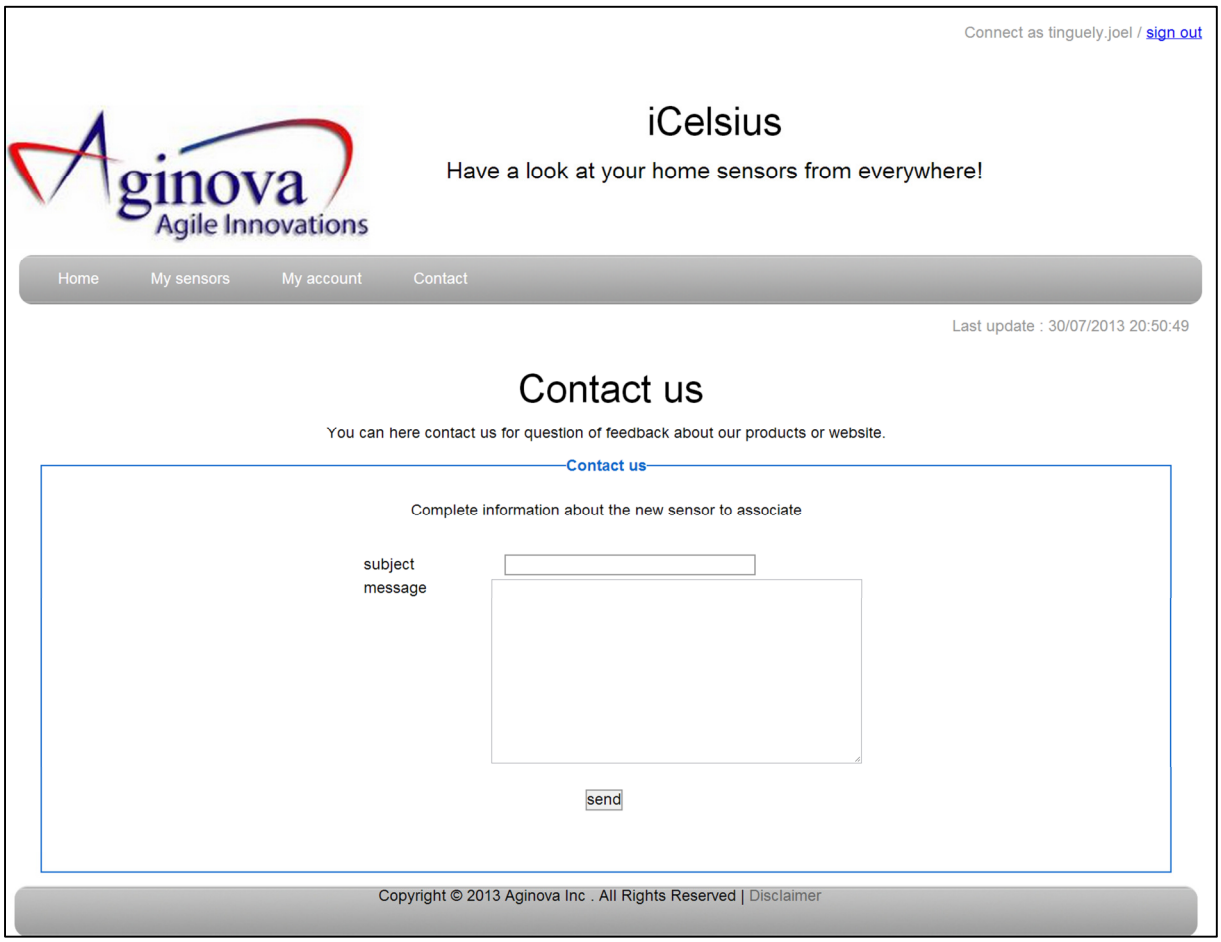


Figure 111 Visualisation de la page Contact

Rôle	Page *Contact* pour que l'utilisateur puisse envoyer des messages à un administrateur de l'application.
Servlet	applicationGeneralServlet.Contact.java
Fichier JSP	user/contact.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 6 Information sur la page Contact

Figure 112 Visualisation de la page My Account

Rôle	Page *My Account* permettant à l'utilisateur de gérer son compte *AppUser*.
Servlet	applicationGeneralServlet.myAccount.java
Fichier JSP	user/myAccount.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 7 Information sur la page My Account

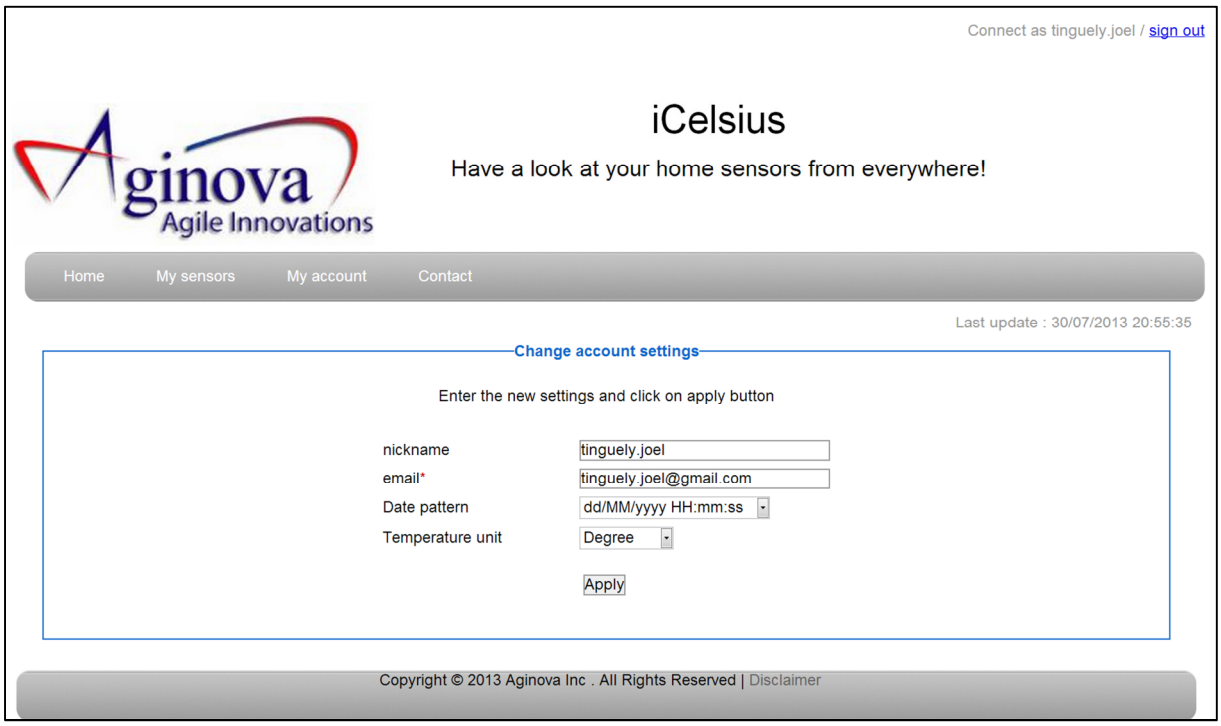


Figure 113 Visualisation de la page de modification des paramètres du compte

Rôle	Page permettant à l'utilisateur de modifier les informations sur son compte.
Servlet	appUser.ModifyAppUser_controller.java
Fichier JSP	user/modifyAppUser.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 8 Information sur la page des paramètres de compte

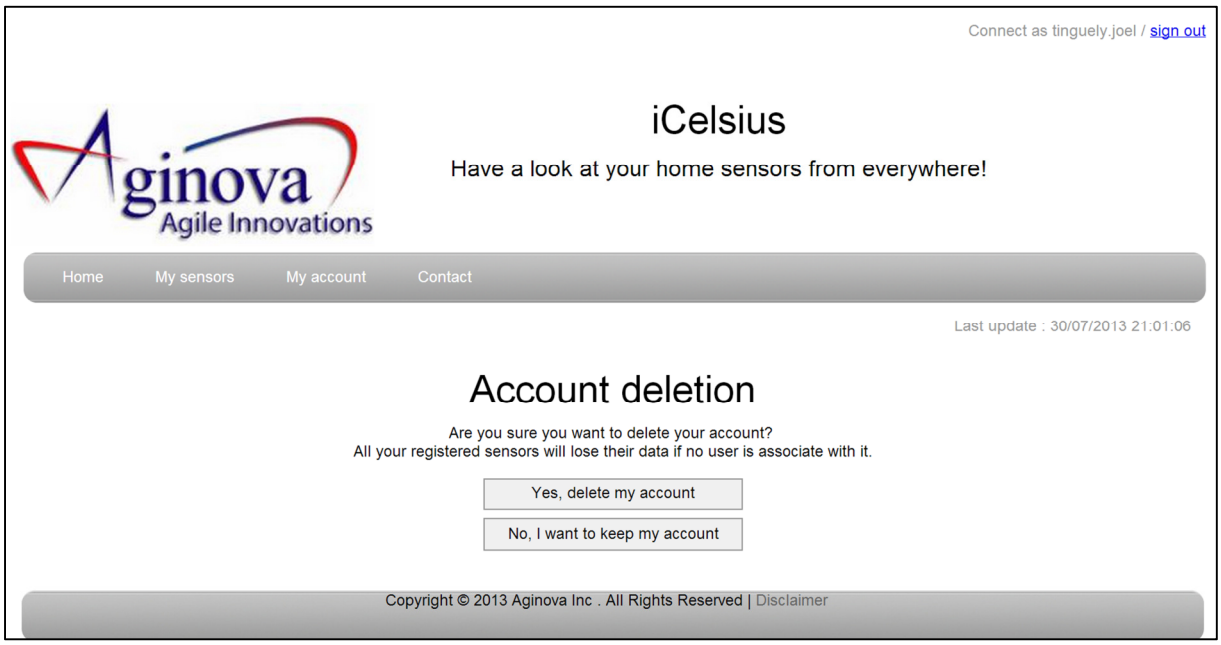


Figure 114 Visualisation de la page de suppression du compte

Rôle	Page permettant à l'utilisateur de supprimer son compte *AppUser*.
Servlet	appUser.DeleteAppUser_controller.java
Fichier JSP	user/deleteAppUser.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application
Remarques	L'objet *AppUser* est supprimé. Tous les objets *UserSensor* possédant une référence sur l'utilisateur sont mis à jour. Si l'utilisateur est le seul *AppUser* associé à un sensor, l'objet *UserSensor* ainsi que toutes les données du sensors sont supprimés.

Tableau 9 Information sur la page de suppression de compte

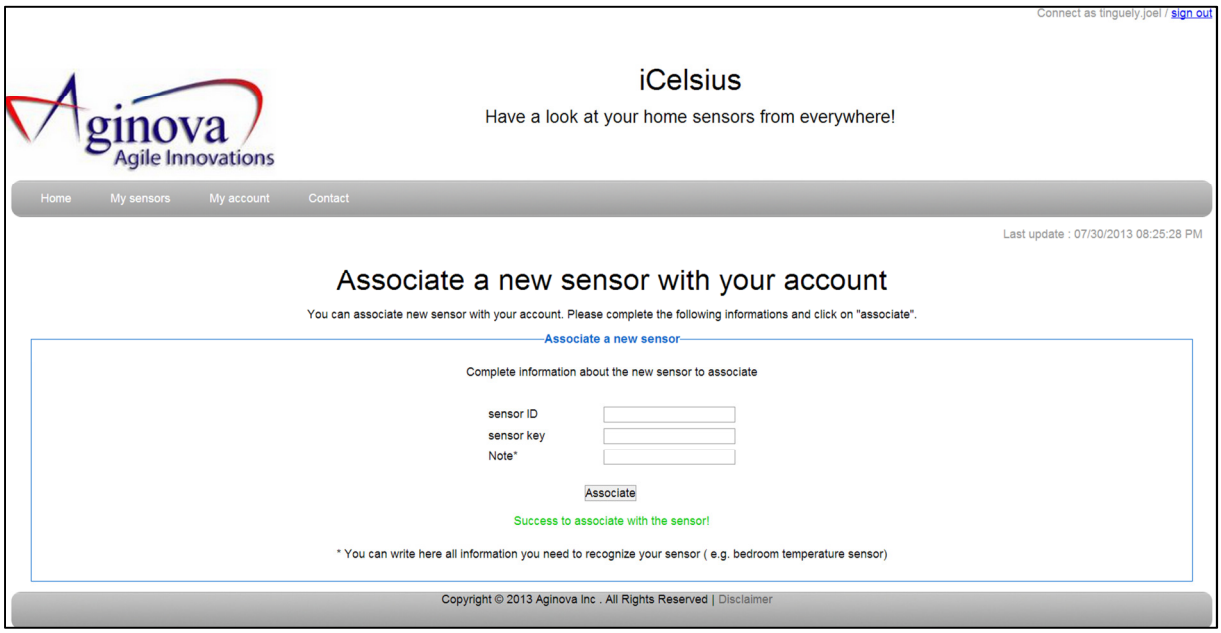


Figure 115 Visualisation de la page pour l'association d'un sensor

Rôle	Page permettant à l'utilisateur de s'associer à un nouveau sensor pour voir les informations provenant de celui-ci.
Servlet	userSensor.AddUserSensor_controller.java
Fichier JSP	user/newUserSensor.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application
Remarques	Le sensor ID et la clé sont deux éléments présents physiquement sur le sensor. Ces éléments permettent de s'assurer que l'utilisateur possède bien le sensor au moment de son association avec celui-ci. Le message en vert indique que l'utilisateur est correctement associé au sensor dont il vient d'entrer les informations.

Tableau 10 Information sur la page permettant l'association d'un sensor

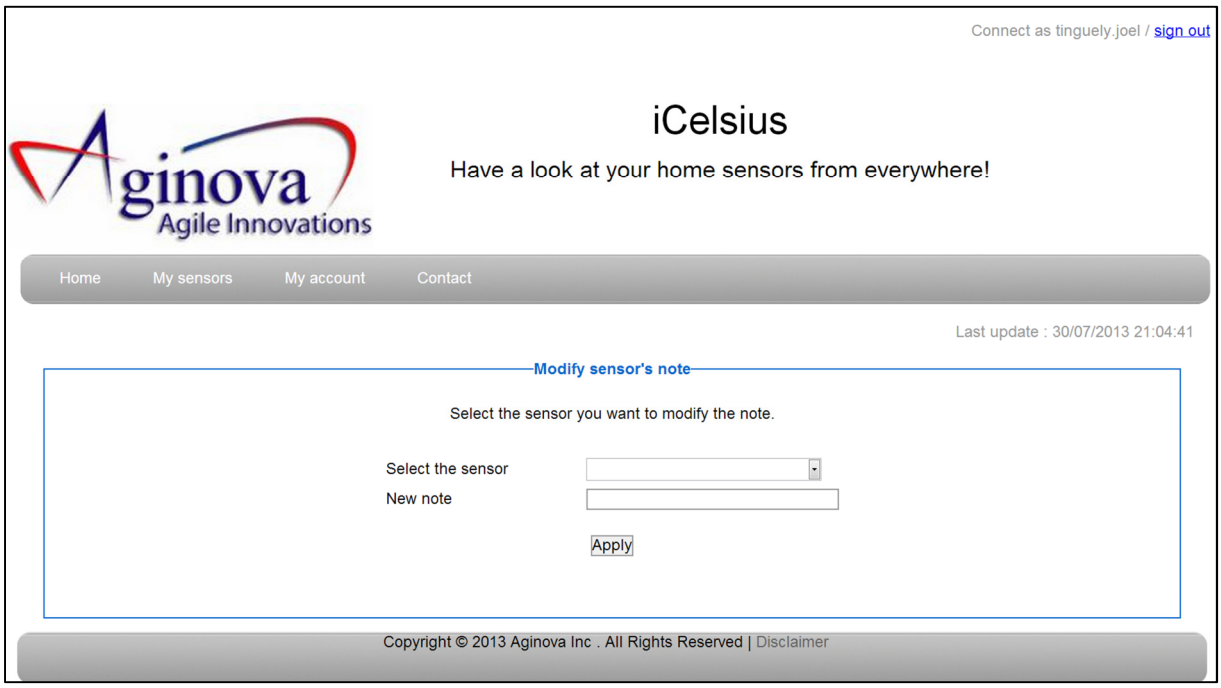


Figure 116 Visualisation de la page de modification du nom du sensor

Rôle	Page permettant à l'utilisateur de modifier le nom donné à un sensor.
Servlet	appUser.ModifyNote_controller.java
Fichier JSP	user/modifyNote.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application

Tableau 11 Information sur la page permettant de modifier le nom d'un sensor

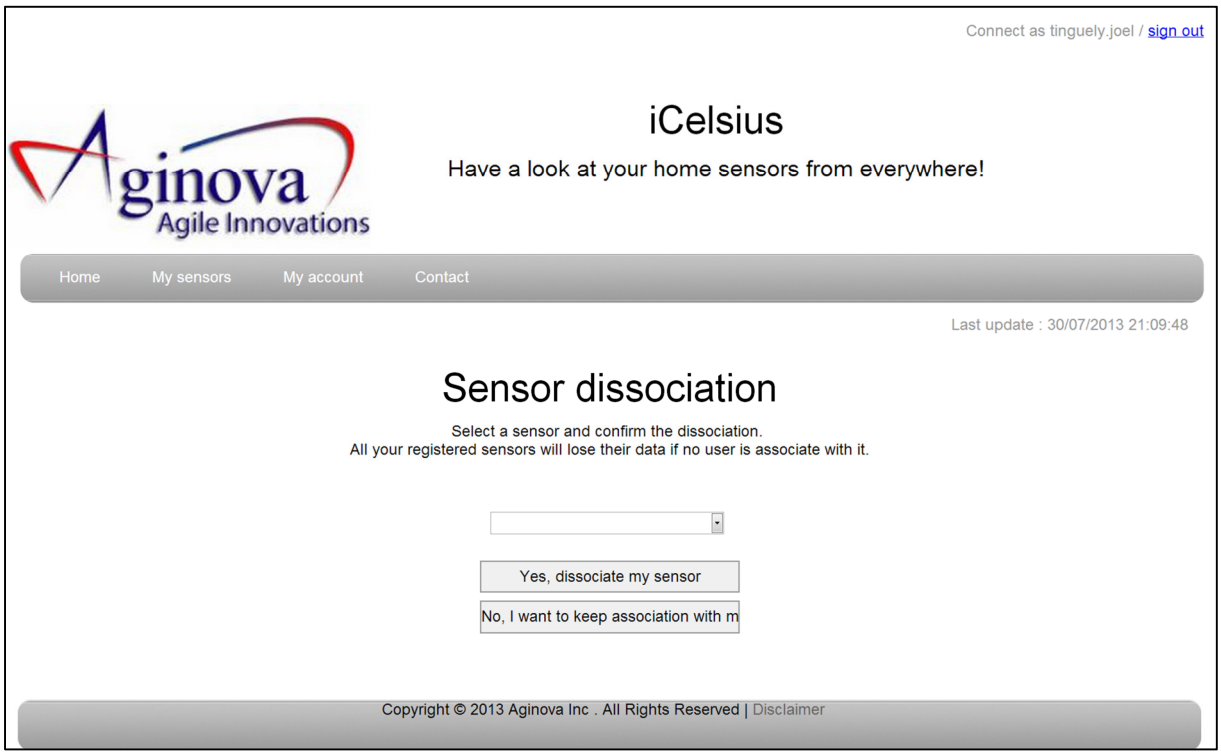


Figure 117 Visualisation de la page de dissociation d'un sensor

Rôle	Page permettant à l'utilisateur de supprimer un de ses sensors.
Servlet	appUser.Dissociate_controller.java
Fichier JSP	user/dissociate.jsp
Droit d'accès	Utilisateurs connectés et inscrits dans l'application
Remarque	Si l'utilisateur est le seul *AppUser* associé à un sensor, l'objet *UserSensor* ainsi que toutes les données du sensors sont supprimés.

Tableau 12 Information sur la page de suppression de sensor

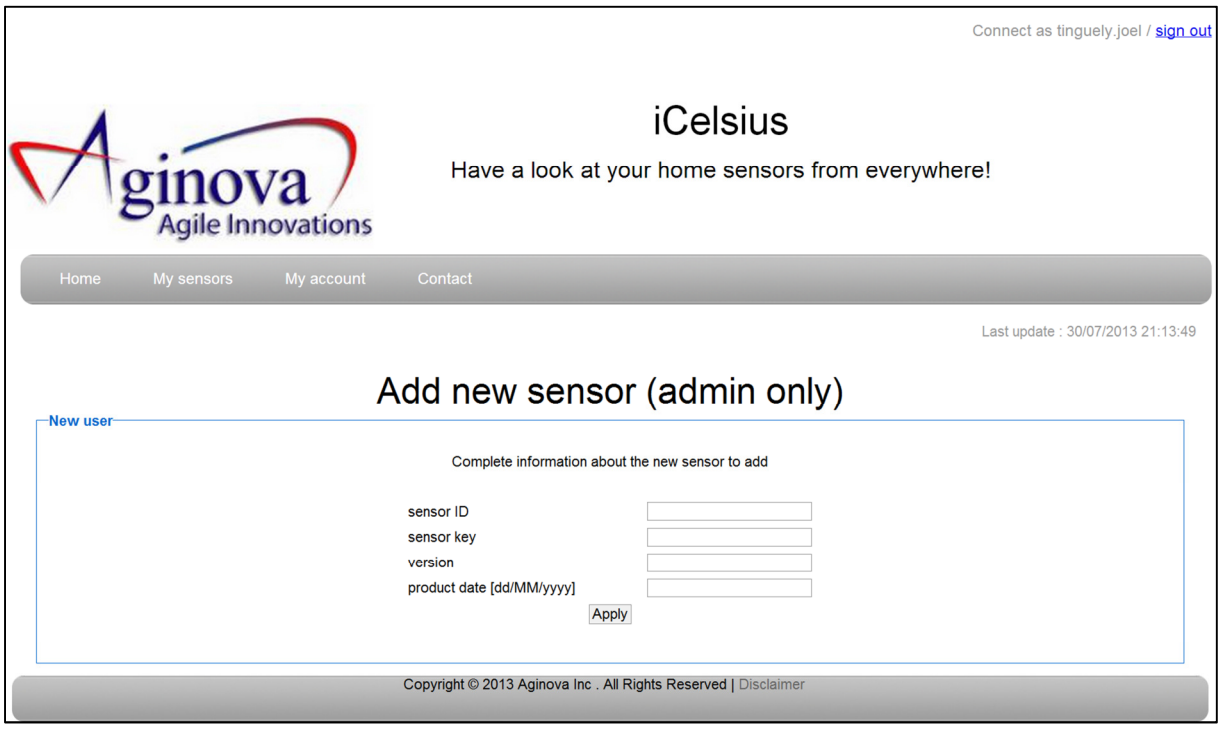


Figure 118 Visualisation de la page d'ajout de sensor

Rôle	Page permettant à un administrateur d'ajouter un sensor à l'application.
Servlet	appUser.AddSensor_controller.java
Fichier JSP	user/admin/newSensor.jsp
Droit d'accès	Administrateurs connectés et inscrits dans l'application

Tableau 13 Information sur la page d'ajout de sensor

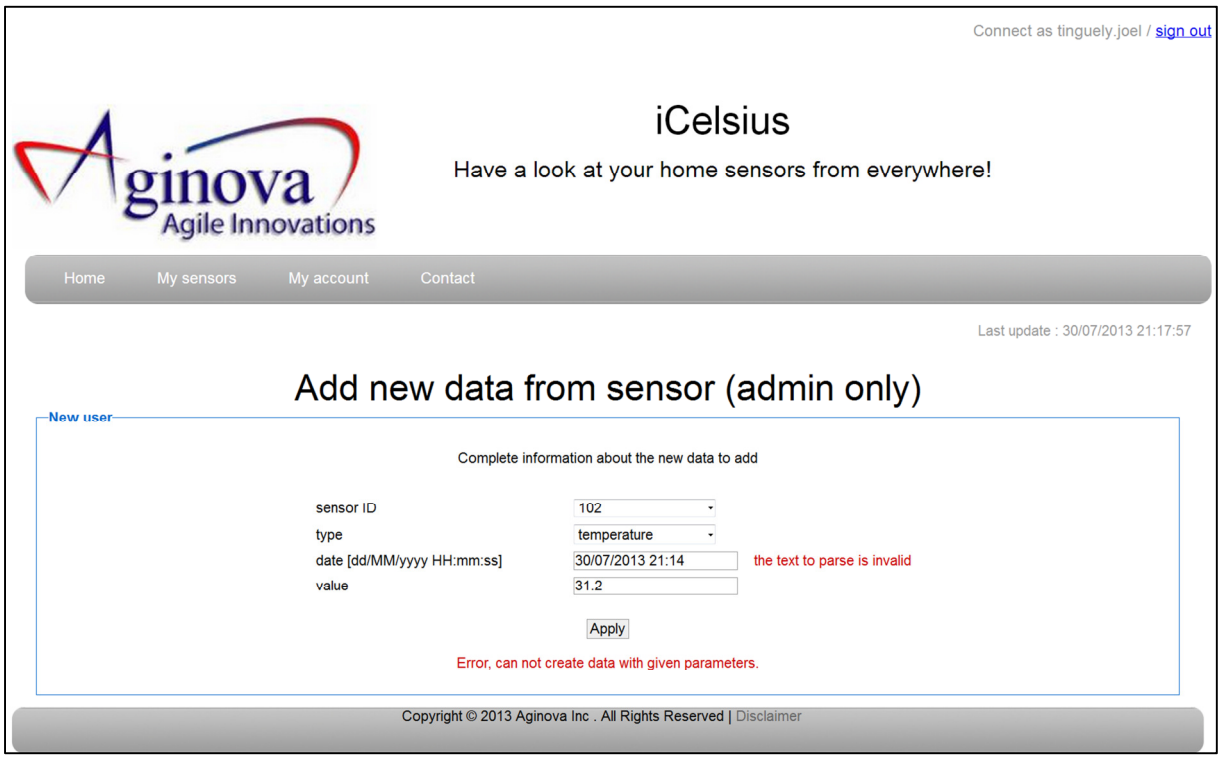


Figure 119 Visualisation de la page d'ajout d'une donnée

Rôle	Page permettant à un administrateur d'ajouter une donnée pour un sensor de l'application.
Servlet	appUser.AddSensorData_controller.java
Fichier JSP	user/admin/addSensorData.jsp
Droit d'accès	Administrateurs connectés et inscrits dans l'application
Remarque	Les messages en rouge montrent les messages d'erreurs qui sont affichés en cas d'erreur de saisie. Dans cet exemple, la date est incorrecte car il manque les secondes.

Tableau 14 Information sur la page d'ajout de donnée

8 Communication avec le sensor

8.1 Installation de l'environnement de développement

8.1.1 Installation du driver pour l'adaptateur JTAG

Par défaut, Windows ne trouve pas de pilote pour notre adaptateur JTAG, *Rainsonnance RLINKUSB Dongle*. Il faut donc installer le driver manuellement. Le driver à installer est fourni dans le fichier *RLinkDvr.zip*.

8.1.2 Installation d'Eclipse

Il est ensuite nécessaire d'installer une version d'Eclipse spécialement conçue pour le développement C et C++. Nous avons choisi d'installer Eclipse Juno CDT version 8.1.2[6]. L'installation de l'environnement fonctionnant avec Eclipse est expliquée dans le document *RZEMFISDK-UG-101812.pdf*, page 20.

Modifier le comportement du build : clic droit sur le projet -> *Properties* -> *C/C++ build* -> *builder settings*. Décocher *Use default build command* et ajouter *C:\dev\aginova\rezolt\Dec_2012\tools\mingw\bin\make* dans *Build Command.* La Figure 120 illustre les modifications à apporter à la configuration du build.

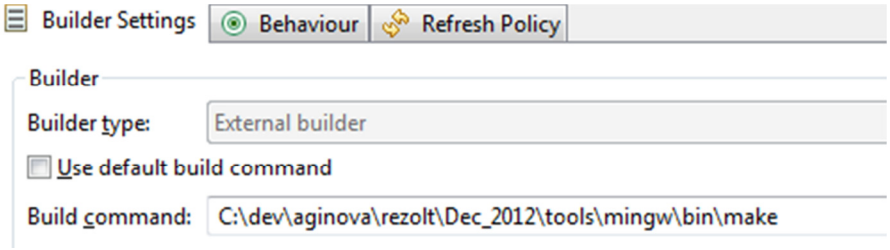


Figure 120 Modification de la commande de build

Afficher la fenêtre *Make target* : *Window -> show view -> make target.* Créer un nouveau target pour le projet. Dans l'onglet *make target* : Clic droit sur le projet -> *new*. Modifier le target name avec la valeur suivante : *sensor-iCelsius_Wireless-FreeRTOS-LwIP-Wiced-MW-Bootloader* puis cliquer sur *ok.*

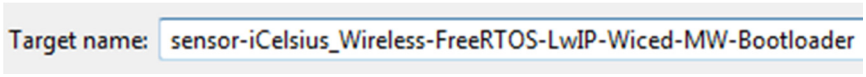


Figure 121 Nom du target

L'environnement de développement est maintenant prêt à être utilisé. Pour compiler un programme pour le bon target, il est nécessaire d'ouvrir la fenêtre *Make Target* et de double cliquer sur le target que nous avons créé. Sur la Figure 122, le double-clic doit donc se faire sur l'élément possédant l'icône verte.

[6] http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/junosr2

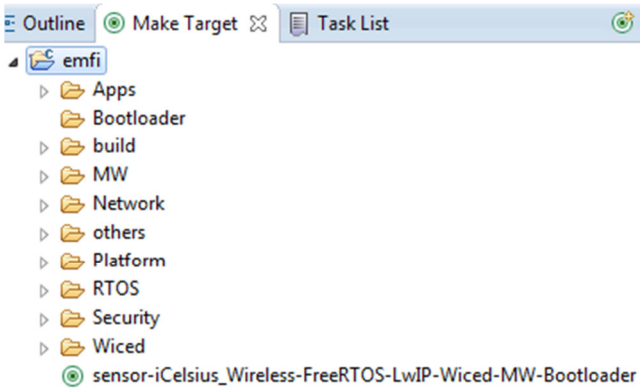


Figure 122 Fenêtre Make Target avec le target créé

Le transfert de l'application compilée sur la cible (flashage) se fait en appuyant sur  , puis sur *opencd-program-complete-release-iCelcius*. La Figure 123 illustre l'action à effectuer pour transférer le programme sur la cible. Lors du transfert, le jumper doit être connecté sur la carte.

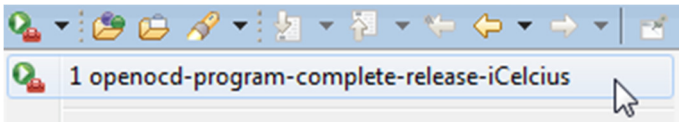


Figure 123 Transfère du programme du la cible

8.2 Modification du code

Le projet fourni comprenait toutes les fonctions nécessaires pour l'envoi des données. La fonction envoyant les données, *app_send_data_packets_over_http*, était alors présente dans le fichier *app.c*. La fonctionnalité ajoutée pour ce projet est l'envoi d'arguments pour transmettre les paramètres sur le serveur.

La Figure 124 illustre l'ajout de code pour l'envoi du *sensorID* depuis le sensor. Du code similaire a été créé pour l'envoi des autres paramètres. La variable *args* est un tableau de *Http_Form_Element*. Ce type est une structure regroupant un nom et une valeur pour les paramètres HTTP. Dans l'exemple ci-dessous, le nom est alors configuré à "*sensorID*". Le numéro unique du sensor, le *sensorID*, est ensuite récupéré de la variable *app*. Ce *sensorID* est une valeur unique définie pour chaque sensor qui est récupérable depuis le programme. Le *sensorID* récupéré est un integer sur 32 bits. Il est nécessaire de convertir cette valeur en chaine de caractères afin de pouvoir la transmettre via HTTP. Pour ce faire, la méthode *sprintf* est utilisée. La méthode *TRACE_INFO* permet d'écrire sur la sortie de debug lors de l'utilisation du terminal.

```
// SensorID
args[0].name = "sensorID";
sprintf(sensorID, "%u", app.sensorId);
args[0].value = sensorID;
TRACE_INFO("%i: %s=%s\n", 0, args[0].name, args[0].value);
```

Figure 124 Envoi du sensorID sur le sensor

Ces arguments sont ensuite envoyés au serveur via la méthode *http_client_post*. Cette fonction envoie une requête POST sur l'URL prédéfinie dans les paramètres. La méthode *app_send_data_packets_over_http* affiche ensuite la réponse reçue du serveur. Actuellement, la réponse est simplement affichée sur la sortie de debug. Par la suite, il est possible de traiter les éléments reçus du serveur à cet endroit.

Le fichier *config.c* permet la configuration du sensor. Grâce à ce fichier, il est notamment possible de définir le réseau wifi sur lequel le sensor se connecte. La Figure 125 est la configuration du SSID du sensor alors que la Figure 126 est la configuration sur canal wifi. Ces deux valeurs seront alors dépendantes de l'utilisateur et devront être modifiables. La Figure 127 est la configuration de l'URL sur laquelle le sensor doit envoyer ses données. Cette URL correspond à l'adresse du serveur suivi du chemin d'accès sur celui-ci. Pour envoyer les données sur le serveur en ligne, il est nécessaire de modifier cette valeur pour y mettre l'URL du serveur en ligne. Cette valeur ne doit pas être modifiable par l'utilisateur. Effectivement l'URL du serveur en ligne ne devrait pas subir de modification.

```
#define CONFIG_SSID_1_FAC_DFLT "TB_Tinguely"
```

Figure 125 Réglage du SSID, config.c

```
#define CONFIG_WEPKEY_ID_1_FAC_DFLT 1
```

Figure 126 Réglage du canal Wifi, config.c

```
#define CONFIG_APP_URL_FAC_DFLT "http://192.168.1.101:8888/sensor/newSensorData"
```

Figure 127 Réglage de l'URL pour le serveur local, config.c

8.3 Type de communication

8.3.1 Envoi de données

La communication entre les sensors et le serveur doit être définie afin que les informations puissent être transmisses correctement. Cette communication se fait via le protocole HTTP, en transmettant les valeurs via les paramètres de la requête. Les paramètres de requêtes seront des chaines de caractères. L'identification de ces paramètres dans la requête s'effectue aussi via des strings.

Le sensor peut envoyer deux types d'information au serveur. Le premier type est une mesure provenant de la probe du sensor. Le sensor capture une mesure via la probe et l'envoie en serveur. L'envoi de la nouvelle valeur se fait via une requête POST sur le serveur avec le chemin */sensor/newSensorData*. Le Tableau 15 fournit les paramètres nécessaires pour l'envoi d'une donnée provenant de la probe du sensor.

sensorID	Le numéro unique du sensor permettant de l'identifier univoquement sur le datastore.
sensorKey	La clé en claire du sensor. Cette clé permet de l'assurer que la donnée provient du sensor possédant le *sensorID* indiqué.
type	Le type de la donnée envoyée. Les types de données supportés par le serveur sont définis dans le fichier *Config.java* du package *utilities* de l'application web. Actuellement, les valeurs supportées pour le paramètre type sont les strings suivants: - *temperature* - *humidity* - *CO2* - *no data*
sensorValue	La valeur de la mesure. L'unité de cette valeur sera automatiquement ajoutée selon le *type* défini. Bien que cette valeur soit d'un type numérique, il est nécessaire de la convertir en string pour pouvoir la passer en argument.
timestamp	Le timestamp du sensor lors de la capture de la valeur. Ce timestamp doit être au format UNIX et représenter l'heure UTC.

Tableau 15 Paramètres pour l'envoi d'une donnée provenant de la probe

Le deuxième type est l'envoi de données concernant le sensor lui-même. Ces données ne proviennent pas de la probe mais des informations sur l'état du sensor. L'envoi de ces valeurs se fait via une requête POST sur le serveur avec le chemin */sensor/newSensorData*. Le Tableau 16 fournit les paramètres nécessaires pour l'envoi de données provenant du sensor lui-même.

sensorID	Le numéro unique du sensor permettant de l'identifier univoquement sur le datastore.
sensorKey	La clé en claire du sensor. Cette clé permet de l'assurer que la donnée provient du sensor possédant le *sensorID* indiqué.
battery	Le niveau de la batterie du sensor. Ce niveau est donné en tension, l'unité est le Volt.
RSSI	La puissance de réception du signal reçu par l'antenne wifi. Cette valeur est fournie en dBm.
codeVersion	La version du software s'exécutant sur le sensor.

Tableau 16 Paramètre pour l'envoi d'une donnée concernant le sensor

Il est possible d'effectuer un envoi d'un de ces types de données ou des deux types dans la même requête. Les paramètres pourront alors être interprétés correctement par le serveur.

8.3.2 Réception de données

Lors de l'envoi d'une donnée via une requête POST, le serveur renvoie une réponse au sensor. Cette réponse peut contenir un message d'erreur ou de réussite ainsi que certaines informations. Pour le moment, seule une donnée de timestamp est présente si le serveur détecte un problème avec le timestamp de la donnée reçue. Cette valeur est présente sous la forme "*timestamp=<new_value>*". La réponse reçue sur le sensor peut alors être parsée pour retrouver le champ *timestamp*. Ce champ n'est pas présent lorsque le serveur ne détecte pas de problème concernant le timestamp de la donnée reçue.

Il est ensuite possible d'implémenter de la même manière la transmission d'autres paramètres par cette méthode.

La deuxième solution pour la réception de données sur le sensor est l'envoi d'une requête GET. Cette méthode n'est actuellement pas implémentée dans la communication entre le sensor et le serveur. Si le sensor effectue une requête GET sur le serveur, il serait alors possible de transmettre des informations au sensor via la réponse provenant du serveur.

9 Conception application Android

9.1 Introduction

Android est un système d'exploitation mobile ayant subi un grand essor depuis l'apparition des smartphones. Le nombre d'utilisateurs de cet OS (Operating System) dépasse même le iOS d'Apple qui était le précurseur dans le domaine. Android se retrouve dans de plus en plus d'appareils qui dépassent largement le cadre des smartphones. On peut notamment retrouver cet OS dans les Google TV et même dans certaines voitures.

Android est sous un contrat de licence qui respecte les principes de l'open source. Il est alors possible de récupérer les sources d'Android et de les modifier sans problème de licence. De plus, l'acquisition et l'utilisation de ces sources sont totalement gratuites. Le seul élément payant de ce système d'exploitation est l'inscription au Play Store afin d'y mettre des applications à disposition.

De nombreuses d'applications sont disponibles pour les appareils fonctionnant avec Android. Ces applications sont pour la plupart regroupées dans un même endroit, le Google Play Store. Ce service est une boutique en ligne où il est possible d'obtenir des applications, des livres, des films ou encore des musiques. Sur le Play Store, il est possible de trouver toute sortes d'applications partant d'un simple hello world jusqu'à des programmes très complets. Ces applications peuvent être payantes ou gratuites. Elles sont pour la plupart développées par des développeurs indépendants qui souhaitent mettre leurs applications à disposition. Ces développeurs utilisent certaines des nombreuses APIs utilisables pour faciliter et rendre plus rapide le travail de développement.

Au vu du rapide développement d'Android et du nombre croissant d'utilisateurs, il est très intéressant de développer une application pour ce système d'exploitation. De plus, les APIs assurent un développement relativement rapide et la licence open source rend le développement très peu onéreux. Un point important est que le développement d'applications Android soit effectué en Java, qui est un langage connu par de nombreux développeurs. C'est pour toutes ces raisons que nous avons décidé de créer une application Android. Les utilisateurs de iCelsius Wireless auront alors la possibilité d'observer les données de leurs sensors directement depuis leur smartphone.

Figure 128 Logo officiel d'Android, Bugdroid

9.2 Installation de l'environnement de développement

1) Télécharger et installer le SDK Android Tools : https://developer.android.com/sdk/index.html *-> USE AN EXISTING IDE -> Download the SDK Tools*
2) Exécuter Android SDK Manager, sélectionner et installer les packages *Tools*, *Android 2.1 (API 7)* et *Extras*
   Remarque: L'installation de la version 2.1 d'Android permet de s'assurer de la compatibilité de presque tous les appareils mobiles actuels. Effectivement, les versions plus récentes

supportent toutes les fonctionnalités de la version 2.1 d'Android. Une très faible partie des utilisateurs utilisant des versions plus anciennes ne pourront alors pas accéder à l'application. Cependant, de grands changements ont été apportés entre la version 1.6 et 2.1 d'Android. Il plus intéressant d'utiliser la version 2.1 au détriment des quelques personnes utilisant des versions antérieures.

3) Installer et exécuter une version d'Eclipse pour les développeurs Java: http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/keplerr -> Sélectionner la bonne version de la machine de travail dans la section *Download Links.*
4) Installation du plug-in Android pour Eclipse. Cet élément est disponible à l'aide de la fonction de mise à jour logiciel. Dans Eclipse: *Help -> Install new Software*. Entrer *https://dl-ssl.google.com/android/eclipse/* dans le champ *work with* et appuyer sur le bouton *add*, puis *ok* dans la nouvelle fenêtre. Installer les composants *Android DDMS, Android Development Tools*, *Android Hierarchy Viewer* et *Android Traceview*
5) Ajouter un émulateur Android. Dans Eclipse: Cliquer sur *Android Virtual Device Manager* () -> *new*. Dans la fenêtre *Create new Android Virtual Device (AVD)*, définir un nom, sélectionner un appareil et configurer le target avec *Android 2.1 – API Level 7* puis cliquer sur *ok*.
6) Configuration de l'émulateur Android. Commencer par exécuter l'émulateur: Dans Eclipse: Cliquer sur *Android Virtual Device Manager* () -> sélectionner l'émulateur créé -> *start…* Dans l'émulateur Android, il est ensuite possible de modifier l'heure, la langue ou le clavier via les configurations Android standard. Vérifier que les options de développement soient toutes cochées dans *Settings -> Application -> Development*.
7) Configuration du vrai terminal: Pour utiliser un vrai terminal Android, il est nécessaire de télécharger le bon pilote correspondant à l'appareil utilisé. Le terminal doit ensuite être configuré en mode développement. Sur Android 4.0 et suivant, il est nécessaire d'activer les options pour les développeurs. Pour ce faire, aller dans *Paramètres -> A propos du téléphone* et cliquer 7 fois sur *Numéro de build*. Ceci fera apparaître une catégorie *Options pour les développeurs* dans le menu *Paramètres*. Dans ce menu, il est nécessaire de cocher *Débogage USB* et *rester activé*. Dans le menu *Sécurité*, il est aussi nécessaire de cocher *Sources inconnues* afin de pouvoir installer notre application sur le terminal.

9.3 Création d'un projet

1) *File -> new -> other*. Ouvrir le dossier *Android* et sélectionner *Android Application Project.*
2) Entrer le nom de l'application, du projet et du package. Le nom de l'application sera le nom visible sur le smartphone et sur Google Play. Les noms de projet et de package ne seront pas publics. Il est ensuite nécessaire de sélectionner les différentes valeurs dans les listes déroulantes. Le *Minimum Required SDK* indique quels appareils pourront voir l'application dans Google Play.

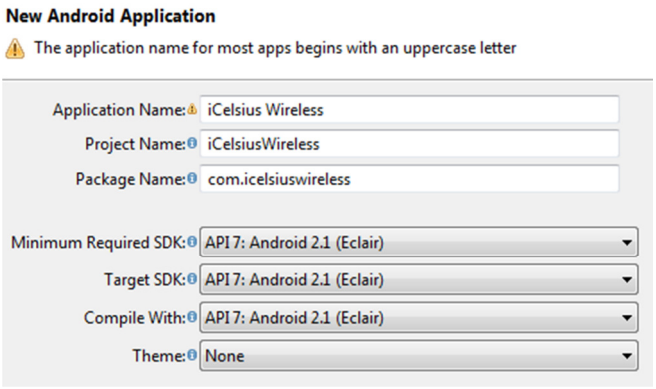


Figure 129 Création du projet Android

3) La fenêtre suivante propose différentes options dans des checkbox. Une des options cochée par défaut est la création d'une icône pour l'application. Cette option permet de définir quelle sera l'image représentant l'application sur les smartphones et Play Store.

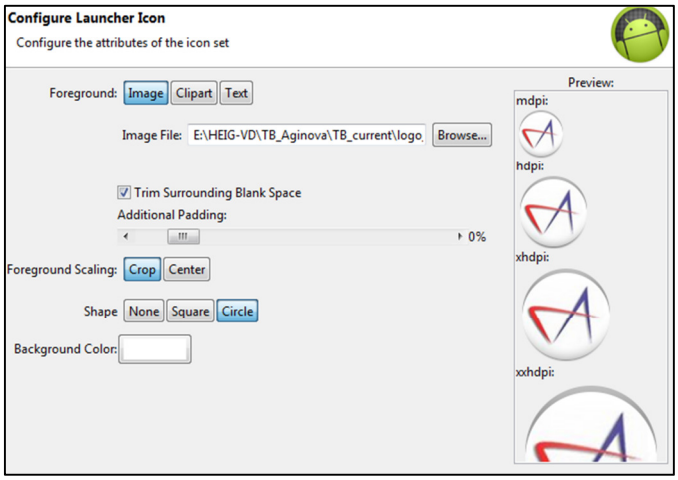


Figure 130 Configuration de l'icône de lancement

4) La fenêtre suivante permet de créer une activité. Pour ce projet, nous décidons de créer l'activité en partant de zéro afin de rester maître de tout.
5) Il est ensuite nécessaire de définir le nom de l'activité et du layout. Le nom de l'activité sera le nom de la classe Java qui contiendra l'activité. Le nom du layout sera le nom du fichier qui contiendra l'interface graphique de l'activité.

9.4 Exécution d'un projet

L'exécution d'un projet Android via Eclipse est très simple. Il suffit d'appuyer sur exécuter ( ) et de sélectionner *Android Application*. Si plusieurs terminaux sont disponibles, Eclipse proposera de choisir sur lequel des terminaux exécuter l'application. L'application sera ensuite automatiquement exécutée sur le terminal choisi.

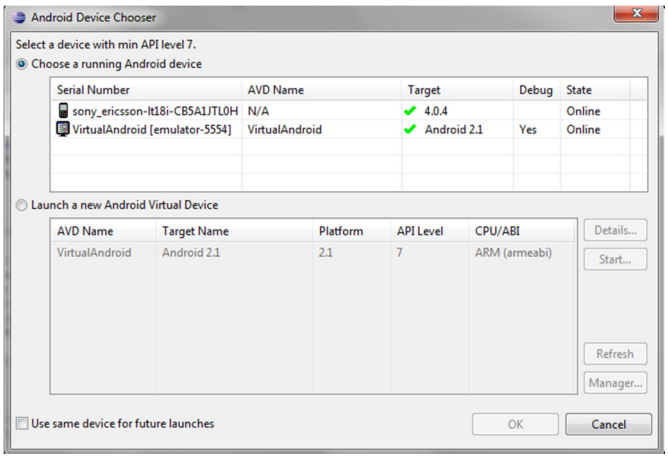


Figure 131 Choix du terminal pour l'exécution de l'application

9.5 Outils de debug d'un projet

Le moyen le plus classique utilisé pour débugger des applications est d'effectuer un affichage sur la sortie standard. A l'instar de Java qui renvoi les affichages dans la console de la machine où s'exécute le programme, le plug-ing Android pour Eclipse propose un outil permettant d'obtenir le même comportement. Sans cet outil, les différents affichages et messages d'erreur ne seraient pas visibles sur la console de l'ordinateur car l'application s'exécute sur un autre appareil.

L'outil proposé s'appelle *Logcat* et il est accessible via le menu *Windows -> Show View -> Logcat* dans Eclipse. Cet outil récupère tous les messages et les affiche dans Eclipse. Il est alors possible de visualiser les affichages effectués via un *System.out.print* ou les stack trace lors des exceptions. Il est aussi possible d'utiliser un système de log provenant du package *android.util.Log*. Les messages peuvent alors être loggés avec des niveaux d'importance différents.

9.6 Fonctionnement

Le système d'exploitation Android est un système open source basé sur un noyau Linux. Les applications Android fonctionnent dans une machine virtuelle nommée Dalvik. Cette machine est basée sur la machine virtuelle Java et permet d'exécuter des programmes écrits en Java sur un environnement possédant des ressources réduites. Les systèmes embarqués tels que les smartphones ne possèdent effectivement pas encore les mêmes capacités que les postes de travail standards. La machine Dalvik permet entre-autre de s'assurer qu'une application ne monopolise le processeur ou ne bloque le système Android. Un utilisateur ne souhaiterait pas qu'une application ne lui permette plus de répondre à un appel téléphonique entrant. La machine Dalvik définit des niveaux de priorités aux applications afin que toutes les tâches importantes restent prioritaires.

Une difficulté supplémentaire lors du développement sur un système embarqué est la variété des composants physiques disponibles. Android est un système d'exploitation fonctionnant sur un très grand nombre d'appareils qui n'ont pas tous les mêmes caractéristiques. L'existence ou non de certain composants, la taille de l'écran ou encore la langue de l'appareil sont des paramètres qu'il est nécessaire de prendre en compte pour le développement d'application Android. Une même application devra gérer plusieurs tailles d'écran et plusieurs langues différentes. L'application crée, *iCelsius Wireless*, n'utilise pas les différents capteurs disponibles sur les smartphones. Ceci simplifie légèrement le travail de développement. La taille de l'écran est aussi un point important pour les applications mobiles. Effectivement, du fait que l'écran de l'utilisateur soit petit, les chances que ce

dernier n'appuie pas sur la bonne touche est grande. Il est alors nécessaire de prendre en compte cette contrainte et de réagir correctement à la pression du bouton retour. La taille de l'écran est aussi une contrainte lors de la création de l'interface graphique. Il est difficile de mettre toutes les informations souhaitées sur un petit écran.

La compilation de code pour Android est effectuée en plusieurs étapes. Le code Java doit premièrement être compilé avec le compilateur *javac* contenu dans le JDK. Les fichiers de code source **.java* deviennent alors du bytecode pour Java, **.class*. Jusqu'ici, le bytecode créé permet d'exécuter le programme sur une machine virtuelle Java. Il est ensuite nécessaire de modifier ce bytecode avec le programme *dx*, contenu dans le SDK d'Android. Cet utilitaire effectue une conversion du bytecode Java en bytecode Dalvik. C'est ce dernier bytecode que la machine virtuelle Dalvik pourra exécuter.

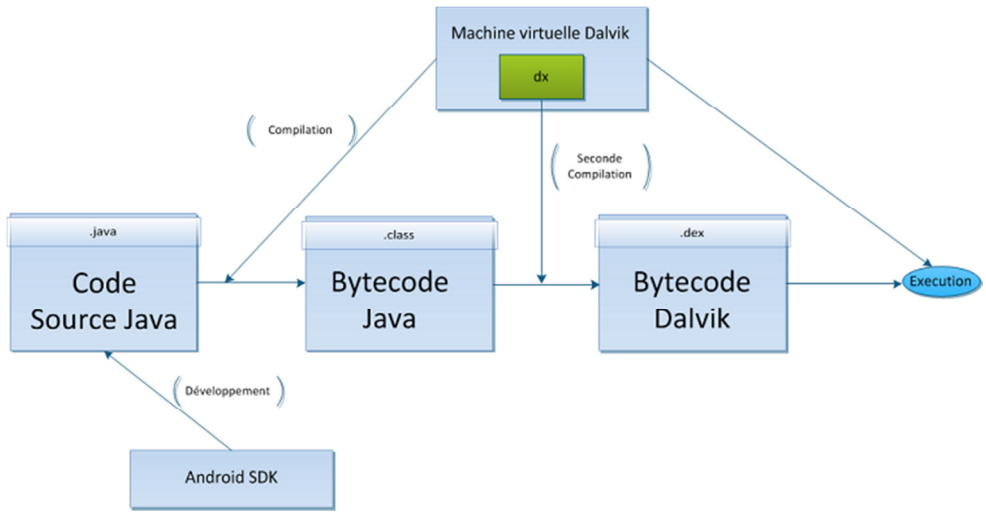


Figure 132 Compilation d'un programme pour Android

Le développement pour Android nécessite de comprendre certains concepts spécifiques à ce système d'exploitation. Chaque OS possède ses propres caractéristiques et il est nécessaire de les connaitre avant de pouvoir effectuer un développement. Les chapitres suivants introduisent certains de ces concepts.

9.6.1 Activité

Une activité correspond à une fenêtre prenant tout l'écran dans une application Android. Chacune de ces fenêtres est composée d'une interface graphique visible de l'utilisateur et d'un contexte donnant des informations sur l'application. Ces deux éléments forment une activité. Comme l'interface graphique d'une activité occupe tout l'espace sur l'écran de l'utilisateur, il ne peut y avoir qu'une seule activité affichée à un instant donné.

Les applications Android permettent de naviguer d'une vue à une autre via l'interface graphique. Chacune de ces vues est alors une activité différente. Il est par exemple possible de cliquer sur un élément d'une liste qui est affichée sur l'écran. Ceci permet alors à l'utilisateur de voir les informations correspondantes à l'élément sélectionné. Dans cet exemple, l'application est composée de deux activités différentes. La première permet d'afficher la liste alors que la seconde permet d'afficher des détails sur un élément choisi.

Une activité n'a pas le contrôle de son état, ce contrôle est effectué par les interactions avec le système et les autres applications. La Figure 133 est le cycle de vie d'une activité, de sa naissance jusqu'à sa mort. Lorsque l'activité changera d'état, une des méthodes présentes dans cette figure sera appelée. Ces méthodes sont des callbacks qui seront appelés lorsque l'utilisateur quitte l'activité, y revient ou effectue d'autres opérations sur son appareil Android.

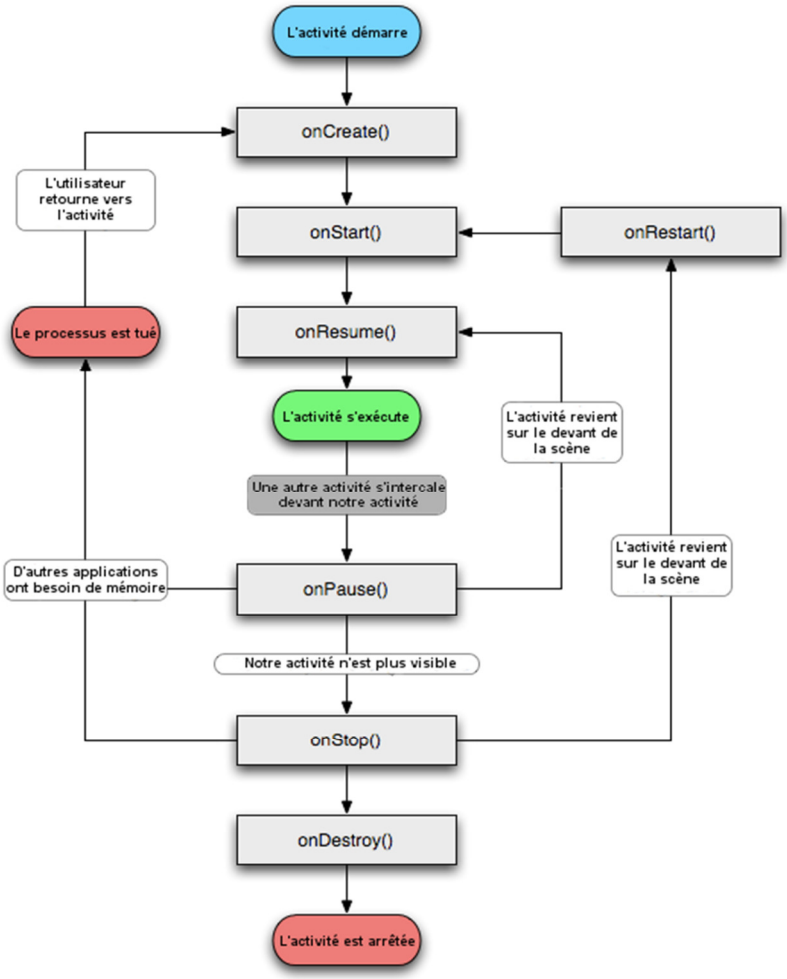


Figure 133 Cycle de vie d'une activité

La méthode *onCreate* est la première qui est appelée lors du démarrage de l'activité. Cette méthode est appelée lorsque l'utilisateur démarre l'activité pour la première fois ou lorsque que le système avait tué l'activité pour récupérer de la mémoire. Cette méthode possède un paramètre de type *Bundle*. Si l'activité est exécutée pour la première fois ou que l'utilisateur avait quitté l'activité correctement, ce paramètre vaudra *null*. Si l'activité redémarre alors qu'elle avait été tuée par le système, ce paramètre permet de récupérer des informations qui auraient été sauvegardées précédemment. Cette méthode devrait définir tous les éléments qui doivent être créés au démarrage de l'activité. Cette méthode est privilégiée pour initialiser l'interface graphique ou lancer les tâches en arrière-plan.

La méthode *onDestroy* est la dernière méthode qui est appelée avant la destruction de l'activité. Il y a trois possibilités différentes pour arriver à cette méthode. La première est que l'activité est dans l'état arrêté et que le système Android a besoin de mémoire et arrête complètement l'application. La deuxième possibilité est que l'utilisateur presse sur le bouton *back* et qu'il n'y a pas d'activité

précédente. La troisième solution est que le code de l'activité appel la méthode *finish* qui termine l'activité en passant par la méthode *onDestroy*. Cette méthode doit arrêter les tâches de fond qui seraient encore en cours d'exécution.

La méthode *onPause* est celle qui est appelée lorsque l'activité entre dans le mode suspendu. Elle n'est alors que partiellement visible et l'utilisateur regarde une boite de dialogue qui s'est ouverte par-dessus l'activité. Cette méthode doit alors libérer toutes les ressources physiques et arrêter toutes les tâches afin que le processeur ne soit plus sollicité par notre activité. Cette méthode doit être la plus rapide possible afin de fluidifier la manipulation de l'utilisateur.

La méthode *onResume* est celle qui est appelée lorsque l'activité retourne au premier plan. Elle est aussi exécutée à chaque lancement de l'activité. Son rôle est d'initialiser les ressources qui seront libérées dans la fonction *onPause*.

La méthode *onStop* est appelée lorsque l'activité entre dans le mode arrêté. Cet état peut survenir lorsque l'utilisateur passe sur une autre activité sans quitter proprement l'activité courante. A ce moment il est nécessaire d'effectuer les sauvegardes nécessaires à la prochaine utilisation de l'activité. Effectivement, dans cet état l'activité peut être détruite à tout moment par Android s'il manque de mémoire.

La méthode *onRestart* est appelée lorsque l'utilisateur revient sur une activité dont l'état était arrêté. Cette méthode permet d'effectuer des actions qui doivent être exécutées uniquement si l'activité sort de l'état arrêté.

9.6.2 Etat de l'application

Avec le système Android, il est possible d'avoir plusieurs applications instanciées à un moment donné, chacune de ces applications possédant plusieurs activités. Néanmoins, il n'est possible d'exécuter qu'une seule activité à la fois. Il est donc nécessaire de gérer l'attente et l'élection des activités. Cette gestion est effectuée grâce à une pile de type LIFO. La dernière activité insérée à la pile sera alors la première à en sortir. L'application qui se situe au sommet de la pile est celle qui est actuellement visible pour l'utilisateur.

Les applications peuvent se trouver dans trois états différents qui permettent surtout de différencier la visibilité des activités. Ces états sont présentés dans le Tableau 17.

Active (running)	L'activité est en cours d'exécution sur l'appareil. L'interface graphique de l'activité en cours est affichée sur l'écran. L'utilisateur consulte alors cette activité et peut interagir avec elle. L'activité est donc au sommet de la pile.
Suspendue (paused)	L'activité est partiellement visible sur l'écran. Ce cas de figure se présente lorsque l'utilisateur utilise l'application est qu'il reçoit un SMS. Une fenêtre se met alors par-dessus l'activité. Android appel la méthode de callback *onPause* lors de l'entrée dans cet état et la méthode *onResume* lors de la sortie de l'état.
Arrêtée (stopped)	L'activité n'est plus du tout affichée à l'écran. L'utilisateur n'utilise plus cette activité pour le moment. L'état des activités est alors mémorisé pour une utilisation ultérieure. L'activité peut être tuée à tout moment si le système a besoin de mémoire. La méthode de callback *onStop* est appelée lors de l'entrée de l'activité dans cet état. La sortie de l'état est défini par l'appel de *onRestart* si l'activité n'a pas été tuée, sinon *onCreate*.

Tableau 17 Etat des applications Android

9.6.3 Ressources

Android est un système d'exploitation qui permet son utilisation sur un très grand nombre d'appareils possédant des caractéristiques différentes. Il est alors nécessaire de prendre en compte ces différentes caractéristiques pour que l'application créée s'adapte au mieux à l'appareil sur lequel elle est exécutée. Les ressources permettent d'adapter une application selon le terminal sur lequel elle est installée. Les ressources sont des fichiers organisés d'une certaine manière pour qu'Android sache quelle ressource utiliser en fonction du matériel sur lequel s'exécute l'application. C'est donc grâce à ces ressources qu'il est possible d'adapter la langue des chaines de caractères ou la taille des images qui seront affichées dans l'application.

Lors de la création du projet, Eclipse nous crée une hiérarchie de répertoire dans le dossier *res* qui contient les ressources. Les ressources peuvent être de plusieurs types (image, texte, layout, …) et sont regroupées dans des dossiers possédants des noms standards. C'est grâce à ces dossiers qu'Android pourra automatiquement s'adapter au terminal. Le Tableau 18 présente les types de ressources les plus utilisés dans les applications Android.

res/drawable	Ce dossier contient tous les dessins et images de l'application. Ce dossier contient donc des fichiers PNG, JPEG ou GIF ainsi que des fichiers XML décrivant des dessins simples.
res/layout	Ce dossier contient les fichiers gérant l'interface graphique de l'application. Cette interface graphique est gérée grâce à des fichiers XML qui représentent la disposition des vues.
res/menu	Ce dossier contient des fichiers XML décrivant la constitution des menus de l'application.
res/raw	Ce dossier contient les données au format brut de l'application. On y retrouve par exemple des fichiers de musique ou des fichiers HTML.
res/values	Ce dossier contient toutes les valeurs ou variables utilisées dans l'application. On y trouve notamment les chaines de caractères qui seront affichées sur l'interface graphique de l'application.

Tableau 18 Types de ressources les plus utilisées

Pour pouvoir s'adapter au matériel, il est nécessaire d'ajouter des quantificateurs aux noms des dossiers de ressources. Les quantificateurs sont des noms qui viennent s'ajouter au nom du dossier de la ressource afin de préciser à Android le type de matériel pour lequel les fichiers dans ce répertoire sont destinés. La syntaxe des quantificateurs est présentée à la Figure 134. On voit qu'il est possible de définir plusieurs quantificateurs pour un même dossier. Si Android ne trouve pas une correspondance exacte avec le matériel utilisé, il utilisera le dossier qui contiendra le plus de quantificateurs correspondant au terminal. Si aucun dossier ne correspond au terminal, Android récupère les ressources dans le dossier par défaut, sans aucun quantificateur.

res/<type_de_ressource>[<-quantificateur 1><-quantificateur 2>…<-quantificateur N>]

Figure 134 Syntaxe des quantificateurs

La Figure 135 représente les dossiers de ressources créés automatiquement par Eclipse. On remarque qu'il n'y a pas de dossier par défaut pour le dossier *drawable* qui contient les images de l'application. Tous les dossiers *drawable* sont suivis d'un quantificateur permettant de préciser la taille de l'écran de l'appareil. Les différents quantificateurs pour les tailles d'écran sont présentés

dans ce **lien**[7]. De la même manière, il est possible de définir des quantificateurs pour définir la langue de l'appareil. Cette option est surtout utilisée sur le dossier *value* qui contiendra notamment les chaines de caractères à afficher sur l'interface graphique. L'utilisation des quantificateurs de langues est présentée dans ce **lien**[8]. Les derniers quantificateurs les plus utilisés sont *-port* et *-land* qui permettent de définir le dossier à utiliser si l'appareil est en mode portrait ou paysage. Ces deux quantificateurs sont le plus généralement utilisés sur le dossier *layout* qui contient l'interface graphique de l'application. Afin d'éviter des incohérences sur les quantificateurs, Android a mis en place un système de priorité qui empêche le développeur de définir un dossier avec deux quantificateurs de langue par exemple.

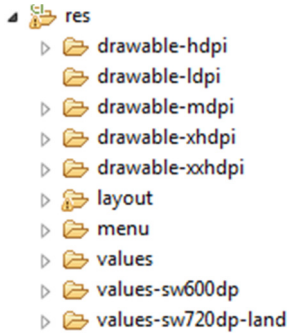


Figure 135 Dossier ressources créé par Eclipse

Le plug-in Android pour Eclipse contient un outil très pratique pour la création de fichiers de ressources. Cet outil permet de choisir simplement les différents quantificateurs du dossier qui contiendra le fichier de ressource. Il n'est alors pas nécessaire de connaitre tous les quantificateurs par cœur. La sélection est effectuée via un assistant avec une interface graphique. Cet assistant apparaît lors de la création d'un nouveau fichier de type *Android XML file*.

Les dossiers de même type mais avec des quantificateurs différents doivent contenir des fichiers de ressources ayant le même nom. Par exemple si le dossier *drawable-hdpi* contient le fichier *ic_launcher.png*, le dossier *dawable-ldpi* devra contenir un fichier avec le même nom. Ce nom sera ensuite utilisé pour récupérer la bonne ressource selon le terminal utilisé. Effectivement, lors de l'utilisation de la ressource dans le code, seul le nom de celle-ci sera donné. C'est donc au système Android que revient le rôle de définir de quel dossier proviendra la ressource en fonction des caractéristiques de l'appareil utilisé.

Lorsque les ressources sont déclarées correctement, il est possible de les utiliser dans l'application Android. Pour se faire, Android génère automatiquement le fichier contenant la classe R qui permet l'utilisation des ressources dans le code. La section *9.6.4 Classe R* explique comment récupérer une ressource dans le code.

9.6.4 Classe R

La classe R est une classe crée et gérée automatiquement par Android. Cette classe permet d'utiliser les différentes ressources définies dans le dossier *res*. A chaque compilation, l'outil *aapt* génère la classe R qui contient des identifiants pour toutes les ressources présentes dans le dossier *res*. Pour chaque type de ressources présent dans le dossier *res*, Android crée une sous-classe dans la classe R.

[7] https://developer.android.com/guide/practices/screens_support.html
[8] https://developer.android.com/training/basics/supporting-devices/languages.html.

Ces sous-classes contiennent des *integer* qui sont des identifiants uniques pour les ressources. Ces identifiants sont générés automatiquement, il ne faut alors absolument pas modifier la classe R manuellement. La Figure 136 nous montre que la classe R se situe dans le dossier *gen* du projet. Ce dossier contient les fichiers générés automatiquement.

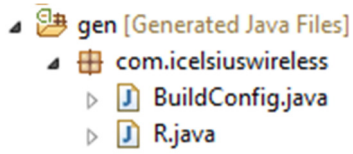


Figure 136 Emplacement de la classe R

L'utilité de la classe R est de récupérer les ressources définies dans le dossier *res*. Dans le code Java, il est possible de récupérer l'identifiant de la ressource via la syntaxe de la Figure 137. Cette identifiant peut ensuite être utilisé avec une des méthodes spécifiques à Android comme le montre la Figure 138. Comme les identifiants sont tous déclarés en *static*, il n'est pas nécessaire d'instancier la classe R pour avoir accès au différents identifiants.

```
[<package_name>.]R.<resource_type>.<resource_name>
```

Figure 137 Récupération d'une ressource en Java

```
setContentView(R.layout.activity_main);
vue = (ListView) findViewById(R.id.list);
```

Figure 138 Utilisation d'une ressource en Java

Il est aussi possible d'utiliser ces ressources dans les fichiers XML qui sont eux-mêmes des ressources. Cette possibilité est très pratique lors de la création de fichiers de layout. Il est par exemple possible de créer un bouton avec un texte provenant d'une ressource de type string. L'accès aux identifiants de la classe R depuis un fichier XML se fait grâce au sigle arobase comme l'illustre la Figure 139. Cet exemple recherche la valeur associée à *hello* dans la classe *string*.

```
android:text="@string/hello"/>
```

Figure 139 Utilisation d'une ressource en XML

9.6.5 Intent

Les *intents* sont les agents qui permettent à une activité de communiquer avec d'autres activités ou d'autres applications. Android utilise massivement ce mécanisme en interne pour la communication entre les différents composants et applications.

Ce document ne présentera pas les *intents* car ils ne sont pas utilisés dans l'application créée. Effectivement, elle ne contient qu'une seule activité et ne communique pas avec d'autres applications, les *intents* ne sont alors pas utilisés. Cependant, il sera nécessaire d'utiliser ce mécanisme dès qu'une autre activité sera créée, il est donc important de mentionner leur existence.

9.6.6 Stockage des données

Dans la plupart des applications Android, il est nécessaire de sauvegarder certaines informations relatives à l'utilisateur ou à l'état courant de l'application. Pour se faire Android permet aux applications de créer des fichiers sur le système de fichiers de l'appareil. La gestion des fichiers se fait alors via les commandes classiques de Java.

Pour l'application créée, la seule valeur qui est sauvegardée est la dernière liste de sensors reçue du serveur. En sauvegardant cet élément, le démarrage de l'application ne nécessite pas d'attendre des données provenant du serveur avant d'effectuer un affichage. Les dernières valeurs reçues seront alors affichées pendant que l'application effectue une requête sur le serveur pour mettre à jour ces valeurs. De plus, si l'utilisateur n'a pas de connexion internet, cette méthode lui permet de visualiser les dernières valeurs reçues.

Les données provenant du serveur sont contenues dans un objet de type *String* au format JSON. Il est alors possible de créer un fichier pour sauvegarder cet élément. Cependant, Android nous met à disposition un élément nous permettant de sauvegarder facilement des objets de type primitif ou des *String*. La classe *SharedPreferences* est effectivement destinée à sauvegarder des informations sur l'application même si celle-ci est détruite par Android ou quittée par l'utilisateur. Les informations sont sauvegardées et récupérables via un identifiant de manière identique à un HashMap.

Comme indiqué dans le chapitre *9.6.1 Activité*, le stockage des données devrait être effectué dans la méthode *onStop* alors que ces mêmes données devraient être récupérées dans la méthode *onCreate*.

9.6.7 AsyncTask

Il est important d'utiliser plusieurs threads dans une application Android. Le thread principal doit éviter d'effectuer un long travail qui ne lui permettrait pas de gérer l'interface utilisateur dans un délai correct. Effectivement, un utilisateur remarquera tout ralentissement de l'application si celle-ci met plus de 100ms à répondre à une action sur l'interface graphique. Passé un certain délai de réactivité, l'application peut même être tuée par Android. Il est donc nécessaire de déléguer les tâches lentes à d'autres threads.

Une activité se compose d'un thread principal gérant l'interface utilisateur (UI) et d'un certain nombre de threads permettant d'effectuer du travail en arrière-plan. Seul le thread principal a la possibilité de modifier l'interface graphique. Si les autres threads essaient d'effectuer des modifications sur l'UI, l'action lèvera l'exception *CalledFromWrongThreadException*. Il est alors nécessaire de mettre en place un système de communication entre les threads afin de pouvoir transmettre des valeurs obtenues lors d'opérations lentes au thread principal. Ce dernier aura alors le rôle de mettre l'affichage à jour.

La machine Dalvik étant dérivée de la machine virtuelle Java, il est possible d'utiliser la classe *Thread* dans les applications Android. Cependant, la gestion des threads demande de bonnes connaissances en programmation concurrente afin d'éviter des problèmes d'inter-blocage. De plus, il est nécessaire d'effectuer la gestion de la communication entre les différents threads. Android nous met à disposition une classe, *AsyncTask*, qui permet de mettre en place simplement un travail en arrière-plan. L'utilisation des *AsyncTask* est donc un moyen plus simple pour gérer des threads.

La classe *AsyncTask* permet d'effectuer du travail en parallèle du thread gérant l'interface utilisateur. Il est nécessaire de créer une classe qui dérive de la classe *AsyncTask* afin de pouvoir surcharger les méthodes utilisées. Lorsqu'on dérive de cette classe, il est possible de définir les types d'objets qui seront pris et retournés par le thread, la classe *AsyncTask* est donc une classe générique. La Figure 140 montre que la classe *UpdateThread* dérive *d'AsyncTask* avec trois paramètres génériques. Le premier indique le type de paramètre qui sera pris en entrée du thread. La deuxième valeur permet

au thread de transmettre l'état de progression de la tâche. Cette valeur peut être récupérée par le thread principal afin de mettre à jour une barre de progression dans l'interface graphique. La dernière valeur est le type qui sera retourné par le thread. Dans la classe *UpdateThread*, le thread créé ne prendra pas de paramètre d'entrée, ne transmettra pas l'état de progression de la tâche et renverra un paramètre de type booléen afin d'indiquer l'état de terminaison du thread.

```
class UpdateThread extends AsyncTask<Void, Void, Boolean>
```

Figure 140 Déclaration de la classe dérivée de AsyncTask

Pour exécuter le thread, il est simplement nécessaire de créer un objet de type *UpdateThread* et d'y appliquer la méthode *execute*. Lorsque *l'AsyncTask* aura terminé son travail, il n'est plus possible de la relancer. Il est donc nécessaire de recréer un objet de type *UpdateThread* pour chaque exécution du thread. La méthode *execute* prend en argument une ellipse contenant des objets du type défini lors de la déclaration de la classe *UpdateThread*. Il est donc possible de fournir un grand nombre d'arguments d'un type précis lors de l'exécution du thread.

La méthode *execute* permet de démarrer le nouveau thread. Trois méthodes de callback sont ensuite appelées dans l'ordre suivant:

1) *onPreExecute()* : Cette méthode s'exécute avant la création du thread. Les instructions qu'elle contient sont exécutées par le thread principal. Tous les éléments qui nécessitent une initialisation sont traités dans cette méthode.
2) *doInBackground(<T>… param)* : Cette méthode est celle qui contient le code qui sera exécuté par le thread créé. Elle prend en paramètre les arguments passés lors de l'appel de la méthode *execute*. Toutes les instructions présentes dans cette méthode seront exécutées en parallèle du thread principal.
3) *onPostExecute(<T> result)* : Cette méthode s'exécute après la terminaison du thread. Les instructions qu'elle contient sont alors exécutées par le thread principal. Le paramètre est la valeur de retour de la méthode *doInBackground*. Dans cette méthode, il est possible de mettre à jour l'interface utilisateur selon le résultat du thread exécuté.

9.6.8 Connectivité réseau

La connectivité réseau est la permission la plus demandée dans les applications Android. L'application créée ne fait pas défaut à cette statistique car il est nécessaire d'avoir un accès internet afin de récupérer les données du serveur.

Avant d'effectuer une requête sur un serveur web, il est nécessaire de vérifier que l'appareil sur lequel s'exécute l'application est bien connecté au réseau. Cette fonctionnalité nécessite une permission supplémentaire, comme le montre la Figure 141. La connectivité réseau nécessite alors la permission *INTERNET* et la permission *ACCESS_NETWORK_STATE*.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Figure 141 Permission nécessaire pour la connectivité réseau

La méthode *isConnectedToInternet* présentée à la Figure 142 retourne un booléen indiquant si l'appareil est connecté à un point d'accès internet valide. La première ligne de cette méthode récupère le gestionnaire de connexion. La deuxième ligne récupère ensuite l'interface active de

l'appareil. Si aucune interface n'est active, cette méthode retournera *null*. Il est finalement nécessaire de vérifier que l'appareil soit connecté via l'interface active.

```
public boolean isConnectedToInternet(){
    ConnectivityManager connectivityManager = (ConnectivityManager) getSystemService(CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connectivityManager.getActiveNetworkInfo();
    return networkInfo != null && networkInfo.isAvailable() && networkInfo.isConnected();
}
```

Figure 142 Contrôle de la connectivité réseau

Si la méthode *isConnectedToInternet* retourne true, l'appareil est connecté à internet. Il est alors possible d'effectuer une requête sur le serveur de Google App Engine afin de récupérer les informations sur les sensors. Cette opération nécessite l'envoi d'une requête HTTP de type GET. La récupération de contenu sur internet doit toujours se faire dans un thread séparé du thread principal. Effectivement, les requêtes HTTP peuvent prendre un certain temps, il ne faut pas bloquer l'interface utilisateur pendant ce temps-là.

La méthode *getNewSensorData* permet d'effectuer une requête sur le serveur afin de récupérer les informations sur les sensors. L'objet *url* permet de définir sur quelle URL la requête va être envoyée. La méthode *openConnection* permet d'ouvrir une connexion sur l'URL définie. Si la connexion s'est bien déroulée, il est possible de récupérer le flux afin de lire les données. Après la lecture, il est nécessaire de refermer le flux avec la méthode *disconnect*. La méthode *readStream* permet simplement de lire le flux d'entrée et de récupérer les données fournies dans un objet de type String.

```
URL url = new URL(SERVER_URL);
HttpURLConnection httpConnection = (HttpURLConnection) url.openConnection();

if (httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK) {
    pageReceived = readStream(httpConnection.getInputStream());
    pageReceived = pageReceived.trim();
    httpConnection.disconnect();
    return pageReceived;
}
```

Figure 143 Récupération du flux sur internet

Il sera ensuite possible d'implémenter de la même manière une requête POST sur le serveur. Comme expliqué dans le chapitre *6.1.1 Méthodes*, la méthode POST permet de transmettre des valeurs au serveur. Pour se faire, il est nécessaire d'ajouter des paramètres dans l'URL de la requête. La Figure 144 est un exemple d'une URL possédant les paramètres *selectedSensor* et *selectedType*.

```
http://apps-test-tb2013.appspot.com/unauthMobile?selectedSensor=103&selectedType=temperature
```

Figure 144 URL pour une requête POST sur le serveur

9.6.9 Fichier Manifest

Le fichier Manifest est un fichier XML indispensable au fonctionnement d'une application Android qui est présent à la racine du projet et porte le nom *AndroidManifest.xml*. Ce fichier permet de définir certaines informations comme le nom de l'application, les ressources nécessaires ou certains paramètres de sécurité. Ce fichier est généré automatiquement par Eclipse lors de la création du projet. Il contient déjà plusieurs paramètres qui ont été définis lors de la création du projet.

Le nœud *manifest* comporte les attributs présentés à la Figure 145. L'attribut *xmlns* permet d'indiquer qu'on utilise l'espace de noms *android*. L'attribut *package* précise le package utilisé pour l'application. Ce package doit posséder un nom unique pour toutes les applications présentes sur le

Play Store. C'est grâce à ce nom que Google reconnait les applications, il n'est alors pas possible de le modifier tout en gardant un suivi de tous les utilisateurs. L'attribut *versionCode* n'est pas visible des utilisateurs, il est utilisé par le Play Store pour connaitre la version actuelle de l'application. Le nom visible des utilisateurs est défini par l'attribut *versionName*.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.icelsiuswireless"
    android:versionCode="1"
    android:versionName="1.0" >
```

Figure 145 Attributs du nœud manifest

Le nœud *uses-sdk* permet de donner des informations sur les versions des APIs que doivent supporter les appareils Android pour pouvoir exécuter l'application. L'attribut *minSdkVersion* indique la version minimale que doit supporter un appareil Android pour exécuter l'application. L'application ne sera pas visible dans le Play Store pour les appareils possédants une API plus ancienne. L'attribut *targetSdkVersion* permet d'indiquer la version de l'API à partir de laquelle l'application peut être exploitée pleinement. Il est en effet possible de rendre une application compatible avec une version plus ancienne de l'API en diminuant les fonctionnalités de l'application. Les valeurs des attributs *minSdkVersion* et *targetSdkVersion* peuvent alors différer.

```
<uses-sdk
    android:minSdkVersion="7"
    android:targetSdkVersion="7" />
```

Figure 146 Attributs du nœud uses-sdk

Le nœud *uses-permission* permet d'indiquer à Android que nous aurons besoin de certaines ressources pour faire fonctionner l'application correctement. Par défaut, l'application n'a accès à aucune ressource qui pourrait nuire à Android, aux autres applications ou à l'utilisateur. Il est nécessaire de déclarer toutes les permissions que nous utilisons. Lors de l'installation de l'application via le Play Store, l'utilisateur sera informé des différentes permissions demandées et devra les accepter pour poursuivre l'installation.

La Figure 147 présente les différentes permissions requises par l'application créée. L'accès à internet est évidamment requis pour demander des informations au serveur. Avant de demander ces information, il est nécessaire de vérifier que l'utilisateur soit connecté au réseau. Pour se faire, nous avons besoin de la permission donnant l'accès à l'état du réseau. La connection avec le serveur nécessite une authentification avec le compte Google de l'utilisateur. Cette utilisation nécessite les permissions *GET_ACCOUNTS* et *USE_CREDENTIALS* permettant la gestion du compte utilisateur.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
```

Figure 147 Permissions utilisées par l'application

Le nœud *application* contient tous les attributs qui caractérisent l'application. De plus, ce nœud contient des enfants comprenant tous les composants de l'application. L'attribut *icon* permet de définir l'icône de l'application. Cette icône est visible lorsque l'utilisateur navigue parmi toutes les applications installées sur son appareil. Le nom de l'application est défini par l'attribut *label*. L'attribut *theme* permet de définir un style qui sera appliqué sur chaque vue de l'application. L'attribut *debuggable* indique que l'application est débogable. Cette option ne doit être activée que

lors de la phase de debug de l'application. L'attribut *allowBackup* indique si l'application doit être prise en compte lors des backups effectués avec l'outil *backup and restore* d'Android.

```
<application
    android:debuggable="true"
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
```

Figure 148 Attributs du nœud application

Le fichier comporte ensuite un ou plusieurs nœuds *activity* qui représentent des activités. Pour rappel, une activité représente une page de l'application. Une application standard possédant plusieurs pages, le fichier de manifest comportera plusieurs nœuds *activity*. L'attribut *name* contient le nom de la classe qui implémente l'activité. La valeur de l'attribut *label* est le nom qui sera affiché en haut de l'activité sur l'écran de l'utilisateur. Lorsque l'utilisateur fait basculer son appareil du mode portrait au mode paysage, l'activité est arrêtée puis redémarrée par Android. Certaines données peuvent alors être perdues. Afin d'éviter ce mécanisme, nous avons décidé de bloquer l'affichage de l'activité en mode portrait. Nous devons donc indiquer que nous ne souhaitons pas arrêter et redémarrer l'application lors d'un changement d'orientation. Pour se faire, nous utilisons l'attribut *configChanges* auquel nous passons la valeur *orientation*. Il est ensuite nécessaire d'indiquer l'orientation souhaitée de l'application. Cette valeur est définie par l'attribut *screenOrientation*. Le nœud *intent-filter* permet ensuite d'indiquer à Android que cette activité est celle à exécuter lors du démarrage de l'application.

```
<activity
    android:name="com.icelsiuswireless.MainActivity"
    android:label="@string/app_name"
    android:configChanges="orientation"
    android:screenOrientation="portrait" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Figure 149 Attributs du nœud activity

9.7 Les vues

Les vues sont les différentes interfaces graphiques qui seront affichées à l'utilisateur. Chaque activité possède une vue qui lui est associée. Ces vues sont contenues dans le dossier de ressource *layout*. Elles sont représentées par des fichiers XML spécifiques à Android.

La création de ces vues peut se faire via une interface graphique proposée par le plug-in d'Eclipse. Cependant, cette méthode ne permet pas d'avoir la même précision que l'écriture du code XML. De plus l'outil d'Eclipse semble comporter quelques bugs récurrents. Cette interface est néanmoins intéressante afin obtenir un aperçu rapide du rendu que donnera code XML écrit.

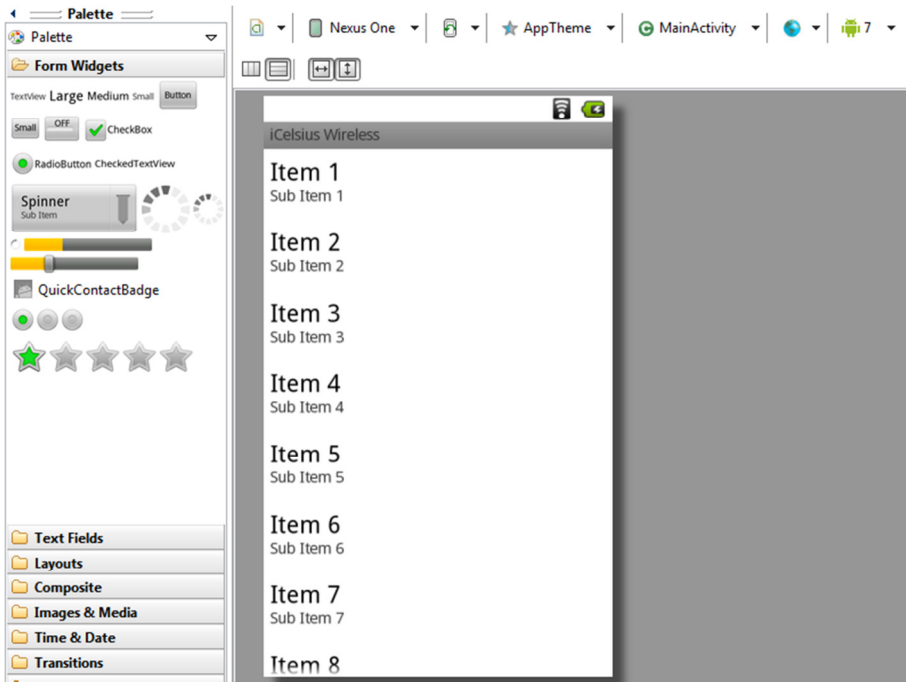


Figure 150 Interface graphique pour la création de vue

Le fichier XML correspondant à la vue doit obligatoirement comporter l'attribut de la Figure 151. Cet attribut permet d'indiquer au compilateur qu'il s'agit d'un fichier XML spécifique à Android.

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

Figure 151 Eléments obligatoires dans un fichier XML de layout

La racine du fichier XML comporte un nœud contenant le nom du layout utilisé. Un layout est une vue spéciale qui peut contenir des vues ou des widgets. Le rôle du layout est de mettre en place les différentes vues afin de créer une interface graphique conviviale. Le rôle de ces vues est de proposer de l'interaction avec l'utilisateur. Un widget est une vue qui ne peut pas en englober une autre. Il s'agit par exemple d'un bouton ou d'une zone de saisie.

Tous nœuds représentant des vues possèdent des attributs en commun. Parmi ceux-ci, les plus utilisés sont *layout_width* et *layout_height* qui permettent de définir la place que prendra la vue sur le layout. Ces attributs peuvent prendre trois types de valeurs. La valeur *fill_parent* indique que la vue prendra la place du parent sur l'axe concerné. La valeur *wrap_content* indique que la vue prendra le minimum de place possible selon le contenu de la vue. Il est aussi possible de définir une valeur numérique à ces attributs. Il est conseillé de donner ces valeurs numériques avec l'unité *dip* ou *dp* qui est une unité indépendante de la résolution de l'écran. Il est effectivement possible de définir une valeur en pixel ou en millimètre mais ceci pose des problèmes lors de l'utilisation de l'application sur des écrans ayant des résolutions différentes. A noter que la taille du texte doit être définie avec l'unité *sp* qui correspond à *dip* mais pour du texte.

9.7.1 Identifiants

Afin que les vues puissent être utilisées par le code, il est nécessaire qu'elles soient associées à un identifiant. Cet identifiant sera automatiquement créé et accessible via la classe R. Pour associer un identifiant à une vue il est nécessaire de rajouter un attribut, *android:id*. La valeur de cet attribut doit

respecter la syntaxe *@+<nomClasse>/<nomIdentifiant>.* Grâce au symbole +, on indique à Android qu'il doit créer un nouvel identifiant. Le nom de la classe des identifiants est généralement *id*.

Les attributs des vues peuvent utiliser des ressources définies dans le dossier *res*. L'accès à une ressource se fait via la syntaxe *@<nomClasse>/<nomIdentifiant>.* Le @ signifie que le code XML va accéder à une ressource. La Figure 152 est un exemple de l'utilisation des identifiants dans un fichier XML. On déclare un nœud *item* et on crée un nouvel identifiant s'appelant *exit*. Le titre de cet *item* est référencé via l'identifiant *exit* de la classe *string* alors que l'icône de l'*item* est référencé par l'identifiant *ic_menu_close_clear_cancel* de la classe *drawable*.

```
<item
    android:id="@+id/exit"
    android:title="@string/exit"
    android:icon="@drawable/ic_menu_close_clear_cancel" />
```

Figure 152 Exemple des identifiants dans un fichier XML

Grâce à ces identifiants, il est possible de récupérer les vues dans une activité afin de modifier l'interface graphique de l'activité. La Figure 153 illustre l'utilisation de la méthode *setContentView* et la récupération d'une ressource de type vue. La méthode *setContentView* permet d'associer une activité à un layout. C'est donc via cette méthode que l'interface graphique d'une activité est définie. La méthode *findViewById* permet de récupérer une vue selon l'identifiant passé en paramètre. Il est ensuite possible d'appliquer des méthodes spécifiques à la vue récupérée afin de modifier les éléments qui seront affichés à l'utilisateur.

```
setContentView(R.layout.activity_main);
vue = (ListView) findViewById(R.id.list);
```

Figure 153 Utilisation d'une ressource en Java

9.7.2 Widgets

Les widgets sont des vues qui fournissent un contenu. Ces vues n'ont pas le rôle de mettre en forme l'interface graphique. Le contenu permet à l'utilisateur d'interagir avec le système ou de fournir des informations à l'utilisateur. Chaque widget possède un grand nombre d'attributs qui lui sont propres. Ce chapitre présente les principaux widgets utilisés pour l'application *iCelsius Wireless*. Pour plus d'information sur les widgets et leurs attributs, consulter la documentation officielle sur ce **lien**[9].

Le widget *TextView* est une vue permettant d'afficher une chaine de caractères. Cette chaine n'est pas modifiable par l'utilisateur et peut être formatée par du code HTML. La Figure 154 représente un *TextView* occupant toute la largeur de l'écran, une hauteur minimale et affichant le texte correspondant à *WelcomeOnApp* définit dans la classe *string*. La taille du texte est définie par l'attribut *android_textSize* alors que sa couleur est définie en valeur hexadécimale par l'attribut *android:textColor*. La Figure 155 est la visualisation correspondante au code de la Figure 154.

[9] https://developer.android.com/reference/android/widget/package-summary.html

```
<TextView
    android:id="@+id/welcomeText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/WelcomeOnApp"
    android:textSize="16sp"
    android:textColor="#00206d" />
```

Figure 154 Widget TextView

Figure 155 Visualisation du widget TextView

Le widget *Button* permet d'afficher un bouton cliquable. Ce widget hérite de *TextView*, il peut donc comporter les mêmes attributs. La Figure 156 représente un simple bouton ayant une largeur maximale et hauteur minimal. Le texte affiché sur le bouton est défini par la valeur correspondant à l'identifiant *refreshButton* de la classe *string*. La Figure 157 est la visualisation du code de la Figure 156.

```
<Button
    android:id="@+id/refreshButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/refreshButton" />
```

Figure 156 Widget Button

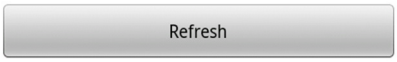


Figure 157 Visualisation du widget Button

Le widget ProgressBar permet d'afficher une barre de chargement à l'écran. Cette fonctionnalité est très utile lors de longues opérations telles que les requêtes sur internet. Ce widget peut se présenter sous la forme d'une barre de chargement indiquant le pourcentage de l'action terminée ou sous la forme d'une *spinning wheel* ne donnant aucune indication sur l'état de l'attente. L'attribut *style* permet de définir quel est le type d'affichage que l'on souhaite.

```
<ProgressBar
    android:id="@+id/progress"
    style="?android:attr/progressBarStyleSmall"
    android:layout_width="30dip"
    android:layout_height="30dip"
    android:layout_margin="10dip" >
```

Figure 158 Widget ProgressBar

Figure 159 Visualisation du widget ProgressBar

9.7.3 Listes et adaptateur

L'application créée affiche les sensors associés à l'utilisateur sous forme de liste. Cet affichage est un moyen classique, propre et efficace d'afficher des informations. La gestion des listes sur Android est divisée en trois parties distinctes: une liste de données, un adaptateur et une vue.

La première partie est la liste de données en Java. Il n'y a rien de spécial dans cette partie car la programmation Android s'effectue avec du code Java standard. Les données peuvent alors être de n'importe quel type et dans n'importe quel type de liste. Dans l'application créée, la liste est de type

List<Sensor>, il s'agit alors d'une liste comprenant des objets d'une classe que nous avons défini. C'est ensuite à l'adaptateur de traiter cette liste pour en extraire les informations voulues.

La deuxième partie est composée d'un adaptateur qui a le rôle de gérer les données. Son rôle n'est donc pas de s'occuper de l'affichage ou de l'interaction avec l'utilisateur. L'adaptateur va récupérer la liste des données et la traiter afin de passer les différents éléments à l'affichage. C'est l'adaptateur qui définira quelles données de la liste doivent être passées à l'affichage. L'adaptateur a le rôle de peupler toutes les vues correspondant aux éléments de la liste. Il traite la liste en tant que données alors que la vue traite la liste en tant que vue.

La troisième partie est la vue de la liste. Son rôle est de créer une interface graphique représentant la liste et de gérer les interactions de l'utilisateur sur celle-ci. L'adaptateur peuple une vue pour chaque élément de la liste. C'est ensuite à la vue de regrouper tous ces éléments afin de les afficher dans un vue de type *ListView*. Les interactions avec l'utilisateur sont gérées par la vue. Comme avec les autres vues et widgets, les interactions avec l'utilisateur sont effectuées grâce à des méthodes de callback comme décrit dans le chapitre *9.7.4 Gestion des évènements*.

9.7.4 Gestion des évènements

Les widgets permettent à l'utilisateur de voir des informations ou des options que nous lui proposons. Il est ensuite nécessaire que l'utilisateur puisse interagir avec l'interface graphique proposée pour sélectionner parmi les options que nous lui soumettons. Lorsque l'utilisateur interagit avec l'interface graphique en cliquant sur un bouton par exemple, Android crée un évènement qui est destiné au code Java pour son traitement. La récupération de l'évènement se fait grâce à un listener qui a le rôle de détecter quand un évènement est levé et de le traiter. Les évènements peuvent être de beaucoup de types différents. Chacun de ces évènements sera associé à une méthode de callback bien précise sur le listener. Ces méthodes ne seront pas présentées ici, pour plus d'informations, voir la documentation officielle d'Android[10]. Les noms des méthodes de callback sont assez explicites pour pouvoir comprendre le code sans explications spécifiques.

Un objet listener est une interface qui possède certaines méthodes de callback qui devront être surchargées pour le traitement de l'évènement. Il est nécessaire d'associer les objets listener à une vue pour que les évènements provenant de celle-ci soit redirigés vers le bon callback. Cette association se fait grâce à une méthode du type *setOn<typeEvenement>Listener (On<typeEvenement>Listener listener)* comme le montre la Figure 160. On voit qu'Eclipse souligne en rouge l'instanciation du listener. Ceci est tout à fait normal car il est nécessaire de surcharger au minimum une fonction afin que ce code soit valide.

```
Button refreshButton = (Button) findViewById(R.id.refreshButton);
OnClickListener listener = new OnClickListener();
refreshButton.setOnClickListener(listener);
```

Figure 160 Association d'une vue et d'un listener

Le moyen le plus simple pour surcharger uniquement la méthode que l'on souhaite utiliser est la classe anonyme. Cette méthode nous permet de spécifier le comportement voulu pour la méthode de callback qui nous intéresse. La Figure 161 illustre la gestion d'un évènement de click grâce à une classe anonyme. Dès que l'utilisateur cliquera sur le bouton possédant l'identifiant *refreshButton*, la

[10] https://developer.android.com/guide/topics/ui/ui-events.html

méthode *onClick* sera exécutée et appellera *updateValues*. La classe Toast permet de créer des affichages temporaires comme le montre la Figure 162 où l'affichage du texte *Update* est effectué dans un objet Toast.

```
Button refreshButton = (Button) findViewById(R.id.refreshButton);
refreshButton.setOnClickListener(new OnClickListener(){
    @Override
    public void onClick(View v) {
        // Perform action on click
        updateValues();
        Toast.makeText(MainActivity.this, "Update", Toast.LENGTH_SHORT).show();
    }
});
```

Figure 161 Gestion d'un évènement avec une classe anonyme

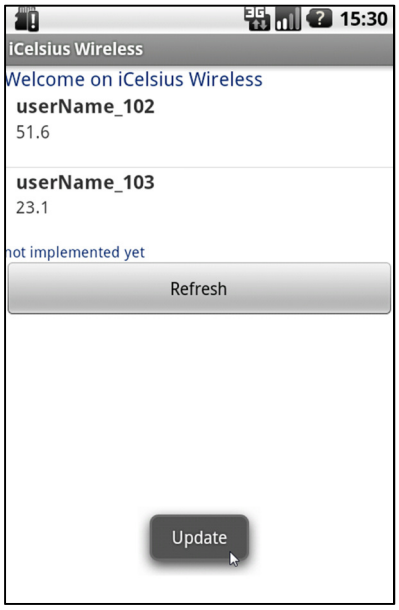


Figure 162 Affichage d'un objet Toast

9.7.5 Layout

Les layouts sont les fichiers contenant les différentes vues et widgets de l'application. Pour que les pages de l'application soient différentes, il est nécessaire de créer plusieurs fichiers de layout. Ces layouts peuvent être de différents types selon le rendu souhaité. Ces types sont présentés dans les paragraphes suivant.

Le layout *LinearLayout* place toutes les vues qu'il contient de manière linéaire dans l'espace de l'application. Il est possible de spécifier si l'on souhaite que la ligne soit considérée comme horizontale (composants placés de gauche à droite) ou verticale (composants placer de haut en bas) via l'attribut *android:orientation*. Le positionnement des différentes vues est ensuite effectué grâce aux attributs *layout_width* et *layout_hight* des vues et du layout. Sur la Figure 162 on voit que les composants sont placés de manière linéaire. Le layout utilisé est alors *linearLayout* avec l'attribut *android:orientation* valant *vertical*. Ce layout est composée d'une zone de texte, d'une liste, d'une deuxième zone de texte et finalement d'un bouton. Chacun de ces éléments déclare l'attribut *layout_width* avec la valeur *fill_parent*, ils prendront alors toute la largeur de l'écran et seront alors placés chacun sur une ligne différente. Ce layout a été utilisé lors de la phase de développement de

l'application. Il est en effet très rapide de rajouter un élément afin de visualiser et tester les fonctionnalités ajoutées.

Le layout *ReletiveLayout* permet de placer les vues qu'il contient les unes par rapport aux autres. Il est alors possible de placer une vue en fonction de la position d'un autre vue. Il est également possible de placer les vues en fonction du layout lui-même. Ce layout est donc très puissant et de plus, il utilise moins de ressources que les autres layouts. Cependant, une modification de l'interface graphique peut vite engendrer une restructuration globale des vues. Si les vues sont mal gérées, il est possible qu'elles se retrouvent superposées dans l'interface graphique. La Figure 163 est la page principale de l'application. Celle-ci est créée avec un layout de type *RelativeLayout*. Cette page ne contient que cinq vues différentes, il est alors possible de les gérer facilement via ce type de layout. Le texte du haut est positionné en haut du layout grâce à l'attribut *layout_alignParentTop*. Le bouton de rafraichissement est placé en bas du layout grâce à l'attribut *layout_alignParentBottom*. La date de dernière mise à jour et l'état de connexion sont ensuite placés en dessus du bouton de rafraîchissement grâce à l'attribut *layout_above* suivi de l'identifiant de la vue correspondant au bouton. La date de mise à jour est alignée à gauche du layout grâce à l'attribut *layout_alignParentLeft*. Finalement, la liste de tous les sensors est située entre le texte du haut et le texte du bas grâce aux attributs *layout_below* et *layout_above*.

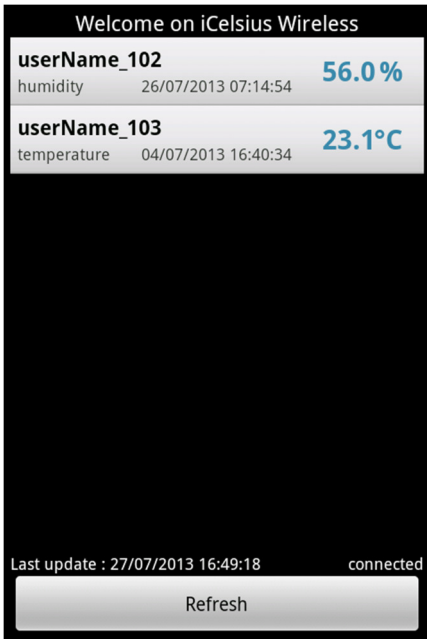


Figure 163 Page principale de l'application avec RelativeLayout

Il existe encore les layouts *TableLayout*, *FrameLayout* et *ScrollLayout*. Comme ils ne sont pas utilisés dans l'application créée, ils ne seront pas présentés dans ce document.

9.7.6 Menu d'option

Tous les smartphones Android sont équipés d'un bouton d'option, que celui-ci soit physique ou digital. Ce bouton permet d'afficher des options supplémentaires qui sont proposées à l'utilisateur. L'avantage d'un bouton menu est d'économiser de la place sur l'écran lorsque l'utilisateur ne souhaite pas voir les différentes options qui lui sont proposées.

La création d'un menu se fait via un fichier XML ayant le nœud racine se nommant *menu*. Chaque option peut ensuite être ajoutée via la balise *item*. Dans le menu de l'application créée, la pression de la touche menu ouvre un petit menu contenant deux options. L'utilisateur a alors le choix de quitter l'application ou d'accéder à un deuxième menu de settings. La Figure 164 représente le premier menu qui est proposé à l'utilisateur lorsque celui-ci presse sur le bouton menu d'Android. Les images du menu sont récupérées gratuitement et légalement car elles sont contenues dans le SDK fourni lors de l'installation du plug-in pour Eclipse. Toutes les icones disponibles pour tous les types d'écrans sont disponibles avec le chemin suivant : *<SDK_path>\platforms\android-7\data\res\drawable-<screenSize>.*

Figure 164 Premier menu d'options

Le bouton *Exit* permet à l'utilisateur de quitter l'application. Cette fonctionnalité fait appel à la méthode *finish* qui appellera le callback *onDestroy* avant d'être détruite. Le bouton *Settings* a pour action de fermer ce petit menu et d'en ouvrir un deuxième, plus grand, qui contient une liste d'options. La Figure 165 représente ce menu. L'application créée n'implémente pas encore les différentes options proposées par ce menu. Lorsque l'utilisateur cliquera sur une des options, un Toast sera affiché lui indiquant que l'option n'est pas implémentée.

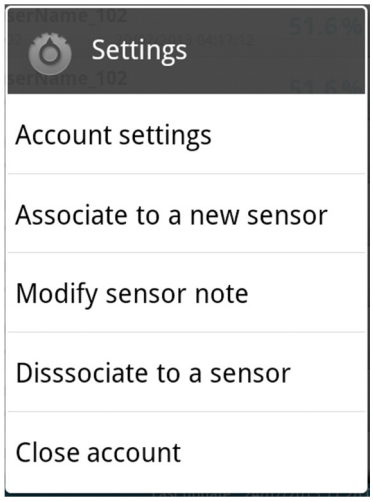


Figure 165 Options du menu Settings

La gestion des interactions de l'utilisateur sur le menu passe par des callback par défaut. Lorsque l'utilisateur appuie sur le bouton menu, le callback *onCreateOptionMenu* est appelé. Dans l'application créée, cette méthode assigne une image au sous-menu *Settings*. Il n'est effectivement pas possible de définir une image pour un sous-menu via le fichier XML. Lorsque l'utilisateur appuie sur un des boutons du menu ou du sous-menu, la méthode de callback *onOptionItemSelected* est appelée. Cette méthode prend en paramètre un *item* de menu afin qu'il soit possible de définir sur quel *item* a appuyé l'utilisateur.

9.8 Présentation de l'application

9.8.1 Package Sensor

La page principale de l'application est une liste présentant les différents sensors de l'utilisateur. Cette liste contient les dernières valeurs reçues de chaque sensor. Pour simplifier la gestion de l'application, une classe contenant toutes les informations sur chaque sensor a été créée. Ces champs sont présentés dans la Figure 166.

```
/** The unique identifier of the sensor in the datastore */
private String sensorID;

/** The note set by the user to recognize his sensor */
private String sensorNote;

/** The type of value. This type is a String. So this application support all type of data if
 * the value and the unit of it is given in string */
private String type;

/** The date of the value. This date must be in UNIX format, the number of seconds from the 1er January 1970 */
private Long   date;

/** The last value of the sensor */
private String value;

/** The unit of the last value of the sensor */
private String unit;

/** The RSSI signal of the sensor. This String must contains the unit after the value*/
private String rssi;

/** The battery level of the sensor. This String must contains the unit after the value*/
private String battery;
```

Figure 166 Champs de la classe Sensor

Le champ *sensorID* contient l'identifiant unique du sensor. Ce champ permet alors par la suite de communiquer au serveur un numéro permettant d'identifier le sensor de manière univoque. Le champ *sensorNote* est défini par l'utilisateur lors de l'association du sensor et de son compte. Il est utilisé pour que l'utilisateur puisse reconnaitre le sensor. Les champs *rssi* et *battery* sont des String qui doivent contenir l'unité. La constitution de ces champs sera alors *<valeur_numérique><unité>*. La date de la mesure est sauvegardée au format UNIX. Il est alors nécessaire de formater cette date avant de l'afficher.

Afin de mettre facilement les objets de type *Sensor* dans une liste qui sera affichée à l'utilisateur, il est nécessaire de créer un adaptateur. Les listes et adaptateurs sont présentés dans le chapitre *9.7.3 Listes et adaptateur*. La classe *SensorListAdapter* implémente l'adaptateur pour les objets de type *Sensor*. Le constructeur de cette classe prend en paramètre une liste de *Sensor*. Il est alors facile de lier une liste de *Sensor* à un adaptateur dans le programme. Cet adaptateur peuple ensuite les vues de chaque élément qui sera affiché à l'utilisateur. Comme les données de date sont contenues en format UNIX, il est nécessaire de passer un objet *SimpleDateFormat* en paramètre afin de formater les dates avant l'affichage.

```
// Set adapter to the list
SensorListAdapter adapter = new SensorListAdapter(this, sensorsObjectList, sdf);
list.setAdapter(adapter);
```

Figure 167 Adaptateur d'une liste de Sensor

La classe *SensorsComparison* permet d'effectuer le tri d'une collection selon la date des valeurs. La date la plus récente sera placée au début de la liste. Ce comparateur permet de trier les *Sensors* afin que la dernière valeur reçue soit celle qui sera affichée tout en haut de la liste dans l'application Android.

```
// Sort sensor by date to have the newer information in the top of the list
Collections.sort(sensorsObjectList, Collections.reverseOrder(new SensorsComparison()));
```

Figure 168 Tri de la liste de Sensor

9.8.2 Classe UpdateThread

La classe *UpdateThread* permet d'exécuter des requêtes HTTP sur le serveur de Google dans un thread séparé du thread principal. Cette classe est définie en tant que classe interne à l'activité pour éviter des problèmes de fuite mémoire. La gestion du thread est effectuée grâce à la classe *AsyncTask*. L'action exécutée par la classe *UpdateThread* est associée au bouton de rafraichissement de l'application. Ce bouton permet à l'utilisateur de demander des nouvelles données au serveur.

La méthode *onPreExecute* de cette classe a pour unique rôle de désactiver le bouton de rafraichissement et d'afficher un widget *ProgressBar* par-dessus le bouton.

La méthode *doInBackground* aura pour seule instruction l'appel à la méthode *getNewSensorsData*. Cette méthode envoie une requête HTTP au serveur et récupère la réponse reçue. Cette réponse est alors enregistrée dans un String qui est renvoyé en valeur de retour de la fonction *getNewSensorsData*. Ce String est ensuite affecté au champ *receivedData* qui est un champ privé de la classe *UpdateThread*.

La méthode *onPostExecute* vérifie que le String reçu du serveur ne soit pas *null*. Si ce n'est pas le cas, ce String est passé à la méthode *setListValues* pour mettre à jour les valeurs dans la liste affichée à l'utilisateur. Le bouton de mise à jour est ensuite réactivé et le spinner d'attente désactivé.

9.8.3 JSON

Les données reçues du serveur sont représentées par des String. Les données sont formatées en JSON. Il est alors possible de les récupérer avec une librairie spécifique à JSON. L'application *iCelsius Wireless* utilise la librairie *org.json* afin de parser la chaine de caractère reçue. Le format du String reçu correspond à l'exemple de la Figure 169.

```
{"lastUpdate":"1374792190",
 "dateFormat":"dd/MM/yyyy HH:mm:ss",
 "sensorsInformation": [
 {   "sensorNote":"userName_102",
     "sensorID":"102",
     "type":"CO2",
     "date":"1374319032",
     "value":"51.6",
     "unit":"%",
     "rssi":"0dBm",
     "battery":"0.0V" } ,
 {   "sensorNote":"userName_103",
     "sensorID":"103",
     "type":"temperature",
     "date":"1372981234",
     "value":"23.1",
     "unit":"°C",
     "rssi":"0dBm",
     "battery":"3.6V" }
 ]
}
```


Figure 169 Format des données JSON reçues

A partir de ce String, il est possible de créer un objet de type *JSONObject* afin de récupérer les différents champs. La Figure 170 représente la récupération d'un champ de type Long. Pour le String de la Figure 169, il serait par exemple possible d'employer ces deux lignes de code avec le paramètre page comprenant ce String et le paramètre *name* valant *lastUpdate*. Les fonctions *getJSONLongValue* et *getJSONStringValue* récupère une valeur de type Long et String grâce à ce principe.

```
json = new JSONObject(page); // Create object from the given JSON String
return json.getLong(name);   // Get the value
```

Figure 170 Récupération d'un champ de type Long

La récupération des informations de chaque Sensor est un peu plus compliquée car elles sont fournies dans un tableau. La Figure 171 représente le code écrit afin de récupérer ces Sensors. Le String contenant les informations JSON est d'abord passé au constructeur d'un objet *JSONObject*. On récupère ensuite le tableau de tous les Sensors dans un objet de type *JSONArray*. Chaque sensor est ensuite inséré dans une liste grâce à une boucle for parcourant chaque élément du tableau JSON récupéré. La fonction *getSensors* prend en paramètre un String JSON et retourne une liste de Sensor grâce au code de la Figure 171.

```
// Create a JSONObject from the String
json = new JSONObject(page);

// Get the array containing all sensors
JSONArray contents = json.getJSONArray("sensorsInformation");

JSONObject[] sensors = new JSONObject[json.getJSONArray("sensorsInformation").length()];
for (int i = 0; i < contents.length(); i++) {
    // Create a sensor from JSON data
    sensors[i] = contents.getJSONObject(i);
    currentSensor = new Sensor(sensors[i].getString("sensorID"),
            sensors[i].getString("sensorNote"),
            sensors[i].getString("type"),
            sensors[i].getLong("date"),
            sensors[i].getString("value"),
            sensors[i].getString("unit"),
            sensors[i].getString("rssi"),
            sensors[i].getString("battery"));

    sensorsList.add(currentSensor);
}
return sensorsList;
```

Figure 171 Récupération des Sensors depuis des données en JSON

9.8.4 Communication avec le serveur

Pour que l'application reçoive des nouvelles données du serveur, elle envoie une requête HTTP GET sur le serveur. Cette requête est adressée à une Servlet renvoyant une page contenant des données formatées en JSON. Cette Servlet est accessible via l'adresse *http://apps-test-tb2013.appspot.com/user/mobile*. La page renvoyée contient les dernières valeurs reçues de chaque sensors associés à l'utilisateur courant. L'application doit ensuite récupérer ces valeurs afin de les afficher à l'utilisateur.

Les données reçues, en format JSON, doivent respecter la structure présentée à la Figure 172. Chaque champ est expliqué dans le Tableau 19.


```
{"lastUpdate":"1374792190",
 "dateFormat":"dd/MM/yyyy HH:mm:ss",
 "sensorsInformation": [
 {    "sensorNote":"userName_102",
      "sensorID":"102",
      "type":"CO2",
      "date":"1374319032",
      "value":"51.6",
      "unit":"%",
      "rssi":"0dBm",
      "battery":"0.0V" } ,
 {    "sensorNote":"userName_103",
      "sensorID":"103",
      "type":"temperature",
      "date":"1372981234",
      "value":"23.1",
      "unit":"°C",
      "rssi":"0dBm",
      "battery":"3.6V" }
 ]
}
```


Figure 172 Format des informations reçues du serveur

lastUpdate	Cette valeur représente une date en format UNIX. La valeur fournie par ce champ indique la date de la réponse provenant du serveur. Elle permet d'obtenir la date de mise à jour des valeurs contenues dans la réponse.
dateFormat	Cette valeur est un String permettant de formater les dates transmises en format UNIX. Ce champ dépend du compte avec lequel l'utilisateur est connecté. Le champ est défini par l'utilisateur dans l'application web afin de formater toutes les dates affichées.
sensorsInformation	Ce champ contient un tableau de valeurs. Chacune de ces valeurs représente un sensor associé à l'utilisateur. Les champs utilisés pour chaque sensors sont présentés ci-dessous.
sensorNote	Le nom du sensor courant. Ce nom est défini par l'utilisateur lors qu'il s'associe au sensor. La valeur de ce champ est dépendante de l'utilisateur courant.
sensorID	L'identifiant unique du sensor. Cette valeur ne sera pas utile pour l'utilisateur mais pour l'application elle-même. Elle permet de reconnaitre les différents sensors de manière univoque sur le serveur comme sur l'application.
type	Le type de la valeur transmise. Ce type est un simple String. Tous les types de valeurs sont alors supportés par l'application.
date	La date de la valeur de mesure transmise. Cette date est en format UNIX. L'affichage de la date se fera selon le champ *dateFormat*. Ce

	champ doit pouvoir être récupéré en tant que *Long*. Tous les sensors seront ensuite triés d'après la valeur de ce champ.
value	Ce champ est la valeur mesurée transmise à l'application. Cette valeur est la dernière mesure reçue du sensor sur le serveur.
unit	Ce champ est un String comprenant l'unité de la valeur reçue. L'unité peut être n'importe quel String. Ce champ, ainsi que le champ type, permettent à l'application de s'adapter automatiquement en cas d'ajout de types de données.
rssi	Ce champ indique le dernier niveau du signal provenant du sensor. Il n'est pas lié à la mesure mais au sensor. L'unité de ce champ doit être contenue dans le String.
battery	Ce champ indique le dernier niveau connu de batterie du sensor. Il n'est pas lié à la mesure mais au sensor. L'unité de ce champ doit être contenue dans le String.

Tableau 19 Champs des informations JSON reçues du serveur

Les informations reçues sont ensuite affichées à l'utilisateur. Celui-ci peut sélectionner un sensor particulier pour obtenir plus de données du sensor. Pour obtenir plus de données, l'application Android envoie une requête HTTP POST sur une Servlet du serveur. Cette Servlet est accessible via l'adresse *http://apps-test-tb2013.appspot.com/user/mobile*.

Lors de l'envoi de la requête POST sur cette URL, il est nécessaire de rajouter deux paramètres indiquant les valeurs que l'on souhaite obtenir en réponse. Le premier paramètre, identifié par *selectedSensor*, est l'identifiant du sensor sélectionné. Le deuxième paramètre, identifié par *selectedType*, est le type de valeur que l'on souhaite observer. La Figure 173 représente l'URL pour l'envoi de données sur le serveur, avec les deux paramètres en fin d'URL.

```
http://apps-test-tb2013.appspot.com/user/mobile?selectedSensor=103&selectedType=temperature
```

Figure 173 Paramètre de requête sur le serveur

Le serveur renvoi ensuite une réponse contenant les informations demandées. Ces informations sont contenues dans une page possédant des données au format JSON. Le format JSON reçu correspond à la Figure 174. Cette figure ne comporte que deux valeurs pour le sensor sélectionné. Dans la vraie application, un grand nombre de valeurs seront présentes dans ce JSON.

```
{"sensor":"userName_103",
 "rssi":"0dBm",
 "battery":"0.0V,"
 "dateFormat":"dd/MM/yyyy HH:mm:ss",
 "sensorData": [
    {
     "date":"1370974390",
     "type":"temperature",
     "data":"34.0",
     "unit":"°C"
    },

      <quantify of other data>

    {
     "date":"1368571675",
     "type":"temperature",
     "data":"10.0",
     "unit":"°C"
    }
]}
```

Figure 174 Format JSON reçu du serveur après un POST

sensor	Le nom du sensor. Ce nom est défini par l'utilisateur lors de l'association du sensor et de son compte. Ce champ indique à quel sensor sont associées les données contenues dans la réponse.
rssi	Ce champ indique le dernier niveau du signal provenant du sensor. Il n'est pas lié à la mesure mais au sensor. L'unité de ce champ doit être contenue dans le String.
battery	Ce champ indique le dernier niveau de batterie fourni par le sensor. Il n'est pas lié à la mesure mais au sensor. L'unité de ce champ doit être contenue dans le String.
dateFormat	Cette valeur est un String permettant de formater les dates transmises en format UNIX. Ce champ dépend du compte avec lequel l'utilisateur est connecté. Le champ est défini par l'utilisateur dans l'application web afin de formater toutes les dates affichées.
sensorData	Ce champ contient un tableau de valeurs. Chacune de ces valeurs représente une donnée d'un certain type provenant du sensor sélectionné. Les champs utilisés pour chaque sensors sont présentés ci-dessous.
date	La date de la valeur de mesure courante. Cette date est en format UNIX. L'affichage de la date se fera selon le champ *dateFormat*. Ce champ doit pouvoir être récupéré en tant que *Long*. Toutes les données seront ensuite triées d'après la valeur de ce champ.
type	Le type de la valeur transmise. Ce type est un simple String indiquant le type de la donnée courante.
data	Cette valeur est la donnée courante. Ce champ contient la valeur de la mesure du type donné pour la date donnée.
unit	Ce champ est un String comprenant l'unité de la valeur reçue. L'unité peut être n'importe quel String.

Figure 175 Champs des informations JSON reçues du serveur après un POST

9.8.5 Gestion des comptes utilisateur

L'application web créée utilise les comptes *Google Account* afin d'identifier les différents utilisateurs sur le site. Tous les utilisateurs d'Android possèdent généralement un de ces comptes qui leur permet notamment d'accéder aux applications sur le Play Store. Ce compte est habituellement enregistré sur l'appareil Android afin que l'utilisateur n'ait pas besoin de se relogger à chaque manipulation nécessitant le compte.

L'application créée devra donc utiliser ce compte enregistré afin de pouvoir s'identifier auprès du serveur. Il est nécessaire de créer un cookie de connexion afin que les communications entre le serveur et l'application Android se fassent correctement. Cette partie de l'application n'a pas pu être implémentée à la date du rendu du travail. Cependant, le code implémentant cette partie de l'application devrait pouvoir s'intégrer facilement dans l'application. Le package *aeauth* contient les essais de codes effectués jusqu'à présent.

En attendant de résoudre ce problème, l'application devait quand même être fonctionnelle afin de pouvoir effectuer des démonstrations. Il a donc été nécessaire de créer une Servlet supplémentaire sur le serveur afin de gérer l'application sans authentification. Cette Servlet est contenue dans le fichier *UnauthMobile.java* présent sur l'application serveur. Elle permet d'accéder à tous les sensors

associés à l'utilisateur *TB.2013.TinguelyJoel*, un compte créé spécialement pour le travail de Bachelor. Cette Servlet est présentée au chapitre *7.10.1.5 Package applicationGeneralServlet*. Elle est accessible depuis le lien *http://apps-test-tb2013.appspot.com/unauthMobile*. C'est donc sur cette URL que l'application Android effectue des requêtes pour obtenir des informations.

9.8.6 Principales méthodes de l'application

9.8.6.1 onCreate

Lors du démarrage de l'application, la méthode *onCreate* commence par inflater les différentes vues. Cette opération consiste à transformer des données XML en objet Java. L'inflatage est une opération qui prend passablement de temps. L'application ne le fait alors qu'une seule est unique fois dans la méthode *onCreate*. Les objets créés sont ensuite sauvegardés dans des variables globales de l'application.

```
// Set view
setContentView(R.layout.activity_main);
refreshButton    = (Button)      findViewById(R.id.refreshButton);
progressSpinner  = (ProgressBar) findViewById(R.id.progress);
lastUpdateText   = (TextView)    findViewById(R.id.lastUpdate);
connectionState  = (TextView)    findViewById(R.id.connectionState);
list             = (ListView)    findViewById(R.id.list);
noSensorText     = (TextView)    findViewById(R.id.noSensorMessage);
```

Figure 176 Inflater les vues dans onCreate

La méthode de callback associée au bouton de rafraichissement est ensuite déclarée dans une classe anonyme. Le rôle de ce callback est d'exécuter le thread gérant la mise à jour des valeurs à chaque pression sur le bouton.

```
// Add refresh button and his listener
refreshButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        // On click, create a new thread to update values
        mProgress = new UpdateThread(MainActivity.this);
        mProgress.execute();
    }
});
```

Figure 177 Déclaration de la méthode de callback du bouton refresh

La méthode *onCreate* essaie ensuite de récupérer des données préalablement sauvegardées lors d'une autre instance de l'application. La méthode *restoreStringData* permet de récupérer une donnée dans l'objet *SharedPreference* d'Android. Cette donnée est identifiée par un String qui est fourni en paramètre de la méthode. La valeur du String est une constante afin s'assurer la cohérence du code entre la sauvegarde et la récupération de la valeur. Si des données sont retournées de la fonction *restoreStringData*, elles sont affichées dans la liste grâce à la méthode *setListValues*.

```
// Get lastData saved when the application exit and check if it contains information
String lastData = restoreStringData(LAST_DATA_FROM_SERVER);
if (lastData != null) {
    setListValues(lastData);
}
```

Figure 178 Récupération de la valeur sauvegardée

Avant de se terminer, la méthode *onCreate* exécute le thread permettant de mettre les données à jour. Les nouvelles données seront alors récupérées depuis le serveur et la liste des sensors sera

automatiquement mise à jour. Il est important de multithreader cette partie du code. Effectivement, en effectuant la requête dans le thread principale, l'utilisateur ne verrait qu'un écran noir en attendant la réponse du serveur. Grâce à ce thread, l'utilisateur a accès aux dernières données sauvegardées pendant le chargement des nouvelles valeurs.

```
// Create a new thread to update values
mProgress = new UpdateThread(MainActivity.this);
mProgress.execute();
```

Figure 179 Exécution du thread de mise à jour

9.8.6.2 onStop

La méthode *onStop* est appelée lorsque l'activité n'est plus visible sur l'écran de l'appareil. L'activité peut alors être détruite à tout moment par Android. C'est donc le bon endroit pour effectuer des sauvegardes de données pour une prochaine exécution.

L'activité garde en mémoire toutes les données nécessaires pour l'affichage de la page d'accueil lors du démarrage de l'application. Toutes ces informations sont contenues dans la réponse du serveur suite à une requête GET. La méthode *onStop* aura donc comme unique instruction d'appeler la méthode *saveStringData*. Cette méthode sauvegarde le String passé en paramètre dans l'objet *SharedPreference* d'Android. Le String *sensorJSONList* est une variable globale contenant la dernière réponse reçue du serveur suite à une requête GET. La valeur sauvegardée est donc un String contenant toutes les valeurs de la page d'accueil au format JSON.

```
// Save the last data received from server
saveStringData(LAST_DATA_FROM_SERVER, sensorJSONList);
```

Figure 180 Contenu de la méthode onStop

9.8.6.3 onCreateOptionsMenu

Cette méthode est appelée lorsque l'utilisateur presse le bouton menu. Cette méthode inflate le menu déclaré dans le fichier menu.xml afin d'obtenir un objet Java. Cette méthode ajoute ensuite une icône au sous-menu Settings. Cette opération n'est pas réalisable dans le document XML.

```
// Inflater to transform a XML menu to an object menu
MenuInflater inflater = getMenuInflater();

// Create object menu from XML menu
inflater.inflate(R.menu.menu, menu);

// Add an icon for the sub menu (unable to do this action in XML)
menu.getItem(0).getSubMenu().setHeaderIcon(R.drawable.ic_menu_preferences);
```

Figure 181 Contenu de la méthode onCreateOptionsMenu

9.8.6.4 onOptionsItemSelected

Cette méthode est appelée lorsque l'utilisateur sélectionne une des options dans le menu de l'activité. Ce callback récupère ensuite l'item sélectionné et effectue des instructions spécifiques à l'item grâce à une instruction de choix multiple (switch, case).

```
// Check which item was selected
switch (item.getItemId()) {
case R.id.option:
    // Nothing to do, the user will automatically see the settings menu
    return true;
case R.id.exit:
    saveStringData(LAST_DATA_FROM_SERVER, sensorJSONList); // Save the last data received from server
    finish(); // Finish the application
    return true;
}
```

Figure 182 Contenu de la méthode onOptionsItemSelected

9.8.6.5 setListValues

Cette méthode permet de mettre à jour la liste des sensors de la page principale de l'application. Les nouvelles valeurs à mettre dans la liste doivent être passée à cette fonction sous forme d'un String JSON. Les données seront alors présentées en format JSON selon l'exemple de la Figure 172.

setListValues récupère d'abord les paramètres *lastUpdate* et *dateFormat* du String JSON. Si le paramètre *lastUpdate* n'est pas valide, le temps courant du système Android est pris. Si le paramètre *dateFormat* n'est pas valide, le code récupère un format par défaut défini dans le fichier *string.xml*. L'affichage de la dernière mise à jour est ensuite rafraichi par la dernière valeur reçue. La valeur reçue étant au format UNIX, il est nécessaire de la formater avec un objet de type *SimpleDateFormat*. En utilisant ce principe, l'heure est automatiquement ajustée pour être affichée en heure locale où se situe l'appareil Android.

Le code transforme ensuite le String contenant les informations JSON en liste d'objets de type *Sensor*. Cette opération est effectuée en appelant la méthode *getSensors*. Si la liste retournée ne contient aucun Sensor, l'affichage de la liste disparait au profit d'un message indiquant qu'aucun sensor n'est présent. Si des sensors sont présents dans la liste retournée par la méthode *getSensors*, les objets *Sensor* sont triés via le comparateur défini par la classe *sensorComparison*. Grâce à ce tri, les valeurs mesurées les plus récemment seront placées en haut de la liste des sensors.

La méthode *setListValues* crée ensuite un adaptateur de classe *SensorListAdapter* et le défini comme adaptateur de la liste de sensor. La liste sera alors créée et formatée selon l'adaptateur. Une méthode de callback est ensuite définie pour gérer l'interaction avec l'utilisateur. Lorsque l'utilisateur cliquera sur un des sensors affichés dans la liste, la méthode *onItemClick* sera automatiquement appelée. Cette méthode récupère l'item sélectionné et affiche une boite de dialogue comprenant des informations supplémentaires à propos du sensor sélectionné. Ces informations supplémentaires sont actuellement formatées pour un unique appareil grâce à des espaces. Pour une utilisation sur plusieurs appareils, il est nécessaire de créer une boite de dialogue personnalisée afin de pouvoir insérer les informations dans un tableau. Cette fonctionnalité n'a pas été implémentée par manque de temps.

9.8.7 Structure du projet

Lors de la création du projet, Eclipse nous crée une structure complète pour une application Android. Le répertoire *src* contient tous les fichiers sources Java. Nous y retrouvons les paquetages créés ainsi que les fichiers **.java* contenant le code source de l'application.

Le package *com.icelsiuswireless* contient les activités de l'application. L'application ne comprenant qu'une activité, ce package ne contient alors que le fichier *MainActivity.java*. Le package *com.icelsiusworeless.aeauth* contient les essais de code pour la mise en place de l'authentification

des utilisateurs. Le fichier *AccountList.java* permet d'afficher une liste à l'utilisateur lui permettant de sélectionner un compte parmi tous ceux associés à l'appareil Android. Le fichier *AppInfo.java* essaie d'effectuer l'authentification avec le serveur. Le package *com.icelsiuswireless.sensor* contient tous les fichiers Java permettant la gestion des objets de type *Sensor*. Le fichier *Sensor.java* contient la déclaration de la classe Sensor ainsi que les getter et setter associés à tous les champs. Le fichier *SensorListAdapter.java* contient l'adaptateur associé à la liste affichant les *Sensor*. Finalement, le fichier SensorsComparison.java est le comparateur utilisé pour trier la liste de sensors.

Le répertoire *res* est le dossier qui contient toutes les ressources utilisées par l'application. On y retrouve les différents types de ressources présentés dans le chapitre *9.6.3 Ressources*.

Les dossiers *drawables* contiennent toutes les images utilisées dans l'application. Chaque dossier est suivi d'un quantificateur afin de spécifier la taille de l'écran pour lequel s'adressent les images du dossier. Les noms des fichiers images contenus dans un dossier correspondent à l'identifiant de la ressource. Il est alors possible d'accéder à une image via le nom du fichier. Les noms de fichiers ne peuvent contenir que des lettres minuscules, des chiffres et des underscores.

Le dossier *layout* contient tous les fichiers XML correspondant aux layouts de l'application. On y retrouve alors le fichier *activity_main.xml* qui définit la page principale de l'application. Le fichier *list_row.xml* définit la mise en page des *Sensor* dans la liste affichée à la page principale. Ce fichier est celui qui est utilisé par l'adaptateur pour peupler la liste. Les fichiers *app_info.xml* et *list_item.xml* sont liés à la gestion des comptes utilisateurs. Ils ne sont alors que présents dans ce dossier pour effectuer des tests.

Le dossier *values* contient des fichiers XML permettant de fournir des informations à l'application. Le fichier *strings.xml* fournit toutes les chaines de caractères utilisées pour l'interface graphique. Il est possible de formater les chaines de caractères lors de leur utilisation par le code Java. Le fichier *styles.xml* permet de spécifier un style pour les différentes vues et widgets utilisés dans l'application. Un style par défaut a été choisi pour constituer la base du graphisme. Il est ensuite possible de modifier ce style via ce fichier.

9.8.8 Rendu graphique

La page d'accueil de l'application affiche une liste de tous les Sensors associés à l'utilisateur courant. Actuellement, l'utilisateur courant est défini comme étant *TB.2013.TinguelyJoel*. La Figure 183 met en évidence les différentes informations présentes sur la page principale de l'application.

Figure 183 Présentation des différentes parties de l'application

La Figure 184 est un agrandissement d'un élément de la liste de sensors. Pour chaque élément, le nom du sensor est affiché en gras en haut à gauche de l'item. En dessous de ce nom, le type de donnée est indiqué. Au centre de l'item, la date de la mesure est affichée à la seconde près. Sur la gauche de l'élément, on retrouve la valeur de la mesure ainsi que l'unité. Toutes les valeurs mesurées seront alignées à la virgule. L'affichage des unités n'est pas optimal comme on peut le voir sur la figure ci-dessous. Ceci est dû au fait que certaines unités occupent une place plus grande que d'autre.

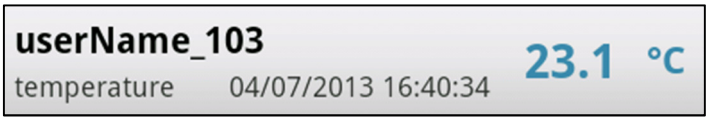


Figure 184 Elément de la liste de sensors

Comme on peut le voir grâce à la barre de défilement, la liste des sensors est scrollable. Un effet de fondu permet d'ajouter un visuel agréable à la liste. Cet effet est bien visible en bas de la Figure 185.

Figure 185 Effet de fondu sur la liste

La Figure 186 est l'état du bouton *refresh* lorsqu'un rafraichissement est en cours. Le bouton devient inactif, il n'est alors pas possible de cliquer dessus. Ceci évite que l'utilisateur lance plusieurs threads de mise à jour. Lorsqu'un thread est en cours, un spinner apparaît sur la droite du bouton. Ce spinner est un élément qui tourne indiquant que l'application est en train de travailler.

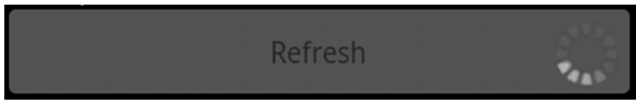


Figure 186 Bouton refresh en cours de rafraichissement

La Figure 187 est la fenêtre qui s'affiche lorsque l'utilisateur clique sur un de ses sensors. On y retrouve le nom du sensor, la valeur, le type, la date de la valeur. De plus, cette fenêtre affiche les informations relatives au RSSI et à la batterie. L'alignement des différentes valeurs de cette boite de dialogue est actuellement effectué grâce à des espaces. Cette solution a été choisie par manque de temps. Pour une utilisation future, il est nécessaire de modifier l'affichage et de créer des boites de dialogues personnalisées.

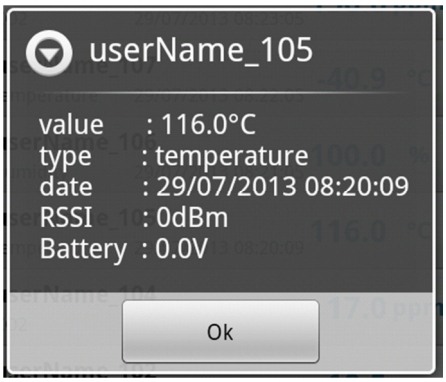


Figure 187 Informations complémentaires sur un sensor

La Figure 188 représente le menu qui s'affiche en bas de l'écran lorsque l'utilisateur appuie sur le bouton menu de son dispositif Android. Ce menu permet à l'utilisateur de quitter l'application ou d'accéder au menu des paramètres.

Figure 188 Premier menu

Le menu des paramètres est représenté à la Figure 189. L'utilisateur peut accéder à différentes fonctionnalités via ce menu. L'application créée n'implémente pas encore les fonctionnalités proposées. Lorsque l'utilisateur choisira une des options, un toast sera affiché lui indiquant que l'option n'est pas implémentée.

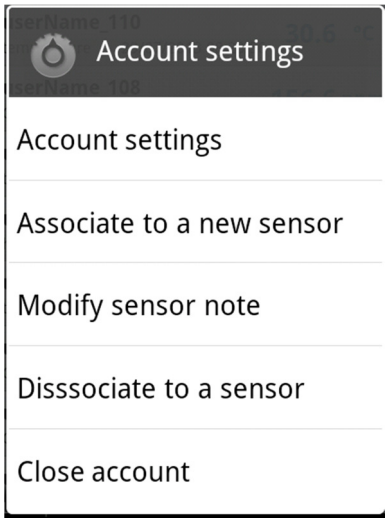


Figure 189 Menu de settings

Lorsque le device Android sur lequel fonctionne l'application n'a pas d'accès internet. Les dernières informations enregistrées sont affichées dans la liste. Le message de statuts indiquera *not connected.* Lorsque l'utilisateur essayera d'effectuer une mise à jour, le toast de la Figure 190 s'affichera.

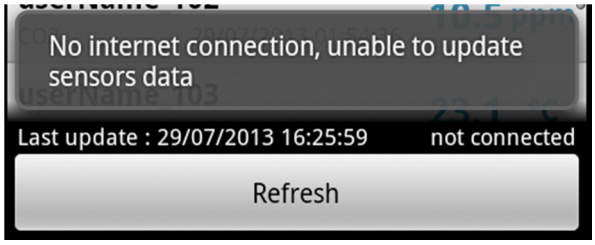


Figure 190 Pas de connexion internet

La Figure 191 est l'affichage que verra l'utilisateur lorsqu'il n'aura aucun sensor connecté à son compte. La Figure 192 est le rendu général de l'application. Le toast affiché indique que la mise à jour des valeurs vient d'être effectuée.

Figure 191 Affichage lorsqu'aucun sensor n'est associé à l'utilisateur

Figure 192 Rendu global

10 Améliorations

10.1 Application web

L'application web créée met en place la structure de base du projet. Il est maintenant nécessaire d'ajouter de nombreuses fonctionnalités afin que cette application soit pleinement exploitable.

Actuellement, l'ajout de sensor est effectué manuellement. Cette opération est alors lente et répétitive car elle doit être effectuée pour chaque sensor. Il serait intéressant de développer une fonctionnalité supplémentaire qui permettrait d'automatiser ce mécanisme. L'idée serait de créer une page supplémentaire dans la console d'administration afin de gérer les entrées dans le datastore. Cette page chargerait un fichier de données dans un format précis, CSV par exemple, et ajouterait toutes les entrées qu'il contient dans le datastore.

La classe *UserSensor* contient un champ nommé *isPremium*. Ce champ permet d'indiquer si le sensor concerné est connecté à un utilisateur qui paye l'utilisation de l'application. Pour l'instant, ce champ n'est jamais utilisé dans le code. Par la suite, il sera donc nécessaire de prendre en compte ce champ afin d'offrir des fonctionnalités supplémentaires aux utilisateurs qui paient.

Le payement est aussi un des points à développer. Les utilisateurs possédant un compte premium auront des avantages sur les autres utilisateurs. Ces avantages seront payants. Il est alors nécessaire de mettre en place un système de payement efficace et sûr afin que les utilisateurs ne rechignent pas à utiliser ce système.

Il peut être intéressant d'effectuer des backups du datastore si celui-ci devait rencontrer des problèmes. Google App Engine nous met à disposition deux méthodes afin de créer des backups. La première est spécifique à la gestion des backups[11]. Cette méthode est encore en phase expérimentale, elle pourrait alors ne plus être supportée dans une version future du SDK. La deuxième méthode pour la création de backup est de prévoir des tâches périodiques qui effectueraient des sauvegardes du datastore[12].

Contrairement à l'application Android qui gère automatiquement le décalage horaire, l'application web affiche l'heure UTC sur toutes ses pages HTML. Effectivement, le serveur ne prend actuellement pas en compte la position géographique de l'utilisateur, il ne peut alors pas définir quelle est l'heure locale de celui-ci. Il serait possible de récupérer cette information dans les paramètres de la requête. Un autre moyen est d'insérer un script s'exécutant sur la machine de l'utilisateur. Ce script aurait alors pour rôle d'afficher la date en prenant en compte le décalage horaire. Ce bout de code aurait la possibilité de communiquer avec la machine locale pour obtenir les paramètres concernant l'heure.

Lorsque l'application Android effectuera une requête POST sur le serveur pour demander des informations sur un sensor spécifique, l'application web récupère le numéro de sensor et le type de données dans les paramètres de l'URL. Il serait judicieux de contrôler ces valeurs avant de les utiliser. Ceci afin d'éviter toutes attaques CSS sur le serveur.

[11] https://developers.google.com/appengine/articles/scheduled_backups
[12] https://developers.google.com/appengine/docs/java/config/cron

Comme indiqué au chapitre *7.9.3 Nombre d'accès au datastore*, il est encore nécessaire de tester la sauvegarde des mesures dans la memCache.

Un dernier point très important est la gestion du graphisme de l'application. Actuellement, l'application est présentée avec une interface graphique très sommaire. Il sera donc nécessaire de modifier cette interface par la suite afin de rendre l'application web plus accueillante. Pour se faire, il existe certains frameworks tels que bootstrap[13] qui permettent de faciliter le travail.

10.2 Communication avec le sensor

Lorsque le sensor envoie une valeur sur le serveur, ce dernier peut retourner certaines informations au sensor. Actuellement, seul le timestamp est présent dans la réponse si la date passée avec le paquet est trop ancienne. Le sensor doit alors mettre à jour sa date afin d'envoyer les prochains paquets avec une date correcte. Il serait possible que le serveur transmette d'autres messages de la même manière. Il pourrait être intéressant que le serveur fournisse un message d'état après chaque transmission. Si la transmission ne s'est pas déroulée correctement, le sensor pourrait alors renvoyer les mêmes valeurs ou les modifier selon le message d'erreur.

Le sensor envoie actuellement une valeur par transmission. Ceci peut vite poser problème en termes de durée de vie de la batterie. Il serait possible d'implémenter l'envoi d'un certain nombre de valeurs à chaque transmission. Ces valeurs pourraient par exemple être formatées dans un String en JSON.

10.3 Application Android

Afin que l'application Android s'adapte à plusieurs langues, il est possible d'ajouter des dossiers dans le répertoire *res* de l'application. Par exemple, pour le français il serait nécessaire d'ajouter le dossier *values-fr* contenant le fichier *strings.xml*. Ce fichier sera une copie du fichier *strings.xml* du dossier *values* mais les valeurs des champs seraient remplacées par du français.

La suite de l'application Android nécessite l'utilisation d'intent afin de pouvoir utiliser plusieurs activités dans l'application. La deuxième activité à créer est celle qui s'occupera de l'affichage des informations concernant un sensor spécifique. Pour cette partie, l'application Android doit effectuer une requête POST sur le serveur. La gestion du POST est déjà mise en place sur le serveur.

L'authentification de l'appareil Android sur le serveur sera une méthode supplémentaire qui viendra se greffer au thread s'occupant de la communication avec le serveur. Il n'est normalement pas nécessaire d'effectuer de grandes modifications pour que cette authentification s'intègre dans l'activité.

[13] http://getbootstrap.com/getting-started/

11 Remarques

La documentation française de Google App Engine n'est pas à jour par rapport à la documentation en anglais. En date du 11.04.2013, trois exemples marquant sont à relever:

1) La version de JRE et du SDK de Java n'est pas la même sur la documentation française et anglaise
2) Le support pour le langage de programmation Go n'est pas encore disponible en français
3) Les quotas généraux limitant les tailles de requêtes ne sont pas les mêmes en français (10Mo) qu'en anglais (32Mo)

La documentation et les exemples de Google ne s'adressent pas à des utilisateurs néophytes. Le suivi de la présentation officiel de Google App Engine nécessite effectivement que le développeur connaisse les principes de Java EE. De nombreux tutoriels sont présents sur internet afin d'apprendre ces concepts.

La modification du code sur le sensor a été effectuée en quelques lignes de code seulement. Le plus difficile a été de trouver où rajouter ces lignes. Effectivement, une méthode était déjà présente afin de gérer l'envoi de données via HTTP. La configuration dans le fichier *config.c* s'est avérée moins évidente que prévue, delà le nombre de *define* présents dans le fichier.

Lors de développement d'application web, certains utilitaires sont très utiles. Parmi ceux-ci, le plug-in Firebug pour Firefox et l'application POSTman pour Google chrome. Le premier permet de visualiser les requêtes et réponses reçues lors de navigations sur internet alors que le deuxième permet d'effectuer ces requêtes. Sans utilitaire spécifique, il n'est pas possible d'envoyer une requête POST sur le serveur. POSTman permet d'envoyer ce type de requêtes en précisant les paramètres à envoyer.

Google propose une solution pour utiliser Google App Engine sur plusieurs plateformes telles qu'un ordinateur normal ou un device Android. Le nom de cette fonctionnalité est *Cloud Endpoints*. Ce service propose de pouvoir utiliser les classes définies dans App Engine sur plusieurs plateformes. Pour se faire, il est nécessaire d'annoter ces classes et de créer plusieurs fichiers supplémentaires. Après avoir lu la documentation et essayé d'implémenter cette solution, je suis revenu à une communication standard. Effectivement, *Cloud Endpoints* n'a pas un grand intérêt pour l'application créée. Il est plus simple de transmettre les informations sous un format standard, JSON, afin que les applications mobiles puissent communiquer avec le serveur. Cette manière de faire implique la création de plusieurs Servlets supplémentaires sur le serveur. Néanmoins cette solution me semble plus propre et respecte très clairement l'aspect client-serveur de l'application.

Ce document comporte beaucoup de titres et sous-titres. Il peut être difficile de lire un tel document sans se perdre. Cependant la recherche d'informations est simplifiée grâce à ces sous-titres. De plus, ces éléments permettent de structurer le document et d'éviter d'avoir des paragraphes de plusieurs pages.

Les codes des applications web et Android possèdent tous les deux une documentation Javadoc. L'écriture de cette documentation est un long travail mais permet de gagner du temps lors de l'utilisation ou de la modification du code. Grâce à cette documentation, il est possible de s'y retrouver facilement dans le code, sans avoir besoin de relire ce document.

12 Conclusion

Le projet rendu permet de visualiser les données de sensors sur son smartphone Android. L'unique point à modifier pour que ce mécanisme fonctionne avec tous les utilisateurs est l'authentification entre le serveur et l'appareil Android.

L'application web créée permet de gérer son compte et ses sensors. Plusieurs modifications sont encore à effectuer afin que cette application soit totalement fonctionnelle. Cependant, l'état actuel de l'application permet l'utilisation des sensors. Les données provenant des sensors sont stockées et il est possible de les visualiser sous forme de tableaux ou de graphiques. De plus, la gestion du compte comporte plusieurs options telles que le choix du format de la date ou de l'unité des valeurs de température. Si plusieurs utilisateurs sont associés à un même sensor, chaque utilisateur pourra définir son propre nom de sensor afin de le reconnaitre aisément.

L'envoi des informations du sensor vers le serveur est opérationnel. Par la suite, il est possible d'améliorer ce mécanisme si l'on souhaite optimiser la durée de vie de la batterie ou éviter la perte d'un paquet.

L'application Android créée permet de visualiser les données des sensors. Ces données peuvent être de n'importe quel type, l'application s'adaptera. Les sensors sont présentés dans une liste comprenant les informations les plus importantes sur chaque sensor. L'utilisateur peut ensuite cliquer sur un sensor pour obtenir plus d'informations sur celui-ci De plus, une structure de menu est mise en place prévoyant la suite du développement de l'application.

Le cahier des charges n'a pas été totalement rempli. L'interface web est plus complète que ce que prévoyait le cahier des charges. Par contre l'application Android ne permet actuellement pas de reconfigurer les sensors comme le prévoyait un des derniers points du cahier des charges. De même pour le dernier point qui prévoyait d'améliorer le design du site web en collaboration avec un designer d'Aginova.

Malgré ces quelques points incomplets, je suis satisfait du travail fourni pour ce projet. Effectivement, en commençant ce travail de Bachelor, je n'avais aucune connaissance en web ou en Android. Il a donc été nécessaire que j'apprenne toutes les notions web, Java EE et Android. Avec ces notions, les langages HTML, CSS et JSP qui m'étaient jusqu'alors inconnus ont dû être appris. Apprendre de nouveaux langages de programmation est cependant bien moins long que d'apprendre des concepts de programmation. Beaucoup de temps a été passé sur des tutoriels afin d'apprendre les principes du web, de Java EE et d'Android. Parmi ces tutoriels, il faut citer ceux du *Site du Zéro* qui permettent d'en apprendre sur beaucoup de sujets en partant de zéro. Ce travail prouve que grâce à internet, il est possible pour des personnes autodidactes de créer des applications relativement complexes.

13 Date et signature

San José, le 01.08.2013

Auteur : Tinguely Joël

14 Sources images et figures

Figure 4 Logo de Google App Engine
https://developers.google.com/appengine/images/appengine_lowres.png

Figure 5 Fonctionnement général de GAE
http://uploads.siteduzero.com/files/369001_370000/369037.png

Figure 6 Choix de type d'authentification
https://developers.google.com/appengine/docs/images/authoptions.png

Figure 26 Fonctionnement Client-Serveur
http://uploads.siteduzero.com/files/369001_370000/369180.png

Figure 18 Arborescence du projet après création
https://developers.google.com/appengine/docs/java/tools/eclipse

Figure 37 Représentation du pattern MVC
http://uploads.siteduzero.com/files/369001_370000/369177.png

Figure 41 Traitement d'une page JSP
http://uploads.siteduzero.com/files/369001_370000/369176.png

Figure 38 Représentation conceptuelle du fonctionnement avec conteneur de Servlets
http://uploads.siteduzero.com/files/370001_371000/370034.png

Figure 42 Portée des objets
http://uploads.siteduzero.com/files/373001_374000/373030.png

Figure 62 Représentation du modèle DAO
http://uploads.siteduzero.com/files/395001_396000/395088.png

Figure 63 Coûts des opérations sur le datastore
https://developers.google.com/appengine/docs/billing?hl=fr#Billable_Resource_Unit_Cost

Figure 128 Logo officiel d'Android, Bugdroid
https://upload.wikimedia.org/wikipedia/commons/thumb/d/d7/Android_robot.svg/96px-Android_robot.svg.png

Figure 132 Compilation d'un programme pour Android
http://uploads.siteduzero.com/files/322001_323000/322271.png

Figure 133 Cycle de vie d'une activité
http://uploads.siteduzero.com/files/322001_323000/322141.png

15 Sources internet

[1] ANDROID HIVE, *Android Custom ListView with Image and Text* [en ligne], http://www.androidhive.info/2012/02/android-custom-listview-with-image-and-text/ (consulté le 29.07.2013)

[2] COMMENT CA MARCHE, *Caractéristiques des servlets* [en ligne], mis à jour en 04.2013 http://www.commentcamarche.net/contents/servlets/servcarac.php3 (consulté le 21.04.2013)

[3] COMMENT CA MARCHE, *Le protocole HTTP* [en ligne], mis à jour en 04.2013 http://www.commentcamarche.net/contents/internet/http.php3 (consulté le 21.04.2013)

[4] DEVELOPER ANDROID, *Authenticating to OAuth2 Services* [en ligne], https://developer.android.com/training/id-auth/authenticate.html (consulté le 29.07.2013)

[5] DEVELOPER ANDROID, *Getting Started with GCM* [en ligne], https://developer.android.com/google/gcm/gs.html (consulté le 29.07.2013)

[6] DEVELOPER ANDROID, Official documentation (API) [en ligne], https://developer.android.com/develop/index.html (consulté le 29.07.2013)

[7] DEVELOPPEZ.COM, *Mapper sa base de données avec le pattern DAO* [en ligne], Herby Cyrille, publié le 08.05.2009. http://cyrille-herby.developpez.com/tutoriels/java/mapper-sa-base-donnees-avec-pattern-dao/ (consulté le 07.04.2013)

[8] DOCS ORACLE, *Class HttpServlet* [en ligne], mis à jour le 10.02.2011 http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html (consulté le 21.04.2013)

[9] DOCS ORACLE, *Interface HttpServletRequest* [en ligne], mis à jour le 10.02.2011 http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html (consulté le 21.04.2013)

[10] DOCS ORACLE, *Interface HttpServletResponse* [en ligne], mis à jour le 10.02.2011 http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html (consulté le 21.04.2013)

[11] FAVICON.CC, *Création de favicon* [en ligne], http://www.favicon.cc/ (consulté le 02.06.2013)

[12] GOOGLE DEVELOPERS. *Google App Engine* [en ligne], mis à jour le 06.02.2013. https://developers.google.com/appengine/ (consulté le 07.04.2013)

[13] GOOGLEAPPENGINE. *Java Enterprise Edition (Java EE) Technologies* [en ligne], mis à jour le 15.04.2012. https://code.google.com/p/googleappengine/wiki/WillItPlayInJava (consulté le

21.04.2013)

[14] JAVAWORLD, *JSP best practices* [en ligne], Dustin Marx, mis à jour le 29.11.2001. http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-jsp.html (consulté le 07.04.2013)

[15] JODA TIME [en ligne], mis à jour le 08.03.2013 http://joda-time.sourceforge.net/ (consulté le 07.04.2013)

[16] NICK'S BLOG, Authenticating against App Engine from an Android app [en ligne], Nick Johnson, publié le 05.05.2010. http://blog.notdot.net/2010/05/Authenticating-against-App-Engine-from-an-Android-app (consulté le 29.07.2013)

[17] OBJECTIFY, *Concepts* [en ligne], mis à jour le 03.05.2012 https://code.google.com/p/objectify-appengine/wiki/Concepts (consulté le 07.04.2013)

[18] ORACLE, *Code Convention for the JavaServer Pages Technology Version 1.x Language* [en ligne], version de février 2003. http://www.oracle.com/technetwork/articles/javase/code-convention-138726.html (consulté le 07.04.2013)

[19] SITE DU ZERO, *Apprenez à créer votre site web avec HTML5 et CSS3* [en ligne], Mateo21, mis à jour le 07.01.2013. http://www.siteduzero.com/informatique/tutoriels/apprenez-a-creer-votre-site-web-avec-html5-et-css3/comment-fait-on-pour-creer-des-sites-web (consulté le 07.04.2013)

[20] SITE DU ZERO, *Créer des applications pour Android* [en ligne], Frédéric Espiau alias Apollidore, mis à jour le 08.01.2013. http://www.siteduzero.com/informatique/tutoriels/creez-des-applications-pour-android (consulté le 29.07.2013)

[21] SITE DU ZERO, *Créez votre application web avec Java EE* [en ligne], Coyote, mis à jour le 30.01.2013. http://www.siteduzero.com/informatique/tutoriels/creez-votre-application-web-avec-java-ee (consulté le 07.04.2013)

[22] SITE DU ZERO, *Montez votre site dans le cloud avec Google App Engine* [en ligne], Mathieu Nebra alias Mateo21, mis à jour le 31.05.2013. http://www.siteduzero.com/informatique/tutoriels/montez-votre-site-dans-le-cloud-avec-google-app-engine (consulté le 05.07.2013)

[23] STACKOVERFLOW, *About el* [en ligne], mis à jour le 27.01.2013 http://stackoverflow.com/tags/el/info (consulté le 07.04.2013)

[24] STACKOVERFLOW, *About JSP* [en ligne], mis à jour le 19.03.2013 http://stackoverflow.com/tags/jsp/info (consulté le 07.04.2013)

[25] STACKOVERFLOW, *About servlet* [en ligne], mis à jour le 20.03.2013 http://stackoverflow.com/tags/servlets/info (consulté le 07.04.2013)

[26] STACKOVERFLOW, *About servlet* [en ligne], mis à jour le 20.03.2013 http://stackoverflow.com/tags/servlets/info (consulté le 07.04.2013)

[27] TUTO MOBILE, *Faire un menu et un sous-menu* [en ligne], Axon, publié le 27.07.2010. http://www.tutomobile.fr/faire-des-menus-et-sous-menus-tutoriel-android-n%C2%B012/27/07/2010/ (consulté le 29.07.2013)

[28] WIKIPEDIA, *Android* [en ligne], mis à jour le 28.07.2013 https://fr.wikipedia.org/wiki/Android (consulté le 29.07.2013)

[29] WIKIPEDIA, *Google App Engine* [en ligne], mis à jour le 05.04.2013 https://en.wikipedia.org/wiki/Google_App_Engine (consulté le 07.04.2013)

[30] WIKIPEDIA, *Hypertext Transfer Protocol* [en ligne], mis à jour le 07.04.2013 https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (consulté le 07.04.2013)

[31] WIKIPEDIA, *Hypertext Transfer Protocol* [en ligne], mis à jour le 05.04.2013 https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol (consulté le 21.04.2013)

[32] WIKIPEDIA, *Liste des codes HTTP* [en ligne], mis à jour le 20.04.2013 https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP (consulté le 21.04.2013)

[33] WIKIPEDIA, *WAR file format (Sun)* [en ligne], mis à jour le 31.03.2013 https://en.wikipedia.org/wiki/WAR_file_format_(Sun) (consulté le 14.04.2013)

16 Table des figures

17 Table des tableaux

18 Annexes

18.1 Journal de travail

05 mars 2013

Place de travail	Aginova, EPFL
Heures	Matin : 10:45 - 12:15
Activité	Prise en main
Détail	Premier contact avec l'entreprise, présentation des ingénieurs. Présentation des produits Aginova. Brève description des projets de chacun.

08 mars 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:15 - 12:15 / Après-midi : 13:15 - 17:00
Activité	Prise en main
Détail	Présentation plus détaillée du projet. Prise de connaissance de Google App Engine notamment sur Wikipedia. Problème de version avec Java. Impossible de supprimer les erreurs sur les fichiers .jsp. Réinstallation de tout l'environnement Java (Eclipse Juno, JDK).

10 mars 2013

Place de travail	à domicile
Heures	Après-midi : 13:15 - 19:30
Activité	Prise en main
Détail	Installation de l'environnement de développement sur une autre machine et écriture de la marche à suivre pour l'installation avec toutes les erreurs rencontrées. Lecture de documentation sur Google Apps Engine.

14 mars 2013

Place de travail	HEIG-VD
Heures	Après-midi : 13:00 - 17:00
Activité	Prise en main
Détail	Création de la première application permettant de se familiariser avec l'environnement. Début de l'utilisation de base de données de Google.

15 mars 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:30 / Après-midi : 13:30 - 17:00
Activité	Prise en main
Détail	Présentation du travail effectué. Discussion général du projet. Présentation du premier cahier des charges. Installation et prise en main de Objectify pour la gestion de base de données de Google Apps Engine. Lecture de la documentation sur Objectify.

17 mars 2013

Place de travail	à domicile
Heures	Après-midi : 13:30 - 17:00
Activité	Datastore
Détail	Lecture de la documentation sur Objectify.

19 mars 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 17:00
Activité	Datastore
Détail	Lecture de la documentation sur le modèle DAO. Début de l'importation d'Objectify dans Eclipse, problème d'importation du fichier jar.

21 mars 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 17:00
Activité	Datastore
Détail	Résolution de l'importation du jar. Début de la mise en place des bases de données avec Objectify et les classes DAO.

22 mars 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:30 / Après-midi : 13:30 - 17:00
Activité	Datastore
Détail	Mise en place du système avec datastore, Objectify et DAO.

27 mars 2013

Place de travail	HEIG-VD
Heures	Après-midi : 14:30 - 18:45
Activité	Datastore
Détail	Mise en place du système avec datastore, Objectify et DAO.

01 avril 2013

Place de travail	à domicile
Heures	Après-midi : 13:00 - 22:30
Activité	Datastore
Détail	Mise en place du système avec datastore, Objectify et DAO. Contrôle du contenu du datastore. Gestion du hash pour les sensors. Ajout du jar d'apache pour le MD5. Test avec plusieurs sensors, plusieurs users. Correction des bugs.

04 avril 2013

Place de travail	à domicile
Heures	Après-midi : 13:00 – 18:00

Activité	Web
Détail	Recherche d'un moyen d'accepter des requêtes http pour la communication avec le sensor.

05 avril 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 – 12:30 / Après-midi : 13:30 – 17:00
Activité	Web
Détail	Recherche d'un moyen d'accepter des requêtes http pour la communication avec le sensor. Lecture de l'article sur Java EE sur le site du zéro. Lors de la lecture de cet article, je remarque que ma conception du code est fausse. J'ai commencé à construire mon code en me basant sur les articles et exemples de la documentation Google. Cette documentation ne reprend pas les bases du JSP et a donc de grandes lacunes pour moi.

06 avril 2013

Place de travail	à domicile
Heures	Après-midi : 13:00 – 18:00
Activité	Web
Détail	Lecture sur le site du zéro. Début du remaniement du code selon la lecture du jour.

07 avril 2013

Place de travail	à domicile
Heures	Après-midi : 20:30 – 23:45
Activité	Rapport
Détail	Mise à jour du rapport. Partie Introduction.

08 avril 2013

Place de travail	HEIG-VD
Heures	Après-midi : 13:30 – 17:30
Activité	Web
Détail	Lecture du site du zéro, création d'un template pour la page.

09 avril 2013

Place de travail	à domicile
Heures	Après-midi : 20:30 – 23:45
Activité	Rapport
Détail	Mise à jour du rapport. Création d'un document Excel pour automatiser le journal de travail. Ceci permet entre autre de créer des graphiques représentatifs du travail.

11 avril 2013

Place de travail	à domicile
Heures	Après-midi : 13:00 – 19:30

Activité	Rapport
Détail	Création du document Excel. Ecriture de la documentation sur Google App Engine.

12 avril 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 – 12:30 / Après-midi : 13:30 – 17:00
Activité	Web
Détail	Réécriture de la vue permettant à l'utilisateur de voir ses capteurs. Avancement du rapport.

14 avril 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 – 19:00
Activité	Web
Détail	Ajout des événements de log et de l'envoi des emails, modification du fichier de déploiement.

15 avril 2013

Place de travail	à domicile
Heures	Après-midi : 21:30 – 23:55
Activité	Rapport
Détail	Ecriture de la documentation sur Objectify

16 avril 2013

Place de travail	HEIG-VD
Heures	Matin : 10:30 – 12:30 / Après-midi : 13:00 – 19:00
Activité	Rapport
Détail	Ecriture de la documentation sur Objectify et le fichier de déploiement. Modification du fichier de déploiement pour intégrer les accès restreints, les filtres et l'activation de HTTPS.

17 avril 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 17:30
Activité	Web
Détail	Mise en place des URL privées pour les admin et les users.

20 avril 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 19:00
Activité	Rapport
Détail	Fichiers index, fichiers de configuration.

21 avril 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 19:00
Activité	Rapport
Détail	Protocole HTTP, Servlet.

23 avril 2013

Place de travail	HEIG-VD
Heures	Après-midi : 13:00 - 17:00
Activité	Rapport
Détail	Vue, JSP.

24 avril 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 23:59
Activité	Rapport
Détail	JSP, EL.

26 avril 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 17:00
Activité	Web
Détail	Rapport: JSTL / Web: début formulaire / Sensor: Soudure des connecteurs.

30 avril 2013

Place de travail	HEIG-VD
Heures	Matin : 10:30 - 12:00 / Après-midi : 13:00 - 16:00
Activité	Web
Détail	Formulaire de nouveau client.

03 mai 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 17:30
Activité	Sensor
Détail	Première communication avec paramètres entre le sensor et le serveur local.

09 mai 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 17:00
Activité	Web
Détail	Formulaire ajout sensor.

10 mai 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 17:00

Activité	Web
Détail	Implémentation des sensorData.

11 mai 2013

Place de travail	à domicile
Heures	Après-midi : 13:30 - 19:00
Activité	Web
Détail	SensorData + début graphique de données.

14 mai 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 23:00
Activité	Web
Détail	Modification des formulaires new sensor et new data pour garder les données si une saisie est erronée.

16 mai 2013

Place de travail	à domicile
Heures	Après-midi : 20:00 - 23:30
Activité	Web
Détail	Ajout du graphique de données.

17 mai 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 17:30
Activité	Web
Détail	Mise à jour du SDK (1.7.5 -> 1.8). Passage de toutes les dates en format UNIX (millisecondes avec Java!). Ajout des options de date et d'unité pour les utilisateurs

28 mai 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 23:30
Activité	Web
Détail	Fin de l'ajout des options de date et d'unité pour les utilisateurs.

29 mai 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 22:30
Activité	Rapport
Détail	Relecture depuis le début jusqu'à Google App Engine non compris

30 mai 2013

Place de travail	à domicile
Heures	Après-midi : 20:00 - 23:30

Activité	Rapport
Détail	Relecture des 50 premières pages.

31 mai 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:15 - 12:30 / Après-midi : 13:30 - 17:45
Activité	Sensor
Détail	Mise en place de la communication avec le serveur. Transmission de valeur ok.

01 juin 2013

Place de travail	à domicile
Heures	Après-midi : 20:30 - 23:45
Activité	Rapport
Détail	Ecriture de la partie Structure du projet.

02 juin 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 18:00
Activité	Web
Détail	Mise à jour des fichiers JSP pour l'inclusion des constantes provenant de Links.java. Début de la mise en place de la Javadoc.

03 juin 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 23:45
Activité	Web
Détail	Correction de bug. Ecriture de la Javadoc.

04 juin 2013

Place de travail	HEIG-VD
Heures	Après-midi : 13:00 - 17:00
Activité	Rapport
Détail	Partie présentation de l'application. Correction des emplacements de fichiers du projet.

07 juin 2013

Place de travail	Aginova, EPFL
Heures	Matin : 09:15 - 12:45 / Après-midi : 14:00 - 18:00
Activité	Web
Détail	Test end to end de l'application avec sensor et probe. Correction des bugs de librairies pour le déploiement.

08 juin 2013

Place de travail	à domicile

Heures	Après-midi : 14:00 - 19:00
Activité	Web
Détail	Mise en place d'un tableau JavaScript pour la gestion du paging. Correction du bug pour envoyer des emails depuis l'application.

09 juin 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 20:00
Activité	Web
Détail	Mise en place du refresh automatique avec gestion du scroll. Correction des bugs observés sur l'application en ligne. Ajout de la Servlet pour les appareils mobiles.

10 juin 2013

Place de travail	à domicile
Heures	Après-midi : 21:00 - 23:00
Activité	Web
Détail	Site web inaccessible car les quotas de read sur le datastore sont dépassés. Début de l'écriture du JavaScript pour la mise à jour uniquement des tableaux au lieu de toute la page.

11 juin 2013

Place de travail	à domicile
Heures	Après-midi : 20:30 - 23:30
Activité	Web
Détail	Ecriture du JavaScript pour les tableaux. Ajout de jQuery pour la gestion des nouvelles données provenant des sensors. Modification du fichier mySensor.java pour inclure un mode de refresh.

14 juin 2013

Place de travail	Aginova, EPFL
Heures	Matin : 10:15 - 13:00 / Après-midi : 14:00 - 18:00
Activité	Datastore
Détail	Explication du problème avec le nombre de requêtes read sur le datastore. Nous cherchons actuellement un moyen de résoudre ce problème. Sans succès pour le moment.

16 juin 2013

Place de travail	à domicile
Heures	Après-midi : 14:00 - 19:00
Activité	Rapport
Détail	Mise à jour de la documentation pour le rendu du rapport intermédiaire.

25 juin 2013

Place de travail	SJSU
Heures	Matin : 09:00 - 12:00 / Après-midi : 13:00 - 17:00

Activité	Web
Détail	Introduction des notes de sensor individuelles pour chaque utilisateur.

26 juin 2013

Place de travail	SJSU
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 16:30
Activité	Web
Détail	Page d'update des settings des accounts. Début suppression account.

27 juin 2013

Place de travail	SJSU
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 16:30
Activité	Web
Détail	Suppression account, modification note.

28 juin 2013

Place de travail	SJSU
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 16:30
Activité	Rapport
Détail	Présentation des modèles AppUser, Sensor, UserSensor et SensorData

01 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:15 / Après-midi : 13:00 - 16:00
Activité	Rapport
Détail	Présentation JavaScript, service memcache, datastore. Modification code pour mise à jour automatique du graphique

02 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:15 / Après-midi : 13:00 - 17:15
Activité	Rapport
Détail	Package appUser, sensor et sensorData. Modification du code pour recevoir proprement les données des sensors

03 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 13:00 - 17:00
Activité	Rapport
Détail	Package applicationGeneralServlet et utilities. Installation de l'environnement de développement pour le sensor.

05 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:30 - 12:00 / Après-midi : 12:45 - 16:30

Activité	Android
Détail	Installation de l'environnement de développement. Rapport.

06 juillet 2013

Place de travail	SJSU
Heures	Matin : 09:00 - 12:00 / Après-midi : 13:00 - 15:00
Activité	Android
Détail	Première compilation, introduction à Android, hello world.

08 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 15:00
Activité	Android
Détail	Lecture du site du zéro jusqu'aux listes. Fin de la journée lors du déclanchement de l'alarme incendie du bâtiment.

09 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 16:30
Activité	Android
Détail	Lecture du site du zéro sur les intents.

10 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 17:30
Activité	Android
Détail	Essai pour le fonctionnement du programme sur la tablette Nexus 7. Enormément de problèmes connus sur le driver. Le PC ne détecte pas la tablette…

11 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:30 - 12:00 / Après-midi : 12:45 - 17:15
Activité	Android
Détail	Continuation des essais pour la tablette…

16 juillet 2013

Place de travail	SJSU
Heures	Après-midi : 13:00 - 17:00
Activité	Android
Détail	Essai d'authentification avec le compte Google sur le serveur.

17 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 17:00

Activité	Android
Détail	Essai d'authentification avec le compte Google sur le serveur.

18 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 17:00
Activité	Android
Détail	Modification du serveur web pour l'envoi de données sans authentification.

20 juillet 2013

Place de travail	SJSU
Heures	Matin : 09:30 - 12:00 / Après-midi : 13:00 - 17:30
Activité	Android
Détail	Création d'une activité qui permet de récupérer les informations sur tous les sensors.

21 juillet 2013

Place de travail	SJSU
Heures	Matin : 10:00 - 12:00 / Après-midi : 13:00 - 17:00
Activité	Rapport
Détail	Ecriture du rapport sur Android.

22 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:30 - 12:00 / Après-midi : 12:45 - 16:00
Activité	Rapport
Détail	Ecriture du rapport sur Android.

23 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:00 - 12:00 / Après-midi : 12:45 - 17:00
Activité	Android
Détail	Création du relativelayout pour la page principale de l'application. Création de la liste et de son adaptateur pour la page principale.

24 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:00 - 12:00 / Après-midi : 12:45 - 17:00
Activité	Rapport
Détail	Ecriture du rapport sur Android.

25 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:00 - 12:00 / Après-midi : 13:00 - 17:00
Activité	Android

Détail	Lecture de la documentation sur les Intents, les threads, l'accès à internet.

26 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 17:00
Activité	Android
Détail	Création de l'Asynctask pour la requête HTTP sur le serveur. Modification de tous les Strings pour l'utilisation du fichier string.xml

27 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 17:45
Activité	Rapport
Détail	Ecriture du rapport sur Android.

29 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 13:00 - 17:30
Activité	Rapport
Détail	Ecriture du rapport sur Android.

30 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 12:45 - 18:00
Activité	Rapport
Détail	Présentation de l'application web et l'application Android

31 juillet 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 13:00 - 17:00
Activité	Rapport
Détail	Ecriture des améliorations, remarques et conclusion.

01 août 2013

Place de travail	SJSU
Heures	Matin : 08:15 - 12:00 / Après-midi : 13:00 - 16:00
Activité	Rapport
Détail	Relecture du rapport, envoi.

18.2 Cahier des charges

Remote iCelsius Wireless specifications

The purpose of this project is to implement the remote communication for the iCelsius Wireless. The iCelsius Wireless is device used to measure temperature and send it to a user device which will display the data. It can works in 3 different modes:

1) Direct mode (the iCelsius Wireless act as a AccessPoint)

2) Infrastructure local (the iCelsius Wireless connect to an existing AP and communicate with a smart as well connected to this AP)

3) Infrastructure remote (the iCelsius Wireless connect to an existing AP send the data to remote server. Device connect to this remote serve from anywhere r in order to receive the data)

This project will focused in part 3. The communication protocol used will be HTTP in order to avoid being blocked by firewall. The server will be implemented for Google App Engine (and obviously also hosted on it). Two mode of user will exist:

1. Standard: data are just routed to the user device
2. Premium: data are stored for long term and user can log to a web interface to visualized them

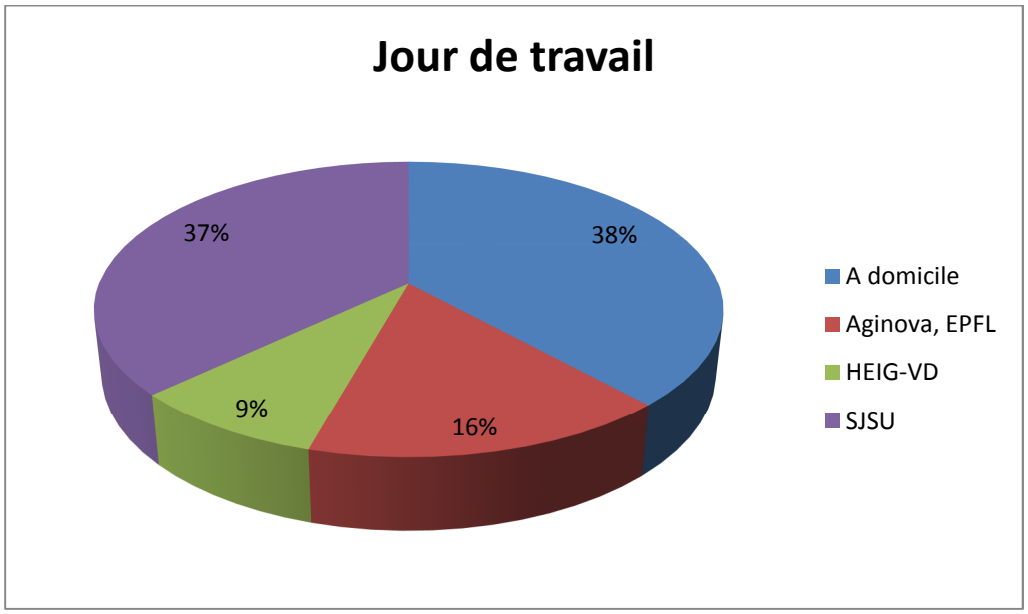
The student will also have to code the embedded part in the senor for http communication with the server.

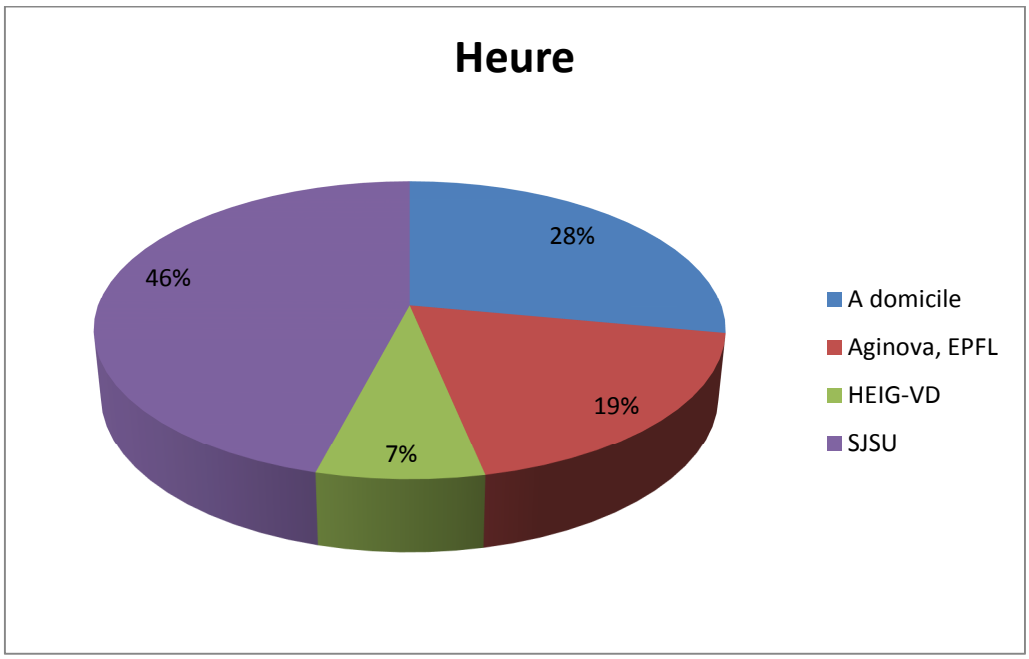
Here is a priority list of the desired features:

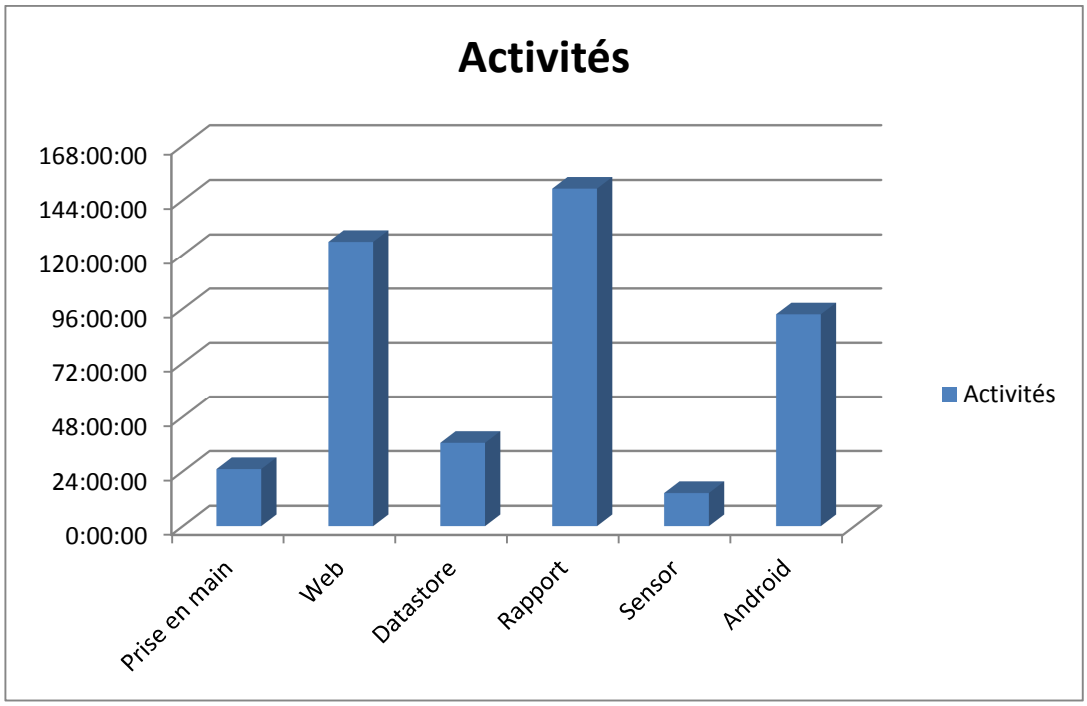
1) Implement basic communication between the iCelsius Wireless and the server using http
2) Develop the mechanism to link a sensor to a user device. Each sensor will have a unique random number only known by user. The server will have this number hashed so it will be able to compare to know if this number is valid.
3) Stored the data in a local database. Each sensor ID may have different type of probe connected to it
4) Implement a basic web interface to see the data stored on the server
5) Implement a basic Android App to receive the data from the server and display it
6) Add more feature to the Android App so it could start reconfiguring the sensor
7) Improve the website design with the help of Aginova Designer

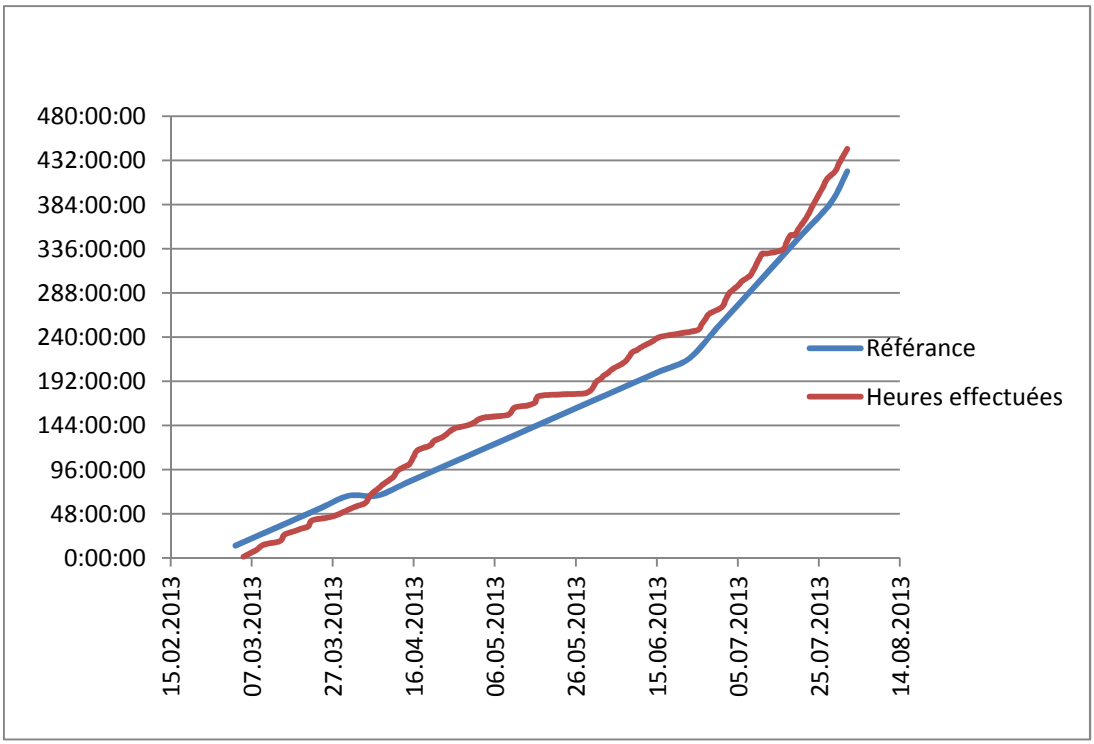
<u>Name Definition</u>
iCelsius Wireless (or Sensor): The Wifi module with a probe connected
Probe: can be temperature, temperature/humidity, CO2, pH, etc...
Sensor ID: Unique number of each sensor
Google ID: Unique login and password identification for final user
User device: Smart Phone or Tablette with our iCelsius Wireless App
Server: The Google Engine App server with our server application running on it
Android App: iCelsius Wireless App running on a Android user device

18.3 Statistiques sur le temps de travail






Activités
168:00:00
144:00:00
120:00:00
96:00:00
72:00:00
48:00:00
24:00:00
0:00:00
Prise en main
Web
Datastore
Rapport
Sensor
Android
Activités


480:00:00
432:00:00
384:00:00
336:00:00
288:00:00
240:00:00
192:00:00
144:00:00
96:00:00
48:00:00
0:00:00
15.02.2013
07.03.2013
27.03.2013
16.04.2013
06.05.2013
26.05.2013
15.06.2013
05.07.2013
25.07.2013
14.08.2013
Référance
Heures effectuées

18.4 Codes d'erreur des réponses HTTP

Codes commençant par 1

Code	Message	Signification
100	*Continue*	Attente de la suite de la requête
101	*Switching Protocols*	Acceptation du changement de protocole
102	*Processing*	WebDAV : Traitement en cours (évite que le client dépasse le temps d'attente limite).
118	*Connection timed out*	Délai imparti à l'opération dépassé

Codes commençant par 2

Code	Message	Signification
200	*OK*	Requête traitée avec succès
201	*Created*	Requête traitée avec succès avec création d'un document
202	*Accepted*	Requête traitée mais sans garantie de résultat
203	*Non-Authoritative Information*	Information retournée mais générée par une source non certifiée
204	*No Content*	Requête traitée avec succès mais pas d'information à renvoyer
205	*Reset Content*	Requête traitée avec succès, la page courante peut être effacée
206	*Partial Content*	Une partie seulement de la requête a été transmise
207	*Multi-Status*	WebDAV : Réponse multiple.
210	*Content Different*	WebDAV : La copie de la ressource côté client diffère de celle du serveur (contenu ou propriétés).
226	*IM Used*	RFC 3229 : ?

Codes commençant par 3

Code	Message	Signification
300	*Multiple Choices*	L'URI demandée se rapporte à plusieurs ressources
301	*Moved Permanently*	Document déplacé de façon permanente
302	*Moved Temporarily*	Document déplacé de façon temporaire
303	*See Other*	La réponse à cette requête est ailleurs
304	*Not Modified*	Document non modifié depuis la dernière requête
305	*Use Proxy*	La requête doit être ré-adressée au proxy
306	(aucun)	Code utilisé par une ancienne version de la RFC 2616, à présent réservé.
307	*Temporary Redirect*	La requête doit être redirigée temporairement vers l'URI spécifiée
310	*Too many Redirects*	La requête doit être redirigée de trop nombreuses fois, ou est victime d'une boucle de redirection.

Codes commençant par 4

Code	Message	Signification
400	*Bad Request*	La syntaxe de la requête est erronée
401	*Unauthorized*	Une authentification est nécessaire pour accéder à la ressource
402	*Payment Required*	Paiement requis pour accéder à la ressource (non utilisé)
403	*Forbidden*	Le serveur a compris la requête, mais refuse de l'exécuter. Contrairement à l'erreur 401, s'authentifier ne fera aucune différence. Sur les serveurs où l'authentification est requise, cela signifie généralement que l'authentification a été acceptée mais que les droits d'accès ne permettent pas au client d'accéder à la ressource (càd utilisateur reconnu essayant d'accéder à du contenu à accès restreint).
404	*Not Found*	Ressource non trouvée
405	*Method Not Allowed*	Méthode de requête non autorisée
406	*Not Acceptable*	Toutes les réponses possibles seront refusées.
407	*Proxy Authentication Required*	Accès à la ressource autorisé par identification avec le proxy
408	*Request Time-out*	Temps d'attente d'une réponse du serveur écoulé
409	*Conflict*	La requête ne peut être traitée à l'état actuel
410	*Gone*	La ressource est indisponible et aucune adresse de redirection n'est connue
411	*Length Required*	La longueur de la requête n'a pas été précisée
412	*Precondition Failed*	Préconditions envoyées par la requête non-vérifiées
413	*Request Entity Too Large*	Traitement abandonné dû à une requête trop importante
414	*Request-URI Too Long*	URI trop longue
415	*Unsupported Media Type*	Format de requête non-supportée pour une méthode et une ressource données
416	*Requested range unsatisfiable*	Champs d'en-tête de requête « *range* » incorrect.
417	*Expectation failed*	Comportement attendu et défini dans l'en-tête de la requête insatisfaisable
418	*I'm a teapot*	"Je suis une théière". Ce code est défini dans la RFC 2324 datée du premier avril 1998, Hyper Text Coffee Pot Control Protocol. Il n'y a pas d'implémentation de ce code.
422	*Unprocessable entity*	WebDAV : L'entité fournie avec la requête est incompréhensible ou incomplète.
423	*Locked*	WebDAV : L'opération ne peut avoir lieu car la ressource est verrouillée.
424	*Method failure*	WebDAV : Une méthode de la transaction a échoué.
425	*Unordered Collection*	WebDAV (RFC 3648). Ce code est défini dans le brouillon *WebDAV Advanced Collections Protocol*, mais est absent de *Web Distributed Authoring and Versioning (WebDAV) Ordered Collections Protocol*
426	*Upgrade Required*	(RFC 2817) Le client devrait changer de protocole, par exemple au profit de TLS/1.0
449	*Retry With*	Code défini par Microsoft. La requête devrait être renvoyée après avoir effectué une action.
450	*Blocked by Windows Parental Controls*	Code défini par Microsoft. Cette erreur est produite lorsque les outils de contrôle parental de Windows sont activés et bloquent l'accès à la page.
456	*Unrecoverable Error*	WebDAV
499	*client has closed connection*	nginx : Le client a fermé la connexion avant de recevoir la réponse. Se produit quand le traitement est trop long côté serveur.

Codes commençant par 5

Code	Message	Signification
500	*Internal Server Error*	Erreur interne du serveur
501	*Not Implemented*	Fonctionnalité réclamée non supportée par le serveur
502	*Bad Gateway* ou *Proxy Error*	Mauvaise réponse envoyée à un serveur intermédiaire par un autre serveur.
503	*Service Unavailable*	Service temporairement indisponible ou en maintenance
504	*Gateway Time-out*	Temps d'attente d'une réponse d'un serveur à un serveur intermédiaire écoulé
505	*HTTP Version not supported*	Version HTTP non gérée par le serveur
506	*Variant also negociate*	RFC 2295 : Erreur de négociation *transparent content negociation*
507	*Insufficient storage*	WebDAV : Espace insuffisant pour modifier les propriétés ou construire la collection
508	*Loop detected*	WebDAV : Boucle dans une mise en relation de ressources (rfc5842)
509	*Bandwidth Limit Exceeded*	Utilisé par de nombreux serveurs pour indiquer un dépassement de quota.
510	*Not extended*	RFC 2774 : la requête ne respecte pas la politique d'accès aux ressources HTTP étendues.