IoT-Framework

Product Report

Project CS

(1DT054) - Autumn 2013

Uppsala University

Arias Fernández, José

Bahers, Quentin

Blázquez Rodríguez, Alberto

Blomberg, Mårten

Carenvall, Carl

Ionescu, Kristian

Kalra, Sukhpreet Singh

Koutsoumpakis, Iakovos

Koutsoumpakis, Georgios

Li, Hao

Mattsson, Tommy

Moregård Haubenwaller, Andreas

Steinrud, Anders

Sävström, Tomas

Tholsgård, Gabriel

February 20, 2014

Abstract

With the advent of low cost wireless connectivity, almost everything is getting connected to the internet, from handhelds to coffee machines, also known as Internet of Things (IoT). This document describes the methodology and development process of this project based on IoT. The project described in this paper is to create a framework for managing services of IoT. The project was developed by fifteen Computer Science master students of Uppsala University during autumn of 2013. The goal of the project was to develop an engine which can gather sensor data from different devices and provide the ability to interact with it.

Contents

Glossary

AMQP Advanced Messaging Queueing Protocol.

AJAX Asynchronous JavaScript and XML.

API Application Programming Interface.

ARIMA Autoregressive Integrated Moving Average.

CSS Cascading Style Sheets.

CRUD Abreviation of the operations *Create, Read, Update and Delete*.

DOM Document Object Model.

ERB Embedded Ruby, the language used for rendering views on the server side with Rails.

HAML HTML Abstracted Markup Language, another language used for rendering views.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS HypetText Transfer Protocol Secure.

IoT Internet of Things.

JSON JavaScript Object Notation, a de-facto standard format used for sending data over the Internet.

MVC Model-View-Controller, an architectural design pattern.

REST Representational Stateful Transfer.

SICS Swedish Institute of Computer Science.

SMS Short Message Service.

SQL Structured Query Language, used by databases to manipulate their data or structure.

SVG Scalable Vector Graphics.

QoI Quality of Information.

TDD Test Driven Development.

UUID Universally Unique Identifier.

URI Uniform Resource Identifier.

URL Uniform Resource Location.

XML Extensible Markup Language.

Chapter 1

Introduction

Nowadays, with the improvements in technology, there are billions of devices worldwide that produce data. Examples of such are temperature sensors, humidity sensors or even the luminosity sensor in a mobile phone. Due to the vast amount of sensors that exist, the amount of data that get produced every second is mind-boggling and it would seem difficult to organize it all in a good and easy way.

There have been many attempts to create systems that allow users to register their sensors and view the produced data, such as SICSth Sense[1], Xively[8] and Thingspeak[7], all of them focus on different features.

This is the reason behind the creation of the IoT-Framework, in an effort to easily view, handle and interact with data streams. Within the system, users can register their sensors, create streams of data (e.g. the temperature in Uppsala), and view them on a graph. In addition, the system supports searching capabilities, helping the user with a full-text query language and phrase suggestions, allowing a user to find such streams using filters (based on the location of the devices or the meta-information provided by the user through tags), sort them using different criteria and rank the found results.

Moreover, users may be interested in the combination of several streams in order to know certain values such as the average, sum, minimum and maximum, instead of the measurements taken in a specific location. Thus, IoT-Framework also allows for this kind of aggregations, creating a *virtual stream*.

Finally, the IoT-Framework also supports the creation of triggers attached to streams and virtual streams. A trigger is a mechanism that notifies the user when a specific criteria is met such as reaching a certain temperature. Currently, the available functions are *lesser*

than, *span* and *greater than*

IoT-Framework is available as an opensource project on GitHub[2] under the Apache 2.0 license.

Chapter 2

Background

2.1 Internet of Things

The Internet of Things (IoT)[3] is a concept that has become more popular lately, the main principle of IoT is to connect hardware to the internet which then can be interacted with or without physical contact. The device (or resource) is uniquely identifiable and will provide data or functionality. The physical limitations of these devices lead to different ways of communicating. A device that is powered all the time could be polled for the latest value or be allowed to push data to the system, while a device that is battery-powered on the other hand might need to turn itself off to conserve energy and would thereby not be available for polling.

The concept of IoT leads to big ideas, where one example would be the smart city[4]. This is a city where sensors are placed all around it and they would monitor, for example, air pollution, amount of traffic or how full a garbage can is. The information could then be used to make smart decisions, for example in the case of traffic monitors it could redirect traffic to lower the risk of traffic jams or with the garbage can sensor one could make garbage collection more efficient by only collecting where it is needed. The ideas from the smart city is starting to be tried out in some cities around the world like Singapore[5].

All of this hardware will be too much to be monitored by humans, so systems need to be built to handle this information and then either provide a good overview of the data, suggest what to do (so a human can easily take the needed decisions) or automatically do it. Such a system would also need to have meta data regarding the devices to be able to decide how useful the data given from the device is.

Ultimately what is envisioned here would be a system that could handle 50 billion devices,

gather data by polling or pushing, have a good visual overview of the data but also be able to create virtual streams(2.2.4), have triggers(2.2.5) on these streams and be able to analyze the data stream by making forecasts. All these things were to be done in a user friendly way where the users could add their own streams to the system.

2.2 Entities

2.2.1 Resource

A *resource* is something that can produce data points, like for instance a smart phone that contains both temperature and light sensors. This resource could then have two streams, one stream for the temperature sensor and one stream for the light sensor.

2.2.2 Data Point

A *data point* is a value generated from a resource, it is associated with a stream and a time. An example of a data point is a temperature value from a smart phone that was recorded at a certain timestamp.

2.2.3 Stream

A *stream* is a continuous flow of data points with metadata about its origin, type and other properties. The source of the stream, which could be a physical device or a website processing data from one or more physical devices, is the element that provides the metadata. The actual data is saved as individual data points where each stream would have a set of them associated with it which would be the history of the stream.

2.2.4 Virtual Stream

A *virtual stream* is a stream that generate data by applying a function on the input data from one or more streams. An example of this could be the average of a set of streams or the difference between the two latest values in the input stream.

2.2.5 Trigger

A *trigger* is a supervisor that holds a set of streams and will run a function when new data is input into one of the streams. If the trigger criteria is met, it will execute some action. An example is a trigger that monitors a temperature stream with the function ”if the value given is less then zero” and the mechanism ”send an SMS to a phone”. As

soon as the temperature drops below zero, the trigger will be launched and the users smartphone will receive a SMS.

Chapter 3

Product Description

3.1 Goals and Scope

The main goal of this project was to establish key functionality that is essential towards Ericsson's vision of a networked society[11]. More specifically the creation of a system where it would be possible to add and search for sensors, visualize, aggregate and make data predictions on and the project was to create as much features as possible for such an application.

At the beginning of the project, scalability and load distribution were the key points and influenced most of the technical decisions taken. However, the resulting system has been designed in such a way so that it can be further extended and improved with additional features. In order to keep the development process more focused, it was decided that other key points, such as security, would not be prioritized.

3.2 RESTful Capabilities

The system architecture allows the four *CRUD* (Create, Read, Update and Delete) functions of persistent storage, following the *REST*(Representational state transfer)[12] principles. These operations can be carried out through the API affecting different kind of resources such as users, streams, virtual streams and triggers.

Moreover, the API supports CRUD operations of resources. This feature is not provided by the front end in order to simplify the interface, expose less windows that the user needs to learn, and enhance the user experience.

Although the API was the only required element in the system that should be designed

in a RESTful manner, the front end environment was also developed following the REST guidelines since the Rails philosophy strongly enforces developers to follow this approach. Thus, both front end and back end were implemented consistently and following the expected standards.

3.3 Streams

The product is built to constantly retrieve information from multiple devices, the core idea of the project is streams of data points which can be displayed, measured and even aggregated in order to compose richer, more complex streams.

Every stream belongs to a unique user registered in the system and the most basic attributes that defines it are the following:

name The name of the stream, which is required.

description A basic explanation about what is being measured.

type The type of the stream, e.g. temperature, humidity, pollution etc.

privacy If it is public for everyone or only available to the owner. In the future the system could support more levels of privacy, but this idea goes beyond the current scope of the project.

location The coordinates of the device.

tags Keywords used for additional metadata for queries.

unit The unit of the data points, e.g. Celsius, Fahrenheit etc.

min_val The lowest value that can be reliably measured.

max_val The highest value that can be reliably measured.

accuracy The positive/negative margin considered in the measurement.

uri The web address of the device.

polling frequency The frequency used for fetching data points.

parser The path to the values in the document which contains the data points.

data type The format of the document, e.g. JSON in the current system.

3.3.1 Smart Stream Creation

A stream could be easily created through a 3 steps wizard. However, the user may be interested in the creation of a set of streams that share the same resource. For example, a user wants to create streams from his smartphone such as the temperature, accelerometer, gyroscope and compass.

To support this, a fast mechanism was developed that makes it possible to create multiple streams simultaneously. This way, the user could type the name of his device helped by autocompletion. When the system shows the results found in the catalog, the user can select the correct resource and receive the list of streams attached to it. Then, the user is able to select the desired templates and specify the resource's UUID[10] (a unique identifier of the device).

Finally, he/she can finish the process creating the set of desired streams.

3.3.2 Graphs

Each stream page shows a 2D graph that displays the data points fetched from the associated resource. If the user wants to see also the predictions of the system, it is possible to render them along with two confidence intervals of 80% and 95.%

One of the main features of the search window is the capability of selecting several streams and show them in a multiline graph in order to compare different streams of the same unit.

These graphs are capable of painting the measured values using a linear interpolation algorithm. The reason for choosing the linear approach is that the other algorithms did not show an intelligible painting.

3.3.3 Predictions

Predictions, or more accurately *time series forecasting*, in the system are handled using the ARIMA (Autoregressive Integrated Moving Average)[44] method in R, described in section 4.3.4. The purpose is to get more detailed information about the future of a datastream than a human could. For practical reasons, the API limits the number of inputs and outputs for predictions. 500 datapoints should still be large enough to pick up on patterns, if there are any.

Though the homepage allows the user to select specific input and output sizes, the API

accepts any integers within the allowed interval. The homepage could fairly easily be altered to allow this as well, but that would add keyboard interactions on sections of the website that is otherwise only interacted with by using a mouse.

The predictions sometimes appear to be of poor quality, and the rule of bad data in means bad data out. A result with just a straight line and a large confidence interval means that the system could not find a pattern even while a human might think there is one. In such cases common sense is recommended in combination with awareness of the cognitive bias known as *clustering illusion* (the tendency to see patterns where actually none exist)[59].

The predictions are also heavily dependent on the size of the input. Since the prediction system only knows about the points it is given to do a prediction on, having a small number of points means the system ignores most of the history. Again, common sense is recommended. While it may be desirable to disregard many historical values (for instance if the values before a certain time are not trustworthy), the predictions can't pick up on patterns among values it is not given.

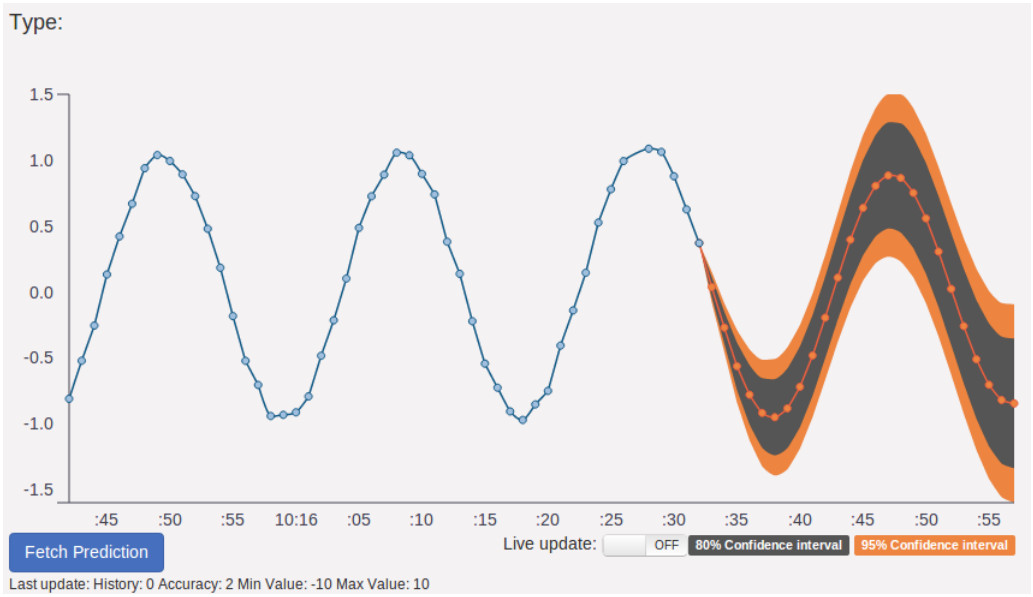


Figure 3.1: Example of a prediction

3.3.4 Subscription

The idea behind the subscription functionality is that the end user, after finding out an interesting stream, may want to save its link to be able to come back to it later. The use case is essentially the same as when the user want to bookmark a web page in a browser.

When the user visit a stream, a "follow" button is displayed on the upper-left corner of the page. By clicking on it, the user will subscribe to that stream. The state of the button will then change and it will display the text "unfollow" instead. If the user click on that button again, the user will then unsubscribe to the stream.

To see all the streams the user have subscribed to, the user can click on the "Subscriptions" tab, where a list of all the links to streams the user is following will be displayed.

3.3.5 Live updates

Live updates allows the user to get real-time information about a stream or a virtual stream. When a user views a stream, they have the option to enable live updates. When a stream produces a data point, it is published to an exchange with a given namespace. An exchange receives data points from a stream and pushes it to a queue associated with that namespace. The exchange has a unique namespace to differentiate it from exchanges related to other streams.

The queue acts as a buffer for new data points, a new queue is created by the exchange for each namespace. The consumer connects to the exchange with the given namespace and listens for incoming data points. The consumer will read the data point, plot the data point on the stream graph. The data point is then removed from the namespace queue. When a user enables live updates, they are acting as a consumer.

The namespace for streams and virtual streams has the following pattern "type.id". The type specifies if the stream is a virtual stream or ordinary stream, the id is the unique identifier of a stream. When a user disables live updates, the queue associated with that user will be deleted by the exchange.

3.3.6 Stream Location

Each stream stores a geographical location. ElasticSearch allows for the location to be stored either as logitude and latitude, or as a geohash. Using this built in functionality

allows the system to make queries based on location and distance. The service has support for this both in the API and on the website in the form of a search filter. The website uses google maps to display the locations as well as extract an area to filter a search, should the user want to.

3.3.7 Rank

All the streams can be ranked by the users. The ranking goes from 0% to 100%, calculating the average of all the rankings performed by the users. Each user can vote once per stream with 1-5 stars, representing 20% per star. With this ranking, users can get information about how dependable a stream is based on the opinions of other users.

3.4 Virtual Streams

A virtual stream is an aggregation of one or more non-virtual streams with a function applied. The functions are currently limited to average (the average value of the parent stream at each point in time), max (highest value among the parent streams at each point in time), minimum (minimum value among the parent streams at each point in time) and sum. The virtual streams are updated whenever any of its parents are updated, making sure the latest value is always the most up to date. Also, diff is a function supported for creating a virtual stream consisting of a single stream and shows how the input stream data alters over time (every input value is compared to the previous one).

With the virtual stream feature the user can get more reliable values by taking the average of multiple sources, or track, for example, the lowest temperature in a whole region.

While it would be possible to add the feature to use virtual streams to form new virtual streams, this introduces some technical difficulties, such as what happens if the user alter a virtual stream A and add virtual stream B that gets its data from A.

3.5 Triggers

Triggers are specified and created by users and can only be set on streams or virtual streams owned by the user. A trigger allows a user to specify under which condition and where an alert should be sent. There are three types of conditions for a trigger that can be set:

- Less than - The trigger will send an alert when the stream/virtual stream have a value smaller than that specified in the trigger.

- Greater than - The trigger will send an alert when the stream/virtual stream have a value greater than that specified in the trigger.
- Range - The trigger will send an alert when the stream/virtual stream have a value within the specified range in the trigger.

A trigger have two possibilities of where it should send an alert:

- User - This will send an alert to the user's alert log on the website and the user can see it and get notified there.
- URL - This will send an alert message to the specified URL as well as to the users's alert log on the website.

3.6 Search

In the main webpage a small search box is visible, allowing the user to do searches. The search results are divided by streams, virtual streams and users. The search uses the syntax defined by *elasticSearch* [13, 14], which provides full text search.

Compared to most of the features of the system, this functionality can be used by both users and guests. So, the user is not required to log into the system to use it.

If the search is done through the API, the following parameters can be used:

Size It allows the user to configure the maximum amount of hits to be returned.

From It defines the offset from the first result the user want to fetch.

3.6.1 Sort and filter

When a user is searching for streams or virtual streams, they have the option to sort and filter the search results. They can be sorted by name, so that the search results appear in alphabetical order, or by user ranking where search results would then be sorted from highest ranked stream to the lowest ranked stream.

Users can also filter search results by the unit, by tags associated to the streams and also by the active state of a stream. Finally, search results can also be filtered by location, where users can interactively set a position and a radius area on the world map to filter out streams that are outside the given area.

3.6.2 Best Rated Streams

In the main webpage, below the search bar the user can see a list of the top rated streams, scrolling each 5 seconds.
With this feature, users and guests can obtain a general idea of the kind of information that this systems gives and can also see in a fast and easy way the most important data for *IoT-Framework* users.

3.6.3 Search autocompletion

When a user performs a new search, each time the user types a letter in the search bar, a list of suggestions are proposed to the user. This suggestions are based on previous searche queries. When the user perform a query with a suggestion, that suggestion adds one point to his own score. If it is a new search, it is added to the list of suggestions with initial score of *1*. Searched keywords with higher scores show up higher in the search box

3.7 User accounts

The user model is defined by the following attributes: username, first name, last name, description, email address and password. Three more columns are automatically added by the Ruby on Rails framework when creating a new model: id, created_at and updated_at. The id variable is an integer used to uniquely identify every user. The uscreated_at variables and updated_at variables are timestamps that store the date when a given user is created and updated.

The username, email address and password attributes are mandatory whereas the first name, last name and description fields are optionals.

Users can create a new account, sign in and sign out from the website. When creating a new account, some basic field validations are performed. As some attributes, such as the username or the email address, should be non-empty, the system checks for their presence before saving them into the database.

Moreover, some attributes are displayed on the application and since they should not take too much space, a check is performed that limits their length to an arbitrary limit, set to 50 characters.

The system also has to make sure that the email address provided by the user is valid. To do so, the system use a regular expression that follows the format "xxx@yyy.zzz". Even though email addresses should be case sensitive according to the standard [15], real life applications do not usually enforce it. It was decided that it should not be enforced in the system either, so before saving email addresses into the database, the system converts them all into lower case.

In order to improve the user experience, when users make submissions that violate some validations, the system resends the form and display messages explaining what went wrong on top of it, e.g. "The email address provided is not valid".

To increase security, user's passwords are encrypted (using the bcrypt function) before being stored, so that if the database is compromised, passwords are not made public and cannot be used. The authentication process works as follow: when signing in, the user submits its password. It is then encrypted, using the same function as the one used when he/she first signed up. If the two encrypted passwords are the same, it means that the raw passwords are the same, since the encryption function is injective. The user is then successfully authenticated.

3.8 Sessions

Once signed in on a website, the user usually want to stay logged in, even if the user close his/her browser and reopen it after a while. The application should be able to remember the user virtually forever, unless the user explicitly clicks on the sign out button. This is made possible using sessions[16]. A session is a semi-permanent connection between two devices, in this case between the device trying to access the website and the server hosting the IoT-Framework.

Sessions are typically handled by cookies[17]. A cookie is a small piece of data sent from a website and stored in a user's web browser. Cookies are used so that it reduces the number of requests to the server. Every time a user signs in, the system stores a token, which is a long enough random string (so that the probability of having two identical tokens is negligible), on the browser and its encrypted version in the server database. Later, if the user visits the website again, the application encrypts the cookie sent by the user, and tries to see if it matches one stored in its database.

3.9 Authorization

The good thing about having an authorization system is that the system can protect pages from improper access. For instance, a user may not want their stream displaying the temperature at home to be accessible to anyon else. By requiring the user to sign in before being able to perform certain actions, the system can check if the user has the right to perform that action. If the user is not allowed to perform an action, the user is redirected to the sign in webpage.

Chapter 4

System Description

4.1 Architecture Overview

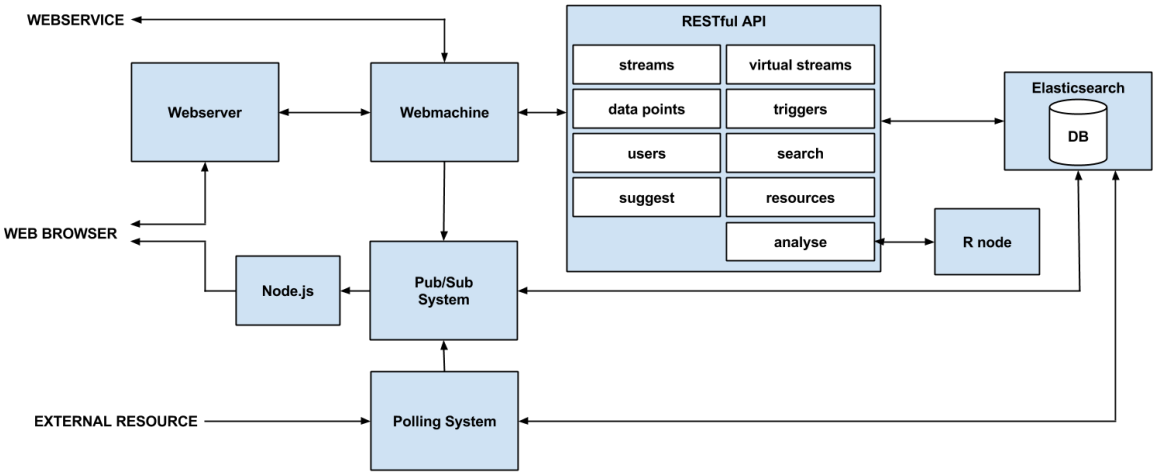


Figure 4.1: Architectural overview.

Fig 4.1 shows a simplified overview of the architecture and how individual modules and systems are connected to create IoT-Framework[2].
The module *Webserver* in fig 4.1 is the front end of the system. It contains a *ruby on rails* server and several other technologies to provide a good user experience, read more in 4.2.
Webmachine module in fig 4.1 is the main interface to the back end of the system. This module contain Webmachine[27] and is where other webservices can connect and use IoT-Framework[2]. This module gets http requests in a RESTful manner and dispatch them to the corresponding modules and sends back information or error codes, read more in 4.3.2.
The *RESTful API* in fig 4.1 module is a container for all the modules that *Webmachine*

dispatch to. The *Webmachine* module only communicate with one of these per http request, read more in 4.3.1.
In fig 4.1 the module *Elasticsearch* contains Elasticsearch[13] which is a document data storage with a high emphasis on search integrated inside. It allows the system to have a close relationship between the actual data and the search functionality, read more in 4.3.3.
The module *R node* in fig 4.1 allows us to calculate predictions and many other statistical computations. The module uses R[42] to do the computations and rErlang[43] to be able to access the R[42] library in Erlang[55], read more in 4.3.4.
Publish/Subscribe System module in fig 4.1 is where data point dependencies are handled, for example virtual streams waiting for data points from its parent streams, and where live updates to the browser is made possible via the *Node.js* module. The *Publish/Subscribe System* module contain several systems RabbitMQ[22] being the most important, read more about the module in 4.3.5.
The *Node.js* module contain Node.js[23] in combination with Rabbit.js[25] and Socket.IO[24] to provide live updates to web browsers supporting web-sockets, read more about Node.js[23] in 4.3.5 and read about live updates in 3.3.5.
The last module *Polling System* allows IoT-Framework[2] to fetch data from external resources at a given time interval, read more about it in 4.3.6.

4.2 Front End

4.2.1 Ruby on Rails

The main technology used in the front end is the well-known Rails framework (version 4). The reason for this choice lies in its ease of use, the high productivity it offers due to the scaffolding capabilities and the huge amount of libraries, called gems, together with its extended documentation and tutorials through out the web.

Rails has aided us by enforcing the REST principles and keeping a clear separation between the business elements, the persistence layer, the control logic and the presentation.

Model-View-Controller

In order to keep this separation, Rails uses the MVC (*Model-View-Controller*) architectural pattern. For each domain element such as a stream, virtual stream or trigger, the system creates a model that represents the entity, enables a set of RESTful routes and

manages the interactions through these routes in the controller methods.

There are multiple languages for views before rendering HTML, such as HAML or Slim, but ERB is used because it is the standard preprocessor. And generally, each resource has several views, represented by the following ERB files:

index display the list of resources created

show display the current resource selected

_form display a form used for creating or editing a resource (only with POST/PUT requests)

new calls the form template

edit calls the form template as well

Regarding the persistence, Rails models inherit by default from the ActiveRecord class. Therefore, whenever the controller calls a model's save method, it will be stored in the local database, which usually is a SQLite or PostgreSQL instance.

However, as a custom storage system using the distributed back end was something needed. Two other libraries were used in order to connect with the engine and keep the data on Elasticsearch. These libraries, *Her* and *Faraday*, allowed us to send HTTP requests instead of using the default capabilities like the local database. The only exception made was in the Users, which were stored in both sides, front end and back end, due to the complexity around the authentication and sessions.

The next figure summarizes the communication between the layers previously described in the application:

1. A user wants to list the users of the app, so he navigates to the url */users*
2. The router knows that a GET request to */users* is bound with the UsersController *index* method
3. In the index method, the controller calls the model using the *User.all* method
4. The User Model fetches the data from the local database (or consumes an external RESTful API as the system are doing in the project using the Her gem instead of ActiveResource)

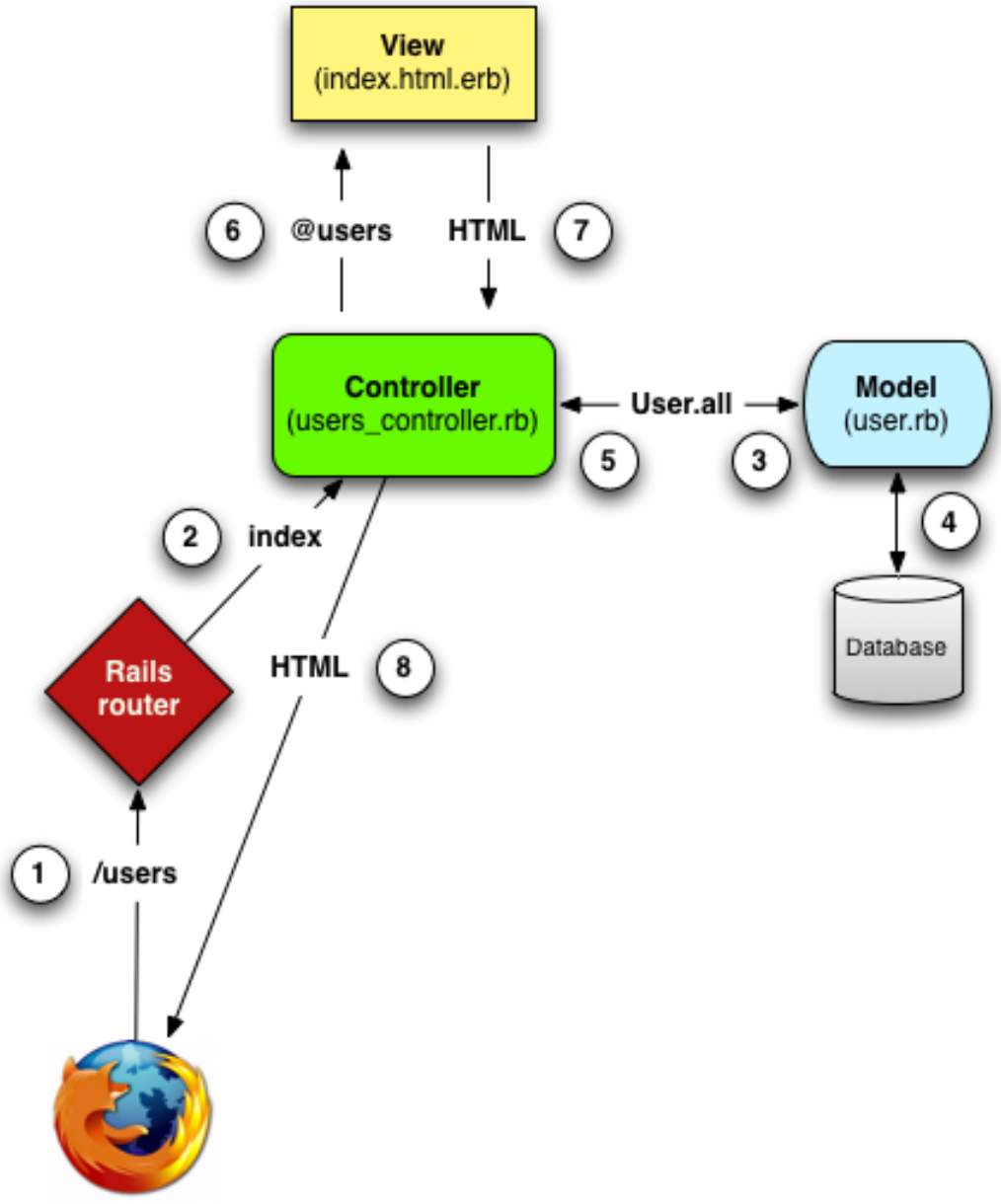


Figure 4.2: This is some text below the picture

5. UsersController stores the data retrieved in the *@users* variable, which is a hash containing all the users

6. Passing the attribute as a parameter to the corresponding view, which is *index.html.erb*

7. The View loops over the collection of users, renders the final HTML and returns it to the controller

8. Finally, the HTML code is delivered and the user is able to see the users found

4.2.2 CoffeeScript

CoffeeScript[32] is a language, developed by Jeremy Ashkenas in 2009 that compiles to JavaScript[18], with the purpose of making the client-side development more idiomatic and simple, with a Ruby-like syntax, expressions and non-statement oriented grammar.

It has been used for handling events produced by the user interaction over the whole system, but specially in windows such as triggers, search, the maps and both stream creation forms.

4.2.3 jQuery

jQuery[33] is a high-level JavaScript library designed to simplify the client-side scripting. It was created by John Resig and released in 2006, producing a new wave in the web development, facilitating the interactions with the DOM[30] API, establishing asynchronous communications with Ajax[31], and implementing basic animations and dynamic effects.

Currently, jQuery is the most popular JavaScript library in the world[34], replacing other older alternatives like Dojo[35] and Prototype[36]. In addition, it is one of the most followed projects on GitHub[37], which has lead to an active community that has created a huge ecosystem of plugins and tools which made web development easier than ever.

JQuery was used as a base library for handling user events and traversing the DOM in order to display dynamic contents.

4.2.4 D3.js and Data Visualization

Data-Driven-Documents, popularly known as **d3.js**[38], is a JavaScript library created around 2010, using standard technologies such as SVG, HTML5 and CSS3. D3 converts

digital data into rich, beautiful, interactive graphs providing dynamism to web applications. It has been used as a data visualization tool due to its high flexibility and cross-browser support.

The library was used with the purpose of visualizing stream data points in real time. In order to achieve this goal, the flow carried out was as follows:

1. First off, when a specific stream was selected by the user, the stream's *Show* window was displayed just before requesting the data points with Ajax.

2. Once the data points were fetched, they were rendered creating new SVG elements that were appended into the DOM tree.

3. Finally, if the user wanted real-time behaviour, he/she could enable the feature and the system would paint new data points dynamically. This action triggers an event which receives asynchronously more data sent by the *Pub/Sub* system, built with Node.js, through Web Sockets.

4.2.5 Bootstrap

Bootstrap[39] is an open-source web design framework created by Twitter in 2010. It allows for quick and beautiful user interfaces using the predefined CSS rules. Essentially, the user can style basic HTML elements, such as input fields or buttons by adding the default CSS classes or even extending them. Bootstrap also comes with a lot of jQuery plugins, which help the overall user experience.

This framework has been used to have a good looking and easy to use website from the beginning without putting too much effort into it. Bootstrap has a built in responsiveness feature so it automatically adapts the users pages for various screen sizes, which also allows for using a smartphone or a tablet to browse the users sites.

4.3 Back End

4.3.1 API

The API works by using Webmachine[27] to handle HTTP request that are made to the API. These HTTP requests are then matched to a set of rules in the dispatcher of Webmachine to see if the request is valid and how it should be handled. These rules

are only based on the URI that the request is made to and will match it to a erlang module that will handle the request. This means that each API module needs to have some functions defined by Webmachine.

1. *allowed_methods*: This function will return a list of allowed methods depending on the URI of the request.
2. *content_types_accepted*: This will return a list of data types accepted, and the function to use for that type, if the request sends data.
3. *content_types_provided*: This function will return a list of data types, and the function to use for that type. The system will return if the request needs some data back.
4. *delete_resource*: This function is used for delete a resource if it is an allowed method.
5. *process_post*: This will handle a POST request if it is allowed.

All these functions are present in most of the API modules, some are not used where the corresponding methods are not allowed. In the system the following API modules exist:

- **streams**
- **virtual_streams**
- **resources**
- **users**
- **datapoints**
- **triggers**
- **search**
- **suggest**
- **analyse**

Here each module will handle all requests regarding the name it has, for example *analyse* will handle all requests that have to do with analyzing the data. The data that is sent to the *streams*, *virtual_streams*, *resources*, *users* and *datapoints* modules needs to be JSON objects and are only allowed to contain certain fields (see Appendix E).

All the API modules use Elasticsearch[13] as the database and will interact with Elastisearch when handling a request. The *virtual_streams* and *triggers* modules will also spawn processes that will in the case of virtual streams handle updating the virtual stream when new data is presented to the system and in the case of triggers run the trigger function when new data is presented to the system and see if this triggers the trigger.

4.3.2 Webmachine

Webmachine[27] is a RESTful API toolkit, written in erlang, which is built on top of Mochiweb[40]. Webmachine makes it easy to integrate a RESTful API to applications written in erlang.

Resource

A application in webmachine is called a *resource*. The resource contains the source code of the users application and together with webmachines API, the user can modify the behaviour of the application based on HTTP method requests but also on other HTTP options. Webmachine provides a toolkit of functions that could be used in the resource.

Dispatching

A webmachine application uses URI dispatching to resources. This makes it possible to distribute multiple URI paths to different resources in webmachine. When webmachine receives a http request, it is handled by the dispatcher. The dispatcher tries to find the resource that matches the requested URI. If a match is found, the dispatcher will run the matching resource. If no match is found, the dispatcher will respond to the http request with a 404 error.

Decision core

After the dispatcher executes the matching resource, the resource is then using a decision core to determine the HTTP response. The decision core together with webmachine will determine which paths to follow given the http request. The HTTP diagram[28] illustrates the flow of processing a webmachine resource from the incoming HTTP request to the resulting response. This diagram is an illustration of the inner workings of the decision.

4.3.3 Elasticsearch

Elasticsearch[13] is a database that focus on having fast and powerful search, which is available through a RESTful API. It is built on top of Apache Lucene[19] which is a high performance database, fully featured text search engine written in Java. It is a NoSQL database and stores JSON documents that are dived up by index and type, it will also update the documents in near-real time.

RESTful API

Elasticsearch has a RESTful API which mean that all communication that needs to be done with Elasticsearch can be done via HTTP requests. In the system this API is used for all communication to Elasticsearch and it gives access to all the features needed. There are plugins that give a more limited graphical view instead, one example is Elastic HQ[20].

Mappings

Mappings in elasticsearch[41] are quite similar to a schema in an SQL database, it defines what kind of fields the document can have and of which field type the fields are. A document in elasticsearch can be seen as a row in an SQL table, where each document (row) have several attributes: e.g. a user has a first name, a last name, an age and a username. The names should be seen as strings, so the *type* should be string, and the age should be of *type* integer.

The users can also specify things such as *index* which would determine how the field can be searched on. If the field is a string the user can decide on how the string should be analyzed, i.e., should the string "my yellow banana" be split up into smaller searchable terms ("my", "yellow" and "banana") or should the string only be searchable as exactly "my yellow banana". The mappings used in this system can be found in the Appendix D.

Settings

In the system Elasticsearch is used to store all the required data. The index setup is to have the index 'sensorcloud' and then for each kind of entity that needs to be saved in the system there will be a type. Elasticsearch was set up on the same computer as the API was running on and made sure that Elasticsearch is not accessible from anywhere outside the computer. This is to make sure that only the API can talk to Elasticsearch directly and everything else needs to talk to the API.

Features used

The features used in Elasticsearch are the basic ones, that being *create*, *read*, *update* and *delete*. Most of the search features present in Elasticsearch were used as well as the advanced update feature of sending update scripts.

4.3.4 Analysis - R

R[42] is an open source programming language for statistical analysis. While several other solutions were considered, the decision to use R was based on that it both did what the system needed for predictions, which was the primary goal, but it also opened up for doing other statistical analysis in the future.

While it would be technically possible to allow users of the system to write their own R code, the functionality is limited to a small interface in the API. This is to prevent users from abusing the system.

rErlang

Using R in the system hinged on being able to interface it with the back end. To this end the open source library rErlang[43] is used, located through the official R-project homepage. Due to being in a half finished state when it was found, some changes and fixes had to be made in order to make it work properly with the system.

The library consists of two parts; one small Erlang part that uses Erlang built in functionality to connect it with the C side, and one fairly large C part. The library is meant to be able to both call R from Erlang and Erlang from R (not just responses), though calling R from Erlang was the only interesting part. The C side communicates with Erlang (at least in part) by writing to the *stderr* buffer, while the *stderr* seems to be used for console outputs, though no console is attached to this process.

R has no real limits on the size of the data for predictions, but sending data to it requires allocation of buffers in C in advance. While it would be possible to code around this limit, the API enforces limits of no more than 500 datapoints as inputs or outputs. It also sets a minimum size, so that users of the system can't ask for impossible predictions.

Predictions

For doing predictions, or forecasts, on a time series (a list of datapoints with timestamps) the system uses a method called ARIMA[44] (Autoregressive integrated moving average).

This method is supposed to be a relatively general one, which means it is not specialized on any specific kind of data.

While the model as such requires some parameters to be set, R has a package that does that automatically. While the predictions may not be perfect for all sets of data, it stands to reason that they are more or less as good as possible for a method run on an arbitrary type of data.

4.3.5 Pub/Sub System

Concepts

- **Stream**: A stream is a flow of data originating from a sensor. A user can subscribe to a stream.
- **Virtual Stream**: A virtual stream subscribes to one or several streams/virtual streams and process incoming data according to a defined function, which could be the average or some other aggregation of data. A user can subscribe to a virtual stream.
- **Data Point**: A data point is a data value to and from a stream/virtual stream. When the system receives data in Json format from an external resource, the system parses it and transforms it to a data point which can be stored and published into the publish/subscribe system, pub/sub system.

Overview

When a new data point arrives for a stream, it is published in the pub/sub system and distributed to all clients who have subscribed to the stream. RabbitMQ[22] has been utilized to implement the pub/sub system. Node.js[23] and Socket.IO[24] are used to interact with the web pages via web-sockets, which allows for support of dynamic visualization.

Clients

A client in the system can be one of the following:

- **Webpage**: A webpage can subscribe to data from the pub/sub system via a web-socket, which enables dynamic visualization.

- **Session**: A session is a logged on user to which we can provide information or alerts about their subscriptions or triggers.
- **Virtual stream**: A virtual stream subscribes to one or several streams and/or virtual streams in order for it to calculate a new value which it in turn can publish.
- **Trigger**: A trigger subscribes to one or several streams or virtual streams in order to check if an alert is to be sent or alternatively any other specified action needs to be taken.

RabbitMQ

RabbitMQ[22] is a message broker and provides robust messaging for many successful commercial websites, including Twitter. It runs on a majority of operating systems and is easy to use. It offers client's libraries for many mainstream programming languages, including Erlang[55]. RabbitMQ is AMQP[52] (Advanced Messaging Queueing Protocol) based protocol and the following are essential concepts:

- **Queue**: A queue is used to cache incoming messages and a client could fetch messages from the queue.
- **Exchange**: A exchange is where data points arrive and distribute to the connected queues according to some rule, for example fanout rule or topic.
- **Publisher/Subscriber**: A publisher is something that sends, publishes, data into the pub/sub system and a subscriber, commonly known as a consumer, is something that listens to specific data that comes in to the pub/sub system.

Why RabbitMQ In the beginning thoughts were to use ZeroMQ[53] as the pub/sub system, because various benchmarks suggested that it was the fastest in terms of throughput. Never the less a change to RabbitMQ was made, because even though it suggested to be slower in throughput, but faster than most others, it offered more functionality since it uses a broker. Other reasons were that RabbitMQ is written in Erlang, the main language of the system, and it also have the capability to cluster, meaning that it is scalable.

How RabbitMQ is used For each type of data point a client can subscribe to, there is an exchange, on which the client can create a queue to receive messages on. In the system there are exchanges for streams, virtual streams, triggers and alerts.

For example, if there is a client that is a virtual stream, which wants data from two streams, it would connect one queue to the exchange for the first stream and another queue to exchange of the other stream. When one of the streams gets a data point that data point is published to the stream's exchange and the virtual stream would get notified.

All exchanges use the *fanout* rule, which mean that if several clients have a queue on the same exchange they would all get the same message at the same time.

Node.js

Node.js [23] is used in combination with Socket.IO[24] to push the latest datapoints of a stream to the front end. Node.js is a platform that is used to build server-side scalable network applications. It utilizes a single-thread loop together with a event-driven I/O model to achieve a high throughput. The scripting language utilized by Node.js is javascript. The Rabbit.js[25] library is used to provide AMQP communication between Node.js and RabbitMQ.

4.3.6 Polling System

The polling system is responsible for pushing and fetching data from the external resources. When the new data arrives, it is published to the pub/sub system after parsing.

Actor Model

In the actor model pattern, each object is an actor. This is an entity that has a mailbox and a behaviour. Messages can be exchanged between actors, which will be buffered in the mailbox. Upon receiving a message, the behaviour of the actor is executed, upon which the actor can: send a number of messages to other actors, create a number of actors and assume new behaviour for the next message to be received.

Of importance in this model is that all communications happen by means of asynchronous message-passing. This allows for to organize a number of processes in a tree. Thus the polling system has been designed as a process tree, one supervisor supervises all the actors.

Framework

In the project, the polling system contains three parts: *polling_supervisor*, *polling_monitor* and *poller*. The pollers are actors and are supposed to handle all communications to the external resources. The polling_monitor supervises the pollers and restart the broken

poller automatically according to user‘s setting. The polling_supervisor is supposed to handle all the programming interfaces to the system.

Each actor has been implemented using gen_server, and has been supervised in supervisor framework. When one poller fetches data, it will push the data to the *parser* module. In this module, JSON data is parsed by the functions in *lib_json.erl* file, text data is supposed to be parsed using the *re* module (but this has not been implemented), which is the built-in regular expression module. The polling_supervisor has also been implemented using gen_server.

Chapter 5

Testing

To make sure that tests worked when pull requests were made on GitHub, Travis[45] was used. A pull request is a request to merge the changes in the users local branch into the main development branch. Travis is a continuous integration[46] tool that automatically runs all the tests that were included in the system code and reports the result as part of a pull request.

5.1 Front End

Throughout development of the front end a test-driven development (TDD) approach was used. This approach consists of writing tests for the expected functonality and design the functionality so that all of the predefined test cases passes. By using this approach, the developer can be confident that the produced code works as intended.

Two kinds of tests were performed: integration tests (using RSpec[47] and Capybara[48]) and unit tests. Integration tests mimic the action of submitting a form, clicking on a button, etc. This made the testing process a lot faster, since manual testing of every page of the website each time new features were added was not needed.

5.2 Back End

In the back end Eunit[49] is used to test the Erlang modules. Every important module has a bi-module that contains tests. These tests were used to make sure that the module still worked as it should after code had been added, removed or changed. The tests check the normal functionality of the module and make sure that non-allowed actions do not work. All these tests can be run using the makefile present in the project folder. The

command 'make test' will run all the tests and return a count of how many tests failed, succeeded and were canceled.

Chapter 6

Related Work

6.1 SicsthSense

This system was made by SICS[56] and has similar base functionality, gathering sensor data and sharing it, as the presented system. Since, the initial idea was taken from SicsthSense[1], the prototype and architectural design of the system is quite similar to theirs. It also has preliminary code for installing it on small devices and android phones which talks with the engine of SicsthSense.

6.2 Xively

Xively[8] (formerly known as Cosm and Pachube) is a division of LogMeIn Inc[6], a leading provider of essential remote services. It was created in 2007 by architect Usman Haque[57]. It also has support for real-time graphs, process historical data pulled from any public feed and send real-time alerts(triggers), similar to the presented system. But they also have functionality to add widgets in websites. It also provides added support to create interactive apps for connected products with various platform support like iOS, Android and Javascript.

6.3 ThingSpeak

ThingSpeak[7] is an open source Internet of Things application and API to store and retrieve data from various devices. With ThingSpeak, the user can create sensor logging applications, location tracking applications, and a social network of things with status updates. It is also integrated with Twitter by which the user can get updates from the users devices by Tweets.

Chapter 7

Conclusion

The main goals of this project was to develop a system based on the principles of a cloud-centric data store for the Internet of Things. Having the ability to interact with and visualize data, and also to get useful information out of it. The project were able to achieve all these goals with added functionalities. The back end was developed using Erlang[55] and the front end was developed using Ruby on Rails[54]. This product can for instance be used as the central system in a smart and sensor-driven home, a data analytics engine with several virtual data outputs from data inputs or a visual status system for sensor information.

Chapter 8

Future Work

8.1 Evaluation

The system is not evaluated on a large scale. Design decisions during the project were always taken with scalability in mind but the system as it is now will probably not handle large amounts of data very well. Therefore an evaluation of the system is needed to see how well the system performs. This is a vital and essential future work as the whole system is based on the principal that should be scalable.

8.2 Scalability

With the consistently increasing amount of data, the system needs to be scalable. ElasticSearch can be run in clusters[50] but Webmachine[27] can not. Right now, Webmachine, the polling system, the pub/sub system, the virtual stream processes and the trigger processes are all running on the same Erlang node. So, the first step would be to break these up on different nodes, that could be run on different machines. This should only require minimum rework of the system to make sure the Erlang nodes can communicate with each other. The next step would then be to break up the individual parts. This would require some reworking of the system, but since the parts of the system are separated they should be able to be reworked separately.

8.3 Security

Currently, the system have basic access control mechanism only. Data exchanged between different services is not encrypted in any form. Anyone can send a data point to any stream through a simple HTTP POST request which can alter the behavior of the stream

and make it faulty. A possible solution is to deploy basic authentication in the back end and using HTTPS.

8.4 Functionality

Lots of features were included in the final product, but there are a few functionalities which were thought of but not implemented due to time constraints. Functionalities such as grouping of streams (if a user would want to see all the streams from his or her smart home on the same page), back end authentication and authorization, and a query language that would allow users to make powerful aggrigations directly in the search.

8.4.1 More Advanced Search

Using Faceted Search[51] which allows users to explore a collection of information by successively applying filters in whatever order they choose.

8.4.2 Stream Relations

Implement the ability to create relations between groups of streams, for example say that these streams belong together as they measure weather data in Uppsala or that they measure temperature in my home, and have information about these relationships.

8.4.3 Improved Triggers

Improving the functionality of trigger so it can actuate a device when a trigger is fired. Also adding the functionality of getting live alerts via for example an SMS or a tweet. Also, creation of triggers in the front end is now limited to only allow triggers on streams and virtual streams that the user own, while the back end do not have this limitation.

8.4.4 Mobile Application

Ability to generate a mobile application for the system, which will provide live feeds from a particular stream, a virtual stream or a group and ability to add triggers.

8.4.5 Added Parsing Capabilities

Right now the parser can only handle data formated as JSON objects. This should be extend to handle formats such as XML and HTML. As mentioned in 4.3.6 handling text data using regular expressions is also something that should be added.

8.4.6 Added Analyzing Capabilities

Right now the system only uses R to do forecasts using the ARIMA[44] (Autoregressive integrated moving average) but R has the capability to run more kinds of statistical analysis. The back end could be extended to allow for more analysis to be done using R.

8.4.7 Quality of Information

The system is missing Quality of Information (QoI), but it would provide many nice features to the system, such as a system ranking on a stream or similar based on its QoI. It could also be used to let a user know if a stream is malfunctioning or providing out of range values.

8.5 Improvements

8.5.1 Live updates

A node.js server needs to run in the back end to enable live updates. Node.js is a good platform that supports multiple concurrent connections however, more suitable options exist. Instead of using node.js, the live update functionality could be reworked by implementing the same functionality in an Erlang module. This would make deployment easier and have a better integration with the overall project.

8.5.2 Virtual streams

In the current system a data point for a virtual stream is created as soon as a data point arrive for any of the parent streams it depend on. This can lead to very inconsistent timestamp intervals since the update frequency for each parent can differ, in addition the time at which they start their interval is also different. One possibility to solve this is to use the time interval, used when creating the virtual stream, in the pub/sub system. This would mean that a data point would not be posted until after an amount of time, regardless if the parent streams have been updated or not.

8.5.3 Timestamps

Timestamps can have different time zones than that in the system, which can create inconsistencies in the timestamp between data points especially concerning virtual streams. Since virtual streams create a new local timestamp for each new data point, its history, created from the timestamps of the parent streams, can initially have a big gap or a big

overlap to the new data points for the virtual stream. To solve this the system need to take into account time zones when creating and displaying timestamps. Time zones can either be provided or derived from the location of the resource.

Bibliography

[1] SicsthSense - Log In. [Online]. Available: http://sense.sics.se/. [2014, January 14].

[2] projectcs13. [Online]. Available: https://github.com/projectcs13. [2014, January 14].

[3] Ashton, K (22 June 2009). That 'Internet of Things' Thing, in the real world things matter more than ideas". RFID Journal.

[4] iot_comic_book.pdf. [Online]. Available: http://www.alexandra.dk/uk/services/publications/documents/iot_comic_book.pdf. [2014, January 14].

[5] *Mahizhnan, A. (1999). Smart cities: The Singapore case. Cities, 16(1), 13–18.*

[6] Remote Access and Remote Desktop Software for Your Computer — LogMeIn. [Online]. Available: https://secure.logmein.com/. [2014, January 14].

[7] Internet of Things - ThingSpeak. [Online]. Available: https://www.thingspeak.com/. [2014, January 14].

[8] Xively - Public Cloud for the Internet of Things. [Online]. Available: https://xively.com/. [2014, January 14].

[9] draft-vial-core-mirror-proxy-00 - CoRE Mirror Server. [Online]. Available: https://tools.ietf.org/html/draft-vial-core-mirror-proxy-00. [2014, January 14].

[10] What is UUID (Universal Unique Identifier) - Definition from WhatIs.com. [Online]. Available: http://searchsoa.techtarget.com/definition/UUID. [2014, January 14].

[11] http://www.ericsson.com/thinkingahead/networked_society [2014, February 19]

[12] Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST). [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [2014, January 14].

[13] Open Source Distributed Real Time Search & Analytics — Elasticsearch. [Online]. Available: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html. [2014, January 14].

[14] Query String Query [0.90]. [Online]. Available: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html. [2014, January 14].

[15] Are Email Addresses Case Sensitive? - About Email. [Online]. Available: http://email.about.com/od/emailbehindthescenes/f/email_case_sens.htm. [2014, January 14].

[16] Session (computer science) - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Session_(computer_science). [2014, January 14].

[17] HTTP cookie - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/HTTP_cookie. [2014, January 14].

[18] JavaScript - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/JavaScript. [2014, January 14].

[19] Apache Lucene - Welcome to Apache Lucene. [Online]. Available: http://lucene.apache.org/. [2014, January 14].

[20] ElasticHQ - ElasticSearch monitoring and management application. [Online]. Available: https://github.com/projectcs13/erlastic_search. [2014, January 14].

[21] tsloughter/erlastic_search. [Online]. Available: https://github.com/tsloughter/erlastic_search. [2014, January 14].

[22] RabbitMQ - Messaging that just works. [Online]. Available: http://www.rabbitmq.com/. [2014, January 14].

[23] node.js. [Online]. Available: http://nodejs.org. [2014, January 14].

[24] Socket.IO: the cross-browser WebSocket for realtime appsjsjs. [Online]. Available: http://socket.io/. [2014, January 14].

[25] rabbit.js/README.md at master squaremo/rabbit.js. [Online]. Available: https://github.com/squaremo/rabbit.js/blob/master/README.md. [2014, January 14].

rabbit.js

[26] Ruby Programming Language. [Online]. Available: https://www.ruby-lang.org/en/. [2014, January 14].

[27] basho/webmachine. [Online]. Available: https://github.com/basho/webmachine. [2014, January 14].

[28] http-headers-status-v3.png. [Online]. Available: https://raw.github.com/wiki/basho/webmachine/images/http-headers-status-v3.png. [2014, January 14].

[29] alavrik/erlson. [Online]. Available: https://github.com/alavrik/erlson. [2014, January 14].

[30] Document Object Model - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Document_Object_Model. [2014, January 14].

[31] Ajax (programming) - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Ajax_(programming). [2014, January 14].

[32] CoffeScirpt. [Online]. Available: http://coffeescript.org/. [2014, January 14].

[33] jQuery. [Online]. Available: http://jquery.com/. [2014, January 14].

[34] jQueryl - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/JQuery. [2014, January 14].

[35] Unbeatable JavaScript Tools - The Dojo Toolkit. [Online]. Available: http://dojotoolkit.org/. [2014, January 14].

[36] Prototype JavaScript framework: a fundation for ambitious web applications. [Online]. Available: http://prototypejs.org/. [2014, January 14].

[37] jquery/jquery. [Online]. Available: https://github.com/jquery/jquery. [2014, January 14].

[38] D3.js - Data-Driven Documents. [Online]. Available: http://d3js.org/. [2014, January 14].

[39] Bootstrap. [Online]. Available: http://getbootstrap.com/. [2014, January 14].

[40] mochi/mochiweb. [Online]. Available: https://github.com/mochi/mochiweb. [2014, January 14].

[41] Mapping — Reference Guide — Elasticsearch. [Online]. Available: http://www.elasticsearch.org/guide/mapping. [2014, January 14].

[42] The R Project for Statistical Computing. [Online]. Available: http://www.r-project.org/. [2014, January 14].

[43] projectcs13/rErlang. [Online]. Available: https://github.com/projectcs13/rErlang. [2014, January 14].

[44] Autoregressive integrated moving average - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average. [2014, January 14].

[45] Travis CI - Free Hosted Continuous Integration Platform for the Open Source Community. [Online]. Available: https://travis-ci.org/. [2014, January 14].

[46] Continuous Integration. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [2014, January 14].

[47] RSpec.info: home. [Online]. Available: http://rspec.info/. [2014, January 14].

[48] Capybara. [Online]. Available: http://jnicklas.github.io/capybara/. [2014, January 14].

[49] Erlang – EUnit - a Lightweight Unit Testing Framework for Erlang. [Online]. Available: http://www.erlang.org/doc/apps/eunit/chapter.html. [2014, January 14].

[50] Overview — Elasticsearch. [Online]. Available: http://www.elasticsearch.org/overview/. [2014, January 14].

[51] Facets[0.90] . [Online]. Available: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/search-facets.html. [2014, January 14].

[52] Home — AMQP. [Online]. Available: http://www.amqp.org/. [2014, January 14].

[53] Distributed Computing Made Simple - zeromq. [Online]. Available: http://zeromq.org/. [2014, January 14].

[54] Ruby on Rails. [Online]. Available: http://rubyonrails.org/. [2014, January 14].

[55] Erlang Programming Language. [Online]. Available: http://www.erlang.org/. [2014, January 14].

[56] Home — SICS. [Online]. Available: https://www.sics.se/. [2014, January 14].

[57] Xively - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Xively. [2014, January 14].

[58] RabbitMQ - Messaging that just works. [Online]. Available: http://www.rabbitmq.com/build-erlang-client.html. [2014, January 14].

[59] Cluster Illusion - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Clustering_illusion. [2014, January 15].

Appendix A

Usage/Tutorial

A.1 Creation of user

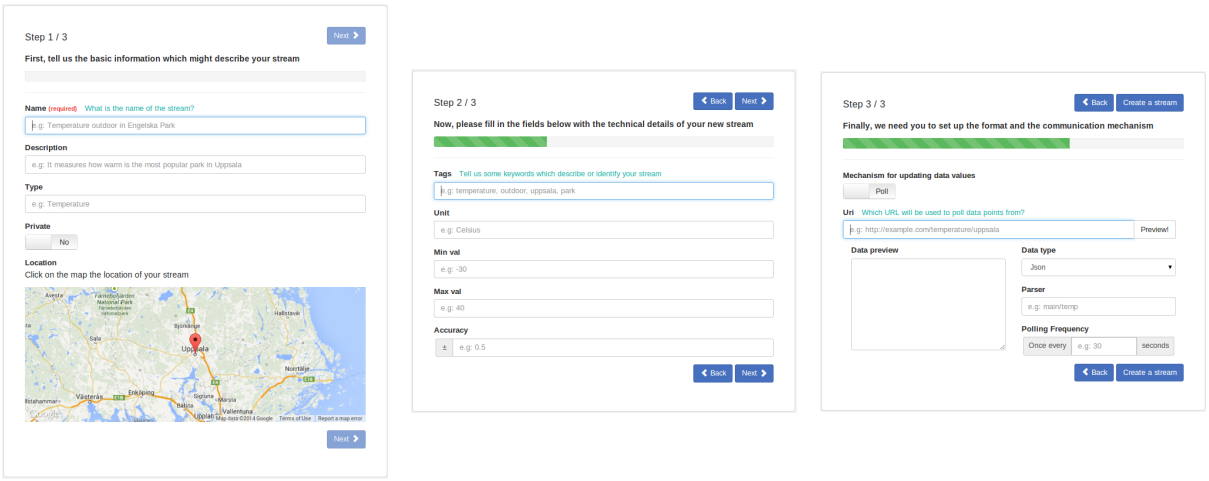
When the user access the website, they are presented with a frontpage. A user could choose to login with an existing account, register a new account or search for public streams. The "sign up" button will allow users to register a new account. It is located on the top-right corner.

The user is then presented with a sign-up form. The sign-up form requires appropriate information in the fields such as username, email, password, firstname and lastname. It also allows users to make their profile private or public.

A.2 Creation of stream

To create a stream, a user needs to be logged in to the website and push the "My streams" button on the top navigation bar. The page presents previously created streams and allows creation of a new stream, creating a stream from a resource or delete all existing streams. If the user chooses to create a stream from a resource, they need to enter a resource name. To create a stream, the user needs to press on the "New stream" button.

The signup process for a new stream is divided into three steps. The first step requires basic information about the stream such as a name, description and type. The second step requires technical details about the stream such as unit of measurement and tags associated to the stream. The last step lets the user choose if the stream should be polled or pushed.



After the sign-up process, a view of the stream is presented. The page shows a graph and basic information such as privacy and active state, triggers that are associated to the stream and a map that shows the position of the streams source of information. Live updates can be activated on this page, this will allow the user to see arriving datapoints plotted on the graph in real-time.

A.3 Creation of virtual stream

Virtual streams can be created from public streams. The user can highlight the streams that they want to include in a virtual stream from the search results page. They can then press the ”Virtual stream” button to create a virtual stream. A virtual stream requires name, description, tags and a starting date. The starting date determines the history size. A function that is applied to the virtual stream can also be chosen.

The virtual stream view is almost identical to the view of an ordinary stream and has the same functionality. The graph shows the results of the included streams after a function has been applied. Selecting the ”My virtual streams” button on the top navigation bar will allow access to virtual streams that a user has created.

A.4 Creation of triggers

Users can create triggers for a stream or a virtual stream by pressing the ”My triggers” button on the top navigation bar. The page will show existing triggers and also allows for creation of triggers. During the creation of a trigger, a function and condition should be provided. They can also choose to push the alert message to a given URI or to display the alert message on a users profile page.

A.5 Following a stream

Users can also follow public streams by pressing the "follow" button on a streams view page. Followed streams can be accessed by pressing the "Following" button on the top navigation bar.

A.6 Search

The search page will show streams, virtual streams and users related to a given search query. Highlighting streams in the search results will show the geographical position of a streams source. Filters can be applied to search queries, by pressing the drop-down button. Search results can be filtered by unit of measurement, tags associated to the streams and also by the active state of a stream. Search results can also be filtered by location, users can interactively set a position and a radius area on the world map to filter out streams that are outside the given area. The search results can be sorted by name so that the search results appear in alphabetical order by user ranking, search results would then be sorted from highest ranked stream to the lowest ranked stream.

Appendix B

Structure of the Code

B.1 Front end

The folders used for the project follows the usual convention in every common Rails app, based on the structure that appears below:

app/ Core application code, including models, views, controllers, and helpers

app/assets Applications assets such CSS, JavaScript files, and images

bin/ Binary executable files

config/ Application configuration

db/ Database files

doc/ Documentation for the application

lib/ Library modules

lib/assets Library assets such as CSS, JavaScript files, and images

log/ Application log files

public/ Data accessible to the public (web browsers) such as error pages

spec/ Application tests

tmp/ Temporary files

vendor/ Third-party code such as plugins and gems

vendor/assets Third-party assets

Other relevant files:

README.rdoc A brief description of the application

Rakefile Utility tasks available via the rake command

Gemfile List of gems (dependencies) used in this app

Gemfile.lock File that ensures that all the app copies use the same gem versions

config.ru A configuration file for Rack middleware

.gitignore Patterns for files that should be ignored by Git

B.2 Back end

The project's folder has the following structure:

config/ Application configuration

doc/ Documentation for the application

include/ Files for storing shared macros and structures

javascripts/ Javascripts for nodejs

priv/ Webmachine's private files

scripts/ Scripts used to install dependencies, execute engine, produce data

src/ Core application code

test/ Application tests

Other relevant files:

.travis.yml Travis configuration file

Makefile Application's make file

rebar.config Rebar configuration file

.gitignore Patterns for files that should be ignored by Git

Appendix C

Dependencies and Libraries

C.1 Front end

As commented before, one of the greatest advantages Ruby has is the great support and development around its community, allowing the developers to reduce unnecessary efforts using small software components called "Gems". The following list explains what is the purpose of each dependency

Framework and Security

rails 4 The well-known web framework

turbolinks gem for downloading the assets dynamically

faker input fake data in the local database and make tests easier

bcrypt-ruby A gem that helps the user keep users passwords private and secure

Communications

jbuilder library to handle JSON conversions

faraday A neat library for making HTTP requests

her A library used to create models and consume RESTful APIs

JavaScript

therubyracer gem which embeds the V8 Javascript interpreter into Ruby

jquery the brand popular library for DOM-manipulation and AJAX

coffee to support CoffeeScript and compile it into JavaScript

uglifier to minify the JavaScript source code

d3-rails JavaScript library for graph visualization

CSS and Styles

sass-rails A pre-processor with more advanced features than simple CSS

bootstrap-sass CSS Framework using the Sass pre-processor

will_paginate Add pagination to the index windows

will_paginate-bootstrap Style pagination with Bootstrap

bootstrap-switch-rails Style checkboxes with Bootstrap

Testing

sqlite Database in the back end

rspec-rails The popular testing framework, RSpec

selenium-webdriver A tool for writing automated tests of websites

capybara Integration test tool. Simulates the user interaction on the app

factory_girl_rails Definition of saved instances for testing purposes

Documentation

sdoc Standalone Ruby documentation generator

Production Environment

pg Gem for creating PostgreSQL database instances

rails_12factor Gem that makes easier the deployment on Heroku

C.2 Back end

For the back end a build tool called *Rebar* was used. Downloading dependencies from git and compiling a project and its dependencies is easily done with Rebar.

RESTful framework

webmachine Provides a small framework for implementing RESTful functionality

Storage and searching

elastic_search Provides storage and advanced searching functionality.

erlasticsearch Erlang interface to elastic_search

Statistical analysis

R Statistical analysis programming language. Downloaded manually, not with Rebar.

rErlang Erlang interface to R.

PubSub system

rabbitmq-server RabbitMQ server, used the Publisher/Subscriber (Pub/Sub) System.

rabbitmq-erlang-client RabbitMQ client using the AMQP protocol.

rabbitmq-codegen Dependency library for building the RabbitMQ client[58].

JSON manipulation

Erlson Library for creating, reading and updating JSON string objects.

Miscellaneous

nodejs Creates websockets which is needed by RabbitMQ libraries.

xsltproc Needed by rabbitmq-erlang-client.

Appendix D

Elasticssearch Mappings

D.1 Datapoint

```
{
  "datapoint" : {
    "_source" : {
      "enabled" : true
    },
    "dynamic" : false,
    "properties" : {
      "stream_id": {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "timestamp" : {
        "type" : "date",
        "format" : "dateOptionalTime"
      },
      "value" : {
        "type" : "double"
      }
    }
  }
}
```

D.2 Pollinghistory

```
{
  "pollinghistory" : {
   "_source" : { "enabled" : true },
    "dynamic": false,
    "properties" : {
      "history":{
      "type" : "object"
      }
    }
  }
}
```

D.3 Resource

```
{
  "resource" : {
    "_timestamp" : { "enabled" : true, "store" : true},
    "dynamic": false,
    "properties" : {
      "name" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "description" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "manufacturer" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "model" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "streams_suggest" : {
        "properties" : {
          "name" : {
            "type" : "string",
            "index" : "not_analyzed"
          },
          "accuracy" : {
              "type" : "float"
          },
          "description" : {
              "type" : "string",
              "index" : "no"
          },
          "min_val" : {
```

```
            "type" : "float"
        },
        "max_val" : {
            "type" : "float"
        },
        "polling" : {
            "type" : "boolean"
        },
        "polling_freq" : {
            "type" : "long"
        },
        "type" : {
            "type" : "string",
            "index" : "not_analyzed"
        },
        "tags" : {
            "type" : "string",
            "index" : "analyzed"
        },
        "unit" : {
            "type" : "string",
            "index" : "not_analyzed"
        }
      }
    }
  }
}
```

D.4 Search_query

```
{
  "search_query": {
    "properties": {
      "search_suggest": {
        "type": "completion",
        "analyzer": "keyword",
        "payloads": false,
        "preserve_separators": true,
        "preserve_position_increments": true,
        "max_input_len": 50
      }
    }
  }
}
```

D.5 Stream

```
{
  "stream" : {
    "_source" : {
      "enabled" : true
    },
    "_timestamp" : {
      "enabled" : true
    },
    "dynamic": false,
    "properties" : {
      "name" : {
        "type" : "multi_field",
        "fields" : {
          "name" : {
            "type" : "string",
            "index" : "analyzed"
          },
          "untouched" : {
            "type" : "string",
            "index" : "not_analyzed"
          }
        }
      },
      "description" : {
        "type" : "string",
        "index" : "no"
      },
      "type" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "tags" : {
        "type" : "string",
        "index" : "analyzed"
      },
```

```
"private" : {
  "type" : "boolean"
},
"unit" : {
  "type" : "string",
  "index" : "not_analyzed"
},
"accuracy" : {
  "type" : "float"
},
"min_val" : {
  "type" : "float"
},
"max_val" : {
  "type" : "float"
},
"polling" : {
  "type" : "boolean"
},
"parser" : {
  "type" : "string",
  "index" : "not_analyzed"
},
"data_type" : {
  "type" : "string",
  "index" : "not_analyzed"
},
"uri" : {
  "type" : "string",
  "index" : "not_analyzed"
},
"polling_freq" : {
  "type" : "long"
},
"location" : {
  "type" : "geo_point",
  "lat_lon" : true,
```

```
  "geohash" : true,
  "geohash_prefix" : true,
  "normalize" : false,
  "validate" : true
},
"resource" : {
  "properties" : {
    "resource_type" : {
      "type" : "string",
      "index" : "not_analyzed"
    },
    "uuid" : {
      "type" : "string",
      "index" : "no"
    }
  }
},
"quality" : {
  "type" : "float"
},
"active" : {
  "type" : "boolean"
},
"user_ranking" : {
  "properties":{
    "average":{
      "type" : "float"
    },
    "nr_rankings":{
      "type": "long"
    }
  }
},
"nr_subscribers" : {
  "type" : "long"
},
"subscribers" : {
```

```
        "properties" : {
          "user_id" : {
            "type" : "string"
          }
        }
      },
      "last_updated" : {
        "type" : "date"
      },
      "creation_date" : {
        "type" : "date"
      },
      "history_size" : {
        "type" : "long"
      },
      "user_id" : {
        "type" : "string",
        "index" : "not_analyzed"
      }
    }
  }
}
```

D.6 Suggestion

```
{
  "suggestion" : {
    "properties" : {
      "resource_id" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "suggest" : { "type" : "completion",
        "index_analyzer"  : "model_analyzer",
        "search_analyzer" : "model_analyzer",
        "payloads" : true
      }
    }
  }
}
```

D.7 Trigger

```
{
  "trigger" : {
    "_timestamp" : { "enabled" : true, "store" : true},
    "properties" : {
      "function" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "outputlist" : {
        "properties" : {
            "input" : {"type" : "float"},
            "output" : {"properties" : {
              "output_id" : {"type" : "string"},
              "output_type" : {"type" : "string"}
            }
          }
        }
      },
      "streams" : {
        "type" : "string",
        "index" : "analyzed"
      },
      "vstreams" : {
        "type" : "string",
        "index" : "analyzed"
      }
    }
  }
}
```

D.8 User

```
{
  "user" : {
    "_timestamp" : { "enabled" : true, "store" : true},
    "_source" : {"excludes" : ["password"]},
    "properties" : {
      "username" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "rankings" : {
        "properties" : {
            "stream_id" : {"type" : "string"},
            "rank" : {"type" : "float"}
        }
        },
      "triggers" : {
        "properties" : {
            "function" : {"type" : "string"},
            "input" : {"type" : "float"},
          "streams" : {"type" : "string"}
        }
      },
      "notifications" : {
        "type" : "object"
      },
      "email" : {
        "type" : "string",
        "index" : "no"
      },
        "password" : {
        "type" : "string",
        "index" : "no",
        "store" : "yes"
      },
        "firstname" : {
```

```
        "type" : "string"
      },
        "lastname" : {
        "type" : "string"
      },
      "description" : {
        "type" : "string"
      },
      "subscriptions" : {
        "properties" : {
          "stream_id" : {
            "type" : "string",
            "index" : "not_analyzed",
            "store" : "no"
          }
        }
      },
      "private" : {
        "type":"boolean"
      }
    }
  }
}
```

D.9 Virtual Stream

```
{
  "virtual_stream" : {
    "_source" : {
      "enabled" : true
    },
    "_timestamp" : {
      "enabled" : true
    },
    "dynamic": false,
    "properties" : {
      "name": {
        "type" : "string",
        "index" : "analyzed"
      },
      "user_id": {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "description" : {
        "type" : "string",
        "index" : "no"
      },
      "tags" : {
        "type" : "string",
        "index" : "analyzed"
      },
      "group" : {
        "type" : "string",
        "index" : "not_analyzed"
      },
      "private" : {
        "type" : "boolean",
        "index" : "not_analyzed"
      },
      "user_ranking" : {
```

```
  "properties":{
    "average":{
      "type" : "float"
    },
    "nr_rankings":{
      "type": "long"
    }
  }
},
"nr_subscribers" : {
  "type" : "long"
},
"subscribers" : {
  "properties" : {
    "user_id" : {
      "type" : "string"
    }
  }
},
"history_size" : {
  "type" : "long",
  "index" : "not_analyzed"
},
"last_updated" : {
  "type" : "date",
  "index" : "not_analyzed"
},
"creation_date" : {
  "type" : "date",
  "index" : "no"
},
"streams_involved" : {
  "type" : "string",
  "index" : "not_analyzed"
},
"function" : {
  "type" : "string",
```

```
        "index" : "no"
      }
    }
  }
}
```

D.10 Virtual Stream Datapoint

```
{
  "vsdatapoint" : {
    "_source" : { "enabled" : true },
    "dynamic": false,
    "properties" : {
      "stream_id":{
        "type":"string",
        "index":"not_analyzed"
      },
      "timestamp":{
        "type":"date",
        "format":"dateOptionalTime"
      },
      "value":{
        "type":"double",
        "index":"not_analyzed"
      }
    }
  }
}
```

Appendix E

Accepted and Restricted Fields in the API

E.1 Streams

E.1.1 Restricted fields when updating

- active
- quality
- user_ranking
- subscribers
- last_update
- creation_date
- history_size

E.1.2 Restricted fields when creating

- active
- quality
- user_ranking
- subscribers
- nr_subscribers

- last_update
- creation_date
- history_size

E.1.3 Accepted fields

- user_id
- name
- description
- type
- tags
- private
- unit
- accuracy
- min_val
- max_val
- polling
- uri
- polling_freq
- location
- resource
- resource.resource_type
- resource.uuid
- parser
- data_type
- location.lon
- location.lat

E.2 Virtual Streams

E.2.1 Restricted fields when updating

- function
- streams_involved
- creation_date
- timestampfrom

E.2.2 Restricted fields when creating

There are no restricted fields when creating.

E.2.3 Accepted fields

- user_id
- name
- description
- tags
- private
- function
- streams_involved
- creation_date
- timestampfrom

E.3 Resources

E.3.1 Restricted fields when updating

There are no restricted fields when updating.

E.3.2 Restricted fields when creating

- streams_suggest

E.3.3 Accepted fields

- name
- description
- model
- manufacturer

E.4 Users

E.4.1 Restricted fields when updating

- username
- subscriptions

E.4.2 Restricted fields when creating

There are no restricted fields when creating.

E.4.3 Accepted fields

- username
- email
- firstname
- lastname
- description
- password
- private

E.5 Data-points

E.5.1 Restricted fields when updating

The data-points are not allowed to be updated.

E.5.2 Restricted fields when creating

There are no restricted fields when creating.

E.5.3 Accepted fields

- stream_id
- timestamp
- value

Appendix F

Installation

F.1 Front end

F.1.1 Requirement

- Linux
- Ruby 1.9.3 (Ruby 2.0 preferred)

F.1.2 Installation

You need to set an option for your shell in order for all of the software to work. Run the following command and read the 'Important' section and follow the instructions.

- make help

Download and compile the dependencies, and compile the project sources

- make install

Run the application

- make run

F.1.3 Usage

In development mode

1. Install the gems needed
    - bundle install
2. Migrate the database

- bundle exec rake db:migrate

3. If you are running the IoT-Framework API locally, then you should modify the API_URL variable declared in the config/config.yml file
   - API_URL : " Put your base URL here : put your port here "
4. Start the Rails server
   - rails s

In production mode

1. Install the gems needed
   - bundle install
2. Migrate the database
   - RAILS_ENV=production bundle exec rake db:migrate
3. Precompile Rails assets
   - RAILS_ENV=production bundle exec rake assets:precompile
4. Modify the 'config/config.yml' file according to your needs.
5. Open the script called sensor_cloud located in the sensor-cloud-website root folder, and modify the USER and RAILS_ROOT variables in accordance to your system settings.
6. Run
   - sudo cp sensor_cloud /etc/init.d/
   - sudo chmod +x /etc/init.d/sensor_cloud
   - sudo update-rc.d sensor_cloud defaults

F.1.4 Running tests

1. Run the tests
   - make test

F.2 Back end

F.2.1 Installing the project

1. Download and compile the linux system dependencies, (only needed once per machine)
    - make install_linux_deps
2. Download and compile the project dependencies, and compile the project sources
    - make install

F.2.2 Running the project

1. Run the application by using startup script (one of the commands below)
    - make run_all
    - sudo ./scripts/sensec.sh start
2. Alternative run (type each in separate shells)
    - make run_rabbit
    - make run_es
    - make run_nodejs
    - export R_HOME="/usr/lib/R"
    - make run
3. To shutdown either close each individual shell or run one of the commands below
    - make stop_all
    - sudo ./scripts/sensec.sh stop

F.2.3 Running tests

1. There are two ways of setting up the environment for testing. Either run the startup script by one of the below commands.
    - make test_setup
    - sudo ./scripts/sensec.sh test_setup
2. Or run each of the following commands in a separate shell

- make run_rabbit
- make run_es
- make run_nodejs
- make run_fake_resource

3. Run the tests
    - make test