Université de Liège
Faculté des Sciences Appliquées

Design of a mobile application for real-time energy consumption monitoring

Author:
Thibaud Ledent

Supervisor:
Prof. Guy Leduc

Academic year 2014 - 2015

A thesis submitted in fulfilment of the requirements for the degree of MSc in Computer Science and Engineering in Management Focus

Jury: Guy Leduc, Benoît Donnet, Laurent Mathy, Vincent Keunen

Master thesis carried out in collaboration with S23Y

Design of a mobile application for real-time energy consumption monitoring

Thibaud Ledent

June 2, 2015

ABSTRACT

The growing awareness regarding new mobile technologies and power consumption brings with it new game-changing challenges. With access to more data, companies and individuals could improve their consumption habits, leading to long-term savings.

This master's thesis aims to provide a mobile application for real-time energy consumption monitoring. Although the idea of reducing energy consumption already exists alongside corporate and individual consumption, intelligent monitoring is an essential asset. Not only could it benefit the client; an energy provider could find various ways to take advantage of knowing precisely how its electrical network is used.

In this context, the design of the system is examined in the course of this report. The application uses data fetched by an existing system and retrieves consumption information to make the data available to the user in an intelligible form. The system is separated into two parts: a back end, a server built upon *Spring Boot*; and a front end, a mobile *Android* application.

Following this section, the project implementation is then detailed through description of the relevant features and our choices to present the data collected in an effective and user-friendly way. Particular attention is paid to the synchronization, distribution and backup of energy consumption. Charts, statistics and relevant responses to abnormal cases are provided. The communication between the application and the server is conducted through a RESTful web service.

Having completed the application, we subsequently focused on testing and validating the system for all the features investigated. As a result, customers can check their power consumption in many different ways, including a flexible line chart of their consumption, a set of statistics about their past day, week, month and year's consumption, or even a comparison with an average customer profile. Finally, other perspectives of development are given, which could potentially complete the working monitoring system which is achieved in this work.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisors. Within the company, Vincent Keunen (co-founder and CEO of S23Y) was never short of good advice, useful comments, remarks and inspiration. Antoine Smolders (co-worker of Vincent since 2013) suggested to me many of the tools used in this work, helped me through the learning process and was always available to answer any questions I had. Furthermore, I would like to thank Professor Guy Leduc from the University of Liège, who has willingly shared his precious time during the last months.

CONTENTS

LIST OF FIGURES

Part I

INTRODUCTION

1

INTRODUCTION

"Let's disrupt the energy sector to reduce the cost of energy for all, develop sustainable energy sources and allow human communities to develop further, thanks to green and cheap energy."
— S23Y's philosophy [27].

We live in an exciting age where mobile computing brings new game-changing challenges. The combination of networking and mobility opens the door to new applications and services of limitless potential. Physical location does not matter anymore. An idea, a product or a piece of information can reach virtually *anyone*, *anywhere*, *anytime*. This last decade has seen the birth of thousands of new applications taking advantage of portability and bandwidth improvements. Mobile phones are now an integral part of our daily lives. They empower people to free their minds, connect more easily, and make smarter decisions. Furthermore, cloud computing creates the potential to put a supercomputer in anyone's pocket. No other modern technology has this reach and this potential.

Today, all the world's information is online and everything is speeding up. Soon, objects of any kind will be provided with the ability to transfer data over a network without requiring human interaction: this is the *Internet of Things* [35]. With this ability comes an exponential growth in information: more data now cross the internet every second than were stored in the entire internet 20 years ago [9]. Many companies already aggregate these data over long periods with the purpose to take advantage of this gold mine. Indeed, when used effectively, *Big Data* allows for more effective interventions, predictions and decisions.

These two ideas are about to revolutionize the way computers are used. Undeniably, new mobile technologies will continue to transform many business sectors simply because virtually every industry is, at some level, information-driven. The energy sector is no exception.

As far as energy is concerned, these innovative technologies could imply a significant shift. The European Union has set an ambitious goal: "to reduce the output of greenhouse gases by 20%, to improve energy efficiency by 20% and to increase the percentage of renewable energy by 20%" [8]. If the traditional way to monitor power consumption for a regular customer is via invoice, there is nonetheless a growing public awareness regarding new technologies.

Measuring your home's energy consumption is the first step toward finding ways to decrease it. This project, called *MyConsumption*, follows this line of thought. It consists of "designing a mobile application for real-time energy consumption monitoring" for companies and individuals. Connected to a smart object, the application retrieves consumption data and information to make them available to the user in an intelligible form. Moreover, different features provide relevant solutions to a set of use cases such as an abnormal consumption.

The idea of reducing energy consumption coexists with consuming. With the ability to monitor consumption, a user could easily adapt their behavior, assess which device is consuming more, and optimize the efficiency of the whole system. Plus, such monitoring opens the door to many other possibilities.

A monitoring system could not only benefit the client; energy providers could also find various ways to take advantage of knowing precisely how their electrical networks are used. By having access to the exact consumption data of every single customer, operators could improve the efficiency of the production and distribution of electricity [40]. In this process, intelligent monitoring is an essential asset. On a large scale and with the right decisions, game-changing challenges and long-term savings could potentially be involved.

Alongside the importance of this problem, there are three reasons behind my choice of this subject. Firstly, it aims to increase the awareness of monitoring systems and their potential energy efficiency measures. Secondly, it tries to simplify and highlight the key steps of the system's implementation. Finally, it is part of a larger open source project with great opportunities, impact and long-term prospects.

After this introduction, this document begins with a description of the project, the methodology followed during this work, and a brief description of the company I worked with. Following this, the design section focuses on the relevant features, user interfaces and use cases. An overview of the architecture and the tools involved in this work is also given. Next, the implementation section of the document describes how we tackled the challenges associated with each feature. It tries to address every issue we faced. Then, a section is dedicated to the validation of the design and the tests of the system. Finally, future development prospects are suggested.

2

STATE OF THE ART

An application designed to monitor energy consumption is not a new idea. The simple proof of this is the large number of actors already present in the market. In this section, we describe what already exists in the world of real-time energy consumption monitoring systems.

The list below is not intended to be fully exhaustive, but should nonetheless provide a good overview of different tools available.

2.1 Real-time energy monitoring systems

As quoted by the ADEME[1], 89% of the costs associated with energy in households are those related to the heating system[2]. Therefore, most solutions involve a mobile application *and* a device to control the heating system, whether it is electric or gas/oil powered. According to the main actors in the market, savings can reach 40% of the current bill, which could have a huge impact when extrapolated over a large scale.

The most well-known smart thermostat comes from Google and is called *Nest* [10]. It is composed of two parts: a thermostat and a device called "*Heat Link*" located on a boiler. The *Heat Link* can turn the heating system on or off. *Nest* will notify the user when (s)he is saving energy. It will also adapt the temperature of the house when no one is there. Moreover, one can schedule the system and remotely control the *Heat Link*. Last but not least, *Nest* tries to learn from the user. By memorizing his/her habits and behaviors, it will automatically tune the configuration of the system for better overall performance.

[1] *Agence de l'Environnement et de la Maîtrise de l'Energie* (http://www.ademe.fr/).
[2] In Ile-De-France (Paris area).

Other tools are available on the market and are quite similar to *Nest*. For example, *Netamo* has an auto-adapt mode to analyze the insulation of the house based on its consumption. It can then tune the configuration of the system to optimize it.

Another one comes from Germany: *Tado*°. It focuses on providing a solution that has good compatibility with many existing heating systems. It also tries to take advantage of the user localization. By using the GPS tracking function of the user's smartphone, *Tado*° can predict when (s)he will come home and make the system more responsive.

Honeywell has a solution with the particularity to provide four devices connected to a central thermostat. Each device can be located in a different room near a source of heat (such as a radiator) which allows to monitor different area.

Other solutions go beyond the monitoring of energy. For example, the *Wiser* solution from Schneider Electric make it possible to monitor and control a dishwasher, a hot water tank, or simply several electrical appliances.

3

PROJECT DESCRIPTION

The first meetings with Antoine and Vincent aimed to shape *MyConsumption*[1]. The purpose of this section is to give an overview of what we set out to achieve.

3.1 Problem statement

The project goal was to develop an application for companies and individuals supplying real-time monitoring of energy consumption. An existing system, a *Flukso* smart meter (see Figure 1), is used to retrieve the data. As far as the architecture is concerned, the application has to be composed of a back end written in Java and a mobile application.

Figure 1: A *Flukso* smart meter.

3.1.1 Back end

The server exposes REST web services to ensure synchronization, distribution and backup of energy consumption data. It also retrieves and distributes pricing information.

[1] *MyConsumption* is the name of this project.

3.1.2 Front end

The *Android* application displays data, provides solutions to different use cases and retrieves information such as energy price estimations at a given moment.

3.1.3 Smart meter

The device used in this project to monitor real-time energy consumption is called a *smart meter*. It is an electronic device that records consumption of energy and communicates that information to a central system (see Figure 2) [41]. In the case of electricity, it is placed close to the electricity meter of a house. A central system gives us access to the data collected.

Figure 2: A smart meter with its central system.

3.1.4 Open source

As a part of a larger solution deployed by the company, the system had to be fully open source. Patrick Herbeuval, who worked on the same project last year, kindly gave us his consent to publish the code he wrote. For the first few months, we worked on a private *Gitlab* repository. Then, in February, Vincent took the decision to move the code to *Github* under the license Apache Version 2.0 [14]. Everything is now available at https://github.com/S23Y/.

An open source project presented a great opportunity for me. Firstly, it means that the resultant code will not be limited to just a half a year of work as it may be useful to others someday. Indeed, contributions are encouraged by forking the repository, and contribute back using pull requests (see the project's readme on *Github*). Secondly, it allowed me to easily reuse other projects, libraries and components available on *Github* (or elsewhere) thanks to the license of the project. Finally, playing with open source tools is a great way to discover and learn from the work of others.

3.1.5 Positioning

The way this project was defined and designed made it a unique solution. Nevertheless, the purpose of this work was not to make an application that controls electrical

appliances. Instead, it aimed to *visualize* and *display* relevant information related to the *monitoring* of energy consumption and, for that reason, represents only a part of the solutions described in the state of the art. As compared to those solutions, the real added value of *MyConsumption* is that it is completely open source and based on recent technologies.

3.2 The first version of MyConsumption

We did not start this work from scratch. Last year, Patrick Herbeuval worked on the same subject during his master's thesis [25]. Here follows a description of the work he carried out.

3.2.1 Back end

The server designed by Patrick was built upon *Spring*, an open source framework used to deploy web applications and to facilitate database access. The server allowed the collection of data from the smart meter API and subsequently kept them in the database. A full API was provided with the server, allowing different kind of access to the data it kept. A scheduled task ran in the background, retrieving the data every ten minutes from the manufacturer's API.

One of the strengths of Patrick's server was its modularity to add compatibility with other smart meters using a strong object-oriented approach. The following points weren't addressed by him:

- Security (i.e. authentication and secured communication with the server and its database);
- Push notifications to mobile devices;
- Processing and analysis of data (such as statistics);
- System deployment in practice (in a private or professional environment).

3.2.2 Front end

Patrick designed a small *Android* application. As shown in Figure 3, this client displayed a graph of the data received from the back end. It also featured a basic user login system and could handle the addition and deletion of sensors. The data fetched from the server were stored in a local database for offline visualization purposes. The following points weren't addressed:

- Receiving notifications from the server (such as an abnormal consumption alert);
- Analysis and evolution of data;
- System preferences;
- The security of the system and a strong authentication mechanism.

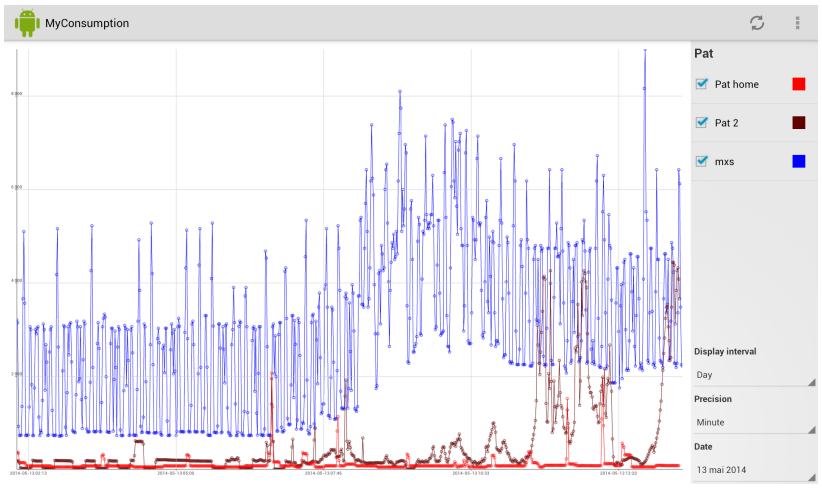


Figure 3: Main screen of Patrick's application.

3.3 Methodology

With regard to the methodology, we followed an agile development approach. From August to May, a meeting was scheduled every two weeks. The first meetings with Antoine Smolders (computer scientist from the University of Liege working for A7 Software since 2013) and Vincent Keunen (CEO of A7 Software and co-founder and CEO of S23Y) allowed us to draw an overall sketch of the application and its desired requirements.

After these initial meetings, we followed an iterative process. Every two weeks we discussed a feature to implement. The goal of the meeting was to arrive at a desired result that could be shown. Between the two meetings, we were able to design and implement this feature. In some ways, this process was similar to a very simplified scrum. During each meeting a report (see the Appendix) was made. It answered the following questions:

- What have I done?
- What am I going to do?
- What causes difficulties?
- What questions have been raised?

This was also the perfect occasion to receive valuable feedback on the work carried out since the last discussion, and to prioritize the feature to implement next.

3.4 How to build and run the project

Two dependency management systems are used in this project: *Gradle* for the app and *Maven* for the server. The following instructions explain how to use this work on a development machine.

3.4.1 Mobile application

With Gradle and Android Studio

The easiest way to build the app is to install *Android Studio* v1.+ with *Gradle* v2.+. You will need the *Android* SDK API 21 and `my-consumption-api` available at *Github MyConsumption* Server (run `mvn clean install` from the root directory of the server to install the API on your machine). Once installed, you can import the project into *Android Studio:*

1. Open File;
2. Import Project;
3. Select `build.gradle` under the project directory;
4. Click OK.

Release

Copy the `.apk` on an *Android* device and install it (installation of apps from unknown sources must be allowed in your settings). Our `.apk` file is available at http://bit.ly/myconsumption.

Test user

A user which owns two sensors is available to test the application with the following credentials:

- username: `thib`
- password: `thib`

3.4.2 Server

With Maven

The easiest way to build is to install *Maven* v3.+ in your development environment. Then, the build is pretty simple:

- Run `mvn clean install` from the root directory.

Deployment

The deployment of the server is described in the Appendix of this document.

3.5 About the company

Software Services For Energy (S23Y) is dedicated to providing innovative software services to the energy sector, worldwide [27]. They launched the *StarfishRespect* open source platform to allow all energy concerned companies to innovate together and create a vibrant ecosystem. *StarfishRespect* is based on the innovative software platform built for Lampiris by Manex (Belgium).

Figure 4: The S23Y logo.

Part II

DESIGN

4

OVERVIEW

Before defining features, use cases and user interfaces we set out to implement, this section provides a high-level view of the system and introduces the tools involved in its conception. The interactions between each part of the system are also outlined. The purpose of this section is not to go into great detail, but rather to explore how the software works in a summarized form. The following chapters will describe its implementation and provide further information.

4.1 Architecture

The Figure 5 illustrates how each part interacts with the others. From this diagram, we see that various actors are involved:

- The *Android* Application, which communicates with its local database (*SQLite*), with our server and with the *Google Cloud Messaging* server;
- Our server[1], which is built upon *Spring Boot*. It provides RESTful services as well as a Java API. It communicates with its local database (*MongoDb*), with the smart meter API and with the *Google Cloud Messaging* server;
- The smart meter manufacturer API, which allows access to their data;
- The *Google Cloud Messaging* server.

4.2 Tools involved

The main tools and concepts related to the mobile application and the server are described below. Some of them may already be well-known to readers, but since they will be used in the following chapters of this work, we thought it relevant to include every definition in the two following sections.

[1] The server is available at http://myconsumption.s23y.com.

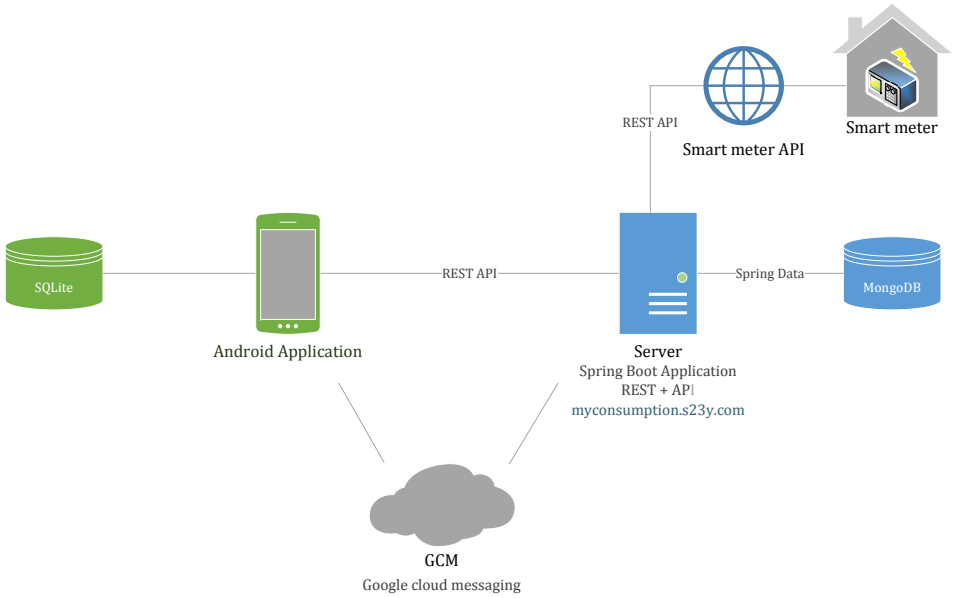


Figure 5: An overview of the system.

4.2.1 Server

IntelliJ IDEA

The Java Integrated Development Environment (IDE) used during this work was *IntelliJ IDEA*. It was developed by JetBrains (formerly known as IntelliJ), and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition [26]. It is a good software for enterprise, mobile and web development and it integrates well with all the latest modern technologies and frameworks described below.

Spring and Spring Boot

Spring is an open source application framework for the Java platform. It supplies many useful features, such as Inversion of Control, Dependency Injection, abstract data access, transaction management, and more [6]. It was conceived in 2002 in response to industry complaints that the Java EE specification was sorely lacking and very difficult to use.

The server designed for *MyConsumption* was first built upon *Spring*, to easily deploy RESTful web services and to facilitate the database access. During the course of this work, we updated the server from *Spring* to *Spring Boot* to ease its configuration. *Spring Boot* makes it easy to create stand-alone, production-grade *Spring* based applications that you can "just run" [32]. *Spring Boot* is a more recent tool, released in 2013.

The reasons behind the choice of *Spring Boot* are the following. Firstly, it supports *Spring* and the first version of the server available. Secondly, it is an open source framework well integrated with Java applications that perfectly suits the needs of the back end described above. Finally, the employees of S23Y have a deep knowledge of *Spring* and were able to guide me through the learning process of its use.
Several modules of *Spring Boot* were used in the project:

- `spring-boot-starter-web`: to deploy the RESTful services;
- `spring-boot-starter-tomcat`: for the deployment on a *Tomcat* server;
- `spring-boot-starter-security`: to add a layer of security;
- `spring-boot-data-mongo`: to provide an integration with the *MongoDb* document database.

Smart meter API

Flusko, the smart meter used to collect the data, deploys an API available at https://api.flukso.net/sensor. A module of our server was dedicated to fetch data from their API and to store them in our own database.

Google Cloud Messaging

The *Google Cloud Messaging* (GCM) service for *Android* made it possible to send data from our server to specific users' *Android*-powered device [18]. The GCM service handles all aspects of queueing of messages and delivery to the target *Android* application running on the target device. It is the standard system in *Android* to implement push notifications and therefore appeared to be the best option.

MongoDb

The choice of *MongoDb* as a database of the server was made and motivated last year by Patrick during his master's thesis.

MongoDb is a NoSQL cross-platform document-oriented database. It is not structured around the traditional table-based relational model. Instead, JSON-like documents with dynamic schemas (called BSON) are used [37]. These documents have the advantage of making the integration of data in certain types of applications easier and faster.

NoSQL databases

The original call for the term "NoSQL" asked for "open source, distributed, non-relational databases" [4]. But there is no formal definition of NoSQL databases. Still, they do have some common characteristics:

- They do not use the relational model;
- They run well on clusters;
- Most of them are open source;
- They are built for the 21st century web estates;
- They are schemaless.

Why are NoSQL databases interesting?

The first reason is because a great deal of application development effort is spent on mapping data between in-memory structures and a relational database. A NoSQL database may better fit the application's needs, and simplify that interaction. The second reason concerns large-scale data. This project may need to support large volumes of data as it aims to keep track of the consumption of many clients. However, as companies capture more and more data, they also want to process it more quickly. With clusters and NoSQL database explicitly designed for this purpose, there is a better fit for big data scenarios. Moreover, it has the advantage to use smaller and cheaper machines.

4.2.2 Mobile application

Android

Android is a mobile operating system based on the Linux kernel and currently developed by Google [36]. Its interface is designed for touchscreens with a mobile vision. At the time of writing, it is the most popular mobile OS with hundreds of millions of mobile devices in more than 190 countries around the world [16]. Furthermore, it is growing fast: every day another million users start to use a new device for the first time. The first version was released six years ago, in 2008. Today, the latest release is *Android* 5.1.1 Lollipop (April 21, 2015). The system is more and more integrated with various Google Services such as Maps, Google+ etc.

The Android SDK and Android Studio

One of the goals of Google, via *Android*, is to create a great community of developers. In order to do so, they provide them a Software Development Kit (SDK) that includes sample projects with source code, development tools, an emulator, and libraries required to build *Android* applications.

Another great tools for developers is *Android Studio*, the official IDE for *Android* application development, based on *IntelliJ IDEA* [17]. It is the IDE used to implement this work.

Support library

Besides the SDK, developers have access to the *Android Support Library*. This is a set of code libraries that provide backward-compatible versions of *Android* framework APIs [22]. It means that applications can use the libraries' features and still be compatible with devices running *Android* 1.6 (API level 4) and up. One of our goals with *MyConsumption* is to support a large set of devices, which is the reason why we used the *Support Library*.

Google Play Services

The goal of the *Google Play Services* is to allow every application to take advantage of the latest Google-powered features. It includes the update system from the *Google Play Store* and other integrations with the Google ecosystem. We used the *Play Services* with the notifications system.

SQLite and ORMLite

As its name suggests, *SQLite* is a light relational database management system. In contrast to many others, it is not a client-server database engine: it is self-contained and serverless [43, 33]. *SQLite* is fully integrated with *Android* within the package `android.database.sqlite` which makes its adoption easy.

ORMLite is a tool used alongside *SQLite*. We chose this tool because it provides lightweight functionalities for persisting Java objects to SQL databases [29]. By adding Java annotations to a class, one can store an instance directly in *SQLite*.

Spring for *Android*

As the *Spring* framework is used on the server side, a good solution to communicate with it is to use *Spring* for *Android*. This is a framework that is designed to provide components of the *Spring* family of projects for use in *Android* applications [31].

5

FEATURES, INTERFACES AND USE CASES

The design process kept us busy for some time, since it was important to think in depth about the core-features of the application before implementing them. A noteworthy point is that all the features described below were designed with the need for off-line synchronization in mind.

5.1 Features to implement

One of the first questions that we tackled was: “what could the relevant features of this app be?”. The following answers were discussed during the meetings.

5.1.1 Graph smoothing

In the beginning of the project, the application was only able to display a graph of the consumption. As you can see in Figure 6, the screen is not really readable due to the high peaks. Allowing the user to smooth the graph with an adjustable slider was thought to be a potentially interesting feature. Based on the data, an easy correction could be calculated for each sample by means of a linear interpolation.

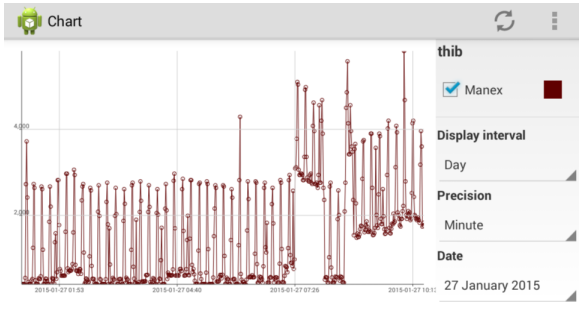


Figure 6: The graph is not easy to read.

5.1.2 Provide an analysis of energy consumption based on statistics

We thought that, given a period (day, week, month, year...), a user should be able to see an analysis of their consumption. For example, information about peak and off-peak consumption[1], average consumption, extrema etc. could be useful.

5.1.3 Savings between periods

This feature is related to the previous one. Based on the idea that we would retain information about a given period, it could be interesting to see the savings (or the losses) made between two periods. For example, one could see how and why the amount of an energy bill is significantly dropping.

5.1.4 Compare one's consumption to a given consumer profile

Couples, average household, big families... All these different profiles have varying bills at the end of the month. It could be interesting for an individual user to see how (s)he compares with standard profiles.

The consumer profile should be relevant enough to be significant for the user. One idea was to take localization into account (since you do not have the same consumer profiles in every country). Another idea was to build those profiles based on public *Flukso* data.

5.1.5 Evolution of consumption

A user's consumption could be impacted by a change of habit or behavior (for example, buying a new dishwasher). One idea was to allow the user to enter a comment on the graph at a given time. With this feature, (s)he could see the advantage associated with that new acquisition. Another feature suggested was to estimate the energy consumption and cost at the end of a period (e.g. a month) by extrapolation.

5.1.6 Manual consumption reading

To target people who do not have a smart meter, it could be useful to enter consumption data manually. However, it is not very likely to see someone using the mobile application this way in a day-to-day usage.

5.1.7 Alerts and abnormal cases

For example, if the consumption is starting to increase in an abnormal way, the system could draw the user's attention to this fact. Any problem related to the connection between the back end and a smart meter should also be reported.

[1] The term peak consumption is the English equivalent of "*électricité de jour*" while off-peak consumption means "*électricité de nuit*".

5.1.8 Retrieving and distributing pricing information

The back end needed to receive electricity price data in some way. A public API could help us to tackle this problem. Moreover, the application should use this information to display the cost associated with the energy consumption over a given period.

5.2 User interfaces and mockups

The second step in the design process was to think about user interfaces by drawing *mockups*[2]. The application offers different screens which were discussed during the meetings. Several of them, such as the login screen, are based on a previous version of the mobile application, and will not be discussed here.

5.2.1 Main screen

The main screen of the application is the one which displays the graph. It is composed of a line chart of the user's consumption and a panel with options to choose sensors, intervals and dates. Although it is largely based on a previous version of the application, several parts have been improved: the integration with the rest of the application; the toolbar; the display of the options on smaller screens; and the smoothing slider. The mockup is shown in Figure 7. Notice the reload button in the upper right corner.

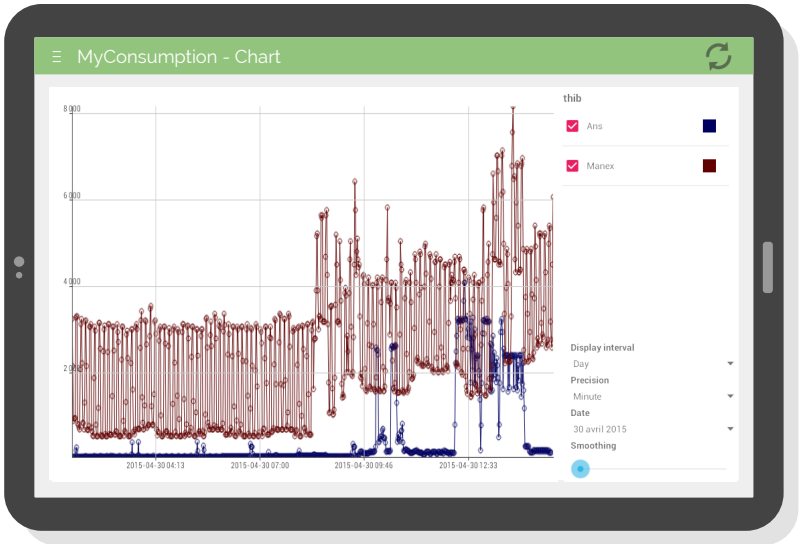


Figure 7: The mockup of the main screen of the application with two sensors.

[2] A *mockup* is a prototype of a design used for demonstration purposes.

5.2.2 Statistics

The second screen we discussed displays the statistics. The challenge here was to provide a lot of information on the same screen. Different colors and a graph were used to draw the user attention to key points. As several periods are needed, the idea was to use tabs to display them easily. To switch between sensors, a drop-down menu was added to the toolbar. The mockup is given in Figure 8.

As far as the statistical items[3] are concerned, we thought it was relevant to include:

- The consumption over the period (kWh);
  - The associated cost in €;
  - The associated environmental footprint (in kg of CO_2);
- The difference between the consumption over this period and the last one (kWh);
- The average consumption (W);
- The maximum and minimum values (W);
- The consumption during high peaks (over the day (kWh)) and off-peaks (over the night (kWh)).

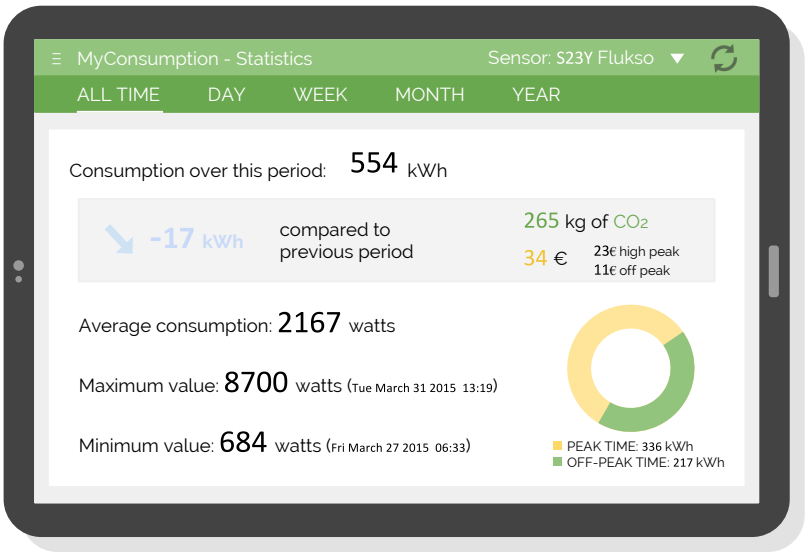


Figure 8: The mockup of the statistics screen.

[3] Note that it is important to differentiate the units associated with the statistics. Kilowatt-hours, or kWh, is an *energy* unit which describes the *total amount* of electricity used or produced over a period of time. Watts, or W, is a *power unit* which describes the *rate* of using or producing electrical energy (or how much is being used right now).

5.2.3 Profile comparison

As discussed above, a relevant feature we identified was for a user to be able to compare their consumption to a standard profile. At the early discussion stage of the work, we did not know how the standard profiles would be computed. This uncertainty is the reason why the mockup proposed in Figure 9 is quite simple. It is composed of:

- A recall of the description of the standard profile selected and a button to modify it;
- A little comparison between the real consumption and the one of a standard profile;
- A graph that highlights the difference between the standard profile and the current consumption.

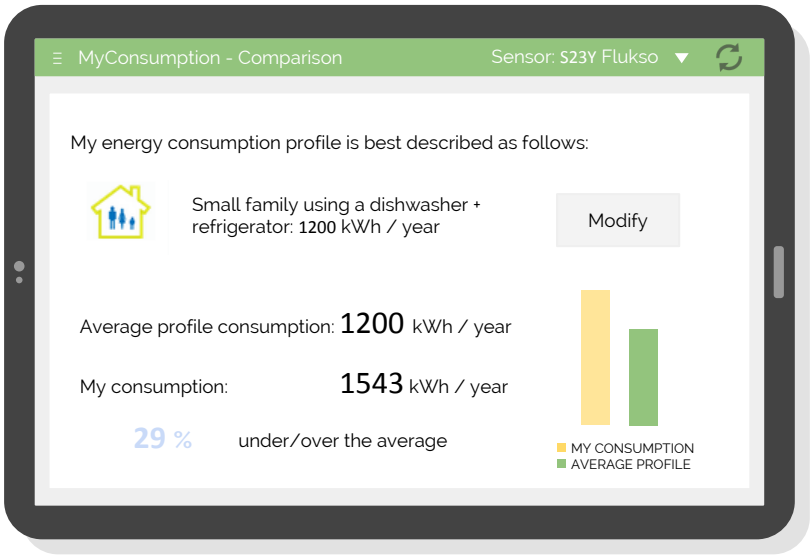


Figure 9: The mockup of the comparison screen.

5.2.4 Settings

Last but not least, a settings screen will allow one to specify different preferences. For example, a user could see a possibility to receive notifications from the server, select their standard consumption profile, enter their annual consumption if (s)he knows it... A description of this screen is given in Figure 10.

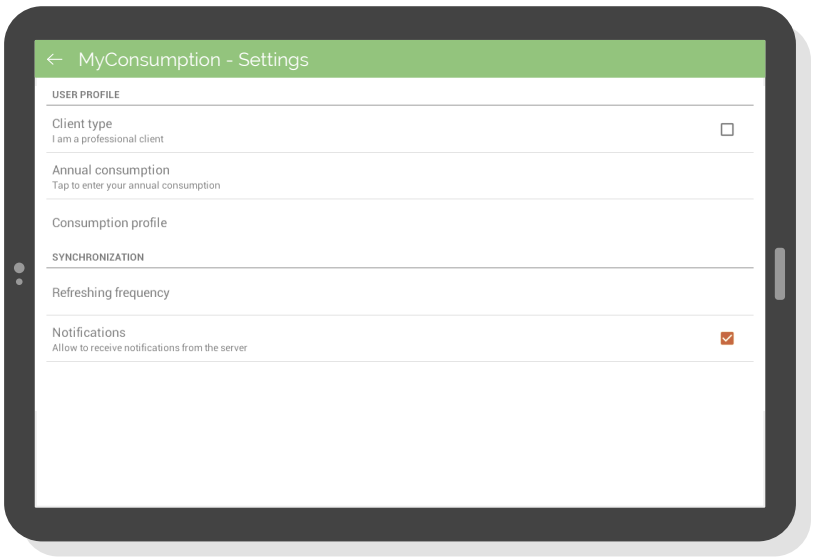


Figure 10: The mockup of the settings screen.

5.3 Defining use cases

As a use case represents a typical interaction between a user and the system [11], not all the features discussed below will be addressed in this section. The use cases defined here correspond to high-level goals and are described as if a virtual camera was filming interactions between the system and its users.

5.3.1 Log in to the application

Actor	User
Goal	Accessing the application by providing a username and password.
Overview	The User starts the application and a form requests them to enter their credentials. If the User has no credentials, an option is available to him/her to create a new account by entering a name and a password. When those credentials are provided on the login screen, the User clicks on a button to access the app. The main screen is then displayed. If the credentials do not match any user, a message is displayed asking the User to reenter them.

Table 1: Log in to the application.

5.3.2 Switch between screens

Actor	User
Goal	Switching from the current screen to another one.
Overview	The User is on a screen of the application. By pushing an icon on the upper left corner of the screen or by sliding from the left of the screen to the center, a sliding panel is shown. This panel offers to the User the option to choose a new screen to display. By clicking on one of them, the current screen is replaced by this one and the sliding panel disappears.

Table 2: Switch between screens.

5.3.3 Smooth the graph

Actor	User
Goal	Smoothing the graph to make it clearer.
Overview	The User is on the main screen of the application which displays a graph of his/her consumption. The User uses a slider to set a smoothing value. When the value is set, the graph is updated.

Table 3: Smooth the graph.

5.3.4 Reload the data

Actor	User
Goal	Reloading the data from the server.
Overview	The User is on the one of the screens of the application which displays the data. The User clicks the reload button on the right upper corner of the screen. The system replaces the current data with a reloading animation. When the reload is complete, the new data appear.

Table 4: Reload the data.

6

SECURITY

For some mobile applications security might not be an issue, but in our case we are dealing with sensitive information. Indeed, if a burglar could access the energy consumption reports of a given house, (s)he could determine whether people are present in the building or not, making theft far easier.

The mechanism that determines whether someone has access to a system is called *authorization* [6]. This process is closely related to *authentication*, which is always the first step in *authorization*: Bob may be *authorized* to view some confidential files of a company, but until he is *authenticated*, the system cannot be sure that this person is really Bob (and if he is not, he should not be *authorized* to access the files).

This chapter aims to address both *authorization* and *authentication* issues related to *MyConsumption*. Solutions are discussed for each specific security concern in the different parts of the system.

6.1 The first version of MyConsumption

The first version of the mobile application had a simple mechanism of username-password authentication: to authenticate and establish his/her identity, the user must enter their username and password when prompted. However, this password was used only on client side to authenticate. Additionally, it was sent in plain text over the internet to the server where it was likewise stored as plain text.

Moreover, the RESTful service had no security mechanism. Every resource was accessible to anyone and the exchanges were unencrypted. Finally, no credentials were provided with the *MongoDb* server, meaning that anyone who could access the same network as the server could also access all data it contains.

6.2 Different authentication mechanisms

To add a layer of security to the system, different possibilities were offered to us. Before describing our implementation, this section describes the various options at hand. It is based on the Part 4: "Securing Your Application with Spring Security" of the book "Professional Java for Web Applications" [6].

6.2.1 Basic authentication

This HTTP authentication protocol enables both authentication requests and authentication challenge responses. To access a resource protected by basic authentication, the username and password are provided and encapsulated in a request with an authorization header that contains the key-word `Basic` followed by the Base64-encoded credentials:

```
GET /support HTTP/1.1
Host: www.s23y.org
Authorization: Basic Sm9objpncmVlbg==
```

Vulnerabilities

Because Base64 is only an encoding algorithm, the credentials are sent in plain text and vulnerable to sniffing. Attackers can access username and password, but also protected resources. This issue can be circumvented by using HTTPS requests (which protects the credentials from snooping (man-in-the-middle) and prevents replay attacks).

Another vulnerability is added if the password is stored in plain text on the server side. However, modern web servers provide mechanisms for storing passwords using one-way hashes.

6.2.2 Digest authentication

Where HTTPS is either not an option or not wanted, a digest access authentication can achieve a certain level of security over HTTP using MD5 checksum algorithm with two different nonces (one nonce for the server and one for the client) and a serial request number (to prevent replay attacks). The password is kept secure as it is never sent over the network and it is stored in hashed format.

Nevertheless, due to recent attacks on MD5, which is now largely obsolete, digest authentication is generally not a security mechanism one should rely on anymore, outside of the security of HTTPS. And even over HTTPS, it is recommended to still use digest over basic when possible.

Client certificate authentication

This is one of the most secure authentication protocols, even if it involves no username or password. It requires HTTPS. The server communicates with the client using public/private key pairs. On the client side, the private key is securely generated and stored on the machine.

The disadvantage of this protocol is that it requires the user to hold a little IT knowledge and users cannot easily use a different computer to authenticate. Nevertheless, it has the huge advantage that credentials cannot easily be compromised.

Claims-based authentication

This mechanism uses a trusted third-party application to authenticate users. For example, one can use Facebook to authenticate to another site.

As shown in Figure 11, when a user attempts to access a protected resource, (s)he is redirected to the third-party application. After successful authentication, (s)he can access the resource with a claim asserting his/her identity. Many different protocols, such as OAuth and SAML, implement this type of authentication.

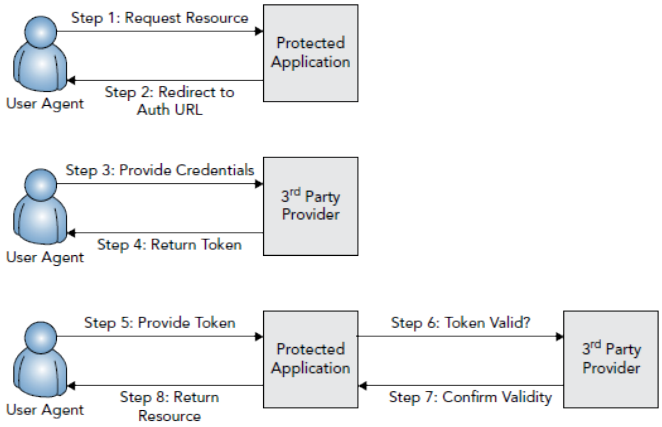


Figure 11: Claims based authentication [6].

Multi-factor authentication

All the protocols mentioned so far only provide one authentication factor. As its name suggests, multi-factor authentication requires the user to perform several steps to identify, making its security advantages significant.

For example, today's websites send a text message to a known user's phone number with a special code to enter. Another example could be to present a client certificate and a username and password.

6.3 Securing the RESTful web service

There were multiple concerns to consider while securing the web service:

- Man-in-the-middle attacks, which can be addressed by using strong SSL mechanisms;
- Stolen authentication tokens;
- User authentication and authorization to perform certain tasks;
- All kinds of HTTP protocol vulnerabilities.

6.3.1 Choosing an authentication mechanism

The digest authentication mechanism presented above was rejected because of the MD5 checksum vulnerabilities. The client certificate authentication did not appear to be convenient either, given the user knowledge it requires. Finally, multi-factor authentication appeared too complex for the goal we wanted to achieve.

That left two possibilities regarding the mechanisms discussed previously: basic authentication over HTTPS and claims-based authentication.

OAuth2

One of the most popular claims-based authentication protocols is OAuth. It allows a user to access a web application by authenticating him/herself with a trusted third-party application (see Figure 11). The credentials of the user are not shared with the web application. It is an excellent protocol for securing API services from untrusted devices, and it provides a good way to authenticate mobile users thanks to tokens authentication [20].

The `spring-security-oauth2` dependency is dedicated to use OAuth2. It required us to choose a trusted third party. As we are targeting *Android* device, Google could have been a good choice. Nevertheless, we chose not to implement this protocol for the following reasons. Firstly, the application would have been adapted to handle a login with a Google account. Secondly, it would have required more configurations on the server side and the documentation of *Spring Boot* related to OAuth was quite poor. Finally, this decision was made in the last month of the project and we opted for a simpler solution, which is described in the next section.

Basic authentication over HTTPS

Enforcing basic authentication for our web services could fit very well with the stateless nature of REST [6]. Its stateless property asserts that each request from any client contains all the information necessary to serve the request. In other words, no client context needs to be stored on the server between requests [39].

Accordingly, the idea we had is to send the credentials with each request. However, there is one major problem: the client application must know and retain the credentials for the entire duration that the web services are in use. This, however, may be fine if the client application runs only on the user side, which is the case in *MyConsumption*. Moreover, a stateless basic authentication would ease the configuration on a distributed server architecture (as there would be no need to centralize session and token IDs).

Finally, basic authentication is easy to set up with the `spring-security` dependency. Although many Java security frameworks are available, *Spring Security* is perhaps the most popular for web applications and, being a *Spring* project, integrates seamlessly with the *Spring* Framework.

This security mechanism was selected and its implementation is described in the next part of the document. A solution to store the credentials is additionally discussed below.

Storing the credentials

There was still one caveat regarding the credentials: they needed to be stored securely. An effective method of doing so is to:

1. Generate a long random salt;
2. Prepend the salt to the password and hash it with a standard cryptographic hash function (SHA256 for example);
3. Save both the salt and the hash in the user's database record.

The function related to this process could be expressed as:

$$pwd = hash(hash(password) + salt)$$

where *password* is in plain text. Then, to validate the *pwd*, one would have to:

1. Retrieve the user's salt and hash from the database;
2. Prepend the salt to the given password and hash it using the same hash function;
3. Compare the hash of the given password with the hash from the database. If they match, the password is correct.

Thanks to the salt, attackers cannot generate a rainbow table to crack the password in seconds. Still, the system must be designed such that the only way attackers can access the salts is by breaking into the database.

6.3.2 Authorization

The `User` object that comes with *Spring Security* is robust as it supports features such as password expiration and accounts being locked. *Spring Security* provides

a system of roles that can be defined. For example, it is possible to define the roles ‘ADMIN’ and ‘USERS’. The admin could access to more resources than other user roles. Moreover, one can prevent the user from accessing resources (s)he does not own (for example, data about another sensor in our specific case) with a simple check regarding the user authenticated.

Part III

IMPLEMENTATION

7

BACK END

This part of the document discusses the implementation process. It is divided in two: this chapter describes the server side of the system and the other depicts the *Android* client. It tries to address every feature and issue we faced with appropriate detail. However, the description below does not faithfully follow the implementation process in every aspect. For further details, it may be interesting to dive into the source code, available at https://github.com/S23Y/.

We wanted the mobile application to communicate with our database over the Internet. In order to meet this need, a simple solution was to exchange JSON objects through a RESTful API, which is the main function of the server. Moreover, as the application is just an interface between the data and the user, the server has to handle most of the computation needs of the project. The desired features of the server are the following ones:

- Deploy RESTful web services and a Java API to exchange data with the mobile application;
- Retrieve and distribute pricing information;
- Manage different users;
- Compute statistics over particular sets of data;
- Retrieve data from the smart meter API;
- Backup them in a database;
- Notify users of abnormal consumption events.

7.1 Structure of the server

The server is divided in two parts: a Java API and a business part. The Java API is just a collection of Java objects that are common to the front end and the back end. The business part is the core of the server. It consists of the three following folders (see Figure 12):

- `java`: the sources of the application divided into several packages;
- `resources`: a folder which contains the properties of the server;
- `test`: some classes used during the tests of the system.

Figure 12: Structure of the server.

7.1.1 Moving from Spring to Spring Boot

The first server designed by Patrick was built upon *Spring*. Alongside its other features, this open source framework simplifies dependency injections, allows the deployment of web applications (like RESTful web services), and facilitates database access.

Sadly, the configuration of the server was extremely time consuming. It took several days to achieve a working configuration on a new machine because some parts weren't set correctly (XML configuration files weren't as modular as they should have been, things tended to crash when refactoring the code etc.). Given this, we decided to move the server to a proper configuration. This move was even more necessary because an open source solution such as the one presented in this work has to be easy to deploy.

There simply couldn't have been a better choice than *Spring Boot* to tackle this issue. Indeed, *Spring Boot* makes it easy to create stand-alone, production-grade *Spring* based applications that you can "just run" [32]. It provides a faster and accessible getting started experience for all *Spring* development. This tool is relatively new, since the project was started on *Github* in 2013. Using *Spring Boot*, we were able to reuse the code of the previous version of the server with a simple configuration.

7.1.2 Maven

Spring Boot comes with *Maven*, a build automation tool used primarily for Java projects. It consists of a `pom.xml` file which lists all the dependencies of the project. For example, the one below imports the web module of *Spring Boot*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Maven has different *lifecycles*. For example, one can easily package its project just by running: `mvn package`. Besides lifecycles, plugins can be added to *Maven* in order to better tune the configuration. For example, the plugin `spring-boot` simplifies the way the server is handled.

7.2 RESTful services

The Representational State Transfer (REST) is a software architecture style, consisting of guidelines and best practices for creating scalable web services [39]. It is a means of expressing specific entities in a system via URL path elements. It minimizes the coupling between client and server components in a distributed application; one of the goals we wanted to achieve.

7.2.1 Web server architecture

The web module of the server is made up of three main packages:

- Entities: each one is an object in its simpler form (e.g. a `User` or a `Sensor`);
- Repositories: each one handles specific access to the database;
- Controllers: each one deploys a particular web service.

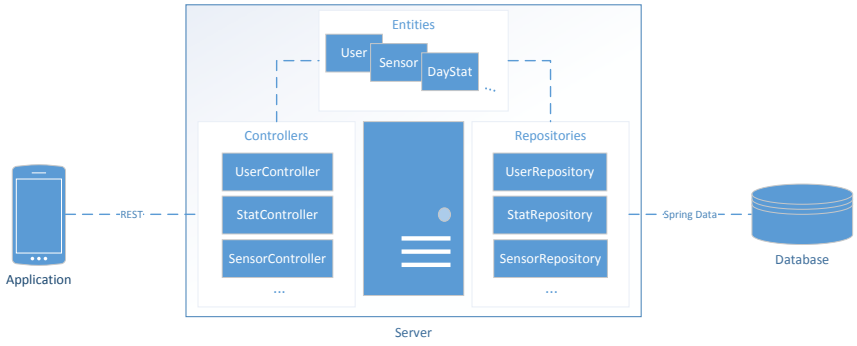


Figure 13: How the server exposes the web services.

As can be seen in Figure 13, each entity is associated with its corresponding controller and repository. For example, the `User` entity represents a typical user (composed of a name, a password, a list of sensors owned etc.). By using the `UserRepository`, one can store a given user on the server database. When the application requests information about a particular user, the `UserController` will try to access the corresponding entity in the database.

7.2.2 REST API

The RESTful API has been built bearing in mind that it should be both friendly to the developer and explorable with a browser address bar. The goal here was to follow some guidelines to make it intuitive, simple and consistent. One of the key principles of REST involves separating the API into logical resources [30]. Our resources are:

- Users;
- Sensors;
- Statistics;
- Notifications;
- Configurations.

Each resource is represented by a controller. *Spring Boot* controllers are considerably easier to implement than *Spring* ones. All of the code of this section has been updated from *Spring* to *Spring Boot*. Below, Listing 1 illustrates how to implement such a controller. The latter will return an array of JSON objects representing each sensor. It is accessible at http://myconsumption.s23y.com/sensors.

```
@RestController
@RequestMapping("/sensors")
public class SensorController {
    @Autowired
    private SensorRepository mSensorRepository;

    @RequestMapping(method = RequestMethod.GET)
    public List<SensorDTO> getAllSensors() {
        return mSensorRepository.getAllSensors();
    }
}
```

Listing 1: Example of a REST controller.

In the snippet of code Listing 1, a lot of annotations are used to tell *Spring Boot* the role of every element. For example:

- `@RestController`: tells that this class is a controller;
- `@RequestMapping("...")`: defines the path to access this resource;
- `@Autowired`: on a property, this annotation allows us to get rid of the setter methods (*Spring* assigns those properties with the passed values or references).

The code above was simplified for readability (primarily, security related elements were hidden). Of note is that critical resources of the RESTful service are protected with a layer of security. The user needs to be authenticated to access a resource and (s)he cannot access content that (s)he does not own. This will be

discussed in a following section about security.

Parameters and other types of request are also supported by *Spring Boot*. The complete description of the API of the RESTful web services is available in the Appendix.

7.3 Java API

Since the application and the server access the same objects between JSON exchanges, a Java API is provided. It gathers several classes that are common to both of them (see the UML diagram in Figure 14). The API is imported to the server and in the application using the dependency injection mechanism. For example, with *Maven*, simply add:

```
<dependency>
        <groupId>org.starfishrespect.myconsumption</groupId>
        <artifactId>my-consumption-api</artifactId>
        <version>1.0.0-SNAPSHOT</version>
</dependency>
```

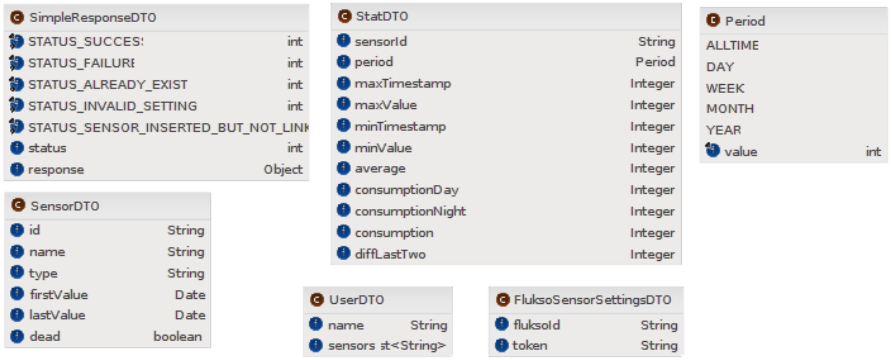


Figure 14: UML diagram of *MyConsumption* API.

7.4 Retrieve and distribute pricing information

In the project description, the back end was made responsible for retrieving and distributing pricing information. To retrieve the prices, one idea was to use a public API. Sadly, we were unable to find one. Moreover, it would be quite complex to design the application at an international level because we would have to find pricing information in every country and implement an automatic retrieval on our server for each.

We also investigated the possibility of retrieving this information from a website such as a price simulator. Again, this approach would not have been sufficiently scalable and reliable as no generic tool could be found to access the data.

We ultimately decided that it was not relevant to investigate or focus further on this retrieval. A workaround was then considered. We defined two values on the server side [28]:

- The peak price: 0.07€/kWh;
- The off-peak price: 0.0525€/kWh.

Moreover, a value to convert a kWh to its CO_2 equivalent (in kg) is also given. It is based on an average over the values found in a study [12] that compares different energy providers which gave us the value of 0.48 kg of CO_2 per kWh.

As far as implementation is concerned, a REST service is defined by a controller to give access to the three default values (peak price, off-peak price, and CO_2 conversion).

7.5 Managing different users

A simple user management system was integrated with the server. A `User` is composed of a name, a password (hashed on the application side, it is never sent in plain text over the network), a list of sensors owned by him/her and a registration ID (an ID associated to the user and the device used with the notification system[1]).

The `UserController` is responsible for creating, deleting and modifying users. Modification includes the addition or deletion of a sensor owned. Users can only access their own resources and they need to be authenticated. This is discussed in the security section of this work.

7.6 The Watcher

The `Watcher` is a task whose purpose is to retrieve data from sensors, compute statistics, and send notifications. It is triggered periodically and it consists of a `Retriever`, a `StatisticsUpdater` and a `Notifier`.

7.6.1 Retriever

Note: this part comes from the previous version of the server.

[1] This will be discussed in a following section dedicated to notifications.

The goal of the retriever is to collect data from a sensor manufacturer API and store this information in our database. As shown in Listing 3, two methods are used: `getAllData()` and `getDataSince()`.

```
public interface SensorRetriever {
        public abstract SensorData getAllData() throws RetrieveException;
        public abstract SensorData getDataSince(Date startTime) throws
            RetrieveException;
        public abstract SensorData getData(Date startTime, Date endTime)
            throws RetrieveException;
}
```

Listing 2: The SensorRetriever interface.

A note about modularity

At the time of writing, only one kind of sensor is supported by the server (*Flukso* sensors). Nevertheless, the design of the retriever allows the simple addition of new types of sensors. Indeed, the retrieval is based on several Java interfaces and generic structures (see Figure 15). If we can match a manufacturer's API with our interfaces and structures, then we can easily extend the compatibility of the server with this new type of sensor. Thanks to this technique, only the retriever needs to be adapted. The main parts of the server and the mobile application need no change at all.

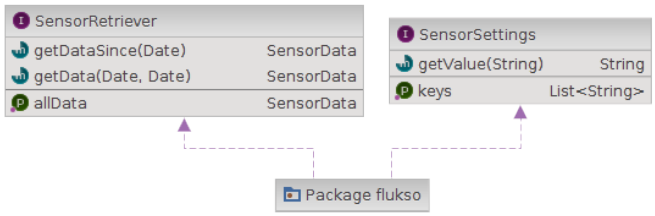


Figure 15: The *Flukso*-dependent code implements two interfaces.

7.6.2 Statistics updater

The `StatisticsUpdater` is a key element of the server because many features of the application are based on it. In particular, as described in the design part of this work, we wanted the user to be able to:

- Get an analysis of their own consumption based on statistics;
- See their savings between periods;
- Compare their consumption to a given consumer profile;
- Be notified in case of abnormal consumption.

To address these four features, we defined five periods: a *day*; a *week*; a *month*; a *year*; and *all time*. For each period, we associated a statistical item. We defined:

- The consumption over the period (kWh);
- The difference between the consumption over this period and the last one (kWh);
- The average consumption (W);
- The maximum and minimum values (W);
- The consumption during high peaks (over the day (kWh)) and off-peaks (over the night (kWh)).

This section explains how these items are computed.

First approach

The first approach described here came from a significant mistake we made. Indeed, we initially designed the `StatisticsUpdater` based on a wrong assumption about the overall system.

The smart meter API of *Flukso* is designed to return pairs of two values. The first one is the consumption (in Watts) and the second one is the corresponding timestamp. But the tricky thing is that the difference between those timestamps is not constant. Here is what the *Flukso* API returns:

- A pair of values every minute for the last 24 hours;
- A pair of values every 15 minutes for the last 7 days;
- A pair of values every day for the last year;
- A pair of values every week for older values.

To compute statistics, it is helpful to have a good understanding of the different periods. At the beginning of this work, therein lay our weakness: we designed and implemented the computation of the statistics by mixing values associated with different periods. Of course, the results achieved did not reflect the reality.

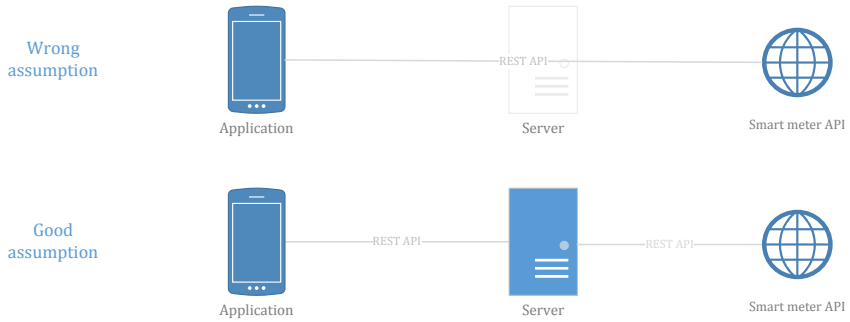


Figure 16: Misunderstanding of the role of our server.

A second approach

The crux of the matter is that we were misunderstanding the role of our server (see Figure 16). Indeed, the first approach was designed with the *Flukso* API in mind. This was incorrect because our application has nothing to do with the API of a specific smart meter provider.

The only values we should have taken into account were those present in the database of our server. As the server is always on, we chose to keep pairs of values with a difference in timestamp of one minute. This is a far better assumption: values are no longer mixed and we were able design the computation of the statistics with our server API in mind. This approach also simplified the implementation.

Day by day

We wanted to compute statistical items over five periods: a day; a week; a month; a year; and all time. We chose to keep a `DayStat` object in our database. It represents the smaller period (see Figure 17). Every day, we compute a new `DayStat` with the latest pairs of values retrieved. Other periods are built by adding `DayStat` objects. For example, a week is simply composed of seven `DayStats` objects.

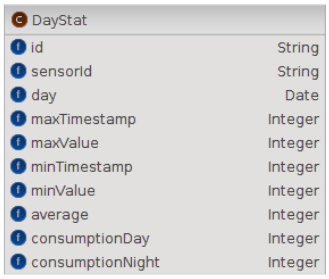


Figure 17: A `DayStat` object.

We have values of consumption, measured in Watts. As a power unit, this value describes how much is being used at a given timestamp. The average consumption (W) is simply the average of the values of the day. The maximum and minimum values (W) are likewise intuitive in their formulation.

To compute the other values, we needed to convert them in an energy unit (Wh per day). For example, to compute the consumption over the period we started by summing each pair of values. As this value corresponds to Watts taken every 60 seconds, summing these values does not mean much. We need an energy (Wh) which is the total amount of electricity used over a period of time. For a day, this means we had to divide by 60 (the numbers of values taken in one hour). The same process was applied to the consumption during high peaks (over the day (kWh)) and off-peaks (over the night (kWh)).

A note about modularity

A noteworthy point is that, thanks to the Data Transfer Object (DTO) model and the Java API we provided, the implementation behind every part of the server (and thus the statistics) remains completely independent of the mobile application. We simply have to update the server and match the Java API, and the client does not need to be modified.

7.7 Database

This section describes one of the goals of the back end: ensure synchronization, distribution and backup of energy consumption data.

In the design section of this work, we explained the choice of *MongoDb*. It should not be forgotten that this database is not structured around the traditional table-based relational model. Instead, JSON-like documents with dynamic schemas are used. These documents have the advantage of making the integration of data in certain types of applications easier and faster.

To communicate with the database, the *Spring Data Framework* was used. It provides a set of *template classes* which are used as a flexible abstraction for working with the data access framework [42]. We used *Spring Data MongoDb*, which focuses on storing data in *MongoDb*. It inherits functionality from the *Spring Data Commons Project*. With such a framework, we do not have to write any *MongoDb* queries because they are written for us.

We defined several entities that are stored in the database. They are described in the UML diagram in Figure 18. For example, a `User` has an ID, a name, a password, a list of sensors and a register ID[2]. This entity is stored in *MongoDb* with the help of a repository.

7.7.1 Moving to Spring Boot

To explain how entities are stored, it is important to describe some work carried out when we updated the server to *Spring Boot*.

The old version of the server was running *Spring* (not to be confused with *Spring Boot*, which simplifies the overall configuration). With *Spring*, specific *MongoDb* operations were used. While this approach enables the programmer to do exactly what (s)he wants, it is prone to errors and needing more code to be written. For example, to access an entity, we would first have to check if it exists before accessing it. Moving to *Spring Boot* made things even simpler.

[2] The register ID is used for the *Android* notification system.

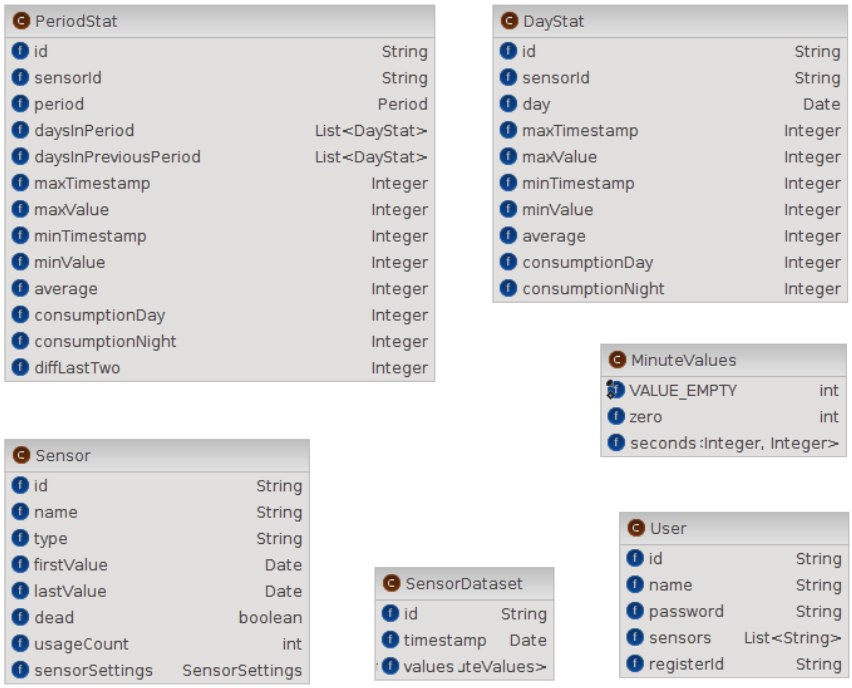


Figure 18: Entities stored in *MongoDb*.

Repository interface

Spring Boot works with repositories. It is an interface that, *out-of-the-box*, comes with many operations, including standard CRUD operations (Create-Read-Update-Delete). The idea is to define queries by simply declaring their method signature. Each repository we created extends some kind of CRUD repository. The latter provides operations to:

1. Save the given entity;
2. Return the entity identified by the given ID;
3. Return all entities;
4. Return the number of entities;
5. Delete the given entity;
6. Return whether an entity with the given ID exists.

For example, the `DayStatRepository` is composed of two functions: one to find a sensor by its ID, and another to find a sensor by its ID and date. Here is the code related to this repository:

It is indeed fairly simple. Following this, it is only necessary to `@Autowired` a `DayStatRepository` where it is needed. After these steps, the two operations above, as well as the basic CRUD operations, are made available.

```
public interface DayStatRepository extends MongoRepository<DayStat,
    String> {
    List<DayStat> findBySensorId(String sensorId);
    List<DayStat> findBySensorIdAndDay(String sensorId, Date day);
}
```

Listing 3: The DayStatRepository interface.

CustomRepository

In the old version of the server, three entities were available: sensor; user; and value. Many operations were already implemented, and we did not want to implement everything again from scratch. A simple solution was to use a `CustomRepository` interface with a custom implementation. The process to use this scheme is described below:

1. Define an interface for our custom code:

```
interface CustomUserRepository {
  List<User> yourCustomMethod();
}
```

2. Add an implementation for this class and follow the naming convention to make sure we can find the class.

```
class UserRepositoryImpl implements CustomerUserRepository {
  private final MongoOperations operations;

  @Autowired
  public UserRepositoryImpl(MongoOperations operations) {
    Assert.notNull(operations, "MongoOperations must not be null!");
    this.operations = operations;
  }

  public List<User> yourCustomMethod() {
    // custom implementation here
  }
}
```

3. By setting our base repository interface to extend the custom one, the system will automatically use the custom implementation:

```
interface UserRepository extends CrudRepository<User, Long>,
    CustomUserRepository {}
```

In this way we gave ourselves the choice to extend our initial work with the simple configuration of *Spring Boot* and to reuse the existing code in the custom repository implementation (the code from the previous version of the server remains).

7.8 Notify users of abnormal consumption events

In the design section we discussed the need for an alert system. For example, if the consumption starts to increase in an abnormal way, the mobile application should draw the user's attention to this fact.

The solution we found is called *push notifications*. These are handled via the *Google Cloud Messaging* for *Android*. This service allows data to be sent from our server to all the users' *Android*-powered device. It handles all aspects of queuing of messages and delivery to the target *Android* application. This section explains its implementation in *MyConsumption*.

7.8.1 Push notifications: the workflow

Three actors are involved here: the mobile application; our server; and *Google Cloud Messaging* (GCM). The interactions between the three are described in Figure 19. The workflow is made up of the following steps:

1. The *Android* device sends a `SENDER_ID` to GCM server for registration;
2. After successful registration, the GCM server returns a `REGISTRATION_ID` to the mobile device;
3. The mobile device sends this `REGISTRATION_ID` to our server;
4. Our server stores the `REGISTRATION_ID` in its database and associates it with the current user of the application;
5. When our server needs to send a notification to a given user, it sends a request to GCM with the associated `REGISTRATION_ID` and message;
6. GCM sends the push notification to *Android* devices.

7.8.2 When to send a notification

We first based the sending of notifications on a simple scheme: every day the system told the user if (s)he was consuming more or less than the previous day. This was fine for testing purposes, but in a real environment, we do not want such intrusive notifications.

A more appropriate approach was outlined. It involves a threshold and a comparison between the user's daily consumption and their weekly consumption. Every day, the server checks:

$$((dayconsumption \times 7) - weekconsumption) > threshold$$

If it is true, then a notification is triggered. The default value of the threshold is 15%.

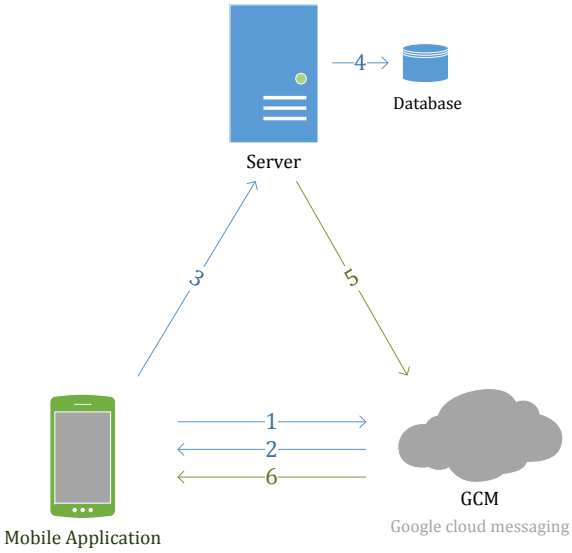


Figure 19: Notifications workflow.

7.8.3 Server implementation

Mobile device registration

We implemented a controller to allow a mobile user to register for the notification system. A POST request at `http://myconsumption.s23y.com/notifs/{name}/id/{regsterId}` sets the `REGISTRATION_ID` to a given user.

Send a notification

Two classes are involved in this process, as outlined below.

The first class, `NotificationMessage`, represents a notification message (see Figure 20). It is composed of different parameters that are described in the GCM server reference. For example, the `collapseKey` identifies a group of messages that can be collapsed, so that only the last message gets sent when delivery can be resumed. We have used the sensor ID as a `collapseKey` to avoid pushing multiple notifications for the same sensor.

The second class, `NotificationSender`, builds the request to send to the GCM server. This class is adapted from a Google repository on *Github*. The POST request is sent to https//android.googleapis.com/gcm/send. It is composed of an authorization header with the API Key of *MyConsumption*[3]. The HTTP body is also described in the GCM server reference.

[3] This public API key comes from the Google API project: https://cloud.google.com/console. It is used as the GCM `SENDER_ID`.

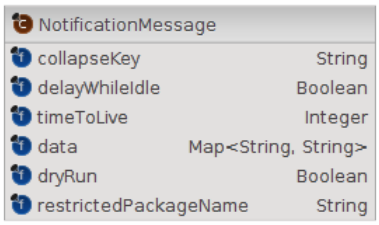


Figure 20: Representation of a notification message.

7.9 Security

In the design section of this work, different security mechanisms were detailed. At the end of the discussion, basic authentication over HTTPS was chosen because it meets the desired level of security while keeping the configuration and the implementation simple. The latter is described in this section.

7.9.1 Authentication

To add the desired level of security, we used the `spring-security` dependency. The two classes involved in the configuration are outlined below. Each one extends a default class of *Spring Security* to tune its configuration.

In `AuthConfig`, we simply bind our `User` entity to the one used in *Spring Security* (`UserDetails`). Thanks to this, users are now recognized by the authentication system of the server.

In `WebSecurityConfig`, we defined some specific policies. Indeed, when creating a new user on the application side, we are not yet able to authenticate. Accordingly, the service to create a user is accessible through anonymous authentication. Another interesting thing to note here is that this approach forces the server to have a stateless behavior, as advised in the design section.

The rest of the work is done by *Spring Security*. Each request requires an `Authorization: Basic` header followed by the Base64 encoded username: password pair. Moreover, we will see in the next chapter that the password is not sent as plain text over the network.

7.9.2 Authorization

Since we mapped our `User` implementation with *Spring Security*, it is possible to retrieve the identity of a specific user when (s)he tries to access a resource. This is done using a particular class of *Spring Security*: `Principal`. Thanks to this, it is possible to check whether the user has the right to access the desired content. In general, the simple check detailed below is enough.

```
if (!(principal.getName().equals(name)))
    return new SimpleResponseDTO(false, "you are not allowed to modify
        this user");
```

Listing 4: Check if a user can access a resource.

7.9.3 Storing the credentials

As the password is hashed on the client side, it is stored as it is received on the server database. The one way hash provide a first layer of security. However, to prevent attacks with rainbow tables, it could be better to use a randomly generated salt with the hash.

Setting up HTTPS for Spring Boot

HTTP protocol vulnerabilities, man-in-the-middle attacks, replay attacks and unencrypted exchanges can be avoided by using HTTPS. With *Spring Boot*, the configuration is simple: in the `application.properties` file, SSL is supported by adding the following information about the certificate.

```
server.port = 8443
server.ssl.key-store = classpath:keystore.jks
server.ssl.key-store-password = secret
server.ssl.key-password = another-secret
```

Listing 5: Setting up HTTPS for *Spring Boot*.

That is all it takes to set up HTTPS. However, even if one can generate a self-signed certificate for testing purposes, a real one will be needed for a release. Sadly, the latter needs to be signed by a certified authority. This process is paying and has not been done in the scope of this work.

8

FRONT END

The role of the *Android* application is to display data fetched from the server. While implementing the client, we tried to minimize the computational load on its side. In the future, this could help in porting the application to another platform. As far as *Android* is concerned, this application runs on 90% of the device active on the *Google Play Store*[1].

This part of the work aims to describe the choices made while implementing the mobile application. First, its structure is detailed. Next, some *Android* related concepts are introduced and described in the case of *MyConsumption*, such as activities, fragments and services. After that, internal communication is addressed. Finally, a section is dedicated to the data management.

8.1 Structure of the application

The package hierarchy of the front end is illustrated in Figure 21. Under the path `app/src/main`, two folders and a file are available:

- `java`: groups the java packages and classes;
- `res`: groups the XML resource files and folders;
- `AndroidManifest.xml`: a file which presents essential information which the system must have before it can run any of the application's code.

The Model-View-Controller (MVC) architecture is used to design *Android* applications. The idea is that any object in an application must be either a model object, a view object, or a controller object [3].

- A *model object* keeps the application's and data "business logic". These objects do not have any knowledge of the user interface; they only aim at

[1] On March 19, 2015.

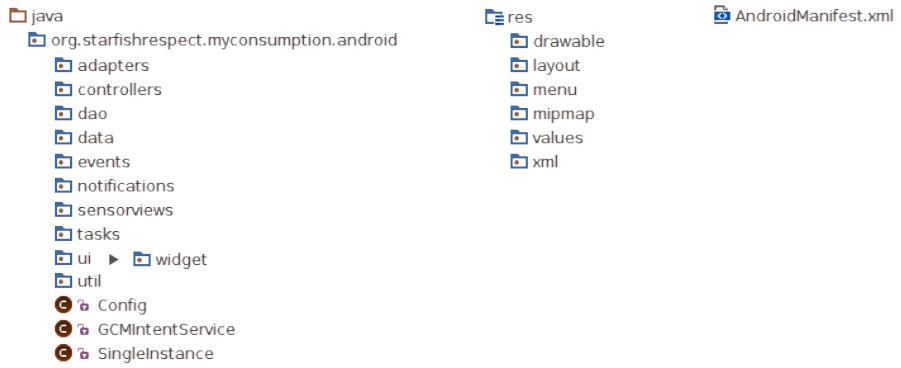


Figure 21: Structure of the application.

holding and managing the data. In *Android*, this category generally includes the classes we create, such as a user, a product in a store etc. Together they compose the *model layer*.

- *Views objects* are the those which know how to draw themselves on the screen. They also know how to respond to user input. Essentially, the things that can be seen on screen are views. This category mainly consists of widgets that are inflated from an XML layout. Together they form the *view layer*.
- *Controller objects* link the view and model objects. They hold the "application logic". They are designed to respond to events triggered by view objects. They also manage the flow of data to and from model objects and the view layer. In *Android*, a controller is typically a subclass of `Activity`, `Fragment` or `Services`.

8.2 XML files

8.2.1 Layouts

A layout describes the visual structure of a graphical user interface, such as the UI for an activity or an application widget. They can be declared in two ways: using an XML file or by instantiating layout elements at runtime. Nevertheless, it is recommended to declare UI in XML because it enables better separation of the application's presentation from the code that controls its behavior. In *MyConsumption*, four types of layouts are defined under the `res/layout` folder. They describe `Activities`, `Fragments`, `Widgets` and `items`. They are all declared using XML files.

8.2.2 Styles

We focused on making the mobile application intuitive and visually attractive. In `styles.xml`, a theme is defined: `Theme.MyConsumption.Base`. It describes the main colors, the default background, and some high level configurations to be used everywhere in the app. The theme of some widgets, such as the header bar, are also defined here.

8.2.3 Resolutions and orientations

Different resolutions are taken into account through the use of fragments (see section below). Moreover, specific dimensions are defined for three types of screen sizes in the three `dimens.xml` files: for `w820dp`[2] screens, `sw600dp` and a default configuration for the others.

As far as the screen orientation is concerned, different layouts are provided for the statistics screen and the comparison one. They both have a specific look in portrait and landscape mode.

8.2.4 Other XML files

Strings, values, arrays, preferences, drawables and menus are also defined in specific XML files, as recommended by the *Android* documentation.

8.3 Activities and fragments

An *activity* is a single, focused thing that the user can do [15]. In brief, it is a window where we can place UI elements. As a user navigates through, out of, and back to the app, the Activity instances transition between different states in their lifecycle (see Figure 22).

A *fragment* is a piece of an application's user interface or behavior that can be placed in an activity [19]. A key feature of fragments is that they can be used as an application's layout. This allows a better modularization of the code and to easy adjustment of UI to different screens (tablets, smartphones, TVs etc.). Just as for activities, fragments have their own lifecycle.

The design part of this report describes user interfaces, mockups and use cases. Their implementation is described below. The files related to this section are located in the package `ui`.

[2] *Density-independent pixel* (dp) is a virtual pixel unit used when defining UI layout, to express layout dimensions or position in a density-independent way [23].

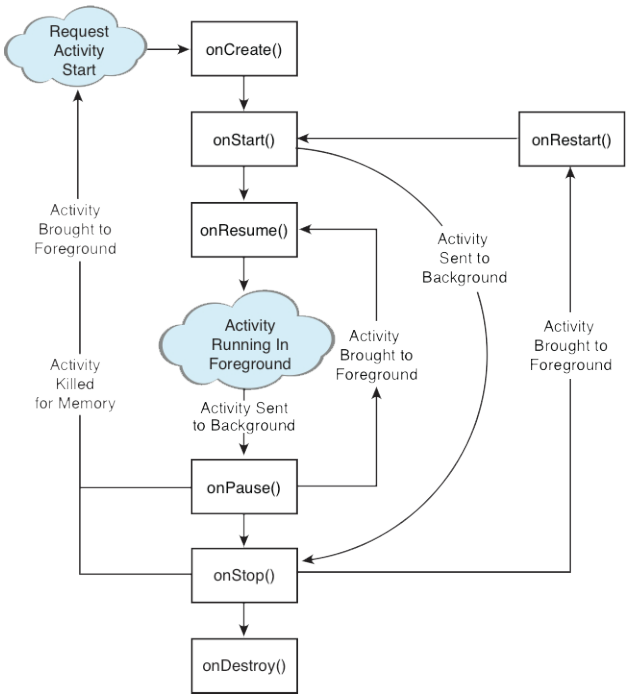


Figure 22: The lifecycle of an *Android Activity* [1].

8.3.1 Login

The `LoginActivity` comes from the previous version of the application. The layout has been improved to better match small screen sizes. This activity is displayed on the first launch of the application. Subsequently, user credentials are stored on the device.

If the user has no credentials, an option is available to create a new account by entering a name and a password. This is handled by the `CreateAccountActivity` (which comes also from the previous version of the application).

A note about security

In the previous version of the application, the user's password was only useful on the client side to log in. In the current version, the server is secured, meaning that we need those credentials for authentication purposes. The plaintext password is hashed and stored in the application's local database with the username. The content of the database can be accessed with root privileges, though the hash provides some degree of protection.

8.3.2 Switching screens

In the design part of this work, a use case describes how the user should switch between screens. A common way to do this on modern *Android* applications is to use a Navigation Drawer (see Figure 23). This is a panel that displays the main navigation options. It is located on the left edge of the screen and is hidden most of the time. It is revealed when the user swipes from the left edge of the screen, or when (s)he touches the app icon in the header bar.

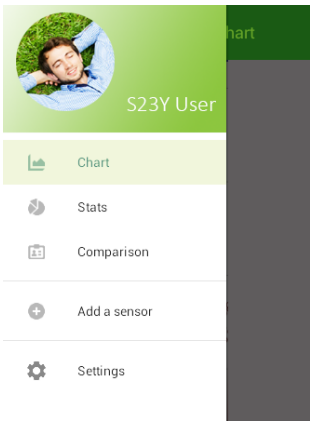


Figure 23: The navigation drawer.

Figure 24: A header bar.

8.3.3 BaseActivity

At the beginning of the project, we had trouble properly managing the activities with the navigation drawer and the header bar (see Figure 24). Different parts of the code were duplicated and interdependent. At a certain point, it became clear that it was simply not scalable.

The design presented below was inspired by one of the most popular *Android* repositories on *Github*: the Google I/O 2014 *Android* App. This application has been a great source of inspiration and a reference while building *MyConsumption*. Indeed, it is full of features, built by Google engineers and widely reviewed. As a result, *MyConsumption* has grown in modularity.

In short, as illustrated in Figure 25, every important screen of the application extends the `BaseActivity`. The latter is an abstract class whose purpose is to

regroup the elements that are reused in every Activity. Two important widgets are defined and handled in `BaseActivity`: the header bar and the navigation drawer. Moreover, common features of the app (such as the reloading option) are also implemented there. Adding a new activity to the navigation drawer is straightforward. Here are the steps in `BaseActivity`:

- Add the title of the new activity to the `NAVDRAWER_TITLE_RES_ID` array;
- Add a corresponding icon to the `NAVDRAWER_ICON_RES_ID` array;
- In the method `populateNavDrawer`(), add the new item to the list;
- Add a case in `goToNavDrawerItem`() to launch the new activity.

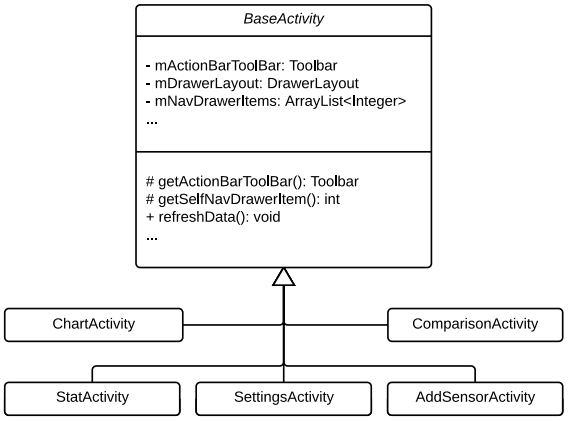


Figure 25: The activities that extend `BaseActivity`.

As a result, the code of a new activity is quite short. The Listing 6 shows the `HelloWorldActivity` class, which is a template that can be reused when creating a new activity. It is short and, thanks to `BaseActivity`, it supports *out-of-the-box* the navigation drawer and the header bar with menu items.

8.3.4 Main screen: ChartActivity

The `ChartActivity` is the first screen one can see when opening the application. This Activity was already present in the previous version of the application but it has been integrated with `BaseActivity` to match the overall design and behavior of *MyConsumption*.

The two fragments `ChartChoiceFragment` and `CharViewFragment` composes this screen. The first fragment displays the option on the right while the second one contains the chart itself. The chart is built with *Achartengine*, a library under the license Apache 2.0. It handles zooming, scrolling, and selecting data by clicking on them.

```
public class HelloWorldActivity extends BaseActivity {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main );
                Toolbar toolbar = getActionBarToolbar();
                toolbar.setTitle("My Consumption - HelloWorld");
                overridePendingTransition (0, 0);
        }

        @Override
        protected int getSelfNavDrawerItem() {
                // set this to have a nav drawer associated with this
                    activity
                return NAVDRAWER_ITEM_HELLO_WORLD;
        }
}
```

Listing 6: The `HelloWorldActivity` which extends `BaseActivity`.

8.3.5 Graph smoothing

In the offices of S23Y, the electrical devices are configured such that high peaks appear in the electricity consumption chart (see Figure 27). For the user, it is not always relevant to see those peaks. He/she might want to see only the general trend of their consumption on the graph. To achieve this purpose, a new option was given to the user to allow him/her to smooth the graph. As illustrated in Figure 26, we used a seek bar which makes it possible to select a value from a range by moving the slider thumb [21].

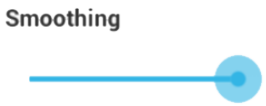


Figure 26: The seek bar.

The smoothing algorithm behind the inteface is a simple moving average. It creates a series of averages of different subsets of the full data set [38]. The reason for this choice is that the number of points displayed on the screen is relatively low (about 500). Consequently, an algorithm with a complexity of $O(n)$ is enough for something computed on the client side. Additionally, the formula is quite simple:

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - x_{n-N}}{N}$$

The result is illustrated in Figure 27.

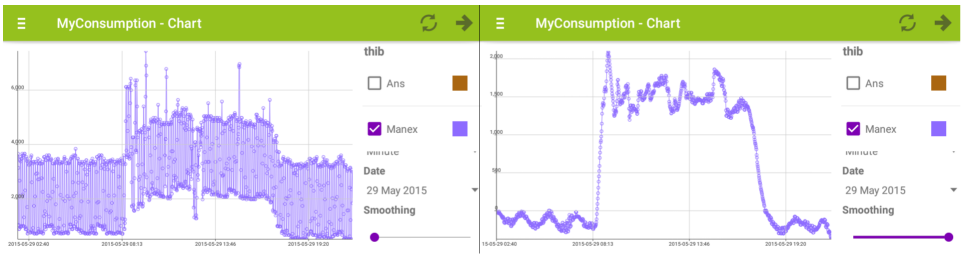


Figure 27: Smoothing option (be aware that the y-axis range has changed).

8.3.6 Statistics

This screen gathers different features described in the design part of this work. The main goal of the `StatActivity` is to provide an analysis of consumption based on statistics, but it also includes the savings between periods as well as the need to retrieve pricing information from our server.

Statistics are computed on the server side; `StatActivity` is only in charge of displaying them. In order to do so, sliding tabs are used. We chose to include a library for this purpose: *PagerSlidingTabStrip* (see Figure 28).

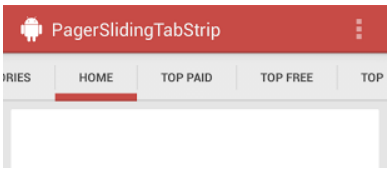


Figure 28: The pager sliding tab strip.

In `StatActivity`, an adapter is used to handle the tabs. Each item of this adapter instantiates a `SlidingStatFragment` which represents the content of each tab. A factory method is used here to create each instance of the fragment. It is the public static method: `SlidingStatFragment.newInstance(...)`.

The `SlidingStatFragment` is composed of `LinearLayouts` to build the UI. The pie chart is created by using a `PieData` object from the *MPAndroidChart* library. The choice of this library was motivated by: the popularity of the library on *Github*; its simplicity; and the large panel of features it offers (which might be useful for future developments).

8.3.7 Profile comparison

The `ComparisonActivity` allows the user to compare their consumption to a given consumer profile. The profile is defined in the preferences of the application and will be discussed later.

The UI of this activity is essentially based on `LinearLayouts` and a `BarChart` object from the *MPAndroidChart* library. From an implementation perspective, this Activity is quite similar to `StatActivity`.

8.3.8 Preferences

The settings of the application are built upon the *Android*'s Preference APIs. Every setting for the app is represented by a specific subclass of the `Preference` class of this API. A screenshot of this Activity is given in Figure 29.

As far as *MyConsumption* is concerned, we chose to group the preferences in two categories: user profile and synchronization. Inside the app, the preferences are easily accessible through a class called `PrefUtils`.

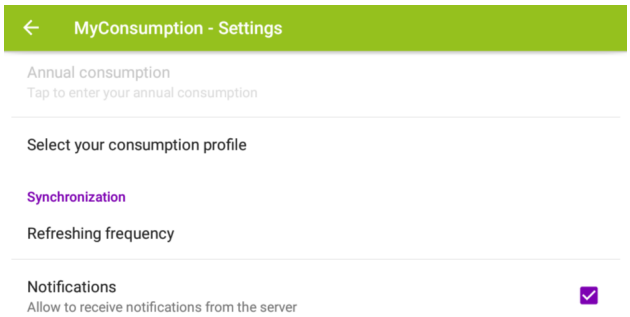


Figure 29: The settings of the application.

User profile

As described in the "Profile comparison" section, each user can choose a consumer profile between the three following ones:

- Studio / apartment with refrigerator: 600 kWh / year;
- Small family using a dishwasher + refrigerator: 1200 kWh / year;
- Average family with electric appliances + electric water heater: 3,500 kWh.

Those profiles were defined based on the CWAPE[3] simulator and a paper of the CREG[4] [13, 7].

Synchronization

A background task refreshes periodically the data from the server. Its refreshing frequency can be set in the preferences to 5, 10, 15 or 20 minutes. Moreover, the user can choose whether he wants to receive notifications.

[3] CWAPE: *Commision Wallonne pour l'Énergie* (http://www.cwape.be/).
[4] CREG: *Commision de régulation de l'électricité et du gaz* (http://www.creg.info/).

8.4 Internal communication

Many elements communicate between each other in an *Android* application. For example, network operations are always performed in a separate thread from the UI[5]. Thus, such an asynchronous task has to communicate the result of its execution with the main thread. This is not the only example: a fragment may want to deliver a particular message to its parent activity; views have to report that they have been modified (e.g. a button clicked); a background task is conducted; etc. This is illustrated in Figure 30.

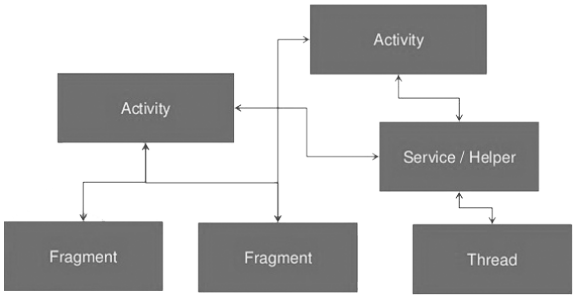


Figure 30: The complexity of internal communications in Android [24].

Of course, there is a way to handle those events in *Android*: by using callbacks. First, we have to define the callback interface:

```
interface MyCallback {
    void callbackCall();
}
```

Then, in a worker (e.g. an asynchronous task), we define this callback as a member of the class and we trigger the event at the right place and time:

```
class Worker {
   MyCallback callback;

   void onEvent() {
      callback.callbackCall();
   }
}
```

Finally, we handle the callback call by implementing the interface:

```
class Callback implements MyCallback {
   void callbackCall() {
      // callback code goes here
   }
}
```

[5] It is done this way to prevent a poor user experience due to unpredictable delays in network connections.

The process presented above is fine for simple operations. However, as soon as we move away from one of the few well-defined scenarios where *Android* can indeed update its views when background tasks result, things become more complex. As advised in the *Android* documentation, we used child classes of `AsyncTask` for all background works. These are launched in a different thread of the UI.

Nonetheless, the common following scenario is hard to manage. As soon as the activity view is displayed, we start an asynchronous task to load data from the server. At the end of its execution, a natural approach would be to tell the `AsyncTask` the modification to make on the view. This, however, is not the right way of doing things. Indeed, if the user rotated the device, the view would be destroyed and replaced by a new one. Thus, the `AsyncTask` would then have a reference pointing to a view which is no longer displayed.

8.4.1 An initial approach

The first solution employed to solve this problem was to use a Singleton which kept pointers to fragments and activities. By assigning tags when creating activities and fragments, we were able to get the references back to them in the Singleton even if they were destroyed by the *Android* system at a given time. Nevertheless, this homemade workaround had some drawbacks: it was hard to maintain and prone to errors.

8.4.2 A better solution: the event bus

The solution we subsequently found comes from a project found on *Github*: the *EventBus*. This is a publish/subscribe event bus optimized for *Android*, which simplifies the communication between components. It acts like the *Observer* pattern (see Figure 31 and 32). It is quite simple and it solves the problem of null references.

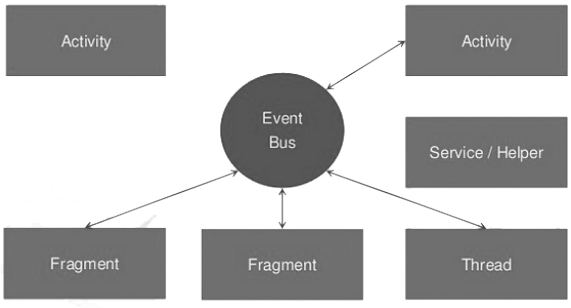


Figure 31: Event bus communication [24].

Firstly, an event must be defined. Luckily, they are simple Java objects which we can design as we please (see Listing 7).

```
public class MessageEvent {
    public final String message;

    public MessageEvent(String message) {
       this.message = message;
    }
}
```

Listing 7: An event message for the *EventBus* [24].

Following this, subscribers (activities, fragments or other components) implement event handling `onEvent` methods that will be called when an event is received. They also need to register and unregister themselves to the bus (see Figure 32 and Listing 8).

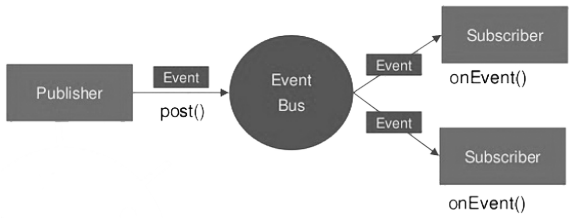


Figure 32: Publish / subscribe with the event bus.

```
@Override
public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);
}

@Override
public void onStop() {
    EventBus.getDefault().unregister(this);
    super.onStop();
}

// This method will be called when a MessageEvent is posted
public void onEvent(MessageEvent event){
    doSomethingWith(event);
}
```

Listing 8: Register and unregister from the *EventBus*.

Last but not least, a simple line is used to post an event from any part of the code. All subscribers matching the event type will receive it:

```
EventBus.getDefault().post(new MessageEvent("Hello everyone!"));
```

8.5 Data management

8.5.1 Reload the data

On the right hand side of the header bar, a button gives the option to reload the data. This is handled in the `BaseActivity` class. The content of the activity disappears while the task is processed, and a loading animation layout is shown.

The reload from our server is composed of two steps. First, the user information and the associated sensors are updated. Then, the values and the statistics corresponding to each sensor owned by the user are updated. As far as the values are concerned, only the new ones are downloaded.

Communication with the server

Each operation between the server and the client uses our Java API. To communicate with the server, the `RestTemplate` class of *Spring Web* is used. It easily handles HTTP requests (GET, POST, DELETE...) to the RESTful services. The conversion between JSON and Java objects is handled by *Jackson*, a tool used for mapping the two. Each request is conducted in a separate thread with the help of an AsyncTask.

Regarding security, a static method from `CryptoUtils` creates the basic authentication headers for each request. They are added to the `RestTemplate` before sending the request.

8.5.2 Off-line synchronization

To ensure off-line usage, the data fetched from the server are synchronized in a *SQLite* database on the mobile device. The structure of this database is quite simple because very little information needs to be stored. *ORMLite* is used to store data that can be represented as objects (e.g. a sensor). Two controllers are used to store the data associated to a user and to the statistics. A particular object, `SensorValueDao`[6] is used to store the list of key-value pairs associated with a sensor.

A class called `DatabaseHelper` was built on top of *ORMLite* to handle the creation, upgrade and deletion of tables. It has also getters for the different types of values.

[6] The Data Access Object (DAO) pattern is used to separate low-level data accessing API or operations from high-level business services [34].

8.6 Configuration, SingleInstance and util classes

The file `Config.java` groups important global settings of the application. For example, the `SENDER_ID` used with the notification system, the public server address, the port and protocol used to communicate with the server, etc.

The class `SingleInstance` groups objects with exactly one instance by using the Singleton pattern. For example, the `DatabaseHelper` is available from `SingleInstance` as it is accessed in different parts of the mobile application.

Moreover, for practical reasons, generic static methods that are not related to a particular object were placed in a package called `util`. Amongst those classes, one is particularly interesting and is described below.

LogUtils

This class is a wrapper around the log system of *Android* to provide a "debug mode" within the application. `LogUtils` provides multiple log levels that are correctly prefixed with "`myconsumption_`" (similar to all Google applications). The different levels are: debug; verbose; info; warning; and error. The debug mode is set in the `Config` class with a boolean variable (when set to false, verbose and debug logs are not shown). We used this system in every new part of the application with the appropriate level. An illustration of this is given in Figure 33.

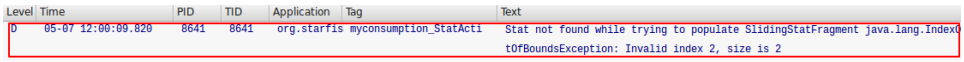


Figure 33: How the logs are displayed in the *Android Device Monitor*.

8.7 Notifications

The notification system has been fully detailed in a section about the server implementation. As far as the mobile application is concerned, one of the first things done by the `BaseActivity` class is to check if the *Google Play Services* are available on the device. If so, they are used to contact the *Google Cloud Messaging* server. A specific class was created for this purpose: `GCMRegister.java`. It is based on the *Android* Documentation: Implementing GCM Client on *Android*.

When the application is launched for the first time, the `GCMRegister` sends the `SENDER_ID` specific to *MyConsumption* to the GCM server for registration. After successful registration, the GCM server returns a `REGISTRATION_ID` to the mobile device. This ID is sent to our server and will be used to send a notification message to this specific device.

Finally, two classes are involved in the process of receiving a notification. The first is the `GCMBroadcastReceiver` which extends `WakefulBroadcastReceiver`. It is a special type of broadcast receiver that creates and manages a partial wake lock for the application. It has only one function (`onReceive()`) called when a notification is triggered and which loads the second class involved in this process: `GCMIntentService`. The latter is in charge of creating the notification on the device.

As this is implemented as a service defined in the `AndroidManifest` file of the application, notifications are shown even if *MyConsumption* is not running (see Figure 34).

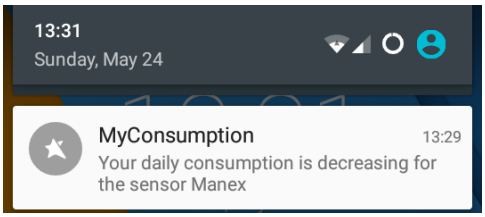


Figure 34: A notification is showing up.

Part IV

TESTS AND DEPLOYMENT

9

TESTS

This last part aims at discussing the different tests of the system as well as its deployment in a realistic practical environment. We start this discussion by assessing the design and the quality of the work done. After that, the different tests performed are described. Finally, the deployment of the system is detailed.

9.1 Design validation

The methodology followed to develop this software allowed us to be modular. Indeed, at regular intervals, we were able to review the work to tune and adjust its behavior accordingly. It helped us to keep our design valid along the way.

The features described in the design part of the document were all validated during one of the meetings with Antoine and Vincent. The graph smoothing, the statistics, the comparison, etc. were much discussed in a constructive spirit. They were only considered achieved after a positive feedback. The same applies to the user interfaces. Some details proposed in the description of the design requirements could not be implemented but workarounds[1] were found during the meetings. For both the features and the user interfaces, the results are conclusive.
As far as the use cases are concerned, their testing involves creating test cases based on the use cases. Thus, the quality of the work done was assessed by following their usage description on the real application in order to test the validity of their functional requirements.

Regarding the modularity of the system, we focused on several concerns. Firstly, the two build automation tools[2] used allow to easily deploy the code by other developers. Secondly, the design of the smart meter data retriever allows to easily add new types of sensors by using a strong Object-Oriented approach. Thirdly, thanks

[1] These workarounds were discussed in the implementation part of the work.
[2] Maven for the server and Gradle for the mobile application.

to the Java API it provides, the server is completely independent of the mobile application; an update does not affect the clients. Finally, in the *Android* application, the way activities extend the `BaseActivity` is also modular regarding the common features shared by these activities.

9.2 Software testing

Of course, during development, the first thing we do is to run our own programmer's "acceptance test": we code, compile, and run [5]. This a daily process: when running the software, we test it. It may just be clicking a button to see if it triggers the expected action. Sometimes, the test is more significant. For example, adding, viewing, editing and deleting a record. But those tests are done, over and over again.

This approach is not the most reliable. It is a boring "random" repetitive work, often not consistent and which does not give precise results and information at the end. However, with the rise of *Agile* and *Test Driven Developments* movements, programmers are encouraged to write automated tests [2]. The following sections give an introduction to more specific tests with their application in this work.

9.2.1 F.I.R.S.T.

The book "Clean Code A Handbook of Agile Software Craftsmanship" [2] advises us to follow five rules (from the above acronym) to write good tests.

- *Fast:* tests should be fast and run quickly (so that they can be run frequently and bugs are found early enough to be fixed easily);
- *Independent:* tests should not depend on each other and one test should not set up the conditions for the next test (which means that one can run the tests in any order);
- *Repeatable:* tests should be repeatable in any environment (in production, on one's laptop...);
- *Self-Validating:* tests should have a boolean output (either they pass or fail);
- *Timely:* tests need to be written in a timely fashion (before the production code that makes them pass).

9.2.2 Unit tests

A unit test examines the behavior of a distinct unit of work. Within a Java application, the "distinct unit of work" is often a single method [5]. It should have a very narrow and well defined scope. For those reasons, unit tests fit particularly well the F.I.R.S.T. principles described above. When such a test fails, it tells the programmer what piece of code needs to be fixed.

In this work, on the server side, we made simple unit tests with *JUnit* (an effective open source unit testing framework for Java). For this purpose, the `spring-boot-starter-test` dependency was used. Three actions of the server were tested:

- Access a non-protected RESTful resource;
- Access a protected RESTful resource;
- See if the database can be accessed.

Under the folder `src/main/test` of the server, three classes implement the tests described above: `ConfigControllerTest`, `UserControllerTest`, `UserRepositoryTest` respectively. They are all based on the same scheme: they used the annotation `@RunWith` (`SpringJUnit4ClassRunner.class`) which tells that the class will be used for a unit test. Moreover, each test is divided in two parts: first a setup which is made and then each unit test. *IntelliJ IDEA* (the IDE used in this work) can be configured to handle the running configuration for the tests. A result of a test is illustrated on Figure 35.

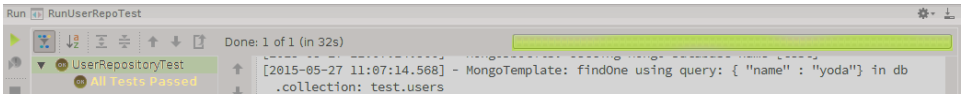


Figure 35: A successful unit test in *IntelliJ IDEA*.

Of course, one could extend those tests to ensure reliable operations of the server on a production machine. To go even further, a continuous integration software could be used to test the proper functioning of the system deployed. Tools like *TeamCity* or *Jenkins* could be appropriate. But the goal of the approach here was to prove that unit tests are easy to implement regarding the design of the server, which is the case.

9.2.3 Integration tests

The purpose of these tests is to verify the correct inter-operation of multiple subsystems. It includes different levels, from testing integration between two classes, to testing integration with the production environment. When it fails, it tells you that the pieces of the application are not working together as expected. Such tests are addressed in the next chapter of this document, where we describe the deployment of the system.

9.2.4 Functional tests

Functional tests involve testing the system as a black box. It checks a particular feature for correctness by comparing the results for a given input against the specification. We have not made such tests in an automated fashion (even if some debug logs provide information about the output of some methods). Nevertheless, while

debugging, functions and methods have been tested by checking the value they returned.

9.2.5 Acceptance tests

Acceptance tests ensure that the functionality meets their requirements. They were done along the way during the meetings with Antoine and Vincent to check if a feature or a use case was correctly implemented. It is similar to an integration test, but with a focus on the use case rather than on the components involved. When such a test fails, it means that the application is not doing what the customer expects it to do.

9.2.6 Regression tests

After integrating a new feature (or maybe fixing one), the programmer should run the unit tests again. This *regression testing* process ensures that further changes have not broken any units that were already tested. When these tests fail, it means that the application no longer behaves the way it used to.

10

DEPLOYMENT

As an integral part of the tests, the deployment of the system was a good opportunity to see if it really works in a realistic practical environment. It involves:

- Two *Flukso* smart meters: one in the office of the company and another at home[1];
- A virtual machine to deploy the server accessible with a public IP;
- Several *Android* devices running the API 16 and 21 of the OS.

The smart meters are easy to set up. Once connected to one's internet connection and to the electrical meter, they send their data to the *Flukso* server. Then, we can associate them to the mobile application so that our back end will fetch and process the data (see Figure 36). As far as the server deployment is concerned, the process is quite straightforward. It is fully detailed in the Appendix of this document.

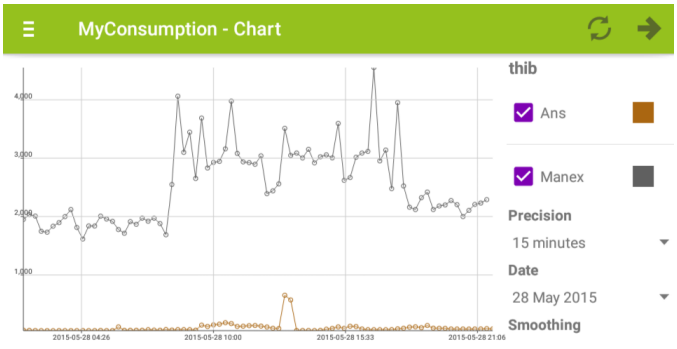


Figure 36: The two smart meters in the mobile application.

[1] Since the energy consumption of the company "Manex" is greater than my home's one, the data of sensor "Ans" are not easy to see...

The result gives us an operating server accessible anywhere. This configuration suits individuals who can easily use the system just by installing the app and a smart meter at home. For companies, the simple deployment of the server in a practical environment allows them to easily use this project. Several devices were used to test the app this real world environment. In the end, the results are conclusive.

Part V

CONCLUSION AND FUTURE WORK

11

CONCLUSION

In the introduction, we expressed the hope that the work in this master's thesis could simplify and highlight the key steps of an energy monitoring system's implementation by providing a good working example. In this final part, we will conclude by describing the progress made towards this goal regarding our design implementation. We will also suggest some future directions that could provide the next steps in the development of this system.

The objective of this master's thesis was to provide a mobile application for real-time energy consumption monitoring. This aim was motivated by the increase in energy efficiency measures where intelligent monitoring is an essential asset. Such an application could benefit both companies and individuals, as measuring energy consumption is the first step toward finding ways to decrease it.

The first part of the project was dedicated to the design of the system. From the data fetched by an existing retriever, we were able to provide relevant information displayed in an effective and user-friendly way. The system was naturally separated into two parts: a back end, a server built upon *Spring Boot*, and a front end, a mobile *Android* application. Use cases, user interfaces and different features were investigated.

The system's implementation was the core of the work. Particular attention was given to the issues involved in the synchronization, distribution and backup of energy consumption data. As a result of the design guidelines drafted in the first chapters, the users of *MyConsumption* can now check their power consumption in many different ways, including a line chart of their consumption, a set of statistics about their past day, week, month and year's consumption or even a comparison with an average customer profile. Moreover, responses to abnormal cases have been implemented. The communication between the application and the server was the crux of the matter, and special care to secure the exchanges was taken.

During the last part of the project, we focused on testing and validating the system for every feature and use cases investigated. Different types of tests have been made at different levels. As far as the back end is concerned, specific unit tests are implemented as a proof of concept of their technical feasibility.

On the other hand, the last phase of the project also revealed some limitations and drawbacks to the system. The consumption profiles given in the comparison feature are not as accurate as expected. We were faced with a lack of comparable information needed to achieve more precise standard profiles. On the server side, our initial idea to retrieve prices and electricity information could not be implemented given the resources found, and consequently a workaround had to be considered. Lastly, the security layer added to the system is a good first step, but it should be developed further.

Finally, by covering a wide range of features, this work has opened the door to other perspectives on future development. Many ideas, such as porting the application to another platform, adding intelligence to the system, and controlling electrical appliances are given in the next section to possibly complete the present working monitoring system.

At the close of the project, my personal opinion is that this work gives a good and extensible energy monitoring system in view of the time allocated to design and set it up. It was a rich and interesting experience; I have learned and discovered many tools and frameworks in a short period of time. In terms of challenges, it was not easy to continue the work of another student. However, the help of the company team guided me through this process. Finally, the opportunity to have a professional working experience in the area of mobile development has been particularly valuable. I was positively surprised by the pleasant working atmosphere as well as the continued interest the company showed in my work.

In conclusion, even if this system still has some unavoidable drawbacks, they do not affect its functionality in any way; the development meets the requirements as defined in the project description and within the scope of this thesis.

12

SUGGESTIONS FOR FUTURE WORK

While drafting this document, many ideas and suggestions for future prospects came to our minds. This final section of the report starts by explaining what could be done if the development process was continued. We then recall features that could not be implemented, but which were drafted during the early meetings. Finally, other ideas and features of larger scope are suggested.

12.1 Improvements to the current system

12.1.1 Logs

The log system used for both server and client has been well implemented. As far as the back end is concerned, the library `slf4j` is used in many parts of the code. In the front end, a wrapper around the *Android* log system is used to integrate the application. Those two systems are sound and complete. Nevertheless, some pieces of code that come from the previous version of the system do not use this log system. For overall consistency, it would be ideal to update them.

12.1.2 Event bus

At the end of the work, a good workaround to easily support internal communication in the mobile application was found. The Event Bus provides a better approach and facilitates the communication between every part of the application. As for the log system, it is not implemented in some old pieces of the code (a few `Async tasks`) that still use the old scheme with *Android* callbacks. This is not a drawback since it works properly, but for consistency they could also be updated.

12.1.3 Security

Security was discussed during the work. The basic authentication mechanism over SSL provide a good layer of security. Nevertheless, some weaknesses of this ap-

proach have been discussed. The password is stored without salt, HTTPS is not in place because of the missing certificate and some design choices may not be the best.

12.2 Features not implemented

Having sketched the features of the mobile application during the first meetings, we then prioritized them. Sadly, two features that were discussed to be implemented at first sight did not get the chance to be developed to completion.

12.2.1 Evolution of consumption

Based on the idea that a user's consumption could be impacted by a change in habit or behavior, we wanted the user to understand how their consumption evolves. The graph available on the main screen of the application already gives him/her a good idea of this. To take the analysis a few steps further, two things were suggested regarding the evolution of consumption:

- Let the user the option to enter a comment on the graph at a given time;
- Estimate the energy consumption and cost at the end of a period (e.g. a month) by extrapolation.

The first feature could be implemented by tweaking the chart library while the second could be achieved with forecasts of data based on a polynomial extrapolation, or using a moving average.

12.2.2 Manual consumption reading

Some people have a look at their electricity meter more than once a year. By allowing the user to enter their consumption manually, one could target those who do not have a smart meter. As it is not very likely that someone would use the mobile application this way, this feature was not made a priority and was left for a possible future work.

12.3 Features of larger scope

12.3.1 Add intelligence to the system

For now, *MyConsumption* is only a monitoring system. As pointed out in the introduction, many existing systems are integrated into larger environments. The tool developed in this work could be extended to be more specific. Below are some ideas.

Control electrical appliances

Most of the systems described in the introduction control a domestic heating system. Similarly, *MyConsumption* could be extended to interact with an existing thermostat. Other electrical appliances, such as a dishwasher, could also be added into the mix thanks to existing remote controlled sockets.

External data

Along the same line of thinking, *MyConsumption* could base its predictions on external data in order to be more relevant. For example, one could extend the analysis we already provide with the localization of the user. Moreover, the evolution of the consumption could take into account weather forecasts to estimate the energy consumption in the near future. Finally, some existing systems try to learn from the habits of the user. This could also be applied to *MyConsumption* (see the *Nest* thermostat by Google, for example).

12.3.2 Server database running on clusters

MongoDb, the database used in this work, is specifically designed to run well on clusters. With many clients, this possibility might be envisaged to balance the traffic loads. Patrick Herbeuval (who worked on this project last year) already gave this some thought, but this could be further developed.

12.3.3 Automated invoices

By assuming a system with many clients, a power supplier could provide an automated invoice processing software. This could replace the current archaic system where we have to note manually our annual consumption on a sheet of paper and hang it in a window of our house.

12.3.4 Smarter smart grid

Power suppliers use smart grids to optimize the efficiency of the production and distribution of electricity. With a monitoring system and access to the exact consumption data of every single customer, the smart grid could be more accurate, leading to large-scale savings. This idea could be expanded as well.

12.3.5 Target other platforms

MyConsumption is only available on the *Android* platform. One could see the value in porting the software to another platform (such as iOS for Apple mobile devices), and/or to a web interface for online reading purposes.

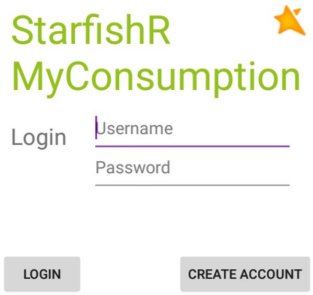
Part VI

APPENDICES

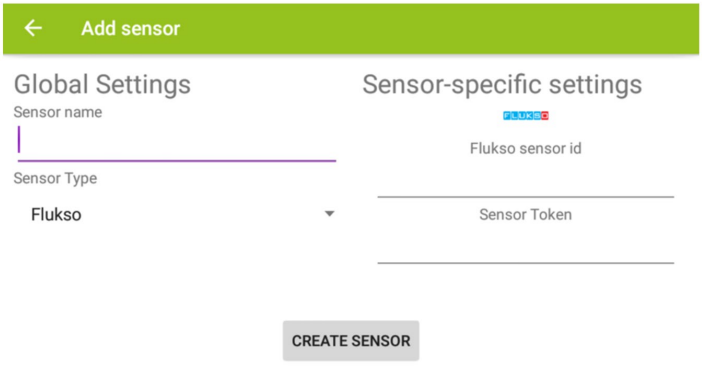
13

MOBILE APPLICATION'S UI

Application's logo.



Login screen.



Add sensor screen.

Figure 37: Logo, login and add sensor.

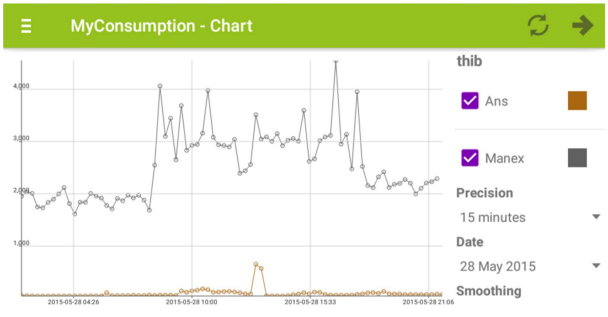
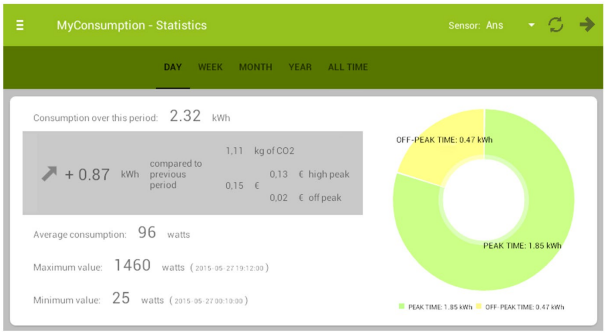
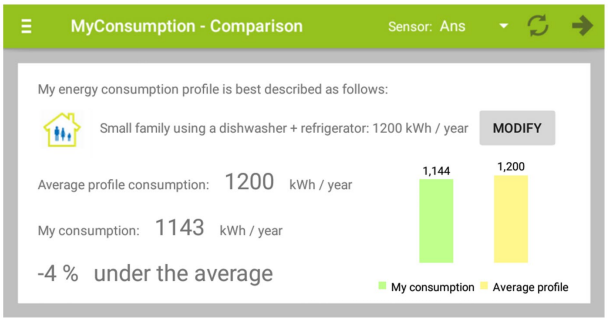


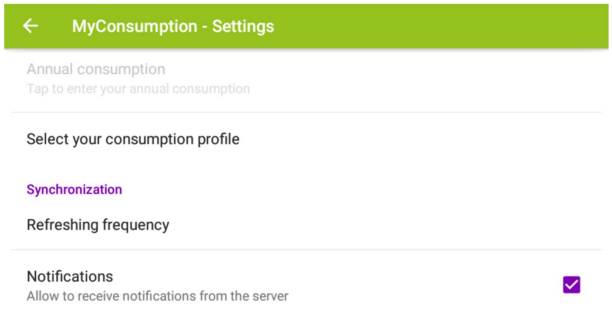
Chart screen.



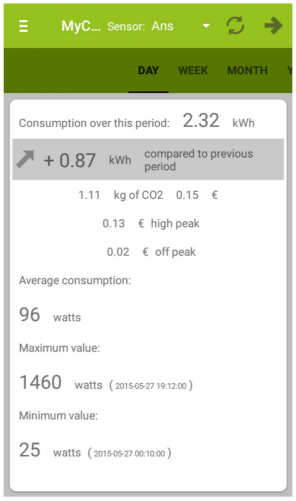
Statistics screen (landscape).



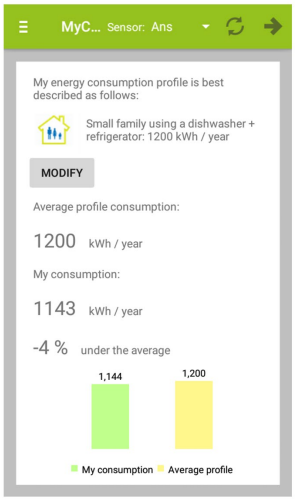
Comparison screen (landscape).



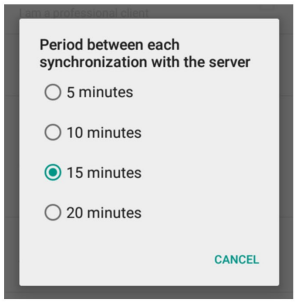
Settings screen.



Statistics screen (portrait).



Comparison screen (portrait).



Settings.

Figure 38: Chart, statistics, comparison and settings.

14

DEPLOYMENT OF THE SYSTEM

Setup

A specific virtual machine (running Debian 7.8 *Wheezy*) is dedicated to host our server. An SSH tunneling and specific ports forwarding were configured so that programmers and clients can access the machine from anywhere. Finally, *Tomcat 8* and *MongoDb 3* were installed on the machine.

As far as the configuration of the server is concerned, we had to add the following dependency to inform *Spring Boot* that *Tomcat* is running on the virtual machine:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
</dependency>
```

Deployment

The process to deploy the server is quite straightforward. First, we run the *Maven* command: `mvn package` which builds a self-contained `WAR` file.

Next, we can clean the virtual machine (if an older version of the back end was already deployed). The following commands are used:

- `ssh thibaud.ledent@212.166.22.110`
- `rm /home/thibaud.ledent/my-consumption-business-1.0.0-SNAPSHOT.war`
- `rm /var/lib/tomcat8/webapps/ROOT.war`
- `rm -r /var/lib/tomcat8/webapps/ROOT`

- `rm /var/log/tomcat8/*`

After that, we need to transfer the `WAR` from the programmer's machine to the virtual one:

- `/path/to/WAR/my-consumption-business-1.0.0-SNAPSHOT.war thibaud.ledent@212.166.22.110:/home/thibaud.ledent`

Finally, we can move the `WAR` to the dedicated folder of *Tomcat* and restart it:

- `ssh thibaud.ledent@212.166.22.110`
- `sudo su`
- `mv /home/thibaud.ledent/my-consumption-business-1.0.0-SNAPSHOT.war /var/lib/tomcat8/webapps/ROOT.war`
- `/etc/init.d/tomcat8 restart`

Two other commands may be useful to display the logs:

- `cat /var/log/tomcat8/catalina.out`
- `tail -f /var/log/tomcat8/catalina.201*-**-**.log`

15

REST API DOCUMENTATION

The documentation below explains how to access our RESTful API to create, read, update and delete consumption data stored on the server. This API is available at http://myconsumption.s23y.com.

Configurations

Ecological footprint

Path /configs/co2
`GET` request that returns the number of kg of CO_2 corresponding to one kWh.

Peak price

Path /configs/day
`GET` request that returns the price (in €) of one kWh during the day.

Off-peak price

Path /configs/night
`GET` request that returns the price (in €) of one kWh during the night.

Notifications

Path /notifs/{*name*}/id/{*registerId*}
`POST` request to set the register id of a user.
Path arguments:

- `name`: String
- `registerId`: String

Sensors

Get all sensors

Path /sensors
`GET` request that returns an array of JSON objects each one representing a `SensorDTO`.

Get a sensor

Path /sensors/{*sensorId*}
`GET` request that returns a JSON object representing a `SensorDTO`.
Path arguments:

- `sensorId`: String

Get the values of a given sensor

Path /sensors/{*sensorId*}/data
`GET` request that returns a list of pairs values.
Path arguments:

- `sensorId`: String

Request arguments:

- `start`: int (optional, default value = 0)
- `end`: int (optional, default value = 0)

Create a new sensor

Path /sensors/
`POST` request to create a new sensor.
Path arguments:

- `type`: String (default value = "")
- `settings`: String (default value = "")
- `name`: String (default value = "")
- `user`: String (default value = "")

Example:

```
{
        "type": "flukso",
        "settings": "{"token":"1d9bcb0d712c3c3266369dd6afdec14f",
            "fluksoId":"9337af875dba89f84a362d44070a2ecf"}",
        "name": "S23Y",
        "user": "thib"
}
```

Edit a sensor

Path /sensors/sensorId
`POST` request to edit a sensor.
Path arguments:

- `sensorId`: String

Request arguments:

- `name`: String
- `settings`: String (default value = "")

Delete a sensor

Path /sensors/{*sensorId*}
`DELETE` request to delete a sensor.
Path arguments:

- `sensorId`: String

Statistics

Get all statistics

Path /stats/sensor/{*sensorId*}
`GET` request that returns an array of JSON objects each one representing a `StatDTO`.
Path arguments:

- `sensorId`: String

Users

Get a user

Path /users/{*name*}
`GET` request that returns a JSON object representing a `UserDTO`.
Path arguments:

- `name`: String

Create a user

Path /users/{*name*}
`POST` request to create a new user.
Path arguments:

- `name`: String
- `password`: String (default value = "")

Associate a sensor to a user

Path /users/{*name*}/sensor/{*sensorId*}
`POST` request to associate a sensor to a user.
Path arguments:

- `name`: String
- `sensorId`: String

Delete a user

Path /users/{*name*}
`DELETE` request to delete a user.
Path arguments:

- `name`: String

Dissociate a sensor from a user

Path /users/{*name*}/sensor/{*sensorId*}
`DELETE` request to dissociate a sensor from a user.
Path arguments:

- `name`: String
- `sensorId`: String

16

MEETING REPORTS

5/08 : 1ère réunion
Ce qui a été fait :
Présentations et introduction.

Ce que je projette de faire :
- Créer un projet *Android* hello world
- Lire le TFE de Patrick Herbeuval
  - Se renseigner sur l'appli, le server, REST, *MongoDb*, *Spring*, *Maven*, *Tomcat*, …
- Définir des cas d'utilisation de l'appli *Android* (usage privé/entreprise)
- Se renseigner sur le monde énergétique (pas seulement ce qu'on consomme * prix de l'énergie). Il faut prendre en compte les taxes, … du secteur.

Ce qui a posé des difficultés :
/

Questions :
/

19/08 : 2ème réunion

Ce qui a été fait :

- Se renseigner sur le monde énergétique
  - Rien de bien intéressant sur internet => chercher une référence plus intéressante
  - Auprès d'un professionnel de l'énergie => bonne idée (on y reviendra)

- Cas d'utilisation de l'appli *Android* (usage privé/entreprise) :
  - Analyse de l'évolution de la consommation moyenne
    - suite à l'achat d'un nouvel appareil
    - suite à un changement quelconque d'habitude (rentré manuellement par l'utilisateur)
  - Comparer sa consommation à un public (ménage moyen, en couple, famille, ...)
  - Permettre d'encoder manuellement sa consommation
  - Voir la différence de consommation si on ajoute un nouvel appareil qui consommerait autant, …
  - Donner la consommation probable à la fin du mois (extrapoler).
  - Quand on doit vendre une maison, on dresse un constat d'énergie de la maison. Voir s'il y a moyen d'approcher le score ou le bulletin que les experts font après expertise énergétique.
  - Générer une alerte si un dépassement de consommation apparaît
  - Energies vertes ? Panneau solaire, éolien, certificats verts, maisons passives, …

- Recherche de références à propos d'*Android* et créer un projet *Android* Hello World

- Lecture du TFE de Patrick et première prise de renseignements sur REST, MongoDb, Spring, Maven, Tomcat, …

Ce que je projette de faire :

- Se renseigner sur le monde énergétique
- Point d'entrée. Par où commencer ?
  - Faire des stats sur le serveur
  - les synchroniser côté client
  - et les afficher dans l'application *Android*

Ce qui a posé des difficultés :

- Installation du SDK.
- Réussir à faire tourner correctement un virtual device

Questions & réponses :

- Gérer au minimum *Android 4*
- Domotique : garder en tête que l'app n'est qu'une interface
  - pas de stockage de données (sauf bd locale pour sync offline)
  - module statistique côté serveur (pas coté app qui récupère uniquement les infos)
  - Augmenter la compatibilité des capteurs : Flukso uniquement ? (j'ai reçu un appareil)
- Synchronisation offline doit être possible

29/09 : 3ème réunion
Ce que j'ai fait :

- Sources du serveur et de l'appli *Android* depuis le dépôt git de Manex
- 23/08 j'ai envoyé un mail à Patrick pour configuration mongodb et serveur
- 22/09 mail Patrick pour dépendances com.google gms
- Tomcat 8.0.9 (http port : 8081)
- Le serveur se lance avec succès
- L'application pose problème.

Ce que je projette de faire :

- Remettre au clair les cas d'utilisation pour qu'on puisse les envoyer chez Lampiris pour avoir un feedback sur ce que les clients demandent vraiment
- Les thermostats connectés *Nest* et *Honeywell*
    - http://www.lesoir.be/647326/article/au-frigo/geeko/2014-09-06/thermostats-connectes-nest-disponibles-en-belgique-en-septembre
    - http://geeko.lesoir.be/2014/09/22/nous-avons-teste-le-thermostat-connecte-dhoneywell/
    - Idées pour case study
    - S'inspirer de leur appli ?
- Continuer à avancer dans le but de faire des statistiques sur le serveur, les synchroniser côté client et les afficher

Ce qui a posé des difficultés :

- Configuration de mongodb et du serveur
- Dépendances com.google gms
- Http 404 du server

Questions :

/

13/10 : 4ème réunion
Ce qui a été fait :

- Fin de la configuration du serveur et de l'application avec Patrick Herbeuval. J'arrive à tout faire tourner (serveur & application) et à accéder aux données du Flukso de Manex
- J'ai continué ma prise de renseignements sur le développement sous *Android*, sur *Spring* et sur *Tomcat*
    - Livres
        - Introduction to Android Application Development
        - Professional Java for Web Applications
    - Internet
        - https://developer.android.com/training/basics
- En cours :
    - Faire un bouton dans la main activity pour lancer une nouvelle activity qui va récupérer les valeurs du Flusko sur une période donnée et en faire la moyenne.

Ce que je projette de faire :

- Continuer à avancer dans le but de faire des statistiques sur le serveur, les synchroniser côté client et les afficher

Ce qui a posé des difficultés :

- S'y retrouver dans tout le code

Questions :

- Vaut-il mieux créer une interface en Java ou en XML ? On dirait que Patrick mixe un peu les deux. J'ai lu que pour des raisons de maintenance, c'était mieux de faire un maximum en XML

27/10 : 5ème réunion
Ce qui a été fait :

- Correction de la configuration du serveur et de l'application. J'arrive à tout faire tourner (serveur & application) et à accéder aux données du Flukso de Manex
- J'ai installé un capteur Flukso chez moi. Je n'ai pas encore affiché les données sur l'application mais ça me permettra de faire des tests à la maison
- J'avance dans "*Android Programming The Big Nerd Ranch*", je construis l'application d'exemples en parallèle. C'est très intéressant. Je me suis créé un dépôt personnel pour les exemples que je fais.
- J'ai tenté d'ajouter la possibilité dans l'appli *MyConsumption* d'afficher les stats. Pour ça, dans layouts -> fragment_graph_choice, j'ai ajouté un bouton (il faudra faire autrement). Dans sensorviews->GraphChoiceFragment, j'ai essayé de faire en sorte que le click sur le bouton modifie la vue (il faudrait ajouter un fragment et remplacer le ChartFragment par le nouveau).

Ce que je projette de faire :

- Continuer à avancer dans le but de faire des statistiques (récupérer les valeurs du Flusko sur une période donnée et en faire la moyenne) sur le serveur, les synchroniser côté client et les afficher

Ce qui a posé des difficultés :

- Avancement assez lent mais je veux faire les choses dans l'ordre : d'abord apprendre *Android* avant de faire des modifications dans l'application de Patrick.

Questions :

- *Sensor is still dead* (depuis le 29 septembre 2014) => données entre le 13 et le 15/10. J'ai lancé le "retriever" aujourd'hui et la semaine passée mais ça n'a pas l'air de fonctionner.
- J'avance très lentement, y a-t-il autre chose que je puisse faire pour me voir avancer ? :
    - Use cases
        - Contacts avec l'équipe *Nest* de Lampiris ?
        - Les décrire plus en profondeur ?
        - ...
    - Utiliser les use cases pour faire des propositions d'UI design (même sur une feuille de papier), s'inspirer des applications *Nest*, *Honeywell*, ... ?
    - Autres recherches ?
    - ...

10/11 : 6ème réunion
Ce qui a été fait :

- J'ai lu plusieurs chapitres de "*Android Programming The Big Nerd Ranch*". J'ai construit l'application exemple en parallèle.
- J'ai tenté d'ajouter la possibilité dans l'appli *MyConsumption* d'afficher les stats.
    - Pour ça, dans layouts -> fragment_graph_choice, j'ai ajouté un bouton (il faudra faire autrement). Dans sensorviews->GraphChoiceFragment, j'ai essayé de faire en sorte que le clic sur le bouton modifie le fragment affiché dans le panneau de gauche
    - Il faut l'ajouter dynamiquement. Pour ça, je dois modifier le code de Patrick car ce fragment-là était *hardcodé* en xml.
    - Cette dernière modification fait planter l'application. Je cherche l'origine du problème.

Ce que je projette de faire :

- Continuer à avancer dans le but de faire des statistiques (récupérer les valeurs du Flusko sur une période donnée et en faire la moyenne) sur le serveur, les synchroniser côté client et les afficher

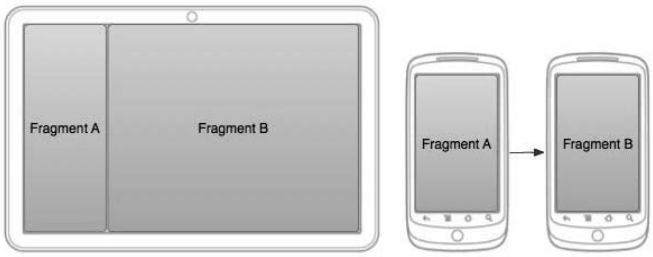
Ce qui a posé des difficultés :

- Beaucoup de tentatives infructueuses dans l'appli *Android*

Question(s) :

- (*Sensor is still dead* (depuis le 29 septembre 2014) => données entre le 13 et le 15/10. J'ai lancé le "retriever" aujourd'hui et la semaine passée mais ça n'a pas l'air de fonctionner).
    - Pas testé depuis

24/11 : 7ème réunion
Ce qui a été fait :

- J'ai tenté d'ajouter la possibilité dans l'appli *MyConsumption* d'afficher les stats. En modifiant l'appli de Patrick, je n'arrive pas à avoir quelque chose qui ne plante pas.
- J'ai recommencé l'appli dans un projet parallèle : myconsumption-android-2 (dépôt privé)
    - LoginActivity -> base de données -> user
    - startMainActivity()
    - Le but est de modifier correctement le code de la MainActivity, sans tout faire planter
    - J'ai créé deux fragments : StartFragment et ChartFragment que je voudrais pouvoir afficher à partir d'une liste déroulante :
    - 

    - Je suis le tutoriel : https://developer.android.com/training/basics/fragments/fragment-ui.html

Ce que je projette de faire :

- Reconstruire l'appli de Patrick pas à pas
- Code de Andaman : ne m'aide pas beaucoup

Ce qui a posé des difficultés :

- Insérer les fragments dans l'appli existante. Assez complexe.

Question(s) :

- Dois-je faire une branche sur le dépôt git de Manex plutôt qu'un dépôt privé ?

Réunion du 02/02

Ce qui a été fait :

Use cases : (usage privé/entreprise (b2b and b2c))

- Fournir une analyse de la consommation
  - Via des statistiques comme : la moyenne, la variance, la valeur min, la valeur max, la moyenne nuit, la moyenne jour, ...
  - Comparer ma consommation à un public (ménage moyen, en couple, famille, ...)
- Analyse de l'évolution de la consommation :
  - Suite à l'achat d'un nouvel appareil
  - Suite à un changement quelconque d'habitude (rentré manuellement par l'utilisateur)
  - Consommation probable à la fin du mois (extrapoler).
  - Économies probables par rapport au mois précédent (en extrapolant par rapport à la consommation de ce mois-ci)
- Permettre d'encoder manuellement sa consommation
- Voir la différence de consommation si on augmente/diminue la température de X degrés (extrapoler)
- Alerte si consommation anormale
- Récupérer et distribuer les prix de l'énergie
- Synchronisation offline
- Augmenter la compatibilité des capteurs : Flukso uniquement ?

Autre

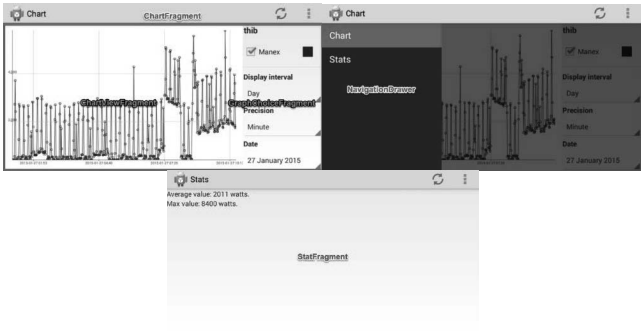
- Quand on doit vendre une maison, constat d'énergie de la maison. Voir s'il y a moyen d'approcher le score ou le bulletin que les experts font après expertise énergétique.

Consommation moyenne

De la liste de uses cases, on avait choisi de commencer par : "calculer la consommation moyenne côté serveur et l'afficher côté application".
Pour afficher la moyenne, j'ai besoin d'un nouvel espace dans l'application.
Problème : comment switcher entre la vue existante qui montre le graphique et une page dédiée aux statistiques ?
Solution : navigation drawer "*The navigation drawer is a panel that transitions in from the left edge of the screen and displays the app's main navigation options*" (https://developer.android.com).

Navigation Drawer

Utilisé dans toutes les dernières versions des applis Google.
Pour l'implémenter, changement de l'architecture de l'appli pour pouvoir ajouter facilement plusieurs fragments.
Chartfragment et Graphchoicefragment hardcodé en XML -> pas très flexible pour passer d'une "vue" à l'autre.



Récupérer dans l'app la moyenne et le max calculés côté serveur

1. On swipe le navigation drawer pour choisir d'aller vers le statfragment
2. Avant de s'afficher, celui-ci lance statvaluesupdater...
3. ...qui lance une asynctask ...
    1. Se connecte au serveur via l'api REST et le nouveau webservice "stat"
    2. http://172.20.1.209:8081/myconsumption/**stat**/**mean**/5429817ae4b025d69733f9f8
    3. http://172.20.1.209:8081/myconsumption/**stat**/**max**/5429817ae4b025d69733f9f8
    4. Récupère les valeurs et les ajoute dans la db locale sqllite de l'appli Android
4. ... Une fois terminé, un callback indique au statfragment que les valeurs sont disponibles dans la db locale de l'appli
5. On les affiche

Problèmes :

1. Ne vérifie pas si une valeur existe déjà dans la db locale Android -> ce n'est pas une "sync" mais juste un "get" => offline mode not supported
2. Les stats ne sont pas stockées dans une db côté serveur mais calculée en live -> mongodb ?

Ce que je projette de faire :

- Fixer un cahier des charges du TFE pour voir si ça correspond aux attentes de Pr Guy Leduc
- Consommation moyenne : offline mode & mongodb

Questions :

- What's next?
- Utilisation de la lib Android Support sans passer par *Maven*
- Pas moyen de faire fonctionner le retriever chez moi. Properties.xml (my-consumption business/resources) :
    - <!--<entry key="mongo_host">mxcsrv20.manex.biz</entry>-->
    - <entry key="mongo_host">127.0.0.1</entry>
    - <entry key="mongo_port">27017</entry>
    - <entry key="mongo_db">starfishr_myconsumption</entry>
- Accès au git de l'extérieur ?
- Gestion des exceptions, quelle est la "norme" en java ? Parce que l'app de Patrick peut planter complètement assez facilement
    - Il fait souvent des try catch avec catch vides
    - Il ne fait jamais de throw new exception machin chose
    - Il n'a pas créé d'exceptions propres à son programme
- Bugs : utiliser les "issues" de gitlab ?
- **Fixer les rendez-vous des prochaines réunions**

Réunion du 16/02 :

Ce que j'ai fait :

- Dépôt open source : Patrick et Guy Leduc sont ok
- Recherche données publiques et API pour les prix de l'énergie (*cf.* API publiques prix énergie et données publique)
- Améliorations GUI :
  - Ajout d'un bouton menu pour afficher le Navigation Drawer et intégrer l'actuel menu settings dans celui-ci
  - Icones de l'application et dans le Navigation Drawer
- Correction d'une série de bugs et changement dans l'architecture de l'appli.
  - J'ai séparé les fonctions présentes dans le Controller de Patrick. En m'inspirant du pattern Model View Controller, j'ai ajouté :
    - SingleInstance
    - UserDao
    - UserController
    - FragmentController
  - Classe abstraite MainFragment pour les fragments présents dans le Navigation Drawer
  - Correction de bugs liés aux changements d'orientation
- Implémentation de la moyenne mobile côté client
  - Fonctionne sauf que l'axe des y est modifié à chaque changement

Ce que je projette de faire :

- Prévoir d'améliorer (suite au rdv avec Guy Leduc) :
  - sécurité : serveur, authentification
  - déploiement du système en pratique (rester opérationnel) :
    - où va être la base de données, ... pour le client ?
    - point de vue sécurité ?
- Tests : y a-t-il une manière de procéder ?
- Passer sous *GitHub*
- Continuer l'implémentation des cas d'utilisation

Questions :

- C'est un peu bizarre de ne pouvoir supprimer un capteur que dans le GraphChoiceFragment
  - Faire une activity dans settings pour gérer les sensors
  - Mockups
- SingleInstance dans pakage DAO ?
- Git Manex :
  - Je suis toujours sur le git Manex, ne le supprimez pas :-)
  - Accès au git de l'extérieur :
    - actuellement
      - ssh: connect to host git.manex.biz port 22: Connection timed out fatal: Could not read from remote repository.
      - Please make sure you have the correct access rights and the repository exists.
    - Vpn ou accès à une machine en ssh de chez moi ? Car je ne sais pas faire grand-chose en dehors de réseau Manex (pas d'accès au server, à la base de données *MongoDB*, ...)

Réunion du 16/03 :

Ce que j'ai fait :

- J'ai modifié fragmentcontroller pour supporter correctement les getfragment...
- Transfert de l'app *Android* vers *Github*
  - Vu que c'est un dépôt public, important que le code soit propre et facile à construire
  - Or *Maven* était *broken* (google support, google service)
    - Transfert de l'app vers *Gradle*
      - Support de *Android* API 15 -> 21
- Textview qui affiche les stats
- Transfert du serveur sur *Github*
  - Renommer le package
    - La config *Spring* plante
  - Réécriture de l'api REST avec *Spring Boot*
    - Ça fonctionne
    - Je suis en train de travailler sur le Retriever pour mettre des données dans la base de données
    - Elles sont dans la bd mais j'ai un petit bug quand je les récupère
- Mail *Flukso* -> on ne m'a pas répondu

Ce que je projette de faire :

- Gérer l'ajout des statistiques dans la *Mongo Db*
- "Watcher" // Retriever
  - Toutes les 10 min fait une mise à jour des statistiques côté serveur pour les stocker dans *Mongo Db*
  - L'application interférerait avec ces données là (on aurait juste un objet à récupérer et pas une série de stats à générer)

Réunion du 31/03 :

Ce que j'ai fait :

- Serveur : passage sous *Spring boot*
  - Controllers pour le web service
  - Repository pour se connecter à la *MongoDB*
  - Passage vers une db locale (je peux travailler de chez moi)
  - Proche de ce que Patrick avait fait mais plus simple et moins de classes
- Appli :
  - Inspirée de l'app Google IO
    - Meilleure gestion des activités
      - Moins d'interdépendances
      - Meilleure gestion des Callbacks
      - Hiérarchie simplifiée (tout descend de la classe abstraite BaseActivity)
    - Navigation drawer
    - Plus stable
  - Architecture plus propre
  - Utilisation de *Gradle*
  - Support de *Android* API 14 -> 21
- Cas d'utilisation
  - ☑ Fournir une analyse de la consommation via des statistiques
    - Sur différentes périodes : all time, day, week, month, year
    - Fait : consommation sur la période, consommation instantanée moyenne, maximum, minimum
      - Récupérées depuis le serveur
      - Db locale *Android*
      - N'affiche que pour un seul sensor, quid du GUI ?
    - À faire ? moyenne nuit, moyenne jour, ...
  - ☑ Économies par rapport à la période précédent
  - En cours : comparer ma consommation à un public (ménage moyen, en couple, famille, ...)
    - Fait : profil utilisateur

Ce que je projette de faire

- Retriever -> Watcher pour le calcul des stats
- Prévoir d'améliorer la sécurité : serveur, authentification
- Continuer les cas d'utilisation :
  - Consommation probable à la fin du mois (extrapoler)
  - Comparer ma consommation à un public (ménage moyen, en couple, famille, ...)

Questions

- Pour les cas d'utilisation :
  - Récupérer les prix de l'énergie (voir les économies en € faites par rapport aux mois précédents). Où ?
  - Facture Manex pour comparer les valeurs obtenues (kWh) ?
- Comparer ma consommation à un public (ménage moyen, en couple, famille, ...)
  - Pendre en compte la localisation (dans le profil utilisateur)
  - Se baser sur données publiques ? 2 mails envoyés à Bart de *Flukso* sans réponse…
- Tests de l'app et du server (performance, ...) : y a-t-il une manière de procéder ? *JUnit*

14/04 : 12ème réunion

Ce que j'ai fait :

- Dépôt *Github*
  - Logo + Ajout d'un README + Bannière en respectant la charte graphique
    - https://github.com/S23Y/myconsumption-server
    - https://github.com/S23Y/myconsumption-android
- Server
  - Retriever -> Watcher
    - Récupère les data des sensors *Flukso* + calcule les statistiques
      - les stats sont calculées toutes les 10 min et ajoutées dans la db serveur (*Mongo DB*)
      - API du serveur modifiée (simplifications)
  - Ajout du calcul de la moyenne jour/nuit en prenant en compte le WE
  - Ajout d'un ConfigController: kg de CO2 par kWh, prix du kWh jour/nuit, ...
  - Sécurité
    - Oauth2 avec *Spring Security*
    - Access token working, can access to protected resource greeting but "Hello, null!"
  - À faire :
    - Chiffrer ce qui est transmis (https ?)
      - Problème : application.properties n'est pas pris en compte par *Spring*...
    - Authentification à la base de données *MongoDB*

    Tests
      - *JUnit* avec Spring Boot Tester
      - Test du GreetingController
- Appli
  - Adaptée par rapport aux petits changements de l'API
  - Statistiques :
    Mockup https://docs.google.com/drawings/d/1xySDd80US3PMY9yyyySyFiaoXCDXH5tE3zYLke5szzo/edit (économies par rapport à la période précédente, ...)
      - *PageSlider* https://github.com/jpardogo/PagerSlidingTabStrip
      - Diagramme jour nuit https://github.com/PhilJay/MPAndroidChart
      - Changements au niveau de la structure de la base de données locale pour stocker les stats, les configs, et le current user
      - Spinner pour choisir un sensor

Ce que je projette de faire

- Prévoir d'améliorer la sécurité : authentification en cours, https, accès à *MongoDB*
  - API : l'utilisateur doit pouvoir accéder uniquement à ses capteurs
- Statistiques :
  - Corriger spinner affiche bon sensor
  - Corriger les calculs diff et jour/nuit côté serveur
- Continuer les cas d'utilisation :
  - Alerte si consommation anormale (il faut gérer les services)
  - Comparer ma consommation à un public (ménage moyen, en couple, famille, ...)
    - Profil utilisateur : encoder toutes les possibilités comme dans CWAPE (http://www.compacwape.be/)

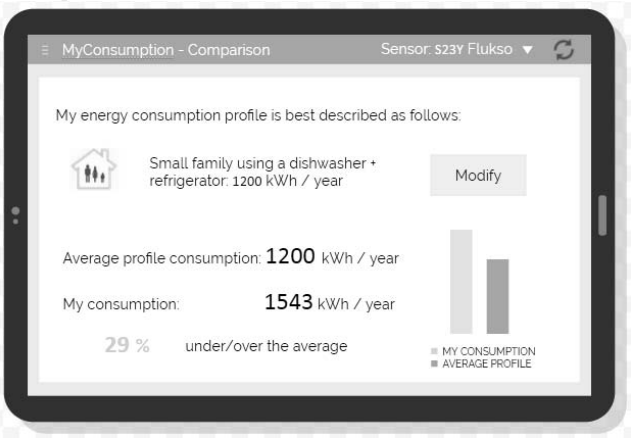
Questions

- J'ai commencé la rédaction TFE

21/04 : réunion

Ce que j'ai fait :

- Profil utilisateur
  - Encoder toutes les possibilités comme dans CWAPE
  - Db *Android* -> utiliser le système de préférences *d'Android*
    - Load au démarrage de l'appli
    - Je m'inspire de l'intégration des préférences de l'appli Google IO
- Comparer ma consommation à un public (ménage moyen, en couple, famille, ...)
  - Mockup



  -
  - implémentation

Ce que je projette de faire

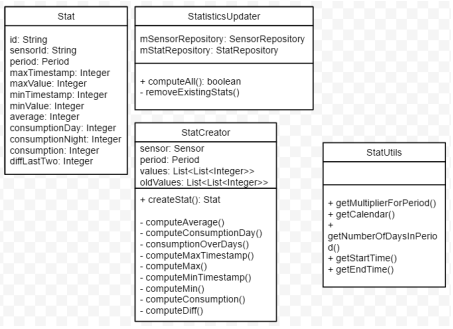
- Prévoir d'améliorer la sécurité : authentification en cours, https, accès à *MongoDB*
  - API : l'utilisateur doit pouvoir accéder uniquement à ses capteurs
- Statistiques :
  - Corriger spinner affiche bon sensor
  - Corriger les calculs diff et jour/nuit côté serveur
- Comparaison :
  - Spinner pour choisir le sensor
- Continuer les cas d'utilisation :
  - Alerte si consommation anormale (il faut gérer les services)

Questions
/

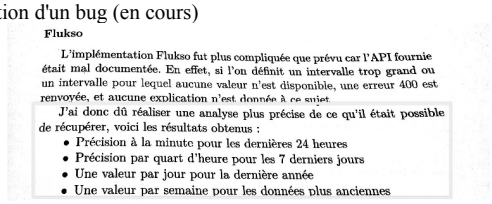
28/04 : 14ème réunion

Ce que j'ai fait :

- Préférences
  - Ne plantent plus
  - Sync delay (alarm)
  - No notification
- Alerte si consommation anormale
  - Alarm and notifier service running in background
  - Problème : pas d'accès à la base données locale *Android* depuis ce thread…
- Stats côté server



  - o
  - Correction d'un bug (en cours)



    - Unit type when retrieving server data: https://www.flukso.net/content/unit-type-when-retrieving-server-data
    - I don't get it: https://www.flukso.net/content/i-dont-get-it
    - A short summary:
      year scale => wh == wh / 7days
      month scale => wh == wh / day
      day scale => wh == wh / 15 minutes
      hour scale => wh == wh / 1 min

Ce que je projette de faire

- Prévoir d'améliorer la sécurité: authentification en cours, https, accès à *Mongo Db*
  - Api : l'utilisateur doit pouvoir accéder uniquement à ses capteurs
- Statistiques :
  - Corriger spinner doit afficher le bon sensor
  - Corriger les calculs diff et jour/nuit côté serveur
- Comparaison :
  - Spinner pour choisir le sensor
- Continuer les cas d'utilisation :
  - Alerte si consommation anormale (il faut gérer les services)

05/05 : 15ème réunion

Ce que j'ai fait :

- App : ajout de l'option "*add sensor*" au navigation drawer
- Simplification de la communication entre les activités, les fragments et les threads au niveau de l'app grâce à un *EventBus*
  - Ca a permis de corriger quelques bugs (rotation de l'écran, reload, ...)
  - Librairie : https://github.com/greenrobot/EventBus
    - EventBus is publish/subscribe event bus optimized for *Android*. *EventBus*... simplifies the communication between components decouples event senders and receivers performs well with Activities, Fragments, and background threads avoids complex and error-prone dependencies and life cycle issues makes your code simpler is fast is tiny (<50k jar) is proven in practice by apps with 100,000,000+ installs has advanced features like delivery threads, subscriber priorities, etc.
- VM avec IP publique et déploiement du serveur
  - correction de dependencies conflicts au niveau de *Maven*
  - period between timestamps is 60 seconds since Tue Apr 28 11:17:00 CEST 2015
  - à voir avec Antoine :
    - ce que je fais quand serveur not connected (period between two timestamps > 60 secs) ?
    - calculer uniquement les stats sur les valeurs correspondant à une période de 60 secondes ?
- (en cours) Correction d'un bug lié au stat côté serveur
  - Sens des valeurs retournées par *Flukso* : à ce timestamp précis, on consomme autant
    - POWER (Watts, or W) is the RATE of using or producing electrical energy (or how much is being used right now).
    - ENERGY (kilowatt-hours, or kWh) is the TOTAL amount of electricity used or produced over a period of time.

maxTimestamp	date		
maxValue	watt		
minTimestamp	date		
minValue	watt		
average	average des values en watt		
consumption	wh		
consumptionDay	wh		
consumptionNight	wh		
diffLastTwo	wh		

    -
- J'ai adapté le controller rest pour les stats (object mapping + API publique myconsumptionserver)
- Pas besoin d'adapter l'app grâce au DTO
- Rédaction du TFE : problem description, design, use cases (features discussed, compromises)

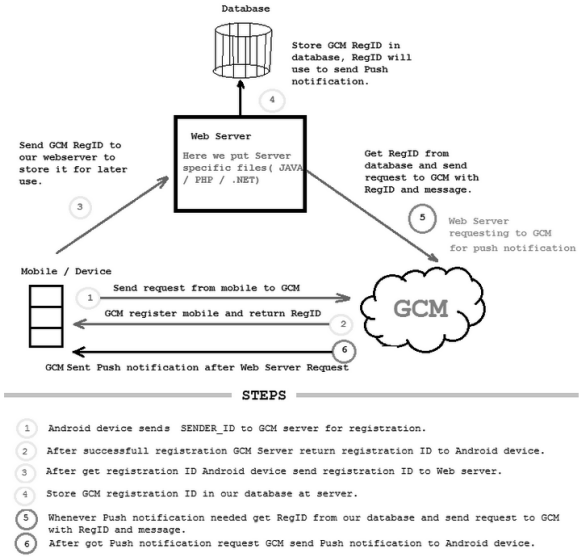
Ce que je projette de faire

- Améliorer la sécurité : authentification en cours, https, accès à *MongoDB*
  - API : l'utilisateur doit pouvoir accéder uniquement ses capteurs
- Statistiques :
  - Corriger spinner affiche bon sensor
  - Corriger les calculs diff et jour/nuit côté serveur
- Comparaison :
  - Spinner pour choisir le sensor
- Continuer les cas d'utilisation :
  - Alerte si consommation anormale (il faut gérer les services)

Réunion du 12/05

Ce que j'ai fait :

- Statistiques :
  - Eventbus, reload
  - Corriger spinner affiche bon sensor
  - Corriger les calculs diff et jour/nuit côté serveur
    - Correction conversion wh en kwh
    - Fix : no values found for this period
- Comparaison :
  - Eventbus, reload
  - Spinner pour choisir le sensor
- Push notification: alerte si consommation anormale



  - Serveur : algorithme pour générer les notifications au bon moment (une fois par jour)
    - Création d'une méthode POST pour stocker le registration ID côté *back end*
      - Il faut associer cet id à un user
      - Coté app, c'est enregistré dans les préférences
    - Api key properties externalised in application.properties
  - Appli :
    - Une notification par sensor maximum (pour ne pas être spammé)
    - Action: ouverture de l'app sur Chartactivity quand on clique sur la notif
- Couleur de l'app, icone & default profile background
- Rédaction du TFE : problem description, use cases, implementation back end front end (en cours)

Ce que je projette de faire

- Améliorer la sécurité : authentification en cours, https, accès à *Mongo Db*
  - API : l'utilisateur doit pouvoir accéder uniquement à ses capteurs
- Continuer les cas d'utilisation :
  - Permettre d'encoder manuellement sa consommation (voir si compliqué, ça peut être utile pour ceux qui n'ont pas *Flukso*)

Réunion du 26/05
Rapport :

- Ce qu'il reste à faire au niveau du rapport
  - Passer le rapport en revue avec Antoine (structure, ...)
  - How to build and run the project + sensor et user de test
  - Reasons why we can't use https server (pas de certificat)
  - When to send a notification
    - Threshold configurable dans paramètres
    - Pas top par rapport aux derniers jours..; ce serait mieux :
      - Par rapport à une fenêtre de temps plus grande (semaine)
      - Ou moyenne accumulante
  - Préférences
    - Consumption profile pour les entreprises ?
  - Décrire singleinstance = classe qui regroupe des singletons: make the constructor private (des singletons et pas de Singleinstance)
  - Tests and validation of the design
  - Structure de serveur à décrire
  - Annexes : server api documentation, ...
  - Todo

Code :

- Amélioration des layouts en portait sur smartphone
  - Possibilité de slider dans les fragments (scrollview)
    - `Chartchoicefragment`
    - `Statfragment` + quand il n'y a pas de données valides, tout cacher
    - Comparison
  - Smoothing with 2 sensors: bug fix
  - Préférences
    - Correction bug des valeurs par defaut pour le profil
  - Tests côté serveur
    - Spring-boot-starter-test (`springjunit`)
    - `Configcontrollertest` accessible sans auth
    - `Usercontrollertest` accessible avec auth
    - `Userrepositorytest`

- Sécurité :
  - Basic authentication
  - L'utilisateur accède uniquement à ses infos
  - Password transmis et stocké hashé mais pas salé

- Ce qu'il reste à faire au niveau du code
  - Clean code
  - Refactoring côté serveur
  - Analyse par *Intellij*
  - Deployer la dernière version du serveur en production

Part VII

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Joseph Annuzzi. *Introduction to Android application development — Android essentials*. Addison-Wesley, Upper Saddle River, NJ, 2014.

[2] Robert Martin. *Clean code — a handbook of agile software craftsmanship*. Prentice Hall, Upper Saddle River, NJ, 2009.

[3] Bill Phillips. *Android programming — the Big Nerd Ranch guide*. Big Nerd Ranch, Atlanta, GA, 2013.

[4] Pramod Sadalage. *NoSQL distilled — a brief guide to the emerging world of polyglot persistence*. Addison-Wesley, Upper Saddle River, NJ, 2013.

[5] Petar Tahchiev. *JUnit in action*. Manning, Greenwich, 2011.

[6] Nicholas Williams. *Professional java for web applications — featuring web-sockets, spring framework, JPA hibernate, and spring security*. Wiley, Indianapolis, Indiana, 2014.

[7] Commission de Régulation de l'Électricité et du Gaz. *Charte de bonnes pratiques pour les sites Internet de comparaison des prix de l'électricité et du gaz*. July 2013.

[8] Europen Smart Metering Industry Group. *Smart Metering for Europe — A key technology to achieve the 20-20-20 targets*. January 2009.

[9] Andrew McAfee and Erik Brynjolfsson. *Big Data: The Management Revolution. Harvard Business Review*, pages 60–6, October 2012.

[10] 01net. *Nest, Netatmo, Wiser... 6 solutions pour réduire la facture de chauffage cet hiver*. http://www.01net.com/editorial/629346/nest-netatmo-wiser-6-solutions-pour-reduire-la-facture-de-chauffage-cet-hiver/. (Visited on 05/14/2015).

[11] Bernard Boigelot. *Object-oriented software engineering*. Université de Liège. 2014.

[12] Jack Clayton. *1 kilowatt-hour — BlueSkyModel*. http://blueskymodel.org/kilowatt-hour. (Visited on 05/27/2015).

[13] CWAPE. *Simulation based on your profile data — CWAPE*. http://www.compacwape.be/proc/simulation?execution=e1s1#. (Visited on 05/24/2015).

[14] The Apache Software Foundation. *Apache License — Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0.html, January 2014. (Visited on 05/14/2015).

[15] Google. *Activity — Android Developers*. https://developer.android.com/reference/android/app/Activity.html. (Visited on 05/24/2015).

[16] Google. *Android Lollipop — Android Developers*. https://developer.android.com/about/versions/lollipop.html. (Visited on 05/19/2015).

[17] Google. *Android Studio Overview — Android Developers*. (Visited on 04/22/2015).

[18] Google. *Cloud Messaging — Android Developers*. https://developers.google.com/cloud-messaging/. (Visited on 05/31/2015).

[19] Google. *Fragment — Android Developers*. https://developer.android.com/reference/android/app/Fragment.html. (Visited on 05/24/2015).

[20] Google. *OAuth for Java Overview — Google Cloud Platform*. https://cloud.google.com/appengine/docs/java/oauth/. (Visited on 05/23/2015).

[21] Google. *Sliders — Google design guidelines*. http://www.google.com/design/spec/components/sliders.html. (Visited on 05/24/2015).

[22] Google. *Support Library — Android Developers*. https://developer.android.com/tools/support-library/index.html. (Visited on 05/23/2015).

[23] Google. *Supporting Multiple Screens — Android Developers*. https://developer.android.com/guide/practices/screens_support.html. (Visited on 04/25/2015).

[24] GreenRobot. *EventBus for Android — SlideShare*. http://www.slideshare.net/greenrobot/eventbus-for-android-15314813. (Visited on 04/25/2015).

[25] Patrick Herbeuval. *Réalisation d'un système de suivi en temps réel de la consommation énergétique sur mobile*. Master's thesis, Université de Liège, Liège, 2014.

[26] JetBrains. *IntelliJ IDEA's website*. https://www.jetbrains.com/idea/. (Visited on 05/14/2015).

[27] Vincent Keunen. *S23Y — Software Services For Energy: Overview — LinkedIn*. https://www.linkedin.com/company/s23y. (Visited on 04/06/2015).

[28] Levver. *Actuele kWh prijs elektriciteit in België*. http://www.levver.be/prijs-elektriciteit-per-kwh/. (Visited on 05/27/2015).

[29] OrmLite. *OrmLite — Lightweight Object Relational Mapping (ORM) Java Package*. http://ormlite.com/. (Visited on 05/23/2015).

[30] Vinay Sahni. *Best Practices for Designing a Pragmatic RESTful API*. http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api. (Visited on 05/27/2015).

[31] Pivotal Software. *Spring — Spring for Android*. http://projects.spring.io/spring-android/. (Visited on 05/23/2015).

[32] Pivotal Software. *Spring Boot's website — Spring*. http://projects.spring.io/spring-boot/. (Visited on 05/14/2015).

[33] SQLite. *SQLite — Home Page*. https://www.sqlite.org/. (Visited on 05/23/2015).

[34] Tutorialspoint. *Design Pattern Data Access Object Pattern — Tutorialspoint*. http://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm. (Visited on 06/01/2015).

[35] WhatIs. *What is Internet of Things (IoT)?* http://whatis.techtarget.com/definition/Internet-of-Things. (Visited on 04/06/2015).

[36] Wikipedia. *Android — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Android_(operating_system). (Visited on 05/19/2015).

[37] Wikipedia. *Mongo Db — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/MongoDB. (Visited on 05/19/2015).

[38] Wikipedia. *Moving average — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Moving_average. (Visited on 05/27/2015).

[39] Wikipedia. *Representational state transfer — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Representational_state_transfer. (Visited on 05/31/2015).

[40] Wikipedia. *Smart grid — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Smart_grid. (Visited on 05/14/2015).

[41] Wikipedia. *Smart meter — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Smart_meter. (Visited on 05/14/2015).

[42] Wikipedia. *Spring Framework — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Spring_Framework#Data_access_framework. (Visited on 05/27/2015).

[43] Wikipedia. *SQLite — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/SQLite. (Visited on 05/23/2015).