International Journal on Science and Technology (IJSAT)
E-ISSN: 2229-7677 ● Website: www.ijsat.org ● Email: editor@ijsat.org

Empowering Energy Efficiency: A Real-Time Mobile Analytics Platform for Intelligent Consumption Monitoring

Oyeronke Ladapo

ronke.ladapo@gmail.com

Abstract.
The increasing demand for sustainable energy solutions calls for innovative tools that empower both consumers and providers to opti- mize energy use. This research presents the development of a novel mo- bile application for real-time energy consumption monitoring, designed to enhance user awareness and enable data-driven decision-making. By leveraging cutting-edge mobile and cloud technologies, the system inte- grates seamlessly with smart IoT devices to collect, process, and analyze energy consumption data. The application features an intuitive inter- face that visualizes usage patterns through dynamic charts, predictive statistics, and anomaly detection algorithms. It provides actionable in- sights for users to identify inefficiencies, adapt behaviors, and achieve long-term energy savings. Additionally, energy providers gain unprece- dented visibility into network usage, unlocking opportunities for optimiz- ing electricity production and distribution. Built upon a robust architec- ture combining a Spring Boot backend and an Android-based frontend, the system ensures reliable synchronization, secure data handling, and responsive user interactions. The findings demonstrate the potential of integrating real-time analytics with mobile technology to address criti- cal energy challenges while fostering sustainability and efficiency across diverse sectors.

1. Introduction

I 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 mat- ter 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. Further- more, 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* [1].

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 [2]. 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%" [3]. 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 con- sumption monitoring" for companies and individuals. Connected to a smart ob- ject, 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 [4]. 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 moni- toring 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 com- pany 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 I tackled the challenges associated with each feature. It

tries to address every issue I 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. Architecture

The Figure 1 illustrates how each part interacts with the others. From this diagram, I see that various actors are involved:

- The *Android* Application, which communicates with its local database (*SQLite*), with the server and with the *Google Cloud Messaging* server;
- The server, 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.

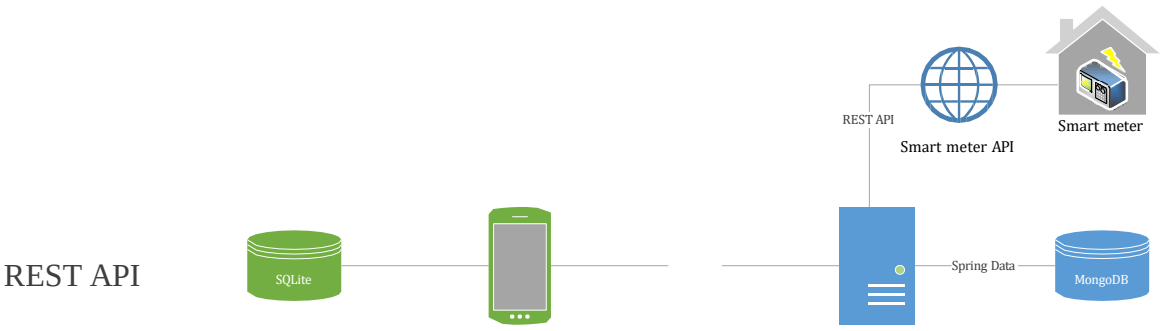

Fig. 1: An overview of the system.

3. 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, I thought it relevant to include every definition in the two following sections.

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 [5]. 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, I 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" [7]. *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* doc- ument 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 the server was dedicated to fetch data from their API and to store them in the database.

Google Cloud Messaging The *Google Cloud Messaging* (GCM) service for *Android* made it possible to send data from the server to specific users' *Android* - powered device [8]. 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 [9]. These docu- ments 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" [10]. 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.

Mobile application

Android *Android* is a mobile operating system based on the Linux kernel and currently developed by Google [11]. 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 [12]. 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* [13]. It is the IDE used to implement this work.

Support library Besides the SDK, developers have access to the *Android Sup- port Library*. This is a set of code libraries that provide backward-compatible versions of *Android* framework APIs [14]. 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 the goals with *MyConsumption* is to support a large set of devices, which is the reason why I 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. I 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 [15, 16]. *SQLite* is fully integrated with *Android* within the package android.database.sqlite which makes its adoption easy.
ORMLite is a tool used alongside *SQLite*. I chose this tool because it provides lightweight functionalities for persisting Java objects to SQL databases [17]. 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 [18].

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.

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 2, 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.

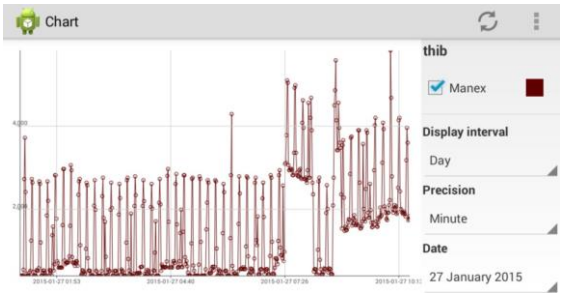

Fig. 2: The graph is not easy to read.

1.1 Provide an analysis of energy consumption based on statistics

I 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.

1.2 Savings between periods

This feature is related to the previous one. Based on the idea that I 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.

1.3 Compare one's consumption to a given consumer profile

Couples, average household, big families... All these different profiles have vary- ing 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.

1.4 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.

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

1.5 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.

1.6 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.

1.7 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.

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.

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 3. Notice the reload button in the upper right corner.

2.2 Statistics

The second screen I 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 4.

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

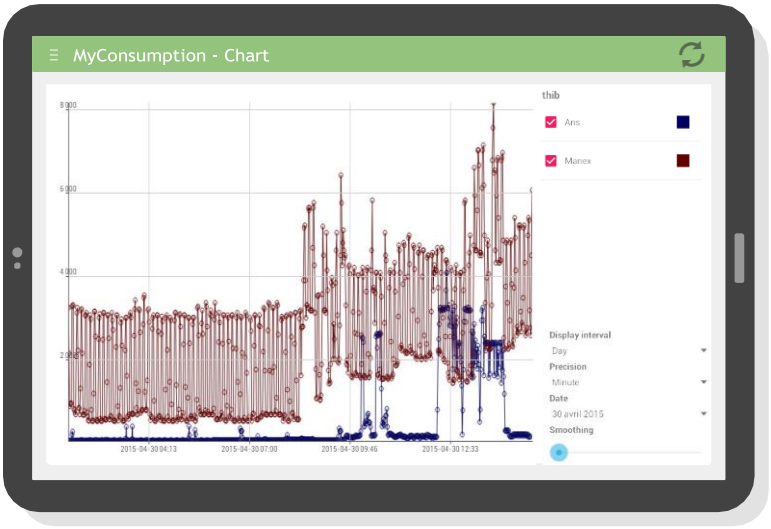

Fig. 3: The mockup of the main screen of the application with two sensors.

As far as the statistical items[3] are concerned, I 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)).

2.3 Profile comparison

As discussed above, a relevant feature I identified was for a user to be able to compare their consumption to a standard profile. At the early discussion stage of the work, I did not know how the standard profiles would be computed. This uncertainty is the reason why the mockup proposed in Figure 5 is quite simple. It is composed of:

[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 elec- tricity 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).

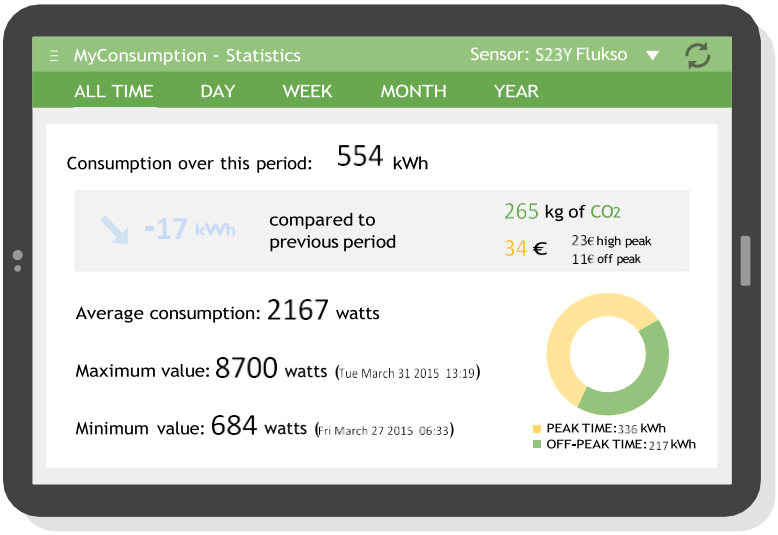

Fig. 4: The mockup of the statistics screen.

- 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.

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 6.

3 Defining use cases

As a use case represents a typical interaction between a user and the system [19], 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.

4 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

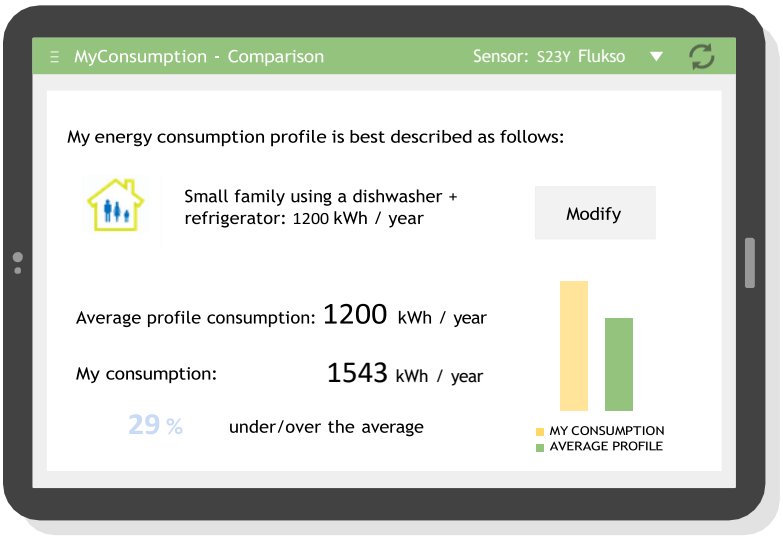

Fig. 5: The mockup of the comparison screen.

the *Android* client. It tries to address every feature and issue I faced with ap- propriate detail. However, the description below does not faithfully follow the implementation process in every aspect.

I wanted the mobile application to communicate with the 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. More- over, 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.

5 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 7):

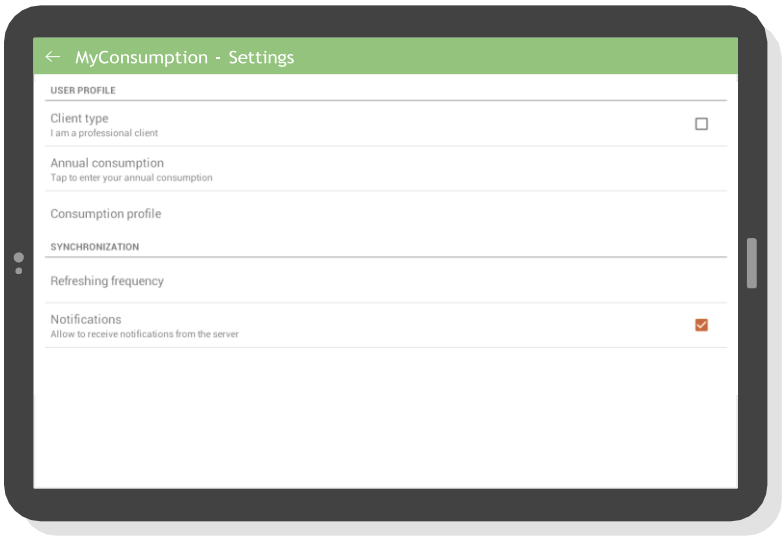

Fig. 6: The mockup of the settings screen.

- 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.

Fig. 7: Structure of the server.

5.1 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, I 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.

5.2 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.
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.

6 RESTful services

The Representational State Transfer (REST) is a software architecture style, consisting of guidelines and best practices for creating scalable web services [20]. 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 I wanted to achieve.

6.1 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.

6.2 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 [21]. The resources are:

- Users;
- Sensors;
- Statistics;
- Notifications;

- Configurations.

In the snippet of code Listing **??**, 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 refer- ences).

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 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.

8 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, I explained the choice of *MongoDb*. It should not be forgotten that this database is not structured around the tradi- tional table-based relational model. Instead, JSON-like documents with dynamic schemas are used. These documents have the advantage of making the integra- tion 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 [22]. I used *Spring Data MongoDb*, which focuses on storing data in *MongoDb*. It inherits functionality from the *Spring Data Commons Project*. With such a framework, I do not have to write any *MongoDb* queries because they are written for us.
I defined several entities that are stored in the database. They are described in the UML diagram in Figure **??**. For example, a User has an ID, a name, a password, a list of sensors and a register ID[4]. This entity is stored in *MongoDb* with the help of a repository.

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

8.1 Moving to Spring Boot

To explain how entities are stored, it is important to describe some work carried out when I 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, I would first have to check if it exists before accessing it. Moving to *Spring Boot* made things even simpler.

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 I 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.

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

9 Tests

This last part aims at discussing the different tests of the system as well as its deployment in a realistic practical environment. I 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.

10 Design validation

The methodology followed to develop this software allowed us to be modular. Indeed, at regular intervals, I were able to review the work to tune and adjust its behavior accordingly. It helped us to keep the 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[5] 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, I focused on several concerns. Firstly, the two build automation tools[6] 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 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 *An- droid* application, the way activities extend the BaseActivity is also modular regarding the common features shared by these activities.

11 Software testing

Of course, during development, the first thing I do is to run the own program- mer's "acceptance test": I code, compile, and run [23]. This a daily process: when running the software, I test it. It may just be clicking a button to see if it trig- gers 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 informa- tion at the end. However, with the rise of *Agile* and *Test Driven Developments* movements, programmers are encouraged to write automated tests [24]. The fol- lowing sections give an introduction to more specific tests with their application in this work.

11.1 F.I.R.S.T.

The book "Clean Code A Handbook of Agile Software Craftsmanship" [24] ad- vises 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);

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

- *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).

11.2 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 [23]. 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, I 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, User- RepositoryTest 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. 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.

11.3 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 I describe the deployment of the system.

11.4 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. I have not made such tests in an automated fashion (even if some debug logs provide information about the output of some methods). Neverthe- less, while debugging, functions and methods have been tested by checking the value they returned.

11.5 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.

11.6 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.

12 Deployment

As an integral part of the tests, the deployment of the system was a good oppor- tunity 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[7];
- 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 connec- tion and to the electrical meter, they send their data to the *Flukso* server. Then, I can associate them to the mobile application so that the back end will fetch and process the data. As far as the server deployment is concerned, the process is quite straightforward. It is fully detailed in the Appendix of this document.

The result gives us an operating server accessible anywhere. This configura- tion 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.

[7] 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...

13 Conclusion and future work

This work presented the design, development, and evaluation of a real-time mobile analytics platform—MyConsumption—that empowers both energy con- sumers and providers through intelligent consumption monitoring. By integrat- ing cutting-edge mobile and cloud technologies with IoT-based smart meters, the system offers a robust, user-friendly solution for real-time data collection, visualization, and analysis. The platform's architecture, built on a Spring Boot backend and an Android client, successfully addresses key challenges such as secure data synchronization, dynamic user interfaces, and actionable insights through statistical analysis. Ultimately, this research demonstrates that timely, data-driven feedback can foster energy-efficient behaviors and facilitate more informed decision-making, thereby contributing to broader sustainability goals.

Future Work

While the current implementation of MyConsumption provides a solid foun- dation, several avenues exist for further improvement:

Enhanced Sensor Integration: Expanding support to additional smart meter types and other IoT devices would broaden the platform's applicability and im- prove data diversity. Advanced Analytics: Incorporating machine learning and predictive analytics could refine consumption forecasts, improve anomaly detec- tion, and offer personalized energy-saving recommendations. Security Improve- ments: Upgrading authentication mechanisms (e.g., introducing multi-factor or claims-based authentication) and integrating more robust encryption methods would further secure sensitive user data.

User-Centric Features: Enhancing the user interface with customizable dash- boards, interactive analytics, and richer notification systems can improve overall user engagement and experience. Scalability and Cross-Platform Adaptation: In- vestigating the scalability of the platform in larger, diverse energy markets and exploring cross-platform compatibility could facilitate broader adoption.

References

1. WhatIs, "What is Internet of Things (IoT)? " http://whatis.techtarget.com/ definition/Internet-of-Things, (Visited on 04/06/2015).
2. McAfee and E. Brynjolfsson, "Big Data: The Management Revolution," Harvard Business Review, pp. 60–6, October 2012.
3. E. S. M. I. Group, "Smart Metering for Europe — A key technology to achieve the 20-20-20 targets," January 2009.
4. Wikipedia, "Smart grid — Wikipedia, The Free Encyclopedia," https://en. wikipedia.org/wiki/Smart grid, (Visited on 05/14/2015).
5. JetBrains, "IntelliJ IDEA's website," https://www.jetbrains.com/idea/, (Visited on 05/14/2015).
6. N. Williams, Professional java for web applications — featuring websockets, spring framework, JPA hibernate, and spring security. Indianapolis, Indiana: Wiley, 2014.
7. P. Software, "Spring Boot's website — Spring," http://projects.spring.io/ spring-boot/, (Visited on 05/14/2015).

8. Google, “Cloud Messaging — Android Developers,” https://developers.google. com/cloud-messaging/, (Visited on 05/31/2015).
9. Wikipedia, “Mongo Db — Wikipedia, The Free Encyclopedia,” https://en. wikipedia.org/wiki/MongoDB, (Visited on 05/19/2015).
10. P. Sadalage, NoSQL distilled — a brief guide to the emerging world of polyglot persistence. Upper Saddle River, NJ: Addison-Wesley, 2013.
11. Wikipedia, “Android — Wikipedia, The Free Encyclopedia,” https://en.wikipedia. org/wiki/Android (operating system), (Visited on 05/19/2015).
12. Google, “Android Lollipop — Android Developers,” https://developer.android. com/about/versions/lollipop.html, (Visited on 05/19/2015).
13. ——, “Android Studio Overview — Android Developers,” (Visited on 04/22/2015).
14. ——, “Support Library — Android Developers,” https://developer.android.com/ tools/support-library/index.html, (Visited on 05/23/2015).
15. Wikipedia, “SQLite — Wikipedia, The Free Encyclopedia,” https://en.wikipedia. org/wiki/SQLite, (Visited on 05/23/2015).
16. SQLite, “SQLite — Home Page,” https://www.sqlite.org/, (Visited on 05/23/2015).
17. OrmLite, “OrmLite — Lightweight Object Relational Mapping (ORM) Java Pack- age,” http://ormlite.com/, (Visited on 05/23/2015).
18. P. Software, “Spring — Spring for Android,” http://projects.spring.io/ spring-android/, (Visited on 05/23/2015).
19. B. Boigelot, “Object-oriented software engineering. Universit´e de Li`ege.” 2014.
20. Wikipedia, “Representational state transfer — Wikipedia, The Free Encyclope- dia,” https://en.wikipedia.org/wiki/Representational state transfer, (Visited on 05/31/2015).
21. V. 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).
22. Wikipedia, “Spring Framework — Wikipedia, - The Free Encyclopedia,” https://en.wikipedia.org/wiki/Spring Framework#Data access framework, (Visited on 05/27/2015).
23. P. Tahchiev, JUnit in action. Greenwich: Manning, 2011.
24. R. Martin, Clean code — a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2009.