Faculty of Sciences and Technology of the
University of Coimbra

Department of Informatics Engineering

Final Internship Report

Integration and Optimization of Energy Data Analysis Systems

Author:
José Ribeiro
jbaia@student.dei.uc.pt

Advisors:
Professor **Alberto Cardoso**
M.Sc. **Rafael Jegundo**

Submitted in Partial Fulfillment of the Requirements for
the Degree of Master of Science (Informatics Engineering)

September 2, 2014

Abstract

Internet of Things (IoT) is a concept that recently became mainstream with the launch of numerous consumer products, some of them dedicated to Home Energy Management. Due of its novelty and immaturity, IoT Platforms should be designed for change. This project aimed to provide an existing data analysis mechanism as a service to third-parties and enable the usage of custom energy management devices by an existing data monitoring application. To achieve this, a service-oriented system was created that allows third-parties to send energy consumption data and receive the classification's result as soon as possible, in isolation of the remaining system; an IoT communication protocol was used to allow energy management devices to send data in degraded network conditions and with low power consumption. This solution enables future projects of the company to reuse its components, such as the generic time-series storage platform developed in the process.

Keywords: Internet of Things, Data Analysis as a Service, Home Energy Management, Loosely Coupled Systems

Acknowledgements

The world's smallest Acknowledgements chapter: my coordinators, the company, my friends, my family and, most importantly, my parents. Thank you all for your help and support during this journey.

Contents

List of Figures

Abbreviations

MQTT MQ Telemetry Transport. 44, 45

MSA Microservices Architecture. 34

SOA Service-Oriented Architecture. 34, 35

SPOF single point of failure. 64

Chapter 1

Introduction

1.1 Context

unplugg is an online platform and application for energy monitoring, analysis and actuation, developed by NumberDiscover. It is based on the use of power meters and smart plugs, devices that allow the recording of consumption data and, in some cases, remote control.

Using these devices, the application is capable of presenting the user with his energy consumption data over time. By analysing that data, the application is also capable of inferring the user's energy consumption pattern for every plug; it is, therefore, capable of predicting whether a device will be in use on a given time of day. This allows the system to create rules of command for automatically controlling the consuming device; by shutting down a plug during the periods of expected inactivity of a given device, the application actively cuts on the standby expenses.

1.2 Goals

The company identifies two main business goals for this internship:

- It should be possible for third-parties to use the data analysis system to obtain rules of command in isolation and independently of the remaining unplugg application.

- It should be possible to receive and process high volumes of highly granular data sent by unplugg's energy management devices.

A revision of the current system should also be performed; during the system's revision, it is of the interest of the company that the main limitations of maintainability, security and performance of the current version of the system are analysed, proposing and implementing solutions that address those limitations and problems.

1.3 Scope

It is important to define the scope of the internship, in order to clarify what is and what is not expected to be studied and developed during its course.

- It is not expected to develop or improve any kind of data analysis algorithm or mechanism; it is only expected to modify the way the overall system interacts with the already existing data analysis components.
- It is only expected to use the already existing hardware solutions and modify the current system in order to be easier to support the future use of other kinds and vendors of devices. It is not expected to develop or improve any kind of hardware.
- It is only expected that the solution's integration process is planned in order to be implemented by the company after the internship's conclusion. It is not expected to integrate the solution with the current platform.
- There will be no involvement of the remaining team in the current system's revision nor in the implementation of the solutions that may arise from such revision.
- There will be no involvement of the remaining team in the implementation of the components that will be proposed to support the new business goals.

1.4 Motivation

The internship and its goals are part of the unplugg platform roadmap, a platform developed and maintained by the company since 2011.

Initially developed as a data visualisation platform, data analysis and control features were later added to allow active interference over the user's energy consumption costs.

1.5 Structure

This internship report is composed by the following chapters:

Introduction In chapter 1, a brief description of the internship's context, goals and motivation is presented.

State of the Art Chapter 2 describes the main modules that compose the current unplugg system, analyses Modern Web Application Architectural and Design Trends and Internet of Things Devices-to-Server Communication Protocols.

Methodology Chapter 3 presents the methodology applied to determine the internship's Requirements, Architecture, the used Software Development methodology, how Testing was performed and the Planning and its execution are analysed.

Product Vision In chapter 4, the Requirement Analysis is presented, as well as the Risks that were verified during the internship's course, and how they were mitigated.

Current System Analysis Chapter 5 lists particular details of the current system's implementation that will be improved by the proposed solution, while presenting some of the possible solutions and their rationale.

Architecture and Design In chapter 6, the proposed architectural style and principles are presented. A description of the contribution of every component of the proposed architecture to the overall system is described.

Implementation Chapter 7 describes how the proposed architecture was implemented taking into consideration the architecture's guidelines, presenting the chosen technologies and the reasoning behind those choices.

Verification and Validation Chapter 8 presents the list of the Verification and Validation tests that were performed on the system.

Conclusions In chapter 9, some remarks about how the proposed system achieves the business goals and complies with the requirements are presented. In addition, possible future studies are listed regarding some technological choices and how this future work may enable new business opportunities.

Chapter 2

State of the Art

2.1 unplugg

unplugg, being an online platform and application, has already a distributed architecture that needs to be studied before addressing the requirements of this internship. As such, a study of the current architecture that enables unplugg to perform energy monitoring, analysis and actuation was performed. It is worth noting that this description is only focused on the components that are relevant for this internship's scope.

An interpretation and critical analysis of its flaws is performed on Current System Analysis chapter.

2.1.1 Overview

An architectural overview of the current system is depicted in Barbosa's internship[1]. This chapter describes the composing subsystems and their components.

2.1.2 Architecture and Design

The unplugg platform is composed by three major subsystems:

unplugg Web Server Composed by the components needed for its energy monitoring features: the web application server, unplugg's Main

Database and the data fetching mechanisms.

On-Off Classification System Composed by the components responsible for unplugg's data analysis/classification features: the `On-Off Classifier` and the `On-Off Classification Results Database` (abbreviated to `Results Database`).

Actuation System Composed by the components needed for unplugg's actuation features: the Classification Starter and the Actuator.

The components composing each subsystem are now analysed in more detail.

2.1.2.1 unplugg Web Server

The unplugg Web Server is composed by three main components:

- `unplugg Web Application Server`
- `Real-time Updates Server`
- `Data Fetchers`
- `Main Database`

unplugg Web Application Server The unplugg Web Application Server is a Ruby on Rails application, with two major functions:

- Serve the unplugg Web Application to Web Clients.
- Receive the energy data sent by one of unplugg's energy consumption data providers, *The Energy Detective*.

The application is responsible for serving the user interface to Web Clients; this is the user's main interaction interface with the application. It allows the user to view his energy consumptions, statistics and control their devices' state when applicable.

It is also responsible for implementing an endpoint for one of the energy data providers, *The Energy Detective* (TED). When a TED's device has

unpublished energy data, TED's servers do an HTTP request on a previously specified web endpoint to allow real-time data push (in this case, unplugg Web Application Server). This endpoint implements the expected interface and receives and stores the energy data in unplugg's Main Database. An endpoint responsible for receiving data by energy data providers will be generically referred to as `Data Endpoint`.

Real-time Updates Server Real-time Updates Server's function is to update, in real-time, the Web Clients with the users' energy consumption data. It is a Faye server[1], a publish-subscribe messaging system. After a Web Client subscribes to real-time updates of a specific device, the server will send every incoming energy consumption datapoint to the client. The server is capable of doing so by hooking to a Consumption's `after_create` Data Model callback, triggered by the creation of a `Consumption`'s instance. A security flaw was detected in the subscription flow that is analysed in Real-time Updates Subscription Flaw (section 5.2.1); a rationale for removing the data model creation hook is exposed in section 6.3.6.1.

Data Fetchers The remaining energy data providers, *Cloogy* and *Current Cost*, serve their data through REST-based Application Public Interfaces. Because they don't support real-time push, the data acquisition is triggered by an hourly cronjob, whose function is to enqueue data acquisition tasks on a work queue. Each task is picked by an available worker that does the HTTP request (a `pull`, in contrast with TED's `push`) and stores the response on `unplugg's Main Database`. These workers are referred to as `Data Fetchers`.

unplugg's Main Database The `Main Database` is a MongoDB database, currently holding collections for the following resources (in their hierarchical order):

User

An unplugg user. A user may have multiple homes.

[1]Faye: `http://faye.jcoglan.com/`

Home
: A home belongs to a user. A home may have multiple meters.

Meter
: A monitoring system that measures power consumption. A meter may have multiple plugs.

Plug
: A smart plug that measures instantaneous power `consumption`.

Consumption
: A consumption represents the energy consumption on a given time instant.

Consumptions are generated at different time rates:

The Energy Detective A consumption every 1 minutes.

Current Cost A consumption every 5 minutes.

Cloogy A consumption every 15 minutes.

It is clear that the `Consumption` collection has a distinct access pattern from the remaining collections: while the `Consumption` collection has massive insertion operations from various components, the remaining collections are seldom modified, being mostly operated through reads instead.

2.1.2.2 On-Off Classification System

The `On-Off Classification System` is responsible for analysing the user's usage pattern regarding a certain device. It is composed by the actual classifier, `On-Off Classifier`, and a database where it outputs its result.

Their high-level functionality is described below.

On-Off Classifiers The classification task is CPU intensive and, as such, the task is currently offloaded to external servers using a service, called IronWorker[2]. This service provides scalable task queues for offloading the main server. For every classification request created by `Classification Starter` (described in Actuation System), an instance of `On-Off Classifier` is created on `IronWorker` that, once it is complete, outputs its result to the `Rules Database`.

Rules Database The `On-Off Classifiers` output rules of automation, named `Qontrols`; these rules describe the detected patterns of energy consumption. These are represented by the time intervals when the plugged devices were classified as turned on or off, by day of the week.

Besides being used as the output database of `On-Off Classifiers`, the `Rules Database` is used by the `Actuator` (described in Actuation System) as its input. This is what enables the `Actuation System` to control the devices' state according to the outputted `Qontrols`.

2.1.2.3 Actuation System

The Actuation System consists of two main components:

- `Classification Starter`
- `Actuator`

Classification Starter The classification process is triggered by a cronjob that runs Classification Starter once a week; its function is to enqueue the devices IDs that should be classified by On-Off Classification System.

Actuator The `Actuator` is the component responsible for applying the rules of automation to users' devices. Once an hour, a cronjob triggers the `Actuator`, that reads the latest rules of automation (`Qontrols`) from `Rules Database` and turns devices on or off, accordingly.

[2]IronWorker: `http://www.iron.io/worker`

2.2 Modern Web Applications Architectural and Design Trends

2.2.1 Service-oriented Architecture

Service-oriented Architecture (SOA) is an architectural and design pattern, where logical and business functions are partitioned into self-contained units of software designed to solve a single concern, named services.

Although no industry standards exist to define what composes an SOA, some principles are widely accepted as the core of what SOA represents [2]:

Standardized Service Contract
: Services within the same service inventory are in compliance with the same contract design standards.

Service Loose Coupling
: Services ensure the service contract is not bound to the service consumers nor to the service logic implementation.

Service Abstraction
: No information other than the necessary for the invocation of a service is included in a service contract, and service contracts are the only source of information about a service.

Service Reusability
: The service is designed so that its business logic may be reused.

Service Autonomy
: Services have control over their business logic, as well as over their runtime execution environment.

Service Statelessness
: Services are separated from their state data whenever possible, in order to reduce resource consumption and be able to handle more requests reliably.

Service Discoverability

Services provide metadata so that they can be discovered and interpreted.

Service Composability

Services are designed to be reused in solutions that may themselves be made up of composed services.

Together, the tenets of Service Design, particularly Loose Coupling, provide agility to the organisation, since loosely-coupled business processes are not constrained by the limitations of the underlying infrastructure.[3]

SOA is often confused with `Web Services` and `WS-*`, a set of specifications which define an interoperable platform supporting an SOA; SOA is an architectural and design pattern, `Web Services` and `WS-*` are one SOA implementation.[4][5]

Because of this confusion, a new trend has surfaced in recent years named Microservices Architecture, that aims to promote the Service-Orientation principles without being associated with `Web Services`/`WS-*`. Microservices Architecture[3] is usually associated with lightweight implementations and loosely coupled interfaces that do not rely on a particular technology or protocol stack.[6][7][8]

The advantages of using an SOA on this particular context are described in Architecture and Design chapter.

2.2.2 Twelve-Factor Application Methodology

The Twelve-Factor Application is a methodology for building modern, scalable, maintainable software-as-a-service applications, written by Heroku cofounder Adam Wiggins.[9] It comprises a set of 12 architectural, design and development principles that aim to promote best software development practices and may be grouped into three distinct categories:[10]

[3]Microservices Architecture is usually focused on promoting a single concern or business capability per service (Single Responsibility Principle), which usually leads to multiple small services, hence the name.

Development and configuration Promote strict separation of code and configuration, explicit declaration of dependencies and parity between environments.

Runtime Promote the design of horizontally scalable applications using multiple independent stateless processes with a share-nothing architecture, with fast startups and graceful shutdowns.

Management and visibility Promote continuous monitoring of the application through the use of log streams and one-off auxiliary processes.

These three categories are loosely correlated with three of the major phases of an application's life cycle: development, deployment and management.

2.3 Internet of Things D2S Communication Protocols

2.3.1 MQTT

`MQTT`, MQ Telemetry Transport, is a broker-based publish/subscribe messaging protocol, designed for use in constrained environments such as an Internet of Things context, where network is considered unreliable or expensive and a small code footprint is required for its use in embedded devices. On June 5, 2014, MQTT 3.1.1 entered a public review period for being advanced as a Candidate OASIS Standard, which ends September 4th, 2014 [11]. MQTT adoption has been growing and is already implemented in similar scenarios in multiple areas such as energy monitoring and healthcare.[12]

2.3.1.1 Architectural and Design Concerns

`MQTT` has some major architectural and design concerns, most importantly:[13]

Reduced bandwidth usage and, consequently, low power consumption. This makes it more suitable for embedded devices scenarios.

This is achieved by reducing the protocol exchanges and by having a compact binary payload with little overhead; this contrasts with text-based protocols such as HTTP, where headers add significant size to the payload. `MQTT` does not enforce any requirements over the payload format.

One-to-Many Message Distribution by using the Publish-Subscribe pattern.

This promotes the decoupling between the producers and consumers.

Multiple Quality of Service settings on a per-message basis.

This allows to use multiple Quality of Service policies over a single connection, allowing to change specific messages delivery guarantees and prioritize others according to their importance. The three available QoS policies are 0 *At most once*, 1 *At least once* and 2 *Only once*.

Support for clean and durable sessions to allow the (optional) persistence of QoS 1 and 2 messages until the client re-connects.

Hierarchical topic-based message filtering with single and multilevel wildcards (`+` for single-level subscriptions, `#` for multi-level subscriptions).

This provides the subscribers with greater flexibility and enables the reduction of the amount of undesired messages, therefore reducing traffic waste; it also allows multiplexing different message topics on a single connection. The wildcard mechanisms promote a semantically organized hierarchy to enable efficient filtering.

Simple implementation in order to reduce the code footprint.

This is achieved by simplifying its API, which as of version 3.1 consists of only 5 methods: connect, publish, subscribe, unsubscribe and disconnect.

2.3.1.2 Security

MQTT does not provide strong Security built-in to the protocol; version 3.1 has support for Identity and username/password Authentication, although it is performed in plaintext (since no encryption mechanism is provided by the protocol itself).[14] Authorization mechanisms are implementation dependent.[15]

The recommended way for providing data privacy is using SSL for encrypting the traffic. TPC/IP port 8883 is reserved with IANA for using MQTT over SSL (in addition to TCP/IP port 1883, for plain MQTT); multiple implementations of MQTT brokers and clients support MQTT over SSL.

2.3.1.3 Performance and Power efficiency (versus HTTPS)

Because MQTT was designed specifically to address fragile networks, its small payloads and protocol exchanges have direct impact in bandwidth consumption and, consequently, in its power usage. A benchmark performing a comparison between HTTPS and MQTT over SSL performance and power usage on an Android device showed MQTT to be both faster and use less energy when sending batches of 1024 1-byte messages, both on 3G networks (5% power saving, 11x faster), as well as on Wi-Fi (33% power saving, 4x faster).[16] The same benchmark also mentions MQTT over SSL as more reliable than HTTPS when receiving, particularly over 3G, where connection re-establishment time is higher.

2.4 Notes on other Internet of Things platforms

During the course of this internship, particularly during the 1st Semester, the intern aimed to study the State of the Art of other Internet of Things platforms in terms of architectural and technological choices. However, most of the relevant companies do not share information about their architecture or infrastructure. Because most of them have surfaced or gained traction in recent years, this is most likely because either they lack the resources to document it or because they rather keep them as trade secrets.

Hence, the analysis was focused on Modern Web Applications architectures and software design principles currently regarded as best practices. The author also analysed distributed systems from major online companies that tackle problems of similar nature, taking into account its architectural considerations and rationale[17][18][19].

Chapter 3

Methodology

Walking on water and developing software from a specification are easy if both are frozen.
— Edward V. Berard *in Life-Cycle Approaches*

3.1 Requirements

User stories were the chosen methodology for the formalisation of the functional requirements. User stories capture through informal language the expected interaction between a user and a system (or between two systems, representing stakeholders).

The traditional template of a User story is:

```
As a <role>, I want <goal/functionality/desire> so that <benefit>.
```

According to Mike Cohn, one of the contributors of the Scrum software development methodology, the `so that` clause may be considered optional[20], which renders:

```
As a <role>, I want <goal/functionality/desire>.
```

Non-functional requirements were formalised based on the high-level business goals of the internship.

This methodology was chosen for the formalisation of Functional Requirements because `User stories` are a simple and brief way of describing a small subset of the desired functionality, by breaking the project into small increments. This leads to a better understanding of how the problem should be approached and gives a method for measuring the degree to which the solution complies with the expected behaviour of the system, as perceived by different stakeholders.

3.2 Architecture

Before architectural decisions could be taken, every architectural constraint needed to be determined, so the newly created system could be compliant with those constraints. Architectural constraints comprise the requirements (both functional and non-functional) and external limitations.

Functional requirements are specified by User stories and, therefore, easy to verify if being addressed by the architecture. However, both non-functional requirements and external limitations drive the architecture development, since the qualities of the system (non-functional requirements) are a result of its components' interactions as a whole, while external limitations should be constantly monitored. In order to address external limitations such as company impositions, every major architectural decision was only executed after consulting and receiving approval by the company.

The rationale behind the solutions taken to address problems of similar nature made by major industry companies (such as Amazon, Facebook, Twitter, Netflix and Heroku) was studied, as well as approaches regarded as current best practices. Particularly, the *Service-Oriented Architectures* were taken as reference because of their rationale and the benefits they bring to the overall system. Heroku's *Twelve-Factor Application* Methodology served as a guideline for the service's design, because of the qualities it provides to its services and, at a higher level, to the system they compose. To address the *Security* requirement, the current system's major interactions were analysed, and current API Authentication and Authorization schemes and practices were examined, particularly for addressing possible security

flaws and assuring the users' data privacy; Twitter's use of an OAuth particular flow (Client Credentials Grant [21]) and REST APIs best practices inspired the design of a component's Authentication/Authorization mechanism. Evolutionary Architecture and Design trends were studied to address the *Maintainability* requirement, which led to the architecture and design of loosely-coupled, high-cohesion components. Asynchronous IO and the use of Event-oriented paradigms were considered as ways of achieving *Performance* on I/O bound applications.

Together, these architectural and design patterns were studied and used to address the requirements identified in Requirement Analysis.

3.3 Software Development

Both business goals of the internship have external dependencies: the `Enterprise Customer`'s expectations of the system and its functionality and the ongoing development of unplugg's energy management devices. Due to the unpredictable nature of the product and the external dependencies, agile methodologies were adopted in order to better adapt to unexpected changes on the internship's scope. In particular, the Lean Software Development (LSD) methodology was used. LSD is characterised by seven principles[22]:

- Eliminate waste
- Amplify learning
- Decide as late as possible
- Deliver as fast as possible
- Empower the team
- Build integrity in (The practice of refactoring the code to keep it simple and clear.)
- See the whole

These principles closely reflect the company's approach to software development and the mindset and personality of the intern; together, they lead to fast development cycles, empower the developer to explore new architectural and design patterns and technologies in a continuous learning process that benefits both the developer and the company. By eliminating waste, such as some bureaucracy imposed by waterfall-like methodologies, the development cycles are kept as lean as possible and allow small functionality increments. Deciding as late as possible while seeing the system as a whole enables an evolutionary architecture to be driven to address the evolving requirements.

The tasks were kept as small as possible in order to enable faster iteration cycles per component.

For tracking the project progress, *Trello*[1] was used to organise, structure and prioritise tasks and the work in progress. Its label and lists system allows to organise the tasks according to the affected component, phase of development and various other metrics. As the project progressed, new tasks were created by breaking down the user stories into small features to be implemented and incorporated in the next release.

For tracking the multiple versions of every component, a Distributed Concurrent Version System (DCVS) was used; the chosen system was Git. Git is the DCVS used by the company, and the intern was already familiar with it. A repository was created by component (service), since its functionality is independent of every other service; this promotes the future independent development of every service, since they are designed to be loosely-coupled. Every Git repository was kept synced with an online version kept on GitHub[2], the online Git service used by the company.

3.4 Testing

In order to assess if the resulting system is adequately tackes the business goals, Verification and Validation tests were performed.

[1]Trello: `http://www.trello.com/`
[2]GitHub: `http://www.github.com/`

3.4.1 Verification

Verifying the system is assessing if it functions as expected; that may be achieved through Unit Testing, Module Testing, Integration Testing and System Testing.

Since the system follows a *Service-Oriented Architecture*, most of the tests that were performed were Integration Tests and System Tests. When considering an *SOA*, Unit Testing covers a very strict scope of code with high maintenance costs; these tests were discarded following the *Eliminate waste* principle of `LSD`, since higher-level testing will cover the same functionality without the development overhead of Unit Testing. Module Tests were implemented for a single component, and are designed to test critical parts of its API[3].

3.4.2 Validation

Validating the system consists of assuring it meets the stakeholders' needs. In this project, that means it is necessary to assure the business goals are met from the perspective of the stakeholders.

Acceptance Tests were performed in order to assess the accomplishment of the business goals; they were designed to be as system wide as possible, in order to cover every stakeholder expectation.

Both Verification and Validation tests are described in Verification and Validation chapter (chapter 8).

3.5 Planning

3.5.1 1st Semester

During the 1st Semester, the study of the State of the Art covered the following topics:

- Distributed Systems Architectures

[3]This component is Aqora, described in section 6.3.1.

- Web Applications Architectures
- Internet of Things *de facto* standards and Devices to Server Communication Protocols
- Inter-node communication protocols and technologies for Distributed Systems and, specifically, Service-Oriented Architectures.
- High-volume and High-Performance Storage Technologies
- Agile Methodologies and Agile Architectures
- Similar Platforms

After the study of the State of the Art, a deep analysis of the architectural flaws of the current system took place, while emphasising the requirements. With the acquired knowledge, a future architecture for unplugg was designed in order to provide the desired technical support. The implementation of the prototype took place and preliminary tests were made.

3.5.2 2nd Semester

The planning of tasks for the Second Semester is illustrated by Appendix 2nd Semester Gantt Planning. An high-level description of the tasks to be fulfilled:

1. Finish the implementation of the unplugg-platform, integrating it with the classification system.
2. Implementation of the WebHook mechanism and the necessary Rules endpoint.
3. Implementation of the Publish-Subscribe System.
4. Integration of the MQTT Broker, including MQTT Broker Bridge.
5. Creation of a WebSocket Server and Integration with the Publish-Subscribe System.

6. Verification of the system through Module Tests and Integration Tests, developed throughout the implementation phase.

7. Validation of the system using simulated energy data to test all of the requirements.

8. Writing of the Final Report.

3.5.3 Plan execution and analysis

During the 2nd Semester, the plan proved to be reasonably accurate, except for task 1 and 2 (listed in Appendix 2nd Semester Gantt Planning):

- The time-series storage service of the `unplugg-platform` was planned to be finished by 15th February. However, after analysis of the possible future uses of the time-series storage service, it was found that adding an additional Entity to the Data Model could could dramatically improve the developer's experience when using the API; this entity would allow to model elaborate scenarios more easily, while keeping the Data Model simple enough for its use on this project's scope. After discussing with the company, it was accepted the potential risk of getting behind schedule would compensate the gain of flexibility. However, the impact of this reimplementation was underestimated and led to a three-week implementation cycle, in addition to the expected one-week of remaining methods implementation, in order to rewrite both the Data Model, the API endpoints, some of the business logic providing data integrity checks and module tests.

- The integration of the unplugg-platform with the classification system took longer than expected, due to the underestimation of adapting its code to the unplugg-plaform. The code of the classification system was rewritten (the reasoning behind this decision is described in section 7.2.6), and this took approximately an extra week of implementation.

This left approximately three weeks to write the report and execute performance tests, which was insufficient. Therefore, the delivery was adjourned to the next phase.

Chapter 4

Product Vision

> *The most difficult part of requirements gathering is not the act of recording what the user wants, it is the exploratory development activity of helping users figure out what they want.*
>
> — Steve McConnell

4.1 Requirement Analysis

4.1.1 Functional Requirements

The internship proposal describes the business goals that need to be addressed during the internship. Those goals correspond to high-level functional requirements for the system:

Data Analysis as a Service
: The system should be able to provide access to the data analysis subsystem to third-parties in isolation and independently of the remaining `unplugg` application.

unplugg's Energy Management Devices
: The system should be able to receive and process data sent by unplugg's energy management devices.

These two high-level functional requirements may be broken down into detailed functional requirements, formalised through user stories.

US1: Enterprise Customer's Data Storage
As an Enterprise Customer,
I **want** the system to be able to store my energy consumption data
so that it may be analysed by the classification system at a later time.

US2: Enterprise Customer's Data Classification
As an Enterprise Customer,
I **want** the system to be able to receive and process classification requests over my energy consumption data,
so that I can receive rules of automation.

US3: Enterprise Customer's Classification Availability
As an Enterprise Customer,
I **want** the system to notify or return the classification result as soon as available.

US4: End User's Real-time Visualisation
As an End User,
I **want** the system to be able to receive energy consumption data from unplugg's energy management devices,
so that I can receive that data in real-time on a browser.

US5: End User's Data Storage
As an End User,
I **want** the system to be able to store energy consumption data from unplugg's energy management devices,
so that I can retrieve it later.

4.1.2 Non-functional Requirements

Non-functional requirements (NFRs) describe the desired system or subsystem attributes.

The non-functional requirements mentioned in the proposal are as follows:

- Maintainability
- Security
- Performance

Given the lack of an objective quantification of Maintainability and Security, the system should be modified in a way that it becomes arguably equal or better under those qualities. In terms of Performance, the new components (and implicit system) should be architected, designed and implemented taking into account what is best for the overall performance of the system; it is not possible to directly compare the performance of the new system with the current one, since the internship aims to implement new features.

4.2 Risk Analysis

During the course of the internship, several risk scenarios for this project were identified. Of those identified potential risks, two eventually became reality. The reasons of why they happened and how they were mitigated are described.

Overly-optimistic Planning During the implementation of the time-series storage service of `unplugg-platform`, it became evident that this service could be adopted by other projects of the company. While it was already functional (both for the system and the other projects), some changes were made in order to ease the adoption of the other projects. The impact of those changes was underestimated, which delayed the remaining project. Due to the complexity of the system, the process of justifying every architectural, design and technological decision during

the writing of the report led to an incremental delay, which made the timely delivery of the report infeasible.

unplugg Energy Management Devices not ready for validation The initial internship's planning took into consideration that unplugg management devices would be ready for the validation phase of the energy management devices data interface (MQTT Broker) was ready. However, that was not verified. In order to mitigate this, an energy consumption data source simulator was implemented for validation.

Chapter 5

Current System Analysis

Even the best planning is not so omniscient as to get it right the first time.
— Fred Brooks, *in The Mythical Man-Month*

5.1 Architecture and Design

5.1.1 Coupling of Components

The current system's interactions between its different subsystems exhibit very high coupling, specifically `Common Coupling`[1]. This kind of coupling occurs between `unplugg Web Server` and `On-Off Classification System` and between `On-Off Classification System` and the `Actuation System`; the interactions and the data sources involved are listed below.

- The `On-Off Classification System` and, in particular, `On-Off Classifiers`, read the necessary consumption data directly from the `unplugg's Master Database`; both the `unplugg Web Application` and the `On-Off Classifiers` have access to the database (the web application for read/write access, the classifiers for read-only access).

[1] Common Coupling occurs when two modules share global data structures.

- The `Actuation System` and, in particular, the `Actuator`, reads the rules of automation directly from `Rules Database`; this database is used by the `On-Off Classifiers` to output the classification's result. Therefore, components of different subsystems (`On-Off Classification System` and `Actuation System`) have access to a shared data source (unplugg for read-only access, the Classification System for write-only access).

This interaction is clearly depicted in the unplugg's Current System diagram.

Sharing a database amongst multiple applications has some disadvantages; in particular:

- Data model changes affect every application. Changes to the data model structure of unplugg's Master Database will affect the Classification System's code, because it is *coupled* to the Master Database's data model. This either causes changes to be more expensive (since they affect the Classification System's codebase as well) or forces the changes to be done in isolation, to allow the database to keep the lowest common denominator for every application accessing the data.

  Likewise, changing the `Rules Database` data model affects `Actuation System`'s code, even though the change driver (the *owner* of the database) is the `Classification System`. The effects stated above are also applicable.

- Concurrency may be an issue. While a database is made to handle concurrency, different applications manipulate data in different ways. The order of the operations may affect the final result, depending on timing. This may, for instance, cause an application to see outdated data due to the operation isolation level (i.e., transaction isolation level).

There are additional consequences to the Classification System's use of the Rules Database for its output:

- While this choice does not pose serious security concerns since the only reader is the `Actuation System` (for scheduling the automation), it raises security issues when considering the `Classification System`'s use by Enterprise Customers, were it to be directly accessed by the users. Because of this, keeping this interaction is not a viable choice.

- Since the output is written to Rules Database, the `Actuation System` has to poll the database hourly to update its cronjob with the (potentially new) automation rules. Even if the external access to the Rules Database by Enterprise Customers was a viable choice, this would not be feasible with a growing number of users, since this would dramatically increase the number of requests, due to every Enterprise Customer's polling needs.

  On the Enterprise Customers use case, it would be an advantage to receive its result *as soon as it is available*, while also being able to request a classification when needed, instead of a fixed-schedule classification process.

The proposed solution addresses these issues and is described in the Architecture and Design chapter.

5.1.2 Data Model

The Enterprise Customer expects to use the unplugg Platform to classify a device's consumption time series data as a sequence of operation periods. Since the Classification System requires access to a considerable range of historical data, it would be inefficient to require the Enterprise Customer to send the entire time range for every classification request, since most of the data would overlap (the amount of overlapping data depends on the request frequency). As such, it is important that the unplugg Platform somehow persist the consumption data in order to allow its reuse by subsequent classification requests.

The unplugg's Data Model, because of its specific use case, presents a highly hierarchical structure. Although this data model fits the unplugg's

End User needs, it does not align with the Enterprise Customer's use case; there is no direct correspondence between the current unplugg's data model entities and the entities that describe the Enterprise Customer's consumption data. It should be noted that this takes into account the interest of an Enterprise Customer to provide the unplugg Platform the strictly necessary data to obtain the classification result (Principle of Least Privilege[2]).

A data model hierarchy capable of abstracting these differences while sufficient to represent the data needed by the Classification System is described in 6.3.1.2.

5.2 Security

5.2.1 Real-time Updates Subscription Flaw

During the Current System Analysis, a security flaw was found while analysing the real-time update mechanism; this subsection describes the flaw, the exploitation scenario, its consequences and a possible workaround.

5.2.1.1 Summary

It is possible for an attacker, either a registered or unregistered unplugg user, to subscribe to real-time consumption updates of any unplugg user's devices. The reason why this is possible is because the real-time updates' server currently does not employ any kind of Authentication/Authorization mechanism, accepting every subscription request as valid; in addition, the topic name format follows a predictable structure.

5.2.1.2 Exploitation Scenario

The steps below describe the basic exploitation flow of the flaw.

1. The attacker establishes a connection with the real-time updates server.

[2]Principle of Least Privilege: restrain the access of a module or actor to the information and resources strictly necessary for its legitimate purpose.

2. The attacker issues a subscription request to a specific device's consumption topic. The topic follows the known structure of `/presence-consumptions-<device_id>`.

3. The server will send the attacker every consumption datapoint received in real-time.

There are multiple ways a valid `<device_id>` could be obtained. Since unplugg's traffic is unencrypted (it uses HTTP, analysed on 5.2.2), it would be possible, for instance, to perform a network sniffing attack on a network during an active unplugg Web Application session (an unplugg user browsing http://unplu.gg on that network). This would allow the attacker to obtain the device ID through the visited URLs or through the consumption updates' traffic. The worst case scenario for the attacker would be to brute-force the device IDs namespace by performing bulk subscriptions; this kind of attack is also possible because the server is not employing any subscription rate limiting mechanism.

5.2.1.3 Consequences

The direct consequence is that the attacker obtains reading permissions over data he should not be authorized to read. While the unauthorized access to this data may seem harmless (as this data may be perceived as useless to users other than the legitimate user), it could, for instance, be used to predict whether the user is at his/her home, based on the decreased energy consumption. However, users data privacy is itself enough of a concern and, as such, this issue should be addressed.

5.2.1.4 Workaround

This flaw may be fixed by implementing an Authentication/Authorization mechanism. This way, the Real-time Updates Server would be able to serve or drop a connection according to the user requesting a subscription to real-time updates of a device; the subscribing client would first authenticate before the server, so Authorization could be established for upcoming

subscription requests on that connection.

This approach is described in the Architecture and Design chapter, applied to WebSocket Server, the component designed to replace the current Real-time Updates Server mechanism.

5.2.2 HTTP use

5.2.2.1 Analysis

Currently, the unplugg Web Application is served using HTTP. Since Edward Snowden's Global Surveillance Disclosures[3] the encrypted Internet traffic has seen global increase during the first half of 2014, according to the Sandvine Global Internet Phenomena reports of *1H 2013* [23] and *1H 2014* [24]; according to the same reports, Europe's traffic in particular has quadrupled (when compared with the same period of the previous year). This increase may be caused by the recent adoption of HTTPS as the default protocol by major Internet services such as Google[4] and Facebook[5]. This also may suggest a raising concern of the users over their data privacy.

Because of the plaintext nature of HTTP, it is susceptible to multiple attack types. While network sniffing is applicable to both HTTP and HTTPS, it is easier for an attacker to extract meaningful information from its plaintext form that allows him to perform powerful attacks such as Session Hijacking or Man-in-the-Middle attacks, for example.

5.2.2.2 HTTPS implementation difficulty and costs

To use HTTPS, one needs to generate an SSL Certificate and configure the service or a proxy server to use it.

[3] *Edward Snowden's Global Surveillance Disclosures* refers to public disclosure made by Edward Snowden in June 2013, through The Guardian and The Washington Post, of top secret documents regarding multiple National Security Agency's programs of global surveillance (namely PRISM) of foreign nationals and U.S. citizens, as well as the involvement of several other non-U.S. security and intelligence agencies.

[4] Google Search had SSL encryption for signed-in users since May 2012 [25], but it was only in September 2013 it became default even for signed-out searches.[26]

[5] Facebook had optional SSL encryption since January 2011 [27], but it was only in July 2013 it became default. [28]

Configuring a proxy server may be the easiest and most flexible approach, since it allows to proxy multiple services using a single certificate, while also keeping the private traffic of a network in plaintext. This is an advantage, since the heavy work of encrypting and decrypting traffic is performed by the proxy server, while allowing the services to reduce their resource consumption; it should be noted, however, this approach assumes the private network is secure, which is an assumption that may only be accepted after analysing the security policies of such network.

The cost of generating a publicly-trusted SSL Certificate (issued by a Certificate Authority trusted by the majority of the web browsers) depends on the issuing company.

After discussing with the company, the cost of generating a certificate was regarded as being a necessary operational cost, given the advantages of providing an SSL connection to the service.

5.2.2.3 Conclusions

Given the multiple benefits of using HTTPS (SSL) and taking into account the implementation difficulty and costs it presents, the author believes it is of the interest of the company and of its users to use encrypted communications on every public facing service, both existing and *to-be-implemented*; that means the only plaintext communications I believe are acceptable are communications performed inside the company's private network.

Chapter 6

Architecture and Design

> *Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure.*
> — Melvyn Conway, *the Conway's Law*

The Architecture and Design chapter describes the architectural style and the design principles followed by every proposed component. It also specifies every component and its functionality.

6.1 Architectural Style and Principles

The chosen architecture for the system is best classified as a Service-Oriented Architecture (SOA) and, in particular, as a Microservices Architecture (MSA). At the service level, the design principles of the Twelve-Factor Application Methodology were adopted.

The combination of these two styles brings desirable properties to the system.

Scalability Since every service is executed as one or more stateless processes, this improves the scalability of the service; vertical scalability comes from delegating the state data management to an external service, reducing the amount of consumed resources per interaction;

horizontal scalability becomes easier, since multiple instances of the service would share no state (a Shared nothing architecture).

It is important to note that, due to its SOA nature, horizontal scalability becomes possible on a per-service basis, meaning that it is possible to run more instances of the services that need more resources, while running fewer instances of the services that have a reduced resource consumption. This contrasts with the reality of scaling out a Monolithic Application, since that would require running multiple instances of the entire application, rather than the parts that require greater resource. [6]

Maintainability A Service-Oriented Architecture leads to multiple small code bases (one per service) instead of a single large code base. Since every code base maintains a reduced amount of logic, it is easier to correct the flaws in the existing functionality in the long term because its complexity is greatly reduced and the code is easier to navigate. Promoting loose coupling between services also increases their maintainability, since modifications to their code base have low or no impact to the remaining services. Both these aspects lead to greater maintainability of the overall system, since its maintainability is a result of the maintainability of its components. [29]

Extensibility is related to the Maintainability of the system. Promoting loose coupling between services and high cohesion per service allows its extension to be performed without impacting the existing system functionality, either by creating a new service (which benefits from loose coupling) or by extending the functionality of an already existing service (which benefits from high cohesion). [29]

6.2 Overview

A representation of the overall system is seen on Figure 6.1 (page 36).

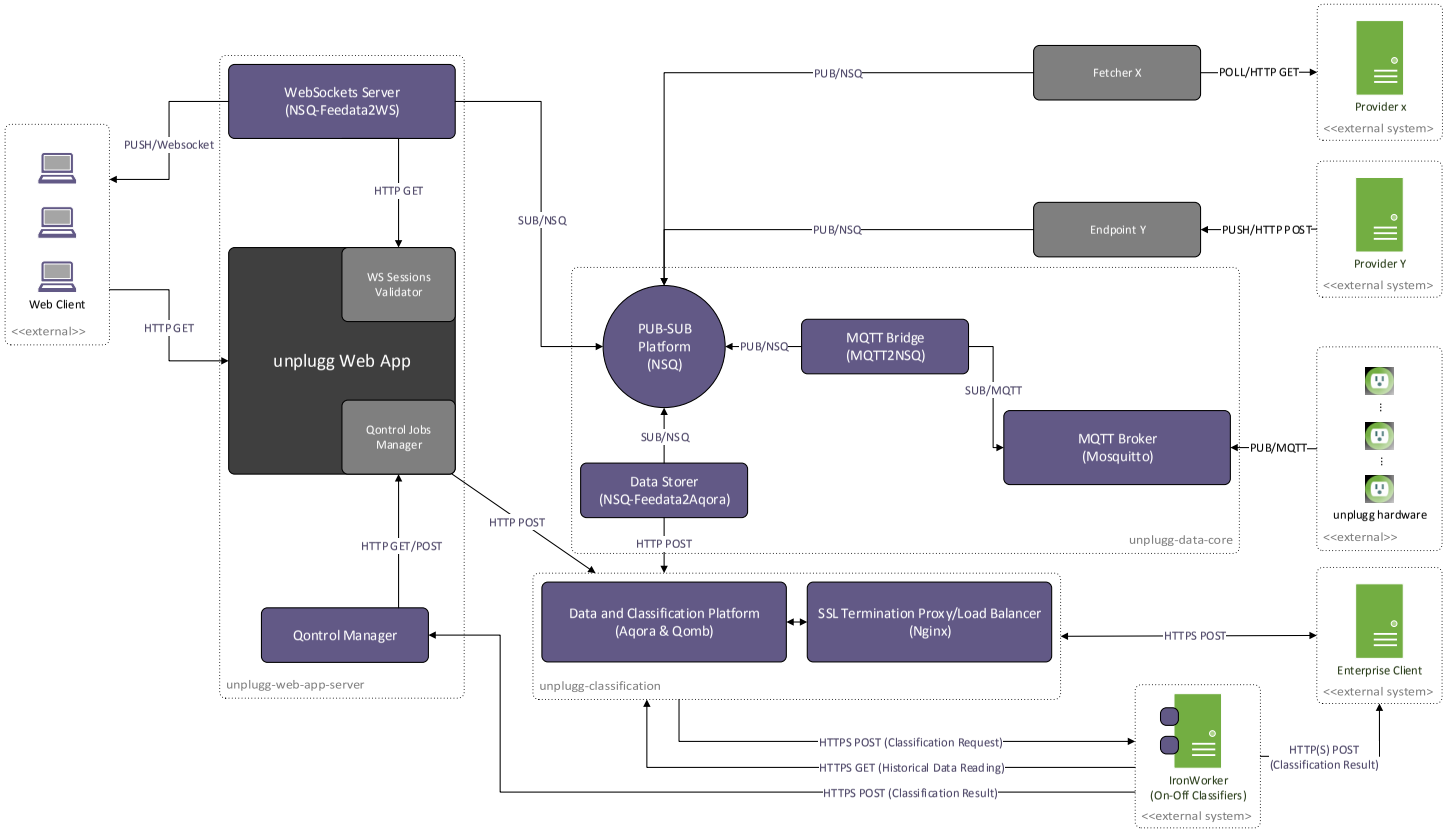

Figure 6.1: System overview diagram.

6.3 Components

In the following subsections, every service's function will be described, in order to clarify its contribution to the system and how it aligns with the business goals.

6.3.1 Aqora

`Aqora` is a time series data storage platform. Its goal is to provide a set of operations over a Canonical Data Model through a simple REST-ish API.

The Canonical Data Model is generic enough to fit both *unplugg* End User's and *Enterprise Customer*'s use cases; that is achieved by designing the Canonical Data Model (CDM) to have direct correspondence to the most relevant entities of both stakeholders' data models, given the problem domain. The set of operations is designed to be as small as possible for the sake of simplicity, while still providing the necessary manipulations over the CDM to persist energy consumption data. As such, *Aqora* classifies as a *multitenant* application, with every Enterprise Customer and unplugg itself being its *tenants*. A view of Aqora and its interactions is represented in Figure 6.2.

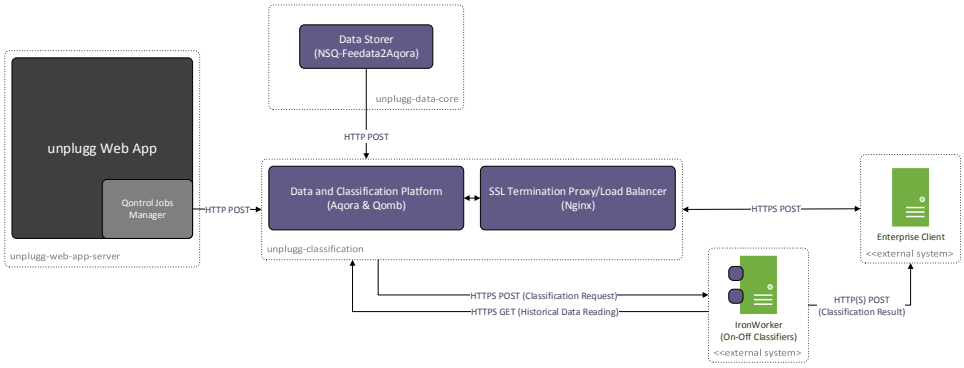

Figure 6.2: Aqora and Qomb's partial architecture view. It shows Aqora and Qomb (colectively referred to as `unplugg-classification`) and their interactions with the system's remaining components.

6.3.1.1 Rationale

Because the Classification System requires access to energy consumption historical data, the absence of a data storage platform would mean every request would have to carry that data. This would be a waste of bandwidth, as most of the data would have already been sent in previous requests; this would also lead to higher transmission times per request, which could create peak loads. Allowing the partial transmission of consumption data allows to spread the total transmission time over multiple requests, averaging the service load over time.

Using a Canonical Data Model (specified in 6.3.1.2) is advantageous because it abstracts the different stakeholders use cases while only preserving the entities that are meaningful to the Classification System; this leads to a reduced set of operations, which simplifies the API for both internal and external consumption.

The choice of using an API as the public entry point for this persistence system (in contrast with providing full raw access to a database) is mainly driven by allowing the implementation of Authentication/Authorization mechanisms. This is an important feature, since this API is expected to be used by multiple tenants, which expect their data to be only accessible by them and the service provider (unplugg).

Providing the persistence system as an API that wraps the available operations over a CDM also enables a higher degree of control over the way the data changes state. This way, additional data consistency is guaranteed, by encapsulating business logic rules on the application providing the API. For instance, this allows the API to enforce the uniqueness of a *Stream*'s name when associated with a given *Feed* (these entities are described in 6.3.1.2).

6.3.1.2 Canonical Data Model

Canonical Data Model is a design pattern that consists on the standardization of a business domain's entities and relationships so it may be used to communicate between different data formats; it is *canonical* since it aims to

simplify the problem domain's representation by being the *common denominator* of the multiple data models involved in the interaction.

In the particular context of Aqora, the use of a Canonical Data Model enables the service's abstraction of any particular business concerns of unplugg End Users' or Enterprise Customers' use case, while providing a standard data model to be used by the Classification System.

Aqora's entities are inspired by Xively's data hierarchy, of which Aqora's data hierarchy is a subset. It has 4 entities, which are described bottom-up in terms of hierarchy:

Datapoint A `Datapoint` represents a numeric `value` with an associated instant in time (`timestamp`). A `Datapoint` belongs to a single `Stream`. Its `timestamp` is unique among its `Stream`'s `Datapoints`.

Stream A `Stream` represents a time series of any given dimension (humidity, temperature, CPU usage, ...). It is an ordered collection of `Datapoints` and belongs to a single `Feed`. Every stream has a string identifier that is unique among its `Feed`'s `Streams`, the `Stream`'s `name`.

Feed A `Feed` is a collection of related `Streams`. The most common use case for a `Feed` is to use it as a representation of a physical device and its `Streams` as the device's multiple sensors, although many other use cases apply. A `Feed` has an identifier that is unique on the `Aqora` platform, the `Feed`'s `id`. Every `Feed` belongs to a single `Application`.

Application An `Application` represents the entity that performs requests of time series data storage and retrieval over `Feeds` and `Streams`. It has an `API key` and an `API secret`, used for Authentication and Authorization.

A Conceptual Data Model Diagram of Aqora's Canonical Data Model is depicted in Figure 6.3, simply representing its entities and their relationships.

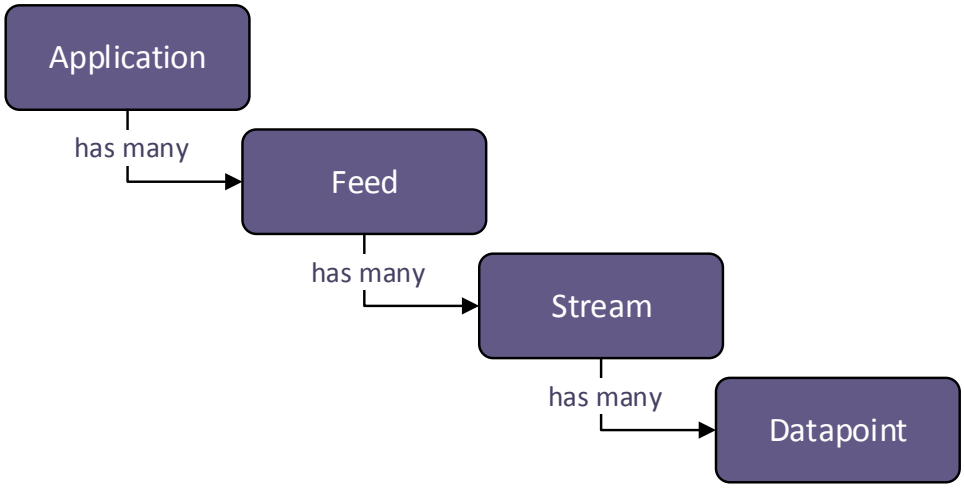

Figure 6.3: Conceptual Data Model Diagram of Aqora's Canonical Data Model. This Canonical Data Model is a linear hierarchical data model; it is capable of representing both major stakeholders' data, abstracting each use case's particular implementations.

Because of this data model abstraction, `Enterprise Customers` are not forced to use the data model hierarchy of `unplugg`, which models end-users' concerns; both parties may store their data in this platform.

The following list describes the `unplugg`'s data model entities correspondence with `Aqora`'s data model entities:

Plug A `Plug` corresponds to an `Aqora`'s `Feed`.

Consumptions (as a group) A `Plug`'s associated `Consumptions` compose a time series that is modelled by an `Aqora`'s `Stream`.

Consumption (as a single instance) Every `Plug`'s `Consumption` instance is modelled using an `Aqora`'s `Datapoint` and is associated with a `Plug`'s *consumptions* `Stream`.

Although an `Aqora`'s `Stream` does not provide added value when used to model the current unplugg End User's use case, the addition of a level of indirection is meant to future-proof the possibility of creating classifications that use multiple time series collected by the same device. This is seen as

a consequence of generically using a `Feed` to model a physical device and `Streams` to model its sensors: because of this model, a classification will be able to use the device's multiple sensors to produce a single result.

6.3.1.3 REST API

Aqora's REST API is designed to provide support to the essential manipulations of the CDM. Its API is kept as small as possible and closely maps the data hierarchy of the data model. The HTTP verbs are used according to their semantics.

6.3.2 Qomb

`Qomb` is a data analysis service. It aims to provide a generic interface for handling data analysis requests over `Aqora Feeds` and `Streams`. It aims to be loosely-coupled with its consumers and to return the classification result using an event-driven approach. The Authentication/Authorization of the classification requests is performed by `Aqora` during the classification's retrieval of data over which the request operates. `Qomb` is designed to offload the classification task to workers that handle the requests; this way, `Qomb`'s own resource consumption is kept as low as possible, assuring the service scales as the number of requests grows. A view of Qomb and its interactions is represented in Figure 6.2.

6.3.2.1 Rationale

`Qomb` is one of the main drivers of the standardization of the persistence system through the creation of a Canonical Data Model and the corresponding CRUD-esque API. The assumption that `Qomb`'s classification operations act over a *known* data model allows the request payload to be kept as small as possible, given the fact that this allows the payload to *refer* to the resources, instead of *sending* them. Since `Qomb` is designed to offload the requests, keeping the requests' payload small is important for the scalability of the service (less time will be spent sending the request's payload to the worker). Using this standardized persistence system as the classification operation's only

data provider also enables the standardization of `Qomb`'s own request interface; this allows the service to accommodate the future addition of different classification operations, as long as they comply with the interface.

For this internship's scope, the only relevant classification operation is On-Off Classifier, described in subsection 6.3.3; however, it should be made clear that this service was specifically designed this way to allow its future extensibility.

Because classification operations are usually long running tasks, it would be inefficient to keep a connection open per request during the process; instead, an event-driven approach called `WebHooks` is taken to provide the consumer with the classification result asynchronously. `WebHooks` are user-defined HTTP(S) callbacks, triggered by web application events; in this case, an event occurs when a `Qomb Classification` finishes its processing and is ready to provide the result. That event occurrence triggers an HTTP/`HTTP(S) POST` to be performed by the `Qomb Classification` on the specified endpoint (provided on the request payload) containing the classification result.

Every `Qomb Classification` expects the request payload to provide the `WebHook URL` where it should publish the result, and every `Qomb Classification` (such as On-Off Classifier) honours this contract. This asynchronous mechanism grants the platform with greater scalability (by not maintaining open connections during the process) while providing *as soon as possible* results.

The Implementation details of this service are described in section 7.2.3.

6.3.2.2 REST API

Qomb's REST API is designed to provide simple methods for requesting classification tasks. Every classification test returns a `task_id`, a unique `id` per received classification request. `task_id` is later used by the requester to validate the authenticity of the published payload.

6.3.3 On-Off Classifier

An `On-Off Classifier` is responsible for classifying an energy management device's time-series of consumption data into a series of `on-off` states (transitions between being turned on and turned off).

The original `On-Off Classifier` has direct access to the `unplugg Web Application`'s `Master Database`, in order to obtain the energy consumption data. A part of `Aqora`'s rationale is to avoid this `Data Coupling` between both components. `On-Off Classifier` has to be adapted to use `Aqora`'s interface and make use of `Aqora`'s `Data Model`.

The current version also uses a database (`Rules Database`) to store its output, that would be later accessed by `Actuator` for the automation to occur. Since there are now multiple stakeholders, the output mechanism will have to be replaced.

The proposed `On-Off Classifier` will classify an `Aqora Feed`'s `Stream` of energy consumption values into an array of `<day of week, hour of day, on-off state>` tuples. The classification request is received through Qomb's interface and offloaded using the IronWorker service (already used by its previous implementation), to satisfy the CPU needs of the process without disrupting the `Qomb` service.

The proposed `On-Off Classifier` will output its classification result using `WebHooks`, as described in Qomb. This mechanism allows the result to be provided asynchronously to the requesting client. As a consequence, `Qontrol Manager` will be the service responsible for receiving classification results on `unplugg`'s behalf).

An instance of `On-Off Classifier` is spawned for every `On-Off` classification request received by Qomb.

6.3.4 Qontrol Manager

`Qontrol Manager` is the `unplugg`'s service responsible for storing the `On-Off` classification results of `unplugg`'s enqueued jobs; it is the service associated with the `WebHook` used by `unplugg` during a request to `Qomb`. The service is the owner of a database, where it persists the array of tuples mentioned in

On-Off Classifier.

Since the service is publicly accessible[1], it must perform a validation of the incoming classification results, in order to avoid data tampering and unauthorized interactions. To allow this, `Qontrol Jobs Manager`, the service responsible for requesting the classification task, generates a random token (`qontrol-token`) that, in conjunction with `task_id` (a *unique* token) is capable of identifying the validity of the request. `qontrol-token` is provided to `Qomb` (and therefore, the worker) as a query parameter of the `WebHook URL`; because of this scheme, this is extensible to provide a more complex method of validation.

When receiving a result, `Qontrol Manager` proceeds to confirm the authenticity of the result by requesting `Qontrol Jobs Manager` to validate the combination of `task_id`, `qontrol-token` and `feed_id`. It does so by performing an `HTTP` request to `Qontrol Jobs Manager`; if the `HTTP` return status is `200 OK`, the result will be accepted and stored in the database; otherwise (`401 Unauthorized`), the result is discarded and the connection with the rogue publisher is dropped.

6.3.5 MQTT Broker

The `MQTT Broker` is the component that receives consumption data directly from unplugg's energy management devices. It is one of the public-facing components of the system. Being a message broker, this is the component that maintains central control over the flow of messages, since it also acts as a message router. As a consequence, it is the only component capable of imposing Authentication and Authorization to `MQTT` publish and subscribe requests.

6.3.5.1 Rationale

Previous research of the company on the area of the Internet of Things pointed to `MQTT` as a probable and preferred choice for a lightweight pro-

[1] `Qontrol Manager` must be publicly accessible since it corresponds to the `WebHook` URL and must be reachable by `On-Off Classifier`.

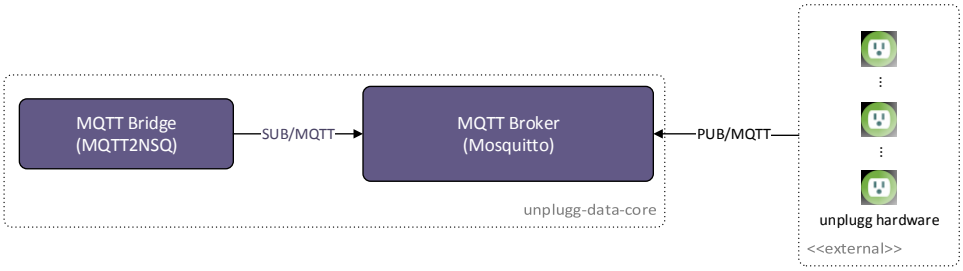

Figure 6.4: MQTT Broker and its interactions on the system.

tocol that suited an IoT scenario and aligned with the company's goal of receiving highly granular energy consumption data from unplugg's energy management devices.

After performing some personal research over the protocol's design principles and performance benchmarks, it was found suitable for the purpose of interacting with the expected kind of devices. However, in order to better comply with the security concern of the company (in this particular case, the privacy of users' data), the protocol's plaintext nature had to be addressed, since this is one of the public-facing servers. As the recommended way to implement encryption on the protocol is to use `MQTT` over `SSL`, the privacy of the data is enforced by making this the only mode of operation supported by `MQTT Broker`.

6.3.5.2 Authentication and Authorization

Since these mechanisms are dependent of the `MQTT Broker` implementation, they are discussed in the MQTT Broker section, in the Implementation.

6.3.6 Publish-Subscribe Messaging Platform/Library

This subsection does not describe a component *per se*; instead, it describes an Enterprise Integration Pattern that has consequences on the remaining Design and Implementation of the system. The implementation of this pattern may spawn no or multiple components[2], that being the reason as to why

[2]A brokered implementation of the Publish-Subscribe pattern will spawn at least one component (the broker), while a brokerless implementation does not spawn any compo-

the title is kept generic. A representation of every component that relies on this pattern is represented in Figure 6.5.

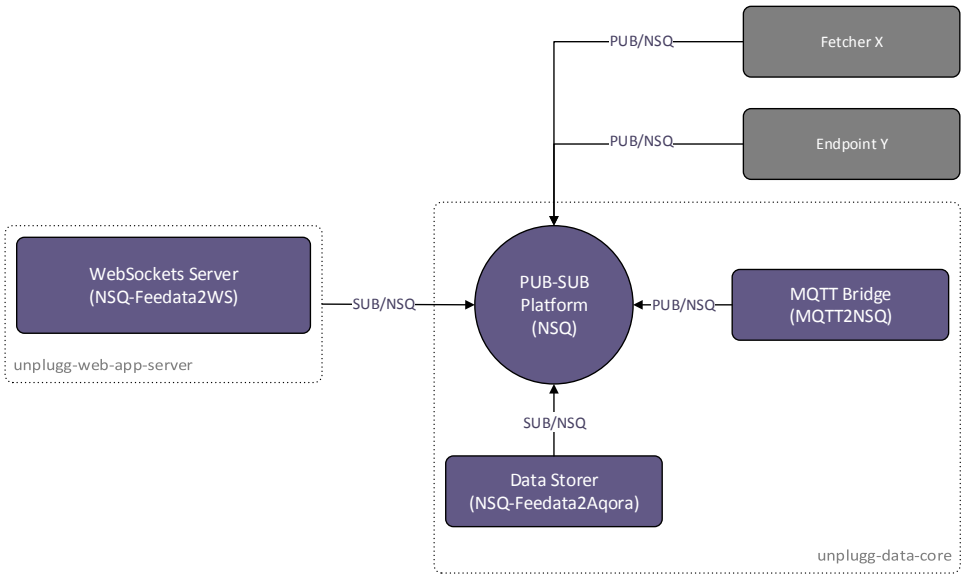

Figure 6.5: Publish-Subscribe Producers and Consumers of Energy Consumption Data.

6.3.6.1 Rationale

Some of the services that compose the system can be seen as `Producers` of energy consumption data. These are the `Data Fetchers`, responsible for collecting energy consumption data from Third-Party Providers, and the MQTT Broker, responsible for collecting energy consumption data produced by unplugg's energy management devices.

Some of them, on the other hand, may be seen as `Consumers` of energy consumption data. These are the Aqora Storer, responsible for storing energy consumption data into Aqora (described in 6.3.8), and the WebSocket Server, responsible for streaming real-time updates through WebSockets to web clients (described in 6.3.9).

nents.

In order to promote loose coupling between energy consumption data `Producers` and `Consumers`, a Publish-Subscribe messaging platform or library is used.

Currently, real-time updates of energy consumption data provided to Web Clients are bound to the `Consumption Data Model` implementation of `unplugg Web Application`: the creation of a `Consumption` instance on `unplugg Web Application Data Model` triggers the broadcast event of an energy consumption update. This implementation shows low cohesion, since it is not the responsibility of a `Data Model Implementation` to provide real-time updates to web clients.

One of the main reasons to implement this solution is to avoid the coupling between a `Producer` (in this particular case, the responsible for creating a `Consumption` instance) and a `Consumer` (in this case, the `Real-time Updates Server`). Because of the current implementation, the only way to extend the access to new energy consumption data to further `Consumers` would be to add additional triggers, which implies that the `Consumption Data Model Implementation`'s cohesion would become even lower.

By using the Publish-Subscribe messaging pattern, either implemented using a broker or a broker-less solution, it is possible to add and remove `Producers` and `Consumers` of energy consumption messages without changing the remaining participants' code.

Implementing this pattern using a solution that is capable of delivering messages with low latencies enables its `Consumers` to receive real-time energy consumption updates. This allows unplugg's End Users to be updated in real-time through the Web Client while also making the data available as soon as possible to the `Aqora` service and, consequently, to the `Qomb` service and the `On-Off Classifier`. This enables unplugg's Enterprise Customers to trigger `On-Off Classifications` using the most recent energy consumption data as soon as possible.

One of the long-term goals of using such a solution is to enable real-time access and **processing** of this data by multiple services, which may enable future business opportunities for the company, without ever disrupting the platform's normal activity.

In order for this solution to be reliable, it should also support reliable delivery of messages; ideally, this would correspond to a delivery with `Exactly Once` semantics, but `At least once` semantics delivery is also acceptable, as long as the `Consumers`' operations are designed as *idempotent* operations.

The implementation details of this pattern are described in NSQ (7.2.5).

6.3.7 MQTT Bridge

`MQTT Bridge` is the service that bridges `MQTT Broker` and `NSQ`; it is a `Messaging Bridge` according to Enterprise Integration Patterns. It is both a subscriber of `MQTT Broker` and a publisher of `NSQ`, which means every message received by the `MQTT Broker` will be published on `NSQ` for the interested NSQ Consumers (Subscribers) access. Its topic subscription is `feedata/#`; in `MQTT`, this corresponds to a multi-level subscription of `feedata`, which matches the energy consumption data topics used by every plug.

6.3.8 Aqora Storer

`Aqora Storer` is the service responsible for persisting every `Feedata` message on `Aqora` using `unplugg`'s credentials. Because the exposed interface for the `Aqora` platform uses `HTTPS`, some latency may be introduced by the overhead of establishing an encrypted connection per message received, if performed synchronously. This latency may be mitigated by concurrently establishing parallel `HTTPS` connections, by running multiple instances of this service. This is possible because:

- This is a stateless service; no state needs to be shared across instances.
- If an `Aqora Storer` instance suffers an unclean shutdown, no message will be lost because of the `at least once` message delivery guarantee.
- `Aqora` was designed to ignore duplicates, in order to comply with an `at least once` message guarantee.

The Implementation details of this component are described in NSQ-Feedata2Aqora (7.2.9).

6.3.9 WebSocket Server

A `WebSocket Server` is used to provide real-time updates to web clients. Using `WebSockets`, the service is able to maintain full-duplex connections with web clients and push the data whenever available, while retaining the possibility of implementing request-based functionality. It provides similar functionality to the `Real-time Updates` mechanism described in Current System Analysis; however, it is designed to do so while promoting loose-coupling. While the previous mechanism relies on `Data Model` operations' callbacks, this service will be a `Subscriber` of the energy consumption data published on the `Publish-Subscribe Platform`. A representation of the partial view of `WebSocket Server` is depicted in Figure 6.6.

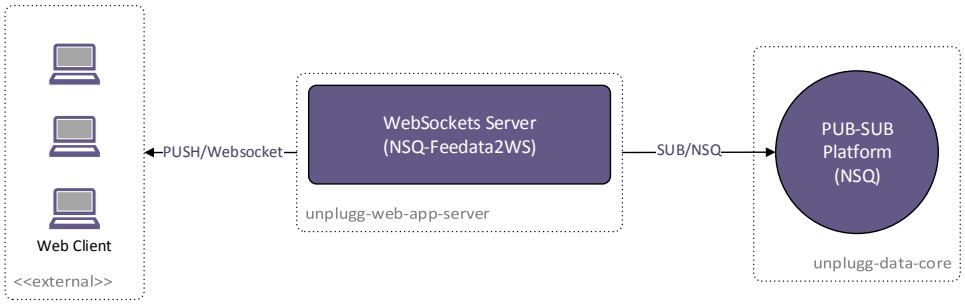

Figure 6.6: WebSocket Server partial architecture view.

In order to avoid the security issue described in Real-time Updates Subscription Flaw, *Authentication* and *Authorization* is mandatory before any client starts receiving energy consumption data. The flow of this mechanism is as follows:

1. A user authenticates before `unplugg Web Application` using the web client, by requesting the `www.unplu.gg` page and providing correct credentials (username and password).

2. When serving a page with real-time updated data (an `Aqora`'s `Feed` graph representing an Energy Management Device), the `unplugg Web Application` generates a random token of sufficient length and adds it to the page. `unplugg Web Application` stores the token and the

user associated with it,

3. The web client establishes a connection with the `WebSocket Server` and provides the token (from now on referred to as `ws-token`) and the list of `Feeds` it is interested in receiving energy consumption updates from.

4. The `WebSocket Server`, unaware of the user requesting the subscription, performs an `HTTP` request to the WS Sessions Validator requesting it to assess if the provided `ws-token` is authorized to receive the intended `Feeds`; this follows the same flow as the one described in `Qontrol Manager`. If the return HTTP status code is `200 OK`, it proceeds to store this association and will begin sending every incoming `Feedata` message that matches one of the `Feeds` through the web client's Web Socket. Otherwise, if the return status code is `403 Forbidden`, the connection is immediately closed, denying unauthorized access to the data.

6.3.10 Reverse Proxy, SSL Termination Proxy and Load Balancer

6.3.10.1 Reverse Proxy

The use of a Reverse Proxy is essential for providing a Unified API: the perception that a single API exists, while in reality it maps to multiple proxied applications. Both Aqora and Qomb APIs are designed and implemented as REST-ish APIs (as described in 7.2.2.1); as these are different applications, they would be reachable through different ports or even IPs or hostnames. The Reverse Proxy allows to abstract the deployment details of these applications by publicly serving both applications through the same host and port (the Reverse Proxy's), with traffic being routed based on the requested resource. Since their URL schemas were designed to be similar, and because Qomb uses the same Authorization scheme as Aqora, Qomb's API may seem as an extension of Aqora's to external users and developers, contributing to the perception that the resulting API (the combination of

both APIs) corresponds to a single application, rather than multiple APIs. This contributes to a better API usage experience, while enabling a dynamic deployment scenario.

6.3.10.2 SSL Termination Proxy

To satisfy the need of using HTTPS, as analysed in 5.2.2, an SSL termination proxy is used to secure every public facing application. This brings some advantages:

- A single certificate may be used to encrypt every request made to the proxied applications, instead of multiple certificates (one per application), which would have costs. This advantage is also related with the Unified API aspect of using a Reverse Proxy.

  This also reduces the access surface of the certificate, since only one process is required to access the certificate. The alternative would be to either make the certificate accessible to multiple applications through the network (insecure) or provide every application with a copy of the certificate (also insecure).

- SSL termination enables public communication to be secured by HTTPS while decrypting the traffic for consumption inside the private network. Allowing the traffic to flow inside the private network in plaintext form saves bandwidth (since the payload is smaller) and CPU usage (since the encryption/decryption process is CPU intensive); it is a good compromise between resource usage and security, given the assumption the private network is secured. This allows the applications to communicate between themselves in plaintext form inside the network while doing their public communications in encrypted form.

  The CPU offload consequence of using an SSL termination proxy is of extreme importance in a SOA context from a scalability point of you, since this allows independent scaling of the SSL termination proxy and every application public-faced by it. This also allows complex

deployment scenarios; for instance, the deployment of the SSL termination proxy on encryption specialised hardware, while deploying the applications in commodity hardware.

6.3.10.3 Load Balancer

In order to be possible for the applications to scale horizontally, it is important that the request load is balanced between every application's instance. Since the Reverse Proxy is the system's entry point for client requests, either the proxied requests are delivered to a load balancer that is aware of the application instances, or the Reverse Proxy is itself the load balancer, therefore being able to proxy the requests directly to the least loaded instance.

The Load Balancer is therefore an important component for the scalability of the overall system.

6.3.11 Specified Components

The components described below were specified components that were not implemented nor planned to be implemented during the internship's course; this is because they need to be implemented in direct integration with the current system, which was clarified as out of scope in the Introduction chapter (section 1.3). Possible implementations are described, and the author's preference and reasoning is provided.

6.3.11.1 WS Sessions Validator

`WS Sessions Validator` is the component responsible for validating the permissions of a `ws-token` to request specific `Feeds` real-time updates. Since the Authentication of a web client's user is provided by `unplugg Web Application`, this application is the only component that is able to establish the relation between an authenticated user and a `ws-token`. `ws-token` is the piece of data that enables a web client to authenticate before `WebSocket Server`, allowing the server to assess the user's authorization to access the requested `Feeds`. This component has two possible implementations:

- It may be implemented on `unplugg Web Application`. When serving the web page, a call to its code would generate, serve and persist the `(ws-token, authenticated user` relation on a database. An additional web endpoint would be added to `unplugg Web Application`'s API, in order to serve `WebSocket Server` validation requests.

- It may be implemented as a service. When serving the web page, `unplugg Web Application` would call this service to generate a `ws-token`, while providing the authenticated user (so it may be retrieved during Authorization). `unplugg Web Application` would serve the `ws-token` returned by the service, while the service would persist the relation, allowing it to respond to future requests by `WebSocket Server`.

`WebSocket Server` expects this component to be compliant with the `HTTPS API` for validating a request, independently of its implementation.

Following the Service-Oriented Architecture approach promoted throughout this internship, the suggested implementation is the latter. This has the advantage of allowing the independent deployment of both components, to address their different workload needs.

6.3.11.2 Qontrol Jobs Manager

`Qontrol Jobs Manager` is the component responsible for requesting `On-Off` Classification requests to `Qomb` on behalf of `unplugg` (End Users); as such, it is the only component able to validate the (`qontrol-token`, `task_id`) combination[3].

Because `Qontrol Manager` is publicly accessible, it must be able to authenticate the received results before storing them. `Qontrol Manager` was designed to expect the `Qontrol Jobs Manager` component to implement an HTTP API with a single method. This method must receive the `task_id`, `qontrol-token` and the `feed_id` fields and must return `200 OK`, in the case of a valid combination of fields. Otherwise, the return status code should

[3] As mentioned in `Qontrol Manager`, `qontrol-token` is generated by `Qontrol Jobs Manager` and added as a query parameter to the `WebHook URL`.

be different (`401 Unauthorized`), allowing the `Qontrol Manager` to drop a rogue client and the published result.

The most important feature of `Qontrol Jobs Manager` is to be able to validate a result received by `Qontrol Manager`. That is achieved by persisting (`task_id`, `qontrol-token`, `feed_id`) during a classification request for later retrieval. However, the implementation itself may be performed in two distinct ways:

- It may be implemented on `unplugg Web Application`. An additional web endpoint would be added to `unplugg Web Application`'s API, in order to serve `Qontrol Manager` validation requests.
- It may be implemented as a service. A simple HTTP API would provide the web endpoint necessary to validate the classification result. An additional method would be created to provide a method for persisting the validating combination, either performing the request itself or performed by another component of the system.

Chapter 7

Implementation

With proper design, the features come cheaply. This approach is arduous, but continues to succeed.
— Dennis Ritchie

After the System's Architecture and Design comes Implementation. Implementation is nothing more than a realization of the technical specifications provided by the Architecture and Design phases. Therefore, this chapter describes the reasoning behind some of the technological choices and service-specific implementation details.

7.1 General Technological Choices

Some of the services described in the Architecture and Design chapter are already available with known implementations such as Reverse Proxy, SSL Termination Proxy and Load Balancer; however, most of the described components/services needed to be implemented from scratch. For implementing those services, the same programming language and runtime environment were used: `JavaScript` was the chosen programming language, while the chosen runtime environment was `Node.js`. A brief description of the language and the runtime are provided, as well as the reasoning behind the choice. A description of the chosen library for implementing the REST APIs is also provided.

7.1.1 Programming Language

`JavaScript` is a prototype-based scripting language with dynamic typing. It is a multi-paradigm language: it supports object-oriented, imperative and functional programming. Being a multi-paradigm language with dynamic typing gives a greater flexibility in terms of programming style, but the choice of the language is essentially a consequence of using the `Node.js` runtime environment, described in the following subsection.

7.1.2 Runtime Environment

Node.js is a runtime environment for server-side and networking applications that uses `Google V8` JavaScript engine to execute code. As an asynchronous event driven framework, it is designed to maximize throughput and efficiency by using non-blocking I/O and asynchronous events that are processed in a single-threaded event-based loop.

7.1.2.1 Rationale

All the services required by the system are I/O bound, due to the nature of their operations: since the CPU intensive tasks (Classification operations) are offloaded to external systems, most of the operations performed inside the system are:

- Maintaining real-time clients' connections, with both incoming and outgoing messages: MQTT and NSQ-Feedata2WS.
- Retransmitting messages between two different message channels: MQTT2NSQ-Feedata.
- Handling requests that either cause database or networking (external APIs) operations: Aqora, Qomb, NSQ-Feedata2Aqora and Qontrol Manager.

Since these operations are I/O bound (no CPU intensive tasks are performed by these services), their performance will most likely benefit from the `Node.js` runtime and its non-blocking I/O and event-oriented approach.

7.1.3 REST APIs

The chosen `Node.js` libray for implementing REST APIs was `express.js`. It is the *de facto* web application framework for `Node.js` applications, being the most depended upon web application framework by the packages registered on `npm` (`Node.js`'s public package registry and package manager), the 6th most depended upon package [30] and the 24th most starred code repository on GitHub [30] (as of August 2014); high profile users include [31] MySpace, Segment.io, Mozilla's Persona, Ghost, LinkedIn[32] and others.

`express.js` is built on top of `Connect`[1], an extensible HTTP server framework; its extensibility is based on the use of plugins known as *middleware*; a *middleware* is essentially a function that operates over a client request (for instance: body validation). Every request received by *Connect* is passed through a *middleware* chain, defined by the developer. This approach has greater flexibility, as it allows the developer to define the exact operations to be performed and adapt them and their order to the developer's needs. There is a wide range of already implemented `Connect` *middleware* available that performs common operations such as compression, request body parsing, cookie sessions' handling, routing, logging and others.

7.1.3.1 Rationale

Multiple reasons play behind the choice of `express.js`:

- My familiarity with the framework, having implemented various projects with it, both personal and academic.
- Although there are other `Node.js` libraries specifically created for API development, they do not provide the same support and flexibility as `Express.js`, either because they aren't as popular as `Express.js` (reduced amount of available information) or because they are designed to solve particular patterns, such as REST APIs. Some of the designed APIs' methods have complex business logic that does not align with CRUD semantics; they also have route-specific *middleware* needs (i.e.,

[1] Connect: `https://github.com/senchalabs/connect`

there are routes that need Authentication, routes that need Authorization, while some are public).

`Express.js` support for route-specific *middleware* chains and support of a finer control over the request routing (complex URL schemas) allows a more modular structure of the code by reusing *middleware chains* on multiple routes. This leads to cleaner code, since non-global *middleware* can be chained on a route's *middleware* chain.

- Its popularity plays an important role on the amount of available documentation, related libraries, the wide range of *middleware* libraries covering numerous common use cases and the amount of discussion of common issues. All these factors allow faster implementation cycles and easier maintenance of the APIs on the long term.

7.1.4 Database

The chosen database for persisting data was `MongoDB`[2]. It is used to persist both `Aqora` and `Qontrol Manager`'s data.

`MongoDB` is the most popular NoSQL database system, as of August 2014. [33] It is a document-oriented, dynamic schemas database; these dynamic schema documents are stored as `BSON` (Binary `JSON`), a `JSON`-like format. `BSON` data types are a superset of `JSON` types, but `BSON` is designed to have a minimum spatial overhead and to be easily traversable, necessary for search operations.

MongoDB allows data queries using user-defined JavaScript functions and provides an aggregation framework based on the `MapReduce` paradigm for condensing large volumes of data. It is capable of providing high availability and horizontal scaling through the use of replica sets (two or more copies of the data) and sharding (horizontal data partition). `MongoDB` also grants the user the control over the *Write concern* of the database (the guarantee that MongoDB provides on a write operation); this allows to fine-tune the trade-off between `write guarantee` and `performance`.

High-profile users include *eBay*[34], *Foursquare*[35] and *SAP*[36].

[2]MongoDB: `https://www.mongodb.org/`

7.1.4.1 Rationale

Multiple reasons lead to the choice of `MongoDB` as the database:

- Proven capability to handle high volumes of data while maintaining performance [37].
- Having dynamic schemas adds flexibility to the data model, allowing future modifications without the need for performing expensive operations.
- Since the chosen Programming Language is `JavaScript` and the REST APIs payloads are in `JSON` format, using a database that stores `JSON`-like structures benefits the development of the project by allowing the usage of a single language and format across the full stack. Furthermore, it is a better and simpler approach to store the structures as-is that reducing them to multiple relations in a relational database. [38]
- The `MongoDB Node.js` Driver[3] fully implements the `MongoDB` API and provides an asynchronous interface for making use of `Node.js`'s event-oriented model.

7.1.5 In-memory Key-value Data store

The chosen `In-memory Key-value Data store` was `Redis`[4]. It is used by Aqora and NSQ-Feedata2WS to maintain an in-memory key-value data store for fast storage and retrieval of small chunks of data; their use cases are described in 7.2.2.3 and 7.2.10.1.

`Redis` is a key-value cache and store with optional durability. `Redis` is the most popular key-value store (as of August 2014 [39]), with high-profile users including [40] Twitter, GitHub, StackOverflow and Flickr.

It is a remotely-accessible dictionary which maps keys to values; its values are not only strings, but also lists, sets, sorted sets, bitmaps, hashes and `HyperLogLogs` (a data structure used to estimate the cardinality of a set)

[3] `node-mongodb-native`: `https://github.com/mongodb/node-mongodb-native`
[4] Redis: `http://redis.io/`

[41]. Because durability is optional, the in-memory nature of Redis grants it with great performance, both on writing and reading operations [42]; the documentation details the time complexity of every operation [43]. It is implemented as a single-threaded single process. It supports transactional operations, replication and sharding.

7.1.5.1 Rationale

`Redis` was chosen due to its proven performance for reading and writing small chunks of data, since durability was not necessary for the needs. `Redis` is used to store temporary data on both (Aqora and NSQ-Feedata2WS) use cases: session details. Since session data is verified for every request, it is necessary to use a storage medium that was capable of providing read/write performance; due to a session's temporary nature, the durability trade-off is acceptable and fit the semantics of the use case.

Because of `Redis` in-memory nature (when durability is off), it is capable of delivering the necessary performance.

7.2 Implemented Components

7.2.1 Reverse Proxy, SSL Termination Proxy and Load Balancer

The chosen software for the functions of Reverse Proxy, SSL Termination Proxy and Load Balancing was Nginx[5]. Nginx is an open source reverse proxy server with a strong focus on high concurrency, high performance and low memory usage.

On 3rd of July, 2013, according to W3Techs, Nginx became the most used web server among the Alexa's Top 1000 websites (surpassing Apache)[44], serving 41.1% of those websites (as of 13 August 2014); it currently serves 21.2% of all the websites which the web server is known to W3Techs[45] (as of 13 August 2014). High profile users include Netflix[46] and Wikipedia and Wikimedia sites[47] (as an SSL termination proxy).

[5]Nginx: `http://nginx.org/`

It has an asynchronous event-driven architecture (in contrast with the typical multi-thread or multi-process approach), which provides more predictable performance under high loads. Nginx has additional functionalities besides reverse proxying, namely load balancing (with health checks), HTTP caching, static file web serving, `gzip` compression and decompression, `IPv6`, `SPDY` and WebSockets support, among others. Since it is capable of reverse proxying HTTPS, Nginx is also a capable SSL termination proxy.

7.2.1.1 Rationale

Nginx was the chosen software because its modular, event-driven, asynchronous, single-threaded, non-blocking architecture achieves greater performance, density and economical use of server resources when compared to Apache, the most used open source web server (being Nginx the second most used). This is because process or thread-based models of concurrency handling have degraded performance when scaling due to thread trashing or excessive context switching. Nginx was created from scratch to handle the C10K problem and, as such, its architecture is made to control both CPU and memory usage and maintain them to a minimum, which allows Nginx to scale non-linearly with the growing number of connections and requests; Nginx claims to handle 10000 simultaneous HTTP keep-alive idle connections using only 2.5MB.[48]

This allows the focus to be on its particular features, since Nginx's impact on performance won't be perceivable.

7.2.2 Aqora

The implementation details of Aqora are now provided.

7.2.2.1 REST API

`Aqora`'s `REST API` was implemented using `Node.js` and `Express.js`, following the rationale described in section 7.1.3.1.

The code was organized following a resource-oriented approach: every resource directory contains its routes and model source code. This contrasts

with a function-oriented source code organization, where models and routes are grouped together.

Routes were implemented using `namespaces`; this allows routes to be hierarchically extended by other routes. This facilitates the process of extending a resource's functionality.

The implemented Authorization and Authentication flow is provided in Appendix B.

7.2.2.2 Database and Data Model

`Aqora` provides an API for persisting time-series data, used by unplugg End Users and Enterprise Customers; as such, this service is expected to have high usage. Particularly, its use case consists of a high volume of write operations and a moderately high volume of read operations. `MongoDB`'s lazy writing to disk (using journaling) with the appropriate *Write concern* fits this kind of usage. [49]

The `Data Model` manipulation was implemented using `Mongoose`, a `MongoDB` object modelling library that provides equivalent functionality to an `ORM` (Object-relational mapping) for `SQL` databases.

`Aqora` provides a route for bulk insertions of `Datapoints`; because `Mongoose` provides an abstraction for the native `MongoDB` driver by wrapping helping methods and fields around objects, the performance of this multiple-document insertion was not optimal, since the process of wrapping objects with those methods and fields causes some overhead. In this particular method, the native `MongoDB` driver was used. Furthermore, every lookup method (API methods that fetched energy consumption data) was implemented with additional configuration in order to prevent `Mongoose` from adding those methods and fields. This modification is reported to have had 3.5x speedup on particular find queries and sets of data. [50]

7.2.2.3 Session Data store

Since the majority of the requests requires Authentication and Authorization, the verification of such conditions should be performed as fast as possi-

ble in order to avoid impacting the round-trip time of every request. A Redis instance is used to cache (Aqora session token, Aqora Application ID) associations after applications (either unplugg Web Application or Enterprise Customers) perform Authentication.

7.2.3 Qomb

Qomb's REST API was also implemented using Node.js and Express.js, following the rationale described in section 7.1.3.1. The JSON payload received during the request is transformed before spawning a Qomb Classifier; the Authorization header is added to the payload, so the worker may be capable of authenticating itself before Aqora *as* the requesting application.

It is important to note that, although this internship's scope only required the On-Off Classification to be provided as a service, this platform was designed and implemented to be easily extended to support other classification mechanisms. This was possible due to the definition of the request payload format as abstract enough to represent any classification operation that operates over a time-series.

7.2.4 MQTT Broker

The chosen MQTT Broker implementation was Mosquitto. Mosquitto is an open source broker implementation of MQTT 3.1 and 3.1.1. It supports Authentication, Authorization, Certificate based SSL/TLS, Pre-shared-key based SSL/TLS, MQTT Broker Bridging[6] and its functionality is extended by plugins, namely complex Authentication/Authorization schemes. In order to guarantee the privacy of the messages, the Certificate based SSL/TLS is the appropriate one, as suggested in section 6.3.5.1. mosquitto-auth-plug (https://github.com/jpmens/mosquitto-auth-plug) supports multiple back-ends for storage and integration with the existing system. A PostgreSQL as back-end for the mosquitto-auth-plug covers both Authentication and Authorization concerns.

[6]Bridging multiple brokers allows them to publish topics on other brokers, allowing client segmentation and data aggregation.

7.2.4.1 Authentication

Authentication is supported by using a username and password combination, with the password stored using `PBKDF2`[7]. Since unplugg energy management devices do not exist at the time of implementation, the integration with the database that creates the credentials used by the devices will be performed manually until an automated process of deploying the devices (with bundled software and credentials) is available.

7.2.4.2 Topic Authorization

Authorization is supported by the Access Control List (`ACL`) functionality of `Mosquitto` and `mosquitto-auth-plug`. For every topic published by an unplugg energy management device, an entry associating the topic and the device's username must be created, as well as the device's permission regarding the topic (write-only). Since the device's topic corresponds to an Aqora `Feed`, this process will be performed manually for the same reasons above until an automated process of device deployment is arranged. After the relation is established, `Mosquitto` is able to perform an `ACL` check and either allow or reject the message published by the client.

7.2.5 NSQ

`NSQ`[8] was the chosen distributed messaging platform to implement the Publish-Subscribe pattern described in Publish-Subscribe Messaging Platform/Library.

`NSQ` is a distributed messaging platform. It is designed for distributed topologies with no SPOF, horizontal scalability, low latencies, data format agnosticism, both suitable for low-throughput and high-throughput workloads; it provides at-least-once message delivery semantics. [51]

NSQ implements the Publish-Subscribe Pattern with topic-based message filtering, with an additional level of message distribution: channels. While a topic is a stream of data, a channel is a logical grouping of consumers

[7] `PBKDF2` is a key derivation function, that applies a pseudorandom function over a salted password and repeats the process a number of times.

[8] NSQ: `http://nsq.io`

subscribed to a given topic. When a consumer performs a `subscribe`, it subscribes to a topic on a given channel; for every message produced on a given topic, every channel of the topic will receive a copy of the message, while only one subscriber per channel will receive the message, in a distributed fashion. This provides load-balancing amongst the channel's consumers. The topic/channel message distribution is represented in Figure 7.1.

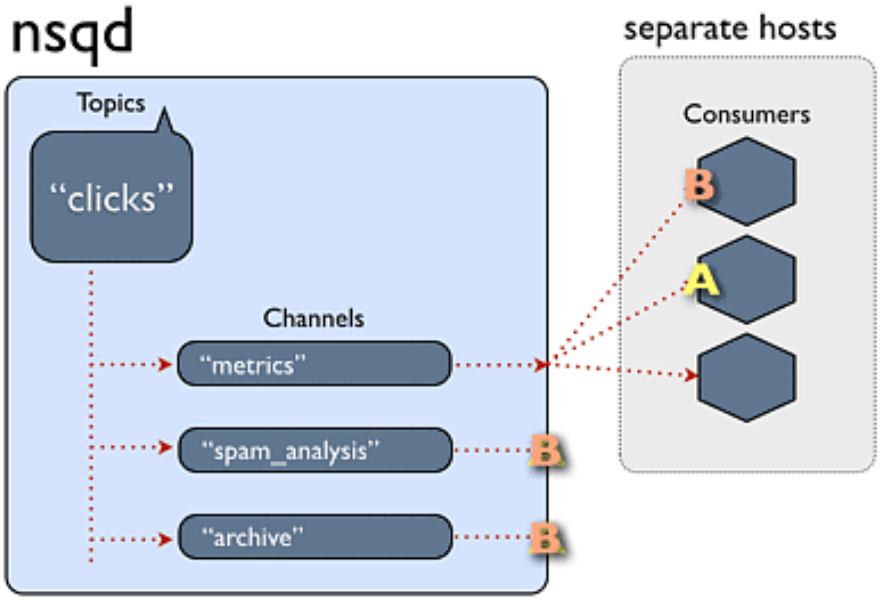

Figure 7.1: NSQ topic/channel message distribution. A copy of message B is received by **every** channel, while only **one** of the *metrics*' consumers receives it. It is worth noting that it was *distributed* (i.e., not sent to the consumer that *just received* message A). Taken from `http://nsq.io/overview/internals.html`.

`NSQ` provides the necessary message passing mechanisms to promote loose-coupling between `unplugg`'s services, particularly between energy consumption data producers and consumers. Consequently:

- `unplugg`'s energy consumption data sources may be added to the unplugg's system without modifying any of the consumers. The data will be stored and broadcasted to its corresponding destination(s) indepen-

dently of the source, as long as it complies with the expected message format. This is what enables NSQ-Feedata2Aqora, for instance, to persist the consumption data on Aqora published to NSQ both by hardware devices and Fetchers/Crawlers, independently of the data's producer.

- Other data consumers may be added to the system to provide different data processing capabilities without causing any code changes to the remaining services; a different channel for the topic would be created so the new consumers could receive a copy of each message.

The low latency provided by `NSQ` is what enables *near* real-time broadcasting of energy consumption data to the web clients, as described in 6.3.9.

7.2.6 On-Off Classifier

Although `On-Off Classifier`'s functionality was not modified, it was re-implemented. Several reasons lead to this re-implementation:

- Since the original code was written during a previous internship, most of its code base was not related to the classification process itself. Multiple classification-unrelated code dependencies were therefore removed during this process.

- The classification algorithm was rewritten on a compact, single file in idiomatic JavaScript for easier reading and future development, since some of the logic was unnecessarily scattered across multiple files.

- A library implementing `Aqora`'s interface was created. This library was also used by NSQ-Feedata2Aqora. This library was named `node-aqora-client`. It is used to provide a simple method interface for `Aqora`'s API.

- The request payload provided by `Qomb` was modified to include the necessary data: the `Bearer Token` provided by the requesting application, necessary to authenticate before `Aqora`; the `Feed ID` and `Stream name`, in order to obtain the energy consumption data; and

the WebHook URL, so the worker is capable to perform an HTTP request on the requesting entity's web server.

The use of the Bearer Token allows the worker to obtain its consumption data using the same authorization method an application would, instead of a root user implementation. This reduces the risk of privilege escalation in the case of a security breach.

7.2.7 Qontrol Manager

For Qontrol Manager's implementation, the REST API was implemented using express.js. The implementation of the service was kept as small as possible due to the low complexity of this problem: one file for the routing code, one for the data model implementation.

7.2.7.1 Database

For Qontrol Manager, both writing and reading are less of a concern when compared to Aqora's use case, since it will be used less frequently (causing a much lower volume of writes); it is used solely by unplugg Web Application when it requests On-Off Classifications. MongoDB was chosen mainly because it operates over dynamic schema documents, which enables future development over the On-Off Classification to be made easily, since the format's flexibility allows the addition or removal of fields without updating every document. The Data Model of a rule of automation (a Qontrol) was implemented using the same library used in Aqora's implementation (section 7.2.2.2), mongoose.js.

7.2.8 MQTT2NSQ-Feedata

MQTT2NSQ-Feedata is the service that bridges MQTT Broker and NSQ; it is a Messaging Bridge according to Enterprise Integration Patterns[52]. It is the component that implements MQTT Bridge as described in section 6.3.7. It is implemented in JavaScript and acts both as a subscriber of MQTT Broker and a publisher of NSQ. Its topic subscription is feedata/#;

in MQTT, this corresponds to a multi-level subscription of feedata, which matches the energy consumption data topics used by every plug. This way, the service forwards every MQTT message with a feedata/{feed_id} topic and publishes it on NSQ under the feedata topic. In order to be authorized to subscribe to every topic, it is implemented as a *superuser*, a user whose subscriptions do not trigger an ACL check.

7.2.9 NSQ-Feedata2Aqora

NSQ-Feedata2Aqora is the component responsible for persisting the energy consumption data in Aqora; it implements the component described as Aqora Storer in section 6.3.8. It subscribes to NSQ messages with feedata as topic, under the channel store. It uses unplugg Aqora's credentials to persist the data in the corresponding Feeds; to do so, it uses the same Aqora library used by On-Off Classifier, node-aqora-client.

NSQ-Feedata2Aqora, as all the other services, was designed to be horizontally scalable. Running multiple instances naturally fits into the message load-balancing provided by the NSQ channel mechanism; this is also enabled by the rejection of duplicates by Aqora (purposely implemented this way to allow this), since NSQ's *At-least-once* delivery guarantees may cause duplication of messages in case of a failed delivery. By running multiple instances, it is possible to establish multiple connections to Aqora in parallel.

7.2.10 NSQ-Feedata2WS

NSQ-Feedata2WS is the Web Socket server responsible for sending Feedata published on NSQ to the corresponding web clients; it implements WebSocket Server as described in the Architecture and Design chapter (section 6.3.9).

It uses the Node.js Socket.IO[9] library. Socket.IO uses a multi-transport abstraction to provide real-time bi-directional event-based communication between server and clients. This transport abstraction (named Engine.IO) transparently chooses the best transport method available to establish the connection. WebSocket is the best method implemented by the library,

[9]Socket.IO: https://socket.io

which means other methods are used only when WebSockets are not available on the client-side. The library's API is event-oriented in order to benefit from the Node.js runtime and supports rooms (groups of clients), namespaces (for multiplexing a single connection), streaming of binary and has both support for volatile messages and used-defined callbacks per-message. [53]

NSQ-Feedata2WS was designed to allow horizontal scalability, meaning multiple instances may be run to serve a greater number of concurrent users. In order to receive the NSQ Feedata messages in a load-balanced way, every instance of the service subscribes to the same NSQ channel for the feedata topic. This has a side-effect: the instance receiving a certain message may not be the instance maintaining the Web Socket connection with the web client. To solve this, a multi-node adapter is used by Socket.IO that requires a Redis instance to enable the multiple instances of the service to communicate with each other, in order to send and broadcast messages to each others' web clients. After configuring the adapter, an instance transparently forwards the message to the correct instance without any further modifications to the code.

7.2.10.1 Feed-Socket Cache

Every time an incoming Feedata message arrives, the service needs to obtain the socket IDs (a Socket.IO internal identifier for a connection) the Feed is associated to, in order to send them the data. In order to cache the (feed_id => [socket_id]) association, a Redis instance is used. This association has an expiration time, in order to guarantee inactive connections are removed from the system, invalidating the WebSocket Session Token. This *temporary data with high-frequency lookups* pattern fits with the Redis internals, as used in Aqora's case.

It is worth noting that this effectively creates a reverse-lookup dictionary necessary for this message routing to be possible, since ws-session-validator is only aware of the inverse relation ((ws_session_token => [feed_id]), ws_session_token corresponding to a unique socket_id).

Chapter 8

Verification and Validation

The following chapter lists the Verification and Validation tests that were executed and their result.

8.1 Verification

8.1.1 Aqora Module Tests

8.1.1.1 Public Endpoints

It should redirect / to the correct version without `Authorization`.
Result: Passed ✓

It should allow access to /v1 without `Authorization`.
Result: Passed ✓

It should give 404 on accesses other than /v1.
Result: Passed ✓

8.1.1.2 Authentication Errors on reserved endpoints

It should throw a 401 when no `Authorization` is provided.
Result: Passed ✓

It should throw a 401 when an invalid `Authorization` is provided.
Result: Passed ✓

It should throw 401 when Basic `Authorization` is provided.
Result: Passed ✓

It should throw 401 when invalid Bearer `Authorization` is provided.
Result: Passed ✓

8.1.1.3 Authentication endpoint

It should throw 401 when not authenticating using Basic `Authorization`.
Result: Passed ✓

It should throw 401 when using an invalid `Authorization` format.
Result: Passed ✓

It should throw 401 when using an invalid API Key/API Secret pair.
Result: Passed ✓

8.1.2 Integration Tests

Integration Test 1
`Qomb` is able to spawn a `On-Off Classifier` per received `On-Off Classification` request.
Result: Passed ✓

Integration Test 2
`On-Off Classifier` is only able to retrieve energy consumption data from Aqora when provided with valid Authentication credentials.
Result: Passed ✓

Integration Test 3
`On-Off Classifier` is able to retrieve energy consumption data from Aqora

in behalf of an authorised owner.
Result: Passed ✓

Integration Test 4
unplugg is capable of requesting `On-Off Classification` requests to `Qomb` and receive its classification result on `Qontrol Manager`.
Result: Passed ✓

Integration Test 5
`Qontrol Manager` is capable of validating the result by requesting `Qontrol Jobs Manager` and analysing its return status code, storing the result if valid.
Result: Passed ✓

Integration Test 6
`Feedata` published through `MQTT` is available to any NSQ client.
Result: Passed ✓

Integration Test 7
`WebSocket Server` is capable of authenticating and authorising users by requesting `WS Sessions Validator`, either accepting or dropping the connection depending on the status code.
Result: Passed ✓

Integration Test 8
`WebSocket Server` is capable of publishing `Feedata` to `Web Clients`.
Result: Passed ✓

8.1.3 System Tests

System Test 1
It is possible to send datapoints to Aqora, request Qomb to analyse those datapoints and receive the classification result using a WebHook.
Result: Passed ✓

System Test 2
It is possible to send datapoints through MQTT and retrieve them through Aqora.
Result: Passed ✓

System Test 3
It is possible to send datapoints through MQTT and receive them through WebSockets on a browser, in real-time.
Result: Passed ✓

8.2 Validation

These tests are used to validate the achievement of both business goals. **Acceptance Test 1** validates the Enterprise Customer's business goal achievement, while **Acceptance Test 2** validates the unplugg's energy management devices business goal.

Acceptance Test 3 validates a use case not required by the business goals; although specific on the data source (unplugg energy management devices), it should be noted that **Integration Test 6** assures that it applies to any energy consumption data source, even Data Fetchers. However, the test was not generically formulated since Data Fetchers were not adapted to the system during the internship.

Acceptance Test 1
As an Enterprise Customer,
I **want** to be able to send energy consumption data
so that I can receive rules of automation.
Result: Passed ✓

Acceptance Test 2
As an unplugg's End User,
I **want** my unplugg energy management devices to be able to send energy consumption data

so that it is available in real-time on a browser.
Result: Passed ✓

Acceptance Test 3
As an unplugg's End User,
I **want** my unplugg energy management devices' energy consumption data to be stored
so that it can be later retrieved and analysed.
Result: Passed ✓

Chapter 9

Conclusions

This internship aimed to study and implement a system capable of exposing the functionality of an existing classification system to third-parties in an isolated way, and capable of receiving energy consumption data sent by the company's own energy management devices. To achieve it, a system was developed that, using a canonical data model, abstracts the different stakeholders data, allowing the existing unplugg Web Application and third-parties to store energy consumption data through a simple REST API; this allows the data's later retrieval and classification into rules of automation. In addition, an Internet of Things protocol was chosen to enable energy management devices to reliably send data to the system even on degraded network conditions, with low energy consumption.

On a first phase, after the Requirement Analysis of the project was completed, the current system limitations were analysed. A study of the state of the art regarding Modern Web Application Architectures, Service-Oriented Architectures and Internet of Things Communication Protocols was performed, in order to contextualise the problem domain. A solution was planned and designed; a prototype regarding a crucial component of its design was implemented to validate its use.

On a second phase, the remaining components were implemented while continuously validating their contribution to the system through the use of user stories.

The author chose to redesign and reimplement a part of the prototype in order to simplify its use by other projects. This led to a delay of the remaining implementation, since the plan had no error margin. While SOA provides benefits to the overall system, it is also complex because of the degree of choices that must be made; this complexity also delayed the system validation phase and the writing of the thesis, due to the justification of the reasoning behind the architectural and technological choices.

The resulting system addresses both business goals. In addition, it is capable of providing the energy consumption data sent by the company's energy management devices in real-time, and lays the foundations for an easier extension of the system by implementing it in a loosely-coupled fashion. No implemented component is a Single Point of Failure of the system, and every implemented component is horizontally scalable, to allow the system to address any number of requests when needed. Energy data producers and consumers may be added to the system without changing any other service's code by making use of the Publish-Subscribe infrastructure.

The additional prototype reimplementation effort enabled an easier adoption of the time-series storage platform by multiple company side-projects, currently using it. The genericness of the classification platform allows its easy extension to support different types of classification over time-series data, in addition to the `On-Off` classification.

Its major contribution is its service-oriented architectural vision that, in conjunction with good software development practices, architecture and design principles enables future service and system-wide extensions to have lower implementation costs. It lays the foundations to future development of different classification mechanisms and real-time data analysis. It shifts the paradigm of the system's pull-based interactions to be push-based; it became an event-driven architecture. The author believes this paradigm shift is important to enable future business opportunities such as Complex Event Processing over energy consumption.

Every identified User Story and Test is addressed by the implemented system. After performing the integration changes, it may be integrated with the current unplugg system.

9.1 Future Work

While the system was designed to enable future extensibility, there's room for improvement on service and system-wide aspects.

- The use of a Time-Series Database as storage database for Aqora, such as OpenTSDB, should be studied in terms of read and writing performance. Besides the performance benefits, a complete time-series database solution such as OpenTSDB offers highly optimised aggregation operations, optimised storage space and tagging mechanisms; these features, particularly tagging, enable complex use cases.

- The use of alternative Internet of Things Communication protocols and protocol stacks should be studied, particularly the recently emerging Thread protocol. Because of the system's loosely-coupled architecture, it will be possible to add support for additional Internet of Things Communication protocols without changing the remaining system.

- The use of alternative binary formats as message and API calls' payloads such as MessagePack and Protocol Buffers should be analysed in terms of performance (comparing to `JSON`. In a more advanced study, the use of binary communication protocols such as Apache Thrift should also be studied in terms of serialisation performance and bandwidth usage.

Appendices

Appendix A

2nd Semester Gantt Planning

	Nome	Duração	Início	Fim	Predecessores
1	Implement remaining unplugg-platform API methods	7d	08/02/2014	15/02/2014	
2	Integrating the unplugg-platform with the Classification System	14d	16/02/2014	02/03/2014	1
3	Implementing the WebHook publish mechanism	7d	03/03/2014	10/03/2014	2
4	Implementing Rules Endpoint	7d	11/03/2014	18/03/2014	3
5	Implementation of the Publish-Subscribe System	7d	18/03/2014	25/03/2014	
6	Setup of a MQTT Broker	7d	25/03/2014	01/04/2014	
7	Integrating MQTT Broker Bridge	7d	02/04/2014	09/04/2014	5,6
8	Setting up WebSocket Server	7d	10/04/2014	17/04/2014	5
9	Implement a data source simulator	14d	17/04/2014	01/05/2014	
10	Verification of the system	18d	01/05/2014	19/05/2014	
11	Validation of the system using simulated data	21d	18/05/2014	08/06/2014	
12	Report writing	28d	09/06/2014	07/07/2014	
13	Presentation preparation	12d	08/07/2014	20/07/2014	

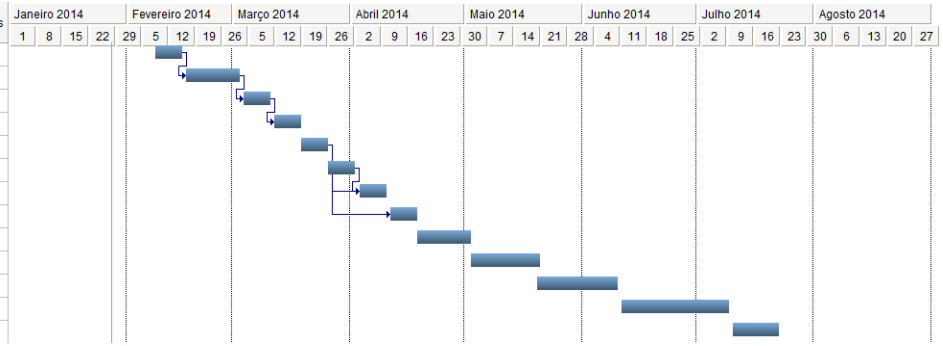

Figure A.1: Gantt of the 2nd Semester Planning

Appendix B

Aqora Auth Spec

B.1 Headers

Every request must have the following headers set:

Content-type: `application/json`

Authorization: `Bearer <base64(bearer_token)>`

The obtention of a `bearer_token` is explained in `POST /v1/auth` method.
Note: Methods may override these headers when specifically specified.

B.2 Status codes

200 (OK) : Operation was successful.

201 (Created) : The resource(s) was/were created successfully.

400 (Bad Request) : Invalid JSON payload, missing/invalid parameters.

401 (Unauthorized) : Invalid, incorrect or expired Authorization.

404 (Not found) : The specified resource does not exist.

500 (Internal Server Error) : An unexpected condition has occurred.

501 (Not Implemented) : A future method, expected to be implemented.

B.3 Authentication/Authorization

B.3.1 Create a session

`POST /v1/auth` (**requires** `Authorization: Basic`)

This method is used to obtain a **Bearer token**: this token is used by the platform to authenticate and authorize an application when requesting *authorized* methods.

It has a method-specific header: **Authorization**: `Basic <base64(api_key:api_secret)>`

The `Authorization` header must be set to `Basic` and should be followed by the `Base64` of the concatenation of the API key and API secret, separated by `:`.

The Authentication mechanism implements Twitter's Application-only authentication, itself based on Client Credentials Grant flow of the OAuth 2 specification[21]. The only differences lie on the expected request and response payloads (both in JSON).

The expected success response is as follows:

```
{
  "status": 200,
  "auth_type": "Bearer",
  "token": <bearer_token>
}
```

This `bearer_token` should be provided on following API calls that require **Authorization**, using the format **Bearer <base64(bearer_token)>**.

B.3.2 Validate a session

`GET /v1/auth` (**requires** `Authorization: Bearer`)

This method is used to verify the success of the obtention of a **Bearer token**. It is *nullipotent*: it has no side-effects.

The expected success response is as follows:

```
{
  "status": 200
}
```

Bibliography

[1] João Barbosa. Automating Energy with the Internet of Things, 2013.

[2] Thomas Erl. *SOA: Principles of Service Design*. Prentice Hall, 2007.

[3] Microsoft. MSDN: SOA in the Real World, Chapter 1 - Service Oriented Architecture (URL: http://msdn.microsoft.com/en-us/library/bb833022.aspx), 2014.

[4] Arnon Rotem-Gal-Oz. InfoQ: SOA != Web Services (URL: http://www.infoq.com/news/2007/07/soa-ws-relation), 2007.

[5] Jason Bloomberg. ZapThink: Divorcing SOA and Web Services (URL: http://www.zapthink.com/2007/06/20/divorcing-soa-and-web-services/), 2007.

[6] Martin Fowler and James Lewis. Microservices (URL: http://martinfowler.com/articles/microservices.html), 2014.

[7] Chris Richardson. Microservices: Decomposing Applications for Deployability and Scalability (URL: http://www.infoq.com/articles/microservices-intro), 2014.

[8] James Hughes. Micro Service Architecture (URL: http://yobriefca.se/blog/2013/04/28/micro-service-architecture/), 2013.

[9] Adam Wiggins. The Twelve-Factor App (URL: http://12factor.net/), 2012.

[10] Heroku. Architecting Applications for Heroku (URL: https://devcenter.heroku.com/articles/architecting-apps), 2014.

[11] OASIS MQTT Technical Committee. 60-day Public Review for MQTT Version 3.1.1 COS01 - ends September 4th (URL: https://www.oasis-open.org/news/announcements/60-day-public-review-for-mqtt-version-3-1-1-cos01-ends-september-4th), 2014.

[12] Richard MacManus. MQTT Poised For Big Growth - an RSS For Internet of Things? (URL: http://readwrite.com/2009/07/22/mqtt_-poised_for_big_growth), 2009.

[13] Dave Locke. MQ Telemetry Transport (MQTT) V3.1 Protocol Specification (URL: http://www.ibm.com/developerworks/library/ws-mqtt/), 2010.

[14] MQTT.org. MQTT.org: Frequently Asked Questions (URL: http://mqtt.org/faq), 2013.

[15] Ian Craggs. MQTT security: Who are you? Can you prove it? What can you do? (URL: https://www.ibm.com/developerworks/community/blogs/c565c720-fe84-4f63-873f-607d87787327/entry/mqtt_security), 2013.

[16] Stephen Nicholas. Power Profiling: HTTPS Long Polling vs. MQTT with SSL, on Android (URL: http://stephendnicholas.com/archives/1217), 2012.

[17] Jeremy Cloud. Decomposing Twitter: Adventures in Service-Oriented Architecture (URL: http://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture), 2013.

[18] Neil Hunt. Netflix Development Patterns for Scale, Performance & Availability (URL: http://www.slideshare.net/AmazonWebServices/dmg206), 2013.

[19] Niklas Gustavsson. Spotify services (URL: http://www.slideshare.net/protocol7/spotify-services-scc-2013), 2013.

[20] Mike Cohn. Advantages of the "As a user, I want" user story template. (URL: http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template), 2008.

[21] Twitter. Application-only authentication (URL: https://dev.twitter.com/docs/auth/application-only-auth), 2013.

[22] Tom Poppendieck Mary Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, Boston, Mass, 2003.

[23] Sandvine. Sandvine Global Internet Phenomena Report: 1st Half of 2013 (URL: https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/sandvine-global-internet-phenomena-report-1h-2013.pdf), 2013.

[24] Sandvine. Sandvine Global Internet Phenomena Report: 1st Half of 2014 (URL: https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf), 2014.

[25] Michael Safyan. Google Inside Search - Bringing more secure search around the globe (URL: http://insidesearch.blogspot.com/2012/03/bringing-more-secure-search-around.html), 2012.

[26] Danny Sullivan. Post-PRISM, Google Confirms Quietly Moving To Make All Searches Secure, Except For Ad Clicks (URL: http://searchengineland.com/post-prism-google-secure-searches-172487), 2013.

[27] Alex Rice. Facebook - A Continued Commitment to Security (URL: https://www.facebook.com/notes/facebook/a-continued-commitment-to-security/486790652130), 2011.

[28] Scott Renfro. Facebook - Secure browsing by default (URL: https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920), 2013.

[29] Randall Degges. Service Oriented Side Effects (URL: http://www.rdegges.com/service-oriented-side-effects/), 2012.

[30] npm Registry. npm Registry: Most Depended-upon Packages (URL: https://www.npmjs.org/browse/depended), 2014.

[31] ExpressJS. ExpressJS Applications (URL: http://expressjs.com/applications.html), 2014.

[32] Shravya Garlapati. Blazing fast node.js: 10 performance tips from LinkedIn Mobile (URL: http://engineering.linkedin.com/nodejs/blazing-fast-nodejs-10-performance-tips-linkedin-mobile), 2011.

[33] DB-Engines. DB-Engines Ranking of Document Stores (URL: http://db-engines.com/en/ranking/document+store), 2014.

[34] Yuri Finkelstein. ZZZ (URL: http://www.slideshare.net/mongodb/mongodb-at-ebay), 2012.

[35] Cooper Bethea. Foursquare: Experiences Deploying MongoDB on AWS (URL: https://www.mongodb.com/presentations/experiences-deploying-mongodb-aws), 2011.

[36] Richard Hirsch. The Quest to Understand the Use of MongoDB in the SAP PaaS (URL: http://www.slideshare.net/mongodb/mongodb-at-ebay), 2011.

[37] MongoDB. MongoDB Performance At Scale (URL: http://www.mongodb.com/scale), 2014.

[38] Zach Cross. IBM developerWorks: Developing mobile apps with Node.js and MongoDB (URL: http://www.ibm.com/developerworks/library/mo-nodejs-1/index.html), 2013.

[39] DB-Engines. DB-Engines Ranking of Key-value Stores (URL: http://db-engines.com/en/ranking/key-value+store), 2014.

[40] Redis. Redis: Who's using Redis? (URL: http://redis.io/topics/whos-using-redis), 2014.

[41] Redis. Redis: An introduction to Redis data types and abstractions (URL: http://redis.io/topics/data-types-intro), 2014.

[42] Redis. Redis: How fast is Redis? (URL: http://redis.io/topics/benchmarks), 2014.

[43] Redis. Redis Command Reference (URL: http://redis.io/commands), 2014.

[44] Matthias Gelbmann. W3Techs: Nginx just became the most used web server among the top 1000 websites (URL: http://w3techs.com/blog/entry/nginx_just_became_the_most_used_-web_server_among_the_top_1000_websites), 2013.

[45] W3Techs. W3Techs: Usage of web servers broken down by ranking (URL: http://w3techs.com/technologies/cross/web_server/ranking), 2014.

[46] Netflix. Netflix: Open Source Software (URL: https://www.netflix.com/openconnect/software), 2014.

[47] Ryan Lane. Wikimedia: SSL Termination (URL: https://wikitech.wikimedia.org/wiki/Https#SSL_termination), 2011.

[48] Andrew Alexeev. The Architecture of Open Source Applications: nginx (URL: http://www.aosabook.org/en/nginx.html), 2012.

[49] David Mytton. MongoDB Benchmarks (URL: https://blog.serverdensity.com/mongodb-benchmarks), 2013.

[50] Vitaly Puzrin. Google Groups/Mongoose Node.JS ODM/Performance of `find()` with moderately large result sets (URL: https://groups.google.com/d/msg/mongoose-orm/u2_DzDydcnA/Lp-Wq14ShZ4J), 2012.

[51] NSQ. NSQ: Features and Guarantees (URL: http://nsq.io/overview/features_and_guarantees.html), 2014.

[52] Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, 2004.

[53] Socket.IO. Socket.IO Documentation (URL: http://socket.io/docs/), 2014.