METRICO SENSOR HUB

ICT-Lab CHIBB

Max Overbeeke (0894897)
INF3A
June 11, 2017

Table of Contents

Introduction

The Metrico sensor hub is built for the CHIBB house at Heijplaat, Rotterdam. CHIBB stands for Concept House Instituut van Bouw- en Bedrijfskunde.

The CHIBB House

The following text is a quote from the website.

Bij CHIBB (Concept House Instituut van Bouw- en Bedrijfskunde) werken studenten onder begeleiding aan het uitwerken van een eigen prototype. CHIBB ontwikkelt een extreem en innovatief concept dat niet alleen duurzaam zal zijn op het gebied van minimaal energieverbruik, maar ook inspeelt op alle facetten van duurzaamheid, dus op grondstofgebruik, Cradle 2 Cradle, leefmilieu, waterhuishouding, ecologie en mobiliteit en wel voor de gehele levenscyclus.

Het project CHIBB wordt komend jaar gerealiseerd. Het gebouw is zodanig ontworpen dat het een nieuwe functie kan krijgen op een andere locatie omdat het gebouw verplaatsbaar is. CHIBB dient tevens als inspiratiebron en testcase op het gebied van duurzaam ontwerpen en gebruik; het lectoraat en de studenten doen onderzoek naar de vele aspecten. De bevindingen van deze testcase kunnen toegepast worden in CHIBB+, een groter gebouw waarvan CHIBB een fragment is.

The ICT-Lab Project

A year ago, MBO TI-students have been asked to design sensors for the CHIBB house. These sensors were to measure variables of the house. Examples are temperature, power usage, humidity, sunlight intensity, etc.

Six months ago, HBO IT-students have been provided with the task to design a hub that collects, store it, and visualize those sensors' data.

A horrible start

After this project's kickoff, we discussed the project assignment with our teachers. It became clear that there were some big challenges ahead regarding sensor data.

Apparently, the sensors that the TI-students had designed, worked on batteries. Thus, all sensors had a maximum uptime of nine hours and then their batteries died. For this reason, all sensors had been removed from the house. This meant that we had no test data at all, except for a few JSON files that were stored on a file server somewhere. In rage, I designed an advanced test data generator.

Next, I noticed that most of the requirements that were summed up in the course guide did not prove any answers to 'why' questions. I made a document as shown in Data simulator.

The Metrico Sensor Hub

The sensor hub has several actors to serve. First, the sensor. The hub must be able to receive data uploads from sensors and store them for later use.

Secondly, the users. A distinction has been made among several user roles. There are three types of roles: visitor, sensor admin and data scientist.

Visitors must be served with some stunning, not boring, visualizations. For this I made a 3D model of the house (see Data simulator).

Sensor admins must be able to create, edit and delete sensors. Enabling and disabling sensors is also part of the functional requirement.

Data scientists have access to a powerful tool called Grafana. This tool is a piece of visualization software that lets users make their own graphs and provide their users with lots of options to do so. Figure 1 is an example of a Grafana dashboard.

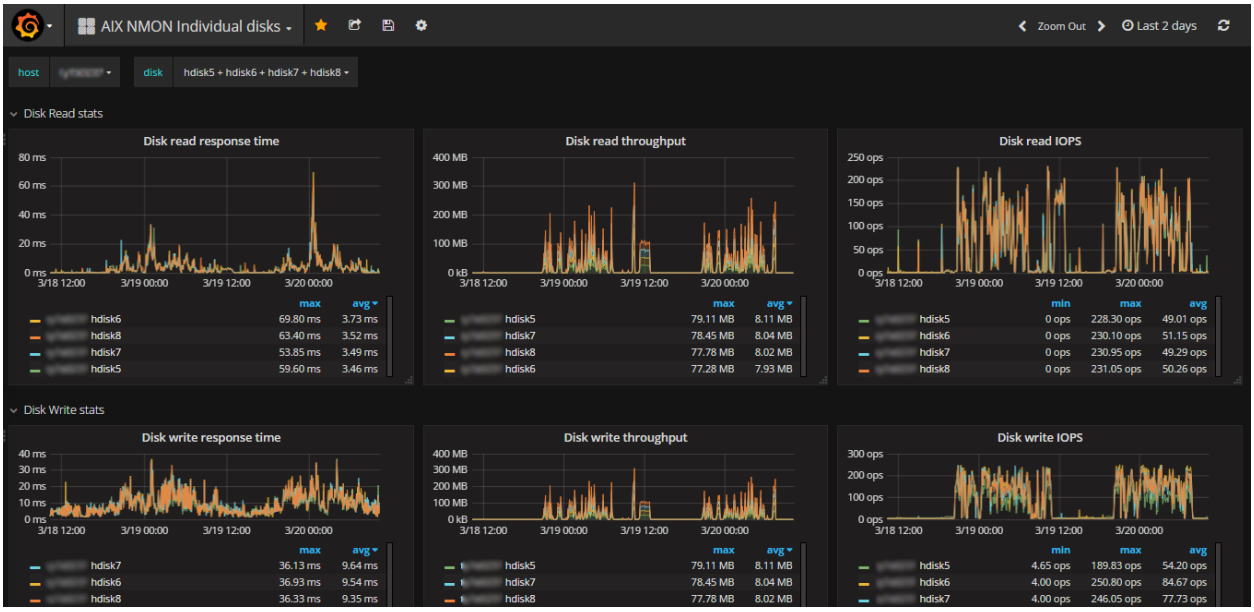


Figure 1 – An example of a Grafana dashboard. In this visualization, the usage of a storage medium is displayed. (source: https://grafana.com/dashboards/1701)

Components

The system is divided in several components. Every single one of them has a single responsibility. This is called a modular architecture. This type of design is useful for collaboration, but also improves the understanding of the system for all parties involved. Errors are easier to find, changes in the source code can be pushed through easier and the system itself will run faster by making use of an event driven framework.

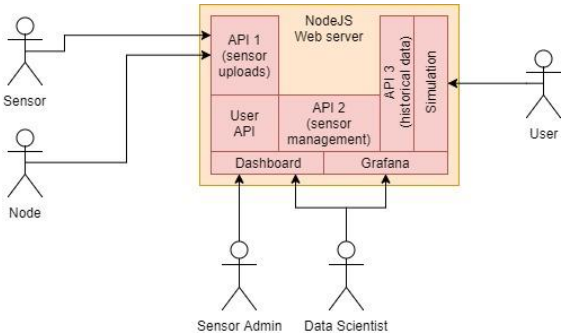


Figure 2 - The top-level architecture of the Metrico sensor hub.

Design choices

When I first looked at the user requirements, a lot of questions arose. I tried designing the system exactly as written in the requirements, which resulted in the following diagram. I was not satisfied and redrew from scratch the design I had in mind. At the time, I wrote the document as shown in Appendix B: Literal translation from user requirements to system design.

I noticed that every single operation invoked a functionality in the server. The server did not operate by itself at all. For this reason, I chose an event driven architecture, which was new for me.

Secondly, I noticed that all functions of the server were to move data around from and to the databases. It'd sometimes check some parameters here and there, format responses, but that was all. It appeared to be an adapter for the databases. Therefore, I chose two powerful databases, and a lightweight HTTP server. One database should handle the relations of users and sensors while the other should store the data points that are uploaded. The lightweight HTTP server should handle traffic and transform data, in addition to being an additional protection layer.

Our product owner, Jan Kroon, told us that they were planning to maybe put a sensor on every square meter of the house if they could handle it. Making a quick assumption that a house's surface area is about 100m^2, that would mean 100 sensors. Worst case scenario, they all send data every second, which results in 100 requests per second. If they're planning to build more houses, this could mean a significant amount of internet traffic. Measurement storage may thus be the biggest challenge here.

I had no idea how many users our product would serve, but I assumed it shouldn't be over 100.000 people. Therefore, I chose for a powerful database, but with a light footprint.

Databases

I've researched SQL databases and time series databases. SQL databases are designed to maintain relationships between tables, and is thus a good use for our users and sensors.

Users are divided by roles and sensors have relations to nodes and sensor types. Nodes are like virtual collections to group sensors together. For example, a hallway can have a certain amount of lights, each of which are connected to a power usage sensor. To avoid confusion among all the other lights in a house, these sensors can be appointed to a virtual node 'hallway'.

Time series databases are up and coming due to the rise in IoT devices. These devices can send data over the internet. These devices also have usage and performance statistics. To monitor these, data is sent to a server every second, minute or hour. Since SQL databases don't work so well with time functions, time series databases are being developed. These databases have a different kind of storage type which increases performance drastically.

Relational (SQL) databases

I considered MariaDB and Firebird databases as a powerful but lightweight SQL databases. The following table contains the features as described by their websites, which were interesting for my case. In the end, I went with MariaDB.

MariaDB	**Firebird**

MariaDB is an enhanced, drop-in replacement for MySQL and is available under the terms of the GPL v2 license. It's developed by the MariaDB community with the MariaDB Foundation as its main steward. **Security** Security is a particular focus for the MariaDB developers. The project maintains its own set security patches on top of MySQL's. For each MariaDB release the developers also merge in all of MySQL's security patches and enhance them if necessary. When critical security issues are discovered the developers immediately prepare and distribute a new release of MariaDB to get the fix out as quickly as possible. **Compatibility** MariaDB is kept up to date with the latest MySQL release from the same branch and in most respects MariaDB will work exactly as MySQL. All commands, interfaces, libraries and APIs that exist in MySQL also exist in MariaDB. There is no need to convert databases to switch to MariaDB. MariaDB is a true drop in replacement of MySQL! Additionally, MariaDB has many nice new features that you can take advantage of.	Don't be fooled by the installer size! Firebird is a fully featured and powerful RDBMS. It can handle databases from just a few KB to many Gigabytes with good performance and almost free of maintenance! Below is a list of some of the Firebird's major features: • Full support of Stored Procedures and Triggers • Full ACID compliant transactions • Referential Integrity • Very small footprint • Fully featured internal language for Stored Procedures and Triggers (PSQL) • Almost no configuration needed - just install and start using! • Big community and lots of places where you can get free and good support • Dozens of third party tools, including GUI administrative tools, replication tools, etc. • Careful writes - fast recovery, no need for transaction logs! • Many ways to access your database: native/API, dbExpress drivers, ODBC, OLEDB, .Net provider, JDBC native type 4 driver, Python module, PHP, Perl, etc.

Other popular databases are Oracle, MSSQL and PostgreSQL. I chose not to use Oracle and MSSQL because of their big footprint and because they're more fitted for major projects with lots of relations and operations. To be honest, I think PostgreSQL could be switched with MariaDB any time, since I feel they are similar. I just went with MariaDB, because I did not care too much about the SQL database, as long as it were stable.

Time series databases

I've looked at three different TSDB engines. I had already worked with InfluxDB and Prometheus. I considered Prometheus, but dropped it eventually since it has a wicked system. Prometheus is made to scrape data from IoT devices, which themselves must be configured as HTTP servers that serve Prometheus. Instead, I want to upload metrics as soon as sensors send them.

The other one was DalmatinerDB. It peeked my interest because of the easy integration with Grafana and its Riak core, used for scalability and resilience. However, I dropped it when I learnt it was still in beta phase (v0.3).

In the end, I went with InfluxDB, which has an awesome storage engine (which is truly worthy of reading about), supports an incredible concurrency level and is highly available with InfluxDB Relay.

Web framework

I decided that using two databases would work the fastest. One database will handle metric data, aggregation and calculations, while the other will be used for user logins, access determination and sensor management.

Keeping this in mind the web server has not a lot of operations to make. It only must forward and transform requests into the appropriate database queries to receive the data the user asks for. This means the programming language of our choice should be able to swiftly communicate with our databases and users, but does not perform heavy calculations. Based on this requirement I compared frameworks and languages like Node.JS and PHP. I also checked for more general-purpose programming languages and their pros and cons. Below you will find a table with a mixture of subjective and objective arguments.

Node.JS

Pros	**Cons**
V8 engine written in C by Google	Single threaded
Async	
Cross platform compatibility	
Real time	
Both frond-end and backend developing	
Over 400k modules	

Python/Django

Pros	**Cons**
Multi paradigm	Relatively slow compared to .NET or Java
Clean and quick code	Scopes might slow down productivity
Lots of modules	

PHP

Pros	**Cons**
Both functional and OOP	Single threaded
Easy database integration	Weak dynamic typing system
Very easy to learn	Updates for bugs creates new bugs
Subjective: Easy integration with HTML but scrambles up document and project structure	
Subjective: Syntax may be called either ‘easy’ or ‘disgusting’	

Ruby on Rails

Pros	**Cons**
Lots of modules and tooling options	Weak multithreading
Test automation is second nature	Hard to find good documentation
Clean code and therefore more productivity	Somewhat slow compared to Node.JS or Go

ASP.NET

Pros	**Cons**
Highly scalable	Not free
Multi paradigm	

Go

Pros	**Cons**
Great concurrency	Still immature
Built in HTTP library	Limited problem solving options compared to mature programming languages
HTML templates with safe string population	
Easy to read and write	
Native	

Side note

I found out that Pug is a script-like language that simplifies HTML into a tabular structure. It should make developing html structure faster, as it does not require opening and closing tags, but instead relies on whitespaces.

Conclusion

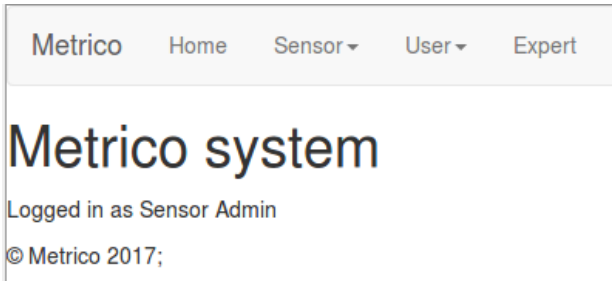
Personally, I'm curious about Node.JS as it is widely used for web development and can handle client side and server side code simultaneously. The only thing I'm worried about is scalability although the server can be easily scaled by using a load balancer or cluster. Since I was already planning to use the database for heavy computations, Node.JS does not have to pick up that part, and that is exactly what it's weakness is.

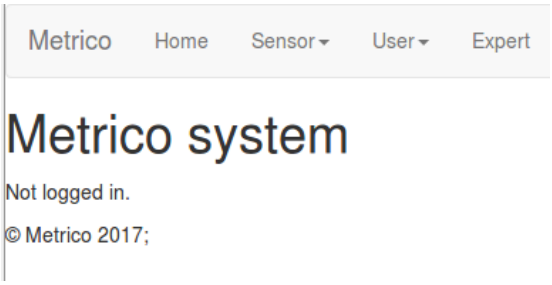
The runners up are Go and ASP.NET. Although Go is generally not used for web development it's a great server language with the basic necessary tools for web development. So is .NET, but with its ASP.NET framework, a lot of fantasies can be made real. This adds an extra layer of complexity though and because it is not free, it is important the Product Owner agrees on using this platform.

Views

In the next section, we will cover the designed views and how they work.

Index



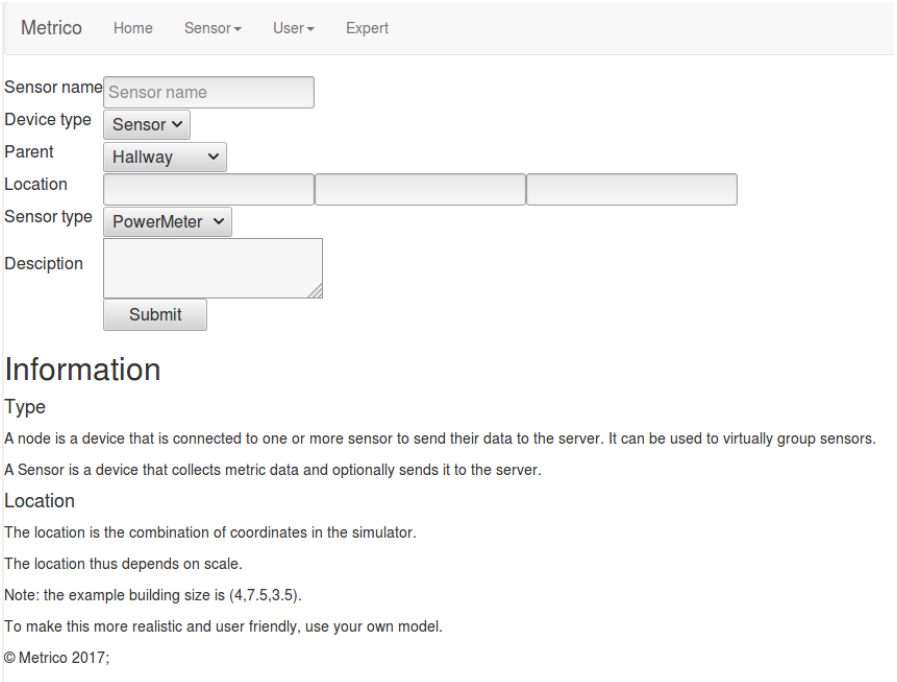


On the home page, the login status will be shown to the user. For now, this is all a user can see on the home page. I planned that the simulator would go beneath the 'logged in' line, but I could not get WebGL to work in time.

Sensor

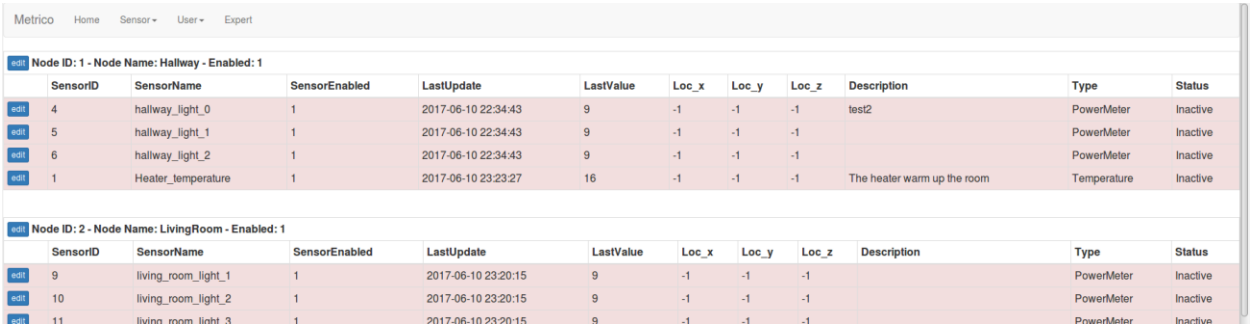
Under this dropdown menu, users will find two options: register and edit. These pages are only accessible as sensor admin.

Register

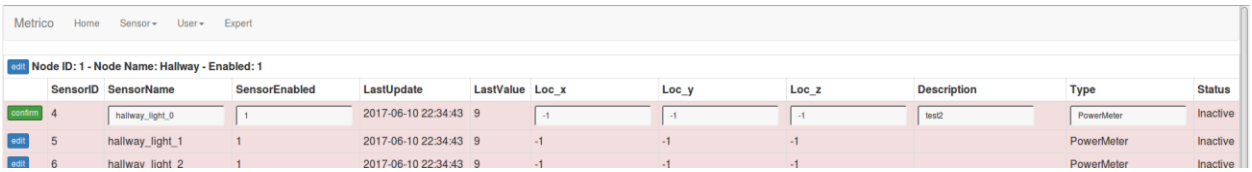


On the register page, sensor admins can add new sensors. Parents and sensor types are pulled from the database and loaded as options. User input is verified client side as well as server side, to ensure no internal errors occur.

Edit



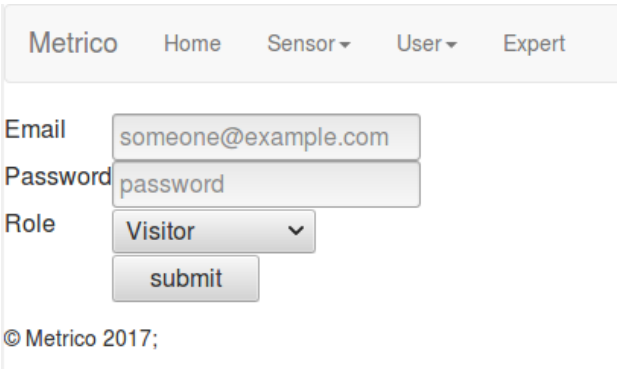
On the edit page, sensors can be viewed and edited. Only values that are logically changeable can be changed.



User

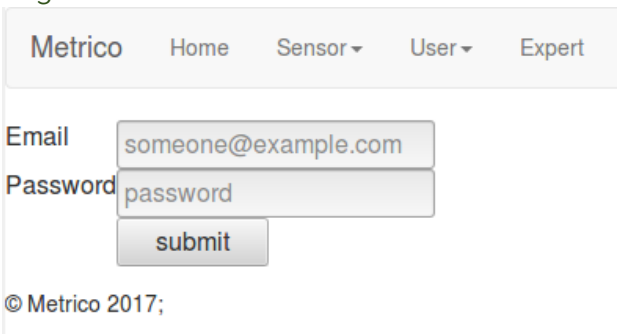
When a user clicks on this drop-down item, two options can be selected: register and login.

Register



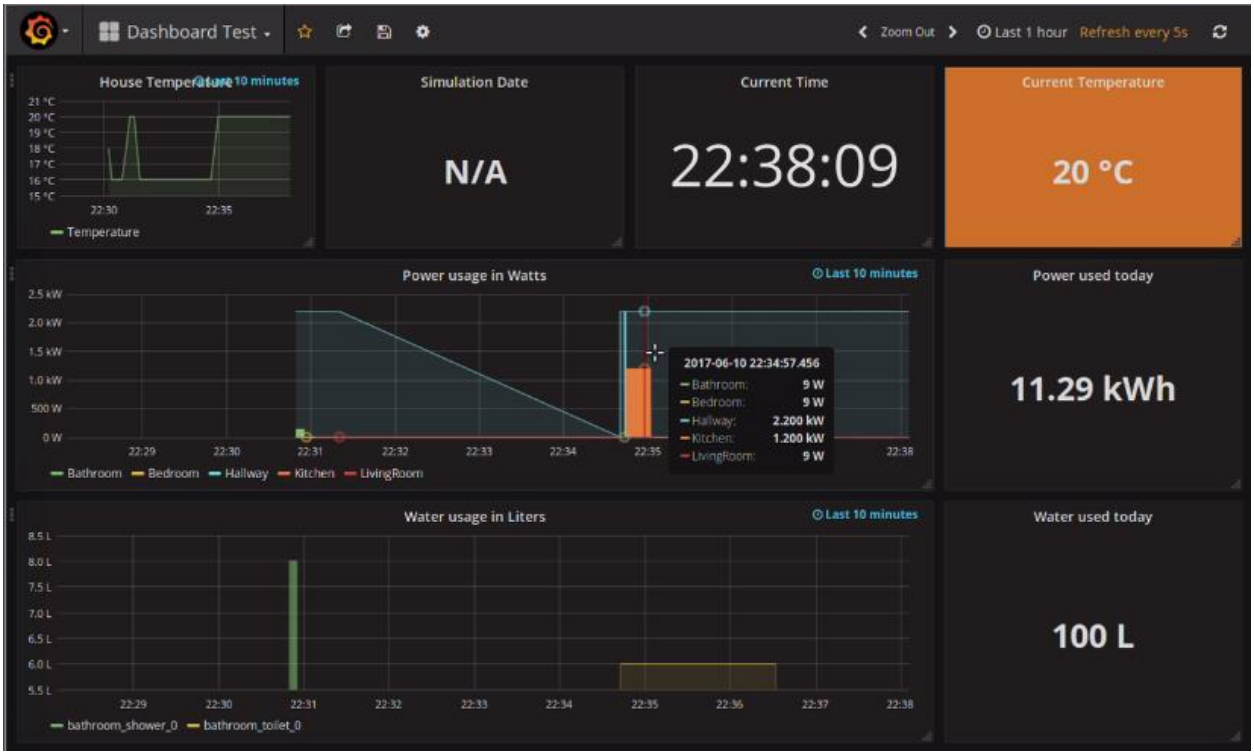
A user must be registered with their email, password and role. Roles besides visitor should be verified by an administrator, but to implement that functionality took too much time.

Login



The login page looks a lot like the register page, which is only natural because a user is logged in with their email and password. When successfully logged in, an access token is sent and will be used for pages that require authentication. Also, the role the user has is set as a cookie, but this is used for visual purposes only.

Expert



The expert button links to the Grafana server. This could be protected with a proxy server. This way, users can not be redirected to Grafana unless they are logged in. Grafana has a HTTP library available.

Data generator

The data generator has been created for generating test data. Since I had no real data available, I decided to create a sims-like program that also uses sensors. The source code is included in the same zip file as this document.

It uses a person's basic needs:

- Hunger
- Fun
- Sleep
- Toilet need

Based on these levels, the simulated person, here called 'Sapphira-chan', acts in their house.

You can see that at 17:30 she entered the hallway. Probably because she just returned from work. The lights in the hallway turn on.

Then she notices she needs to use the bathroom, thus she moves to the bathroom, where the lights turn on as well. In the hallway, the lights are turned off.

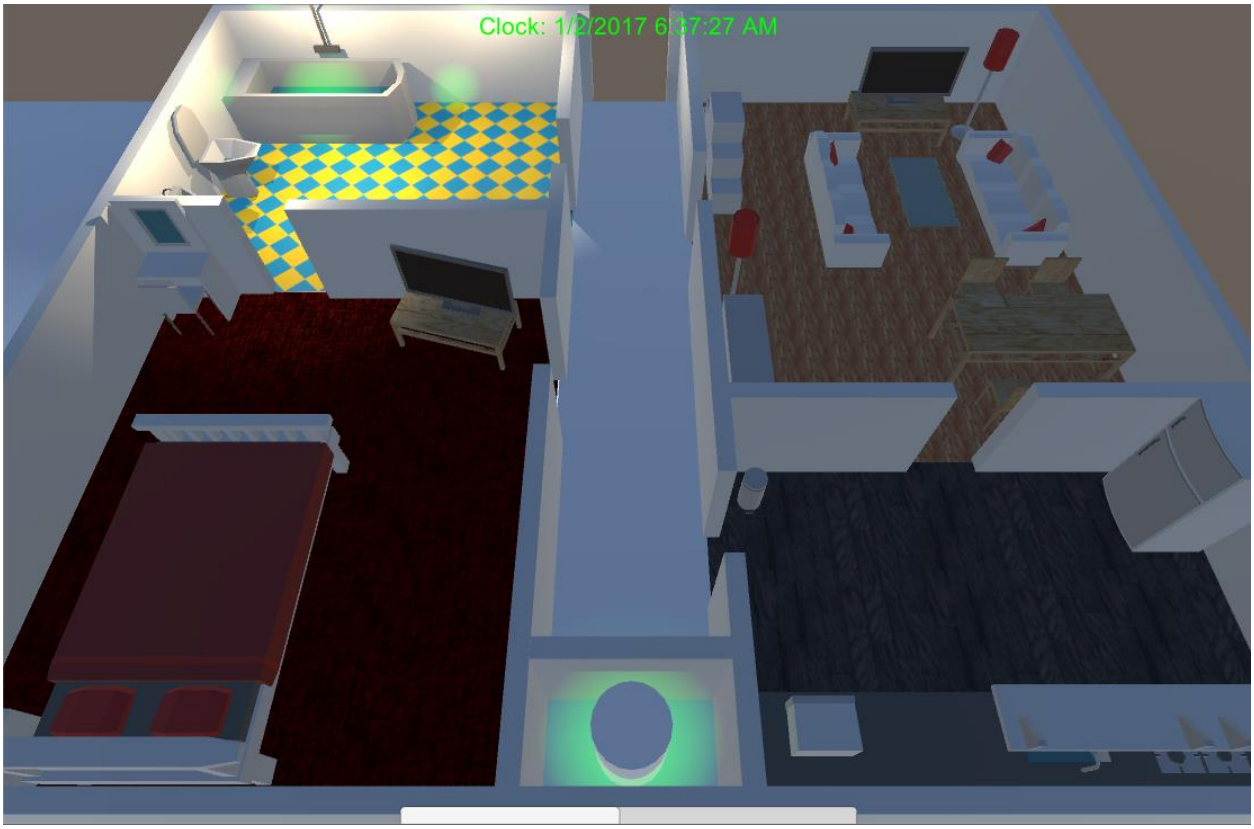
So, she uses the toilet for a minute or two and flushes. All this is monitored by sensors.

```
Sapphira-chan entered the Hallway
activated hallway_light_0.
activated hallway_light_1.
activated hallway_light_2.
[Mon 02-01-2017 17:31:00] Sapphira-chan is thinking.
[Mon 02-01-2017 17:31:00] Sapphira-chan uses the toilet.
Heater is running (room temp: 16).
Sapphira-chan left the Hallway
deactivated hallway_light_0
deactivated hallway_light_1
deactivated hallway_light_2
Sapphira-chan entered the Bathroom
activated bathroom_light_0.
activated bathroom_light_1.
activated bathroom_light_2.
activated bathroom_light_3.
[Mon 02-01-2017 17:32:00] Sapphira-chan does go_bathroom.
Heater is running (room temp: 16).
Average requests per second: 5.933333
[Mon 02-01-2017 17:33:00] Sapphira-chan starts doing __use_toilet.
Heater is running (room temp: 16).
[Mon 02-01-2017 17:34:00] Sapphira-chan is doing __use_toilet.
Heater is running (room temp: 16).
[Mon 02-01-2017 17:35:00] Sapphira-chan finishes __use_toilet.
Heater is running (room temp: 16).
activated bathroom_toilet_0.
[Mon 02-01-2017 17:36:00] Sapphira-chan does __start_flush.
Heater is running (room temp: 16).
deactivated bathroom_toilet_0
[Mon 02-01-2017 17:37:00] Sapphira-chan does __stop_flush.
Heater is running (room temp: 16).
Sapphira-chan left the Bathroom
deactivated bathroom_light_0
deactivated bathroom_light_1
deactivated bathroom_light_2
deactivated bathroom_light_3
Sapphira-chan entered the Hallway
activated hallway_light_0.
activated hallway_light_1.
activated hallway_light_2.
[Mon 02-01-2017 17:38:00] Sapphira-chan does go_back.
Heater is running (room temp: 16).
[Mon 02-01-2017 17:39:00] Sapphira-chan is thinking.
[Mon 02-01-2017 17:39:00] Sapphira-chan 's hunger level is 310.
[Mon 02-01-2017 17:39:00] Sapphira-chan prepares dinner.
Heater is running (room temp: 16).
Sapphira-chan left the Hallway
deactivated hallway_light_0
deactivated hallway_light_1
deactivated hallway_light_2
Sapphira-chan entered the Kitchen
activated kitchen_light_0.
activated kitchen_light_1.
activated kitchen_light_2.
activated kitchen_light_3.
activated kitchen_light_4.
[Mon 02-01-2017 17:40:00] Sapphira-chan is cooking for 45 minutes..
[Mon 02-01-2017 17:40:00] Sapphira-chan uses the stove for 36 minutes..
activated kitchen_stove_0.
[Mon 02-01-2017 17:40:00] Sapphira-chan starts doing <lambda>.
Heater is running (room temp: 16).
```

Later, as you can see, she starts preparing dinner and uses the stove. This stove uses electricity, not gas, which is also monitored. This way, I could get a significant amount of various data, that may lead to finding correlations.

Side note: All correlations are programmed by myself, so it's not that special anymore.

Data simulator



The simulator is made in Unity3D 5.6. Programs made with Unity can be exported as WebGL, which means you can run your programs in the web browser. Unfortunately, I could not get this to work in time. Above you will find a snapshot of the simulation.

Appendix A: User requirements

GATHERED CASES FROM 'DETAILED REQUIREMENTS' AS SPECIFIED IN 'SENSOR DATA HUB' AT FEBRUARY 16, 2017. COLLECTED 21(+1) QUESTIONS REGARDING 11 REQUIREMENTS (MARCH 2, 2017).

Sensor hub

This part contains the requirements concerning uptime and DNS.

The hub should be highly available ("99.99% uptime") [**REQ_1**].

The hub is split in three parts (Api**X**-sensorhub.hr.nl) [**REQ_2**]:

- API 1 will be used for sensors to upload data.
- API 2 will be used to manage sensor nodes.
- API 3 will be used to retrieve actual and historical sensor data.

Questions

1) Regarding REQ_1:
   - Why must the sensor hub be highly available?
   - Which API must be highly available?
   - To what extend is 99.99% achievable?
2) Regarding REQ_2:
   - Why use three different APIs if all can be done within one?
   - Where will the dashboard be located?
   - Doesn't this collate with other groups' implementations?
   - In what format must API 3 send its data?
   - Is API 3 responsible for visualizations?
   - In what API is the dashboard integrated?

Sensors and Data

This part contains the requirements concerning sensors and the data they will transmit.

Sensor data upload

A sensor should be able to upload data to an API. A packet must contain:

- One OR more data point(s) [**REQ_3**],
- A token representing the sensor [**REQ_6**].

A sensor must be identified as a sensor [**REQ_6**]. To upload data, the following must be done:

1) If a sensor is registered, jump to 4.
2) Register at *Sensor Data Hub*.
3) Receive a token to send with every data upload.
4) Upload data with the token and one or multiple data points.

Sensor data storage

Data points (in general) must be stored for future access (historical data) [**REQ_4**].

A data upload must contain the items specified in 'Sensors and Data

This part contains the requirements concerning sensors and the data they will transmit.

Sensor data upload'. If the data does not contain the necessary items, the upload will be ignored [**REQ_6**].

Sensor status [REQ_5]

- A sensor can be 'active'.
- - When a sensor has not sent data for two or less **intervals**.
- A sensor can be 'intermittent failures'.
    - When a sensor has not sent data for three **intervals**.
- A sensor can be 'inactive'.
    - When a sensor has not sent data for thirty **intervals**.

Dashboard and Visualization

This part contains the requirements concerning dashboard access, visualization and management.

Dashboard access

The dashboard will have public and private views [**REQ_7**].

- There must be a public view for **some** sensor data,
- There must be a **private view** for user with an account.

Questions

3) What is SOME sensor data for the public view, IF NOT assuming REQ_8?
    - What type of data?
    - Must the data be an aggregation or average of multiple sensors' data?
    - Must the data be raw or visualized?
4) What must the private view contain, IF NOT assuming REQ_9?
    - Does it contain the same visualizations as the public view?
    - Are there any functionalities like sensor management available?
    - Are those functionalities bound to a specific account type (admin/manager/student)?

Dashboard views

There is a dashboard view for the public containing [**REQ_8**]:

- Dynamic visualizations of the actual situation AND historic trends.
- Graphs SHOULD NOT contain mathematical graphs, but SHOULD contain heat maps or 3D-models.

There is a dashboard view for experts containing [**REQ_9**]:

- Visualizations of single data points over time from one sensor.
- Visualizations of correlations between different sensor measurements.
- Visualizations of averages of measurements where applicable and useful.
- Visualizations must be interactive.

Questions

5) Regarding REQ_9:
   - Must the visualization for experts be customizable?
     - IF YES, then may we use sophisticated dashboards to give experts access to use the gathered data to its full extent (e.g. Grafana)?
   - Must the customized view for experts be saved, so it can be loaded when the site is visited again?

Dashboard tweaks

Dashboard visualization charts must be adjustable to a selected time range (today, last week, last month, etc.) [**REQ_10**].

Data visualization gives more insight if combined with other datasets (e.g. room temperature + weather events) [**REQ_11**].

Questions

6) Regarding REQ_10:
   - Must the setting of a time range be saved, so it can be loaded when the site is visited again?
7) Regarding REQ_11:
   - Must users be able to upload 3rd party datasets?
     - If YES, then must the 3rd party data
   - Must the data be retrieved from (weather) APIs?
8) Regarding the **course guide**:
   - The course guide points out that a user manager must be able to create, remove and stop sensor nodes (Dashboard, bullet 1).
     - Should users be able to EDIT in addition to *create* and *remove*?
     - Should users be able to START, in addition to *stop*?
     - What processes represent the actions mentioned above?
     - Who is a manager?
     - Should users be appointed to be managers?
       - Who is eligible to do so?

Requirements

Requirement	Priority	Description
REQ_1	M	The Hub should work 99.99 % of the time.
REQ_2	M	There will be an api1-sensorhub.hr.nl (at this moment 145.24.222.139) that can be used by sensor nodes to upload new sensor measurements, an api2-sensorhub.hr.nl that can be used for sensor node management and an api3-sensorhub.hr.nl that can be used by different front-end applications to retrieve actual and historical sensor data.
REQ_3	M	A sensor data upload can contain one or more sensor node measurements.
REQ_4	M	All sensor measurements are stored for future use as historical data.
REQ_5	M	If a sensor node has not sent sensor data for more than 3 normal intervals it gets status 'intermittent failures', in case sensor data is not received for more than 30 normal intervals it gets status 'inactive'.
REQ_6	M	It should not be possible that everyone or everything can upload data to the Sensor Data Hub. Someone who wants to send data to the Sensor Data Hub, should first register and receive a token that should be sent with every sensor data message. API calls without a valid token should be ignored.
REQ_7	S	Not everyone is entitled to see all sensor data / graphs / visualizations. There should be a public view with some sensor data and private views for viewers with an account like for example the architecture students or the owner of the house.
REQ_8	S	For interested visitors there should be dynamic visualizations that show the actual situation of CHIBB and historic trends. Preferably these are not mathematical graphs, but floorplans with heat maps or 3D-models.
REQ_9	S	For experts the overviews should not be just graphs of a single sensor, but also graphs that show the correlation between different sensor measurements or the temperature distribution in CHIBB calculated from multiple temperature sensors at different locations. Preferably these graphs are interactive, i.e. experts can choose what they want to see.
REQ_10	S	It should be possible to quickly switch between different historic views, like today, last week, last month, last year etc. Look for inspiration at how Morningstar gives views on stock prices of companies: http://www.morningstar.com/stocks/xnas/msft/quote.html
REQ_11	S	Data gives more insights if the can be compared with other data. The data of temperature sensors will be more meaningful if they are compared with general weather data (KNMI) or data of the weather station on the roof of the CHIBB concept house.

Appendix B: Literal translation from user requirements to system design

In the previous document, we looked at the requirements that are given for this project and now we sketched it in a system-like way, which resulted in the diagram Figure 4 - System design from requirements. In this document, we will consider the numerous errors made while drafting requirements for this project and suggest improvements for the CHIBB monitoring system. Side note: the design contains a lot of assumptions on connecting data streams.

Summarizing flaws

The first notable flaw is the specification of three subdomain APIs with different purposes. Each having one use case specified per API while the rest of the requirements do not specify where the remaining functionality belongs. As shown in the diagram, the sensor data management API has literally no use except for tunneling sensor data from the central server to the dashboard.

Per the course guide, the sensors are to be created, removed and stopped by users. Editing and starting are no part of the requirements, which are logically linked.

User information and privileges

Meanwhile, the dashboard has two different data streams, one of which is the sensor data that is being tunneled. The other one is user information (register requests and login requests). The user information data stream is not described in any of the requirements. Furthermore, it is not specified which registered user can view/edit sensor status. This means that I can delete every sensor from the system if I am registered.

Conclusion

The grouping of the APIs has gone horribly wrong; a lot of use cases remain uncovered in the requirements and users can abuse their privileges as they like.

The following suggestions are to be strongly followed if users should be able to work securely and the system should be fast.

- There should be at most 1 API. Just one for the sensors' data and a separate project for the users' dashboard.
- The dashboard should be connected to the server directly, using database roles to ensure safety (e.g. the dashboard can only read and the sensors can only write).
- Users must be grouped into specific user roles to ensure correct privileges (sensor admin, data analyst, etc.).

When these three requirements are followed, Figure 5 - System design as advised will appear.

Suggestions for improvements

Notable changes are the two access layers over the database. The first wall is a user access control layer which checks for privileges of users and denies permissions where not granted. Optionally, we can move error handling and RDBMS related tasks to the RDBMS itself, but reading and writing to the RDBMS will require a two-step verification (firstly the user login and secondly checking privileges of the user in question). The user

and sensor databases in the RDBMS will be separated to ensure sensors will never be able to access user data. Users, however, if privileged can access sensor data if necessary (changing sensor status). This is still a questionable requirement, as you probably never will want to ignore incoming data.

The second wall is the sensor access control. This access layer manages the incoming write requests coming from sensors, along with the read requests from users. The sensor access control layer is placed on the time series database. It will be a pass-through for new data and requested historic data. The access control layers in depth will look something like the schema below.

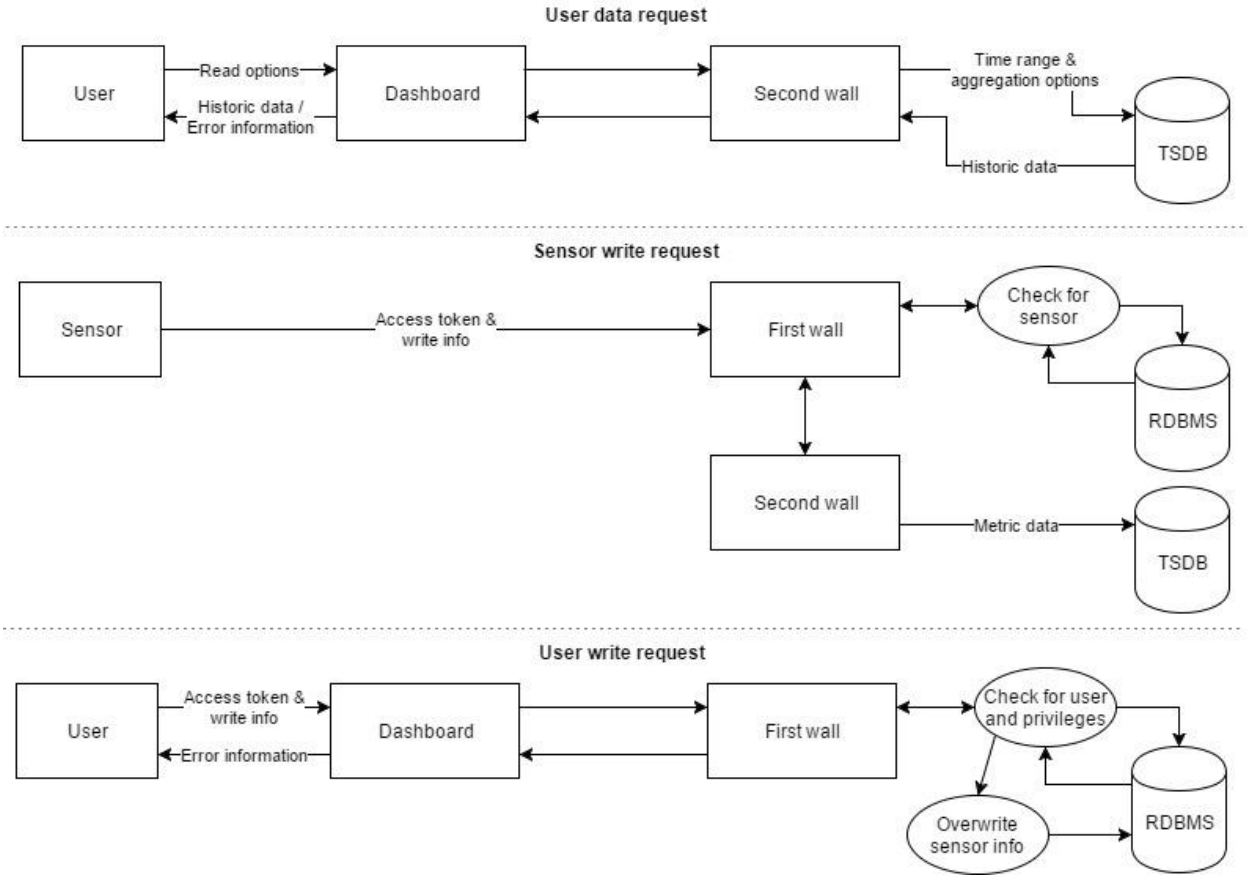


Figure 3 - User access control layers

The top case represents a user requesting metric data from the server (e.g. metrics for models for public view). The middle case represents a sensor requesting to write new metric data to the TSDB. Though it is not visible in the diagram, if the action of checking for an existing sensor fails, the metric data is discarded. The bottom case represents a user requesting to change a sensor's status.

System designs

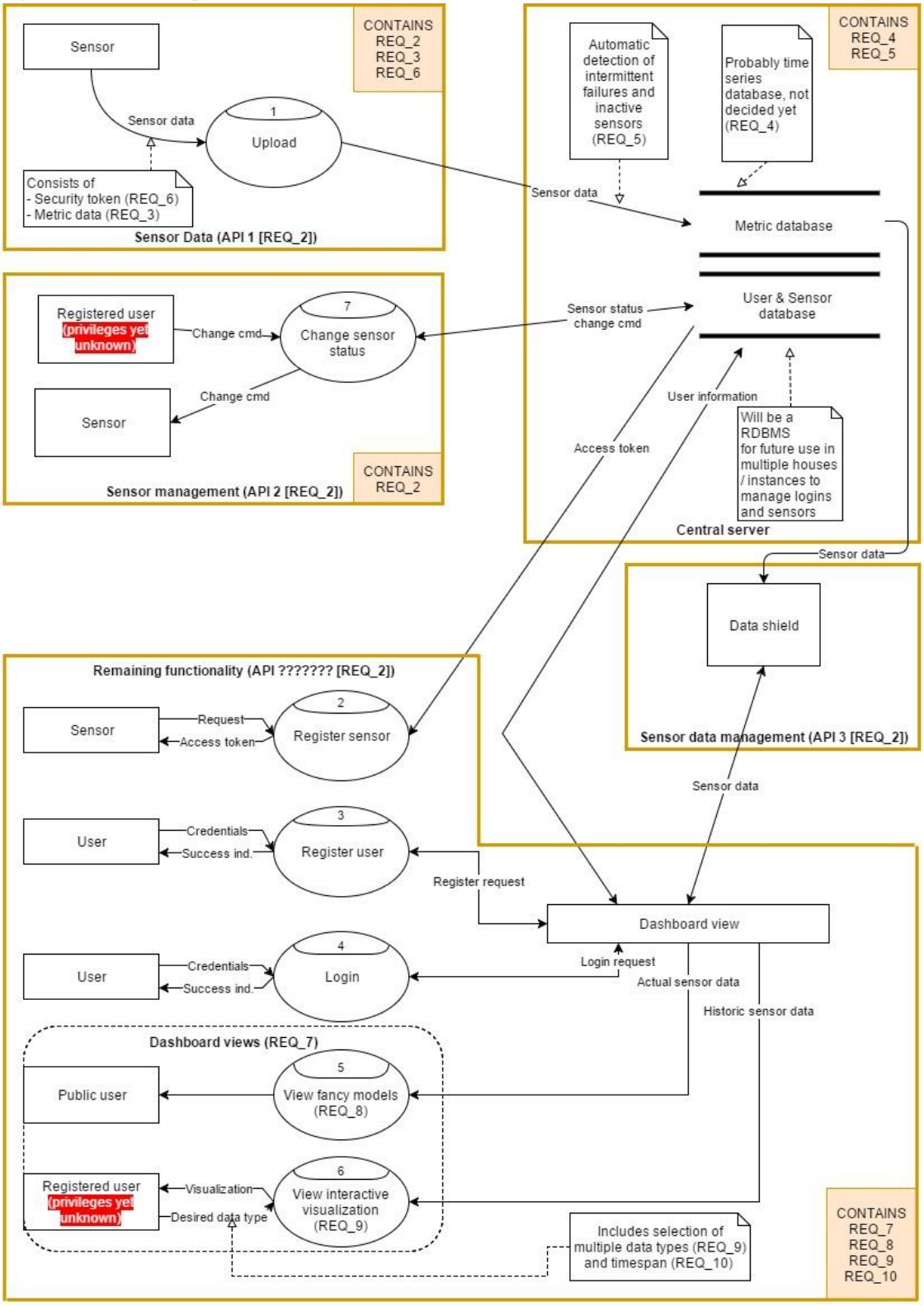


Figure 4 - System design from requirements

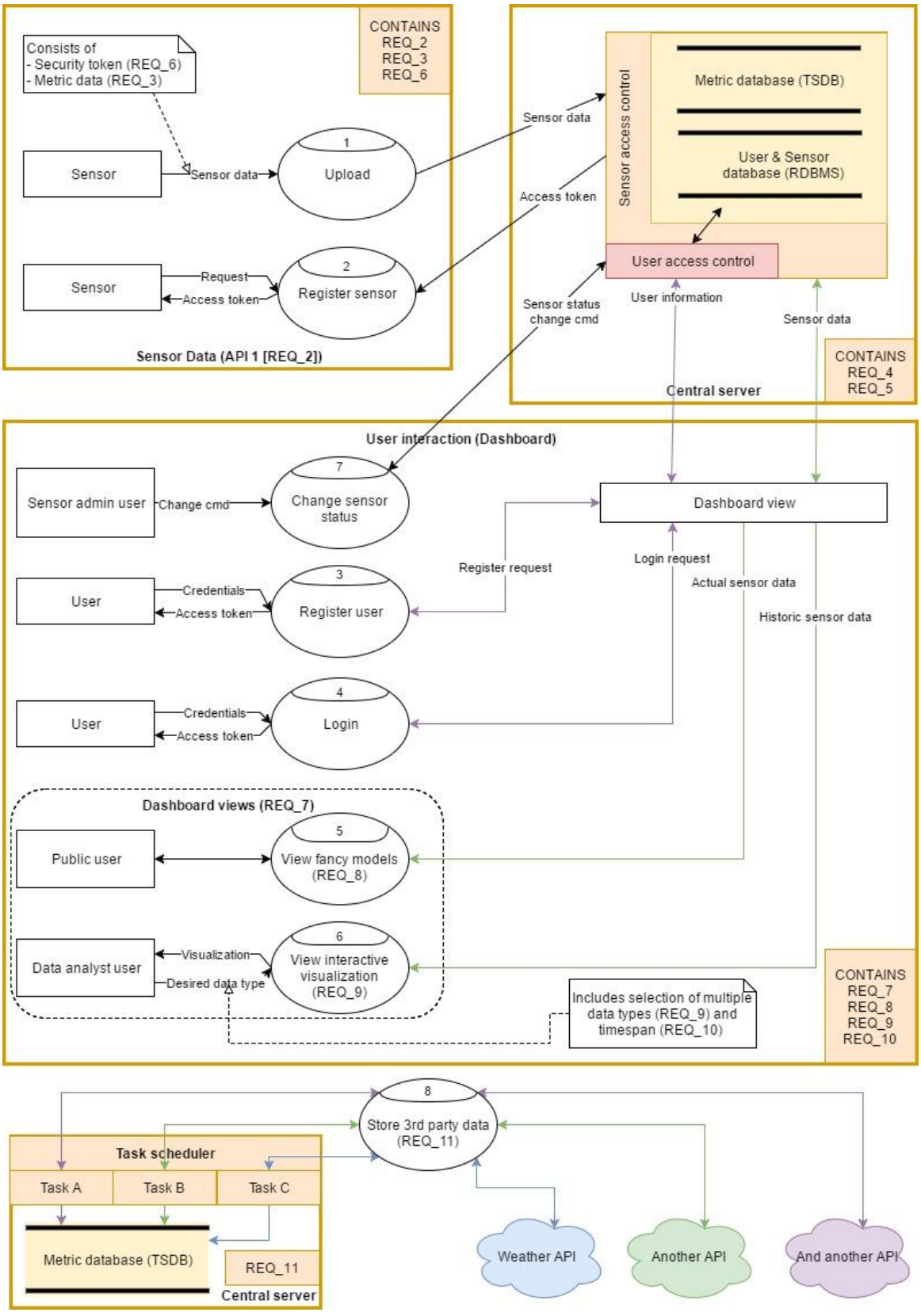


Figure 5 - System design as advised

Appendix C: Use cases and data streams

In the following document, we will see which data streams and related use cases exist in the system as it is sketched right now.

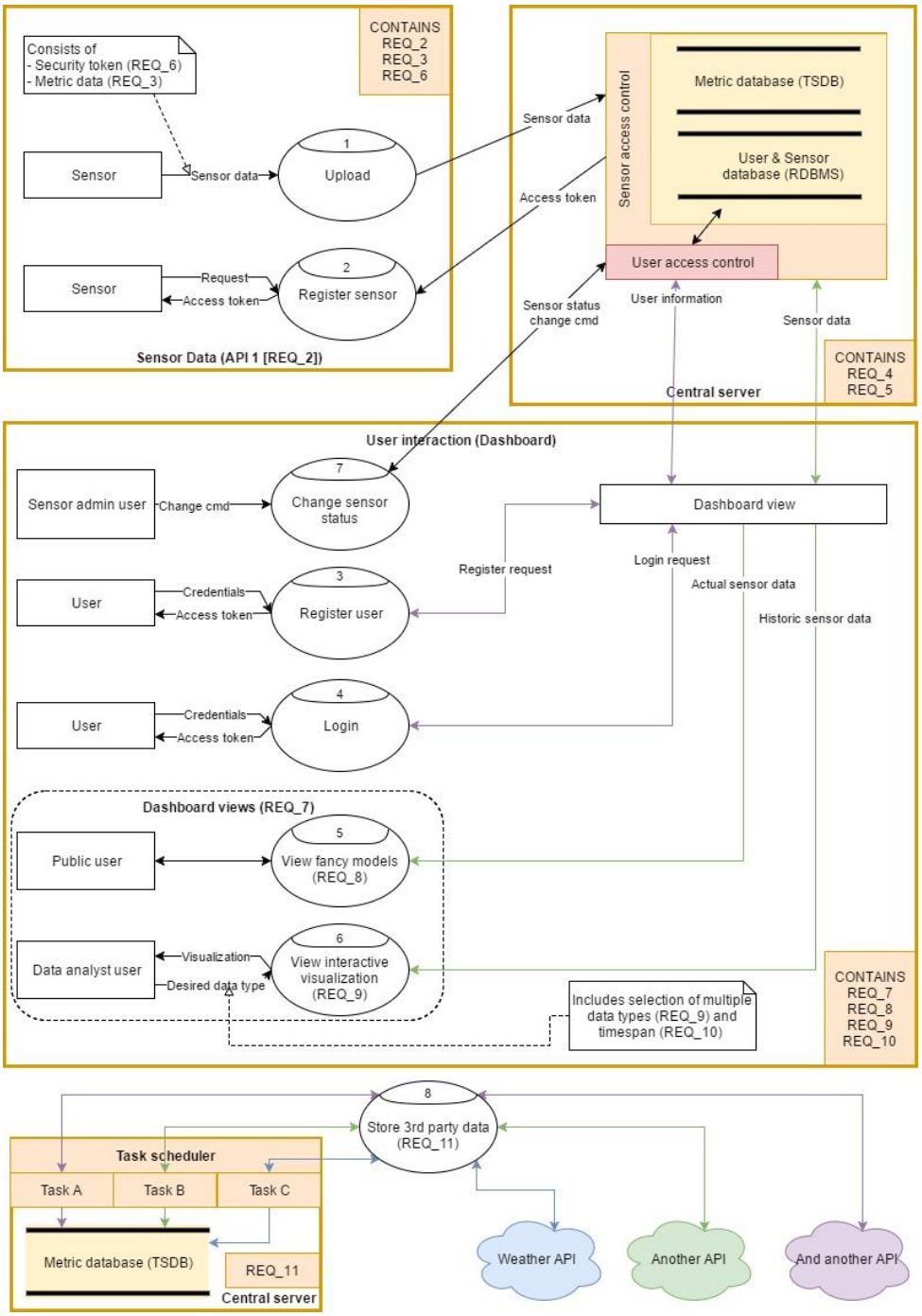


Figure 6 - System design sketch

All use cases are numbered. We will go over them one by one writing out how the data flows from A to B and what the process will look like.

Upload

Functional requirement: A sensor can upload **data** which must be valid and **stored** in a **metric database**.

This requirement is derived from requirements 3, 4 and 6.

The data consists of a:

- Security token, which is received by the Register sensor function,
- Metric data (one or more points), which is received from a sensor.

The requirements of storing valid data will be as follows:

- The security token must be checked;
    - If it is valid, the data must be stored,
    - If it is invalid, the data must be ignored.

The database must be a metric database, because the data will be saved as a combination of a timestamp and a metric value, which technically means a measurement (thus an integer or real number). For this reason, time series databases are strongly advised.

Related Requirements

Requirement	Priority	Description
REQ_3	M	A sensor data upload can contain one or more sensor node measurements.
REQ_4	M	All sensor measurements are stored for future use as historical data.
REQ_6	M	It should not be possible that everyone or everything can upload data to the Sensor Data Hub. Someone who wants to send data to the Sensor Data Hub, should first register and receive a token that should be sent with every sensor data message. API calls without a valid token should be ignored.

Register sensor

Functional requirement: A sensor must be registered to submit sensor data.

There are no requirements or limits on giving out access tokens by the requirements that are given. There are two options regarding registering.

Open (No authentication)

Registers can be added by simply visiting a URL and receiving an access token instantly.

Closed (Two-way authentication)

The sensor admin can add a sensor by ‘creating’ a new sensor in the dashboard. This function shows a code of four to eight digits. The sensor must then visit a URL POSTing the code to authenticate and register. Then, the access token is given to the sensor. This means there are two stages and thus two URLs.

Related Requirements

Requirement	Priority	Description
REQ_6	M	It should not be possible that everyone or everything can upload data to the Sensor Data Hub. Someone who wants to send data to the Sensor Data Hub, should first register and receive a token that should be sent with every sensor data message. API calls without a valid token should be ignored.

Register & Login user

Functional requirement: A user must be registered and logged in to see expert data and/or manage sensor nodes.

There are no requirements or limits on users registered and their permissions. However, it is strongly suggested to implement some form of limiting privileges, to avoid misuse of sensor administration when logged in.

This requirement is necessary to meet requirements 7, 8 and 9.

Requirement	Priority	Description
REQ_7	S	Not everyone is entitled to see all sensor data / graphs / visualizations. There should be a public view with some sensor data and private views for viewers with an account like for example the architecture students or the owner of the house.
REQ_8	S	For interested visitors, there should be dynamic visualizations that show the actual situation of CHIBB and historic trends. Preferably these are not mathematical graphs, but floorplans with heat maps or 3D-models.
REQ_9	S	For experts, the overviews should not be just graphs of a single sensor, but also graphs that show the correlation between different sensor measurements or the temperature distribution in CHIBB calculated from multiple temperature sensors at different locations. Preferably these graphs are interactive, i.e. experts can choose what they want to see.

View models

Functional requirement: A **non-registered user** should be able to **view models** of the **actual situation** of the CHIBB house. These models should be a form of 3D models or floorplans with heat maps.

This requirement is derived from requirement 8.

Related Requirements

Requirement	Priority	Description
REQ_8	S	For interested visitors, there should be dynamic visualizations that show the actual situation of CHIBB and historic trends. Preferably these are not mathematical graphs, but floorplans with heat maps or 3D-models.

View interactive visualization

There are extraordinary tools available for creating stunning visualizations. Fine examples are Grafana and Kibana.

Figure 7 - An example of a Grafana dashboard

Figure 8 - An example of a Kibana dashboard

In the figures above you see a lot of different types of graphs and calculations. All of this is available for free and would be an incredible asset for expert data scientists. Dashboards like these can also be created and saved by users themselves to be viewed later or in real time.

Naturally we could dive into the world of D3JS, also creating beautiful graphs. However, these graphs are not customizable by the user and every single piece of the graph must be coded by hand. This will cost us a considerable amount of time compared to using advanced software to visualize the many different data points we receive from sensors.

The best interest is probably to pick the best of both. We could generate some graphs or KPIs on the public dashboard, just for fun for the public. On the other hand, the data scientists can get to work with the advanced software.

Change sensor status

Adding and removing sensors are trivial. In the paragraph Register sensor we explained the possibilities of adding sensors. Removing sensors would be the act of removing an access token or declaring it temporarily invalid. Users must have the appropriate privileges to do so, which is explained in more detail in the paragraph Accounts with privileges and user roles (REQ_7).

Store 3rd party data

Storing 3rd party data will be done by automated tasks that can be scheduled. The tasks will have direct access to the databases (because they are not considered outsiders). All the work of extracting, parsing and uploading will be done by these tasks themselves, or a global task manager if possible.

Non-functional requirements

High availability (REQ_1)

Non-functional requirement: The system must be highly available (99.99% uptime).

Per the calculations by Wikipedia, 99.99% uptime means 8.66 seconds per day or 4.38 minutes per month downtime. We will need to search for databases and web servers that are high availability-ready and support clustering/replication, such as InfluxDB's Relay or PostgreSQL' pgpool-II.

System design (REQ_2)

Non-functional requirement: The system must be split into three APIs.

Questionable. API 1 is for sensor uploads. This is fair. Sensor node management doesn't need an API as far as our knowledge goes. If API 3 should be used for downloading raw data, fair enough, but still doesn't make sense, because that's not what the system is for.

The system design as shown in Figure 6 - System design sketch is highly recommended to avoid horrible designed data streams and twisted links between multiple components. Sensor node management can be integrated in the dashboard and made available for sensor admins. Downloading raw data isn't the problem and surely can be made available for public (or anyone with privileges), but since the data is already present in databases, there is no reason for the API to be there.

Sensor status updates (REQ_5)

Sensor status updates can be either static or dynamic in the following ways.

Static

Every time a metric upload is valid and added to the metric database, a notification will be sent to the RDBMS to update the sensor's 'last update' field. When a list of nodes is requested by the dashboard, that field is checked for the appropriate status *(Active, Intermittent failures, Inactive)*.

Dynamic

When a list of nodes is requested by the dashboard, all timestamps from the last submitted valid metric from each sensor are checked and given the appropriate status in the dashboard.

Figure 9 - Static vs Dynamic status updates compares the two approaches.

The most optimal situation for performance and independency of the writing and reading mechanisms is reached during the Static form. In this form, there is no dependency on data which is required from multiple sources so the waiting time is reduced, which results in a better overall performance for the system.

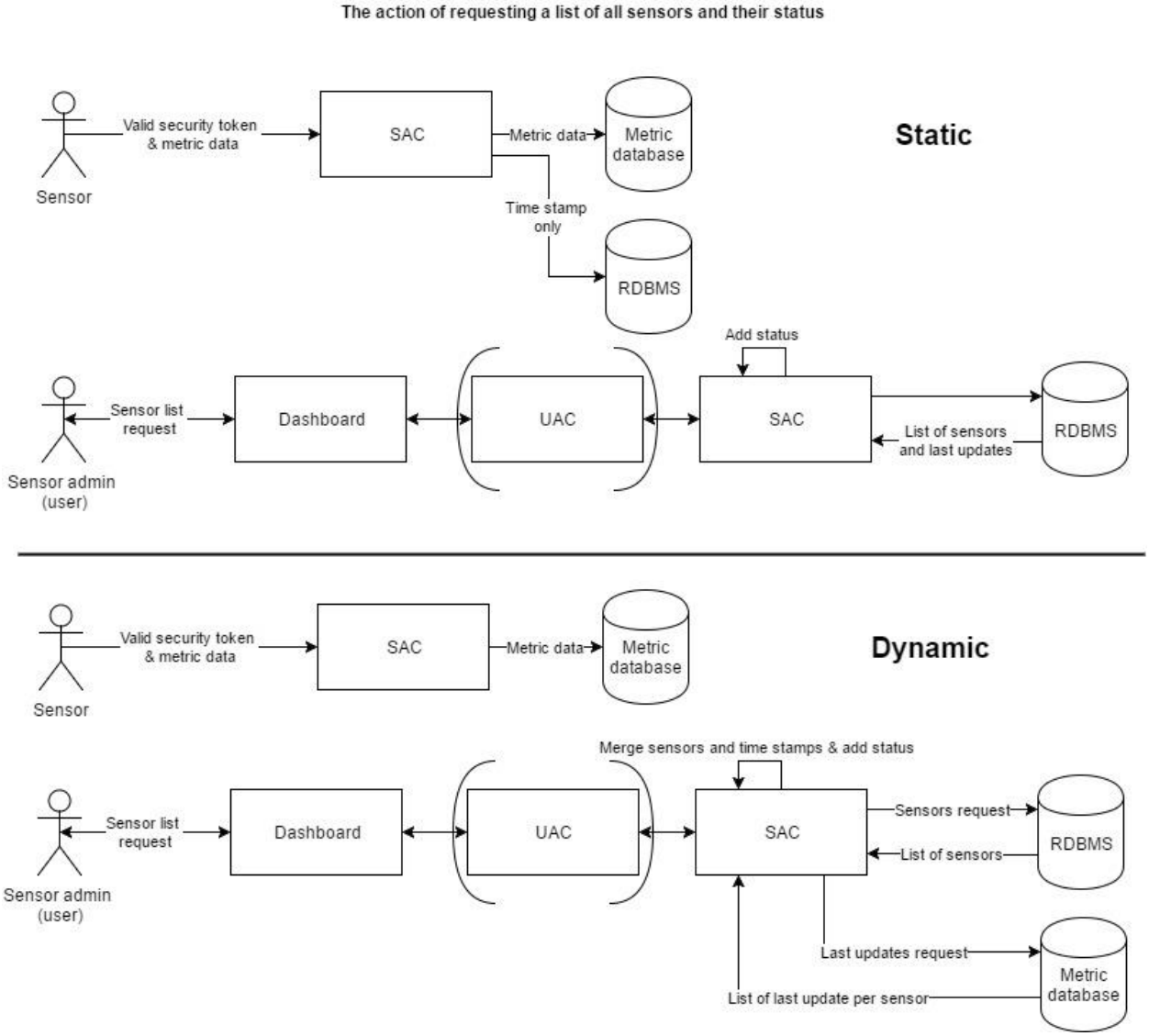


Figure 9 - Static vs Dynamic status updates

Accounts with privileges and user roles (REQ_7)

As there are different functions for different users, it is a good idea to split up user types and give each the appropriate roles and privileges. For example, a data analyst would probably never want to add or remove a sensor, but is interested in custom graphs to check for correlations. The other way around, a sensor admin needs to know what the health of the sensors is to replace sensors in time and configure them. They may be interested in what the runtime for a sensor was and how many uploads were done, but to go to the extent of building custom graphs and see measures on specific times is unlikely.

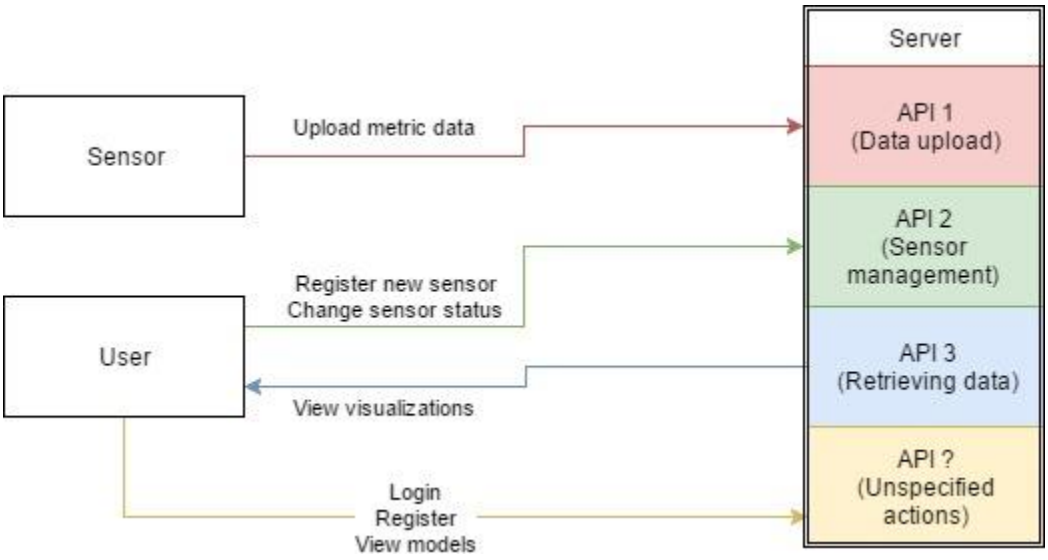
Related Requirements

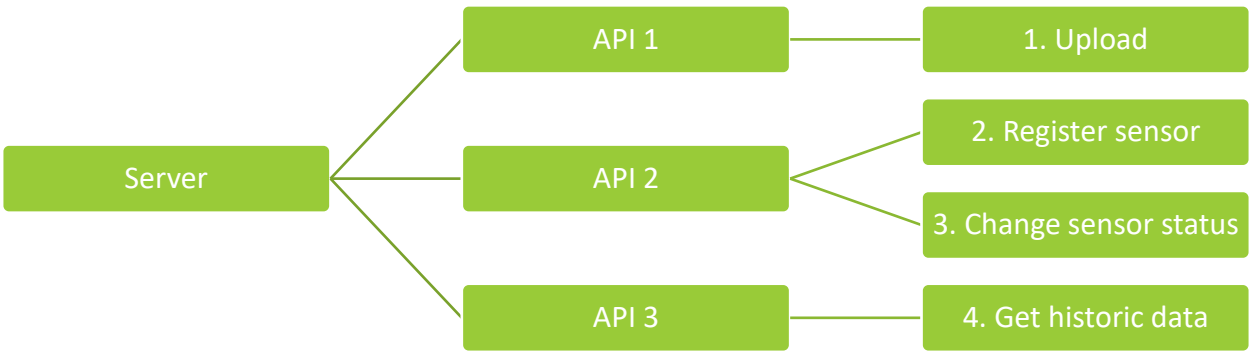
Requirement	Priority	Description
REQ_1	M	The Hub should work 99.99 % of the time.
REQ_2	M	There will be an api1-sensorhub.hr.nl (at this moment 145.24.222.139) that can be used by sensor nodes to upload new sensor measurements, an api2-sensorhub.hr.nl that can be used for sensor node management and an api3-sensorhub.hr.nl that can be used by different front-end applications to retrieve actual and historical sensor data.
REQ_5	M	If a sensor node has not sent sensor data for more than 3 normal intervals it gets status ‘intermittent failures’, in case sensor data is not received for more than 30 normal intervals it gets status ‘inactive’.
REQ_7	S	Not everyone is entitled to see all sensor data / graphs / visualizations. There should be a public view with some sensor data and private views for viewers with an account like for example the architecture students or the owner of the house.

Appendix D: API use cases

BELOW YOU WILL FIND THE EVENTS GROUPED IN APIS AS SPECIFIED BY THE PRODUCT OWNER AND SPECIFIC INFORMATION ABOUT EVENTS AND DATA FORMATS (MARCH 19, 2017).

Actions grouped by API





Side note before reading

Some API operations mention ineligibility. For now, we assume checking the users' privileges is done by providing a cookie header containing a random user token provided by the server. This cookie is obtained when the user logged in successfully.

Requests and responses

In the section below, actions as specified above are described in detail. These details specify the expected processes including errors, data types and client-server interaction.

API 1 – Sensor Data Upload

This API is used for the sole purpose of uploading data.

Upload

Event No.	Requirement	Request Type	Data Type	URL Path
1	REQ_2 / REQ 3	POST	JSON	/upload

The uploader must be a sensor with an access token, gained by registering through API 2. The data itself can be one or multiple data points. A timestamp is expected to be provided. However, if none is given, the server must supply one before uploading data to the metric database.

NB: It is possible to configure the data structure by hand. The sensor admin should be able to do this. A small interpreter-like script must be written for this. Naturally, the config must remain operable.

Data format

```
{
   "access_token":"6Y9RBl6UjWk74A7EHPEasiHcWV2aMQNj",
   "labels":[
      "02/02/2017 13:41",
      "02/02/2017 13:42",
      "02/02/2017 13:42"
   ],
   "data":[
      353,
      354,
      369
   ]
}
```

Expected return codes

Expected result on success	Error (invalid access token)	Error (malformed data)
Status code 200 (OK)	Status code 400 (Bad Request)	Status code 400 (Bad Request)

API 2 – Sensor Management

This API is used to register and manage sensors.

Register

Event No.	Requirement	Request Type	Data Type	URL Path
2	REQ_6	GET / POST	JSON / Plain text	/register

Registering a new sensor is done by getting a code from the server as a user, and making a request from the sensor to the server using that code. The user who wants to add a new sensor must be a sensor admin. Doing this, two layers of security are used to prevent unwanted data submissions from random systems and preventing regular users from registering sensors. The process is visualized in Figure 10 - Process of registering a new sensor.

The sensor admin first GETs the page designated for sensor management. When the admin wants to add a new sensor, a button is pressed and a window with a token appears. This token must be submitted to the same URL as the ajax request. This can be done using tools like *curl*. The access token is returned in plain text.

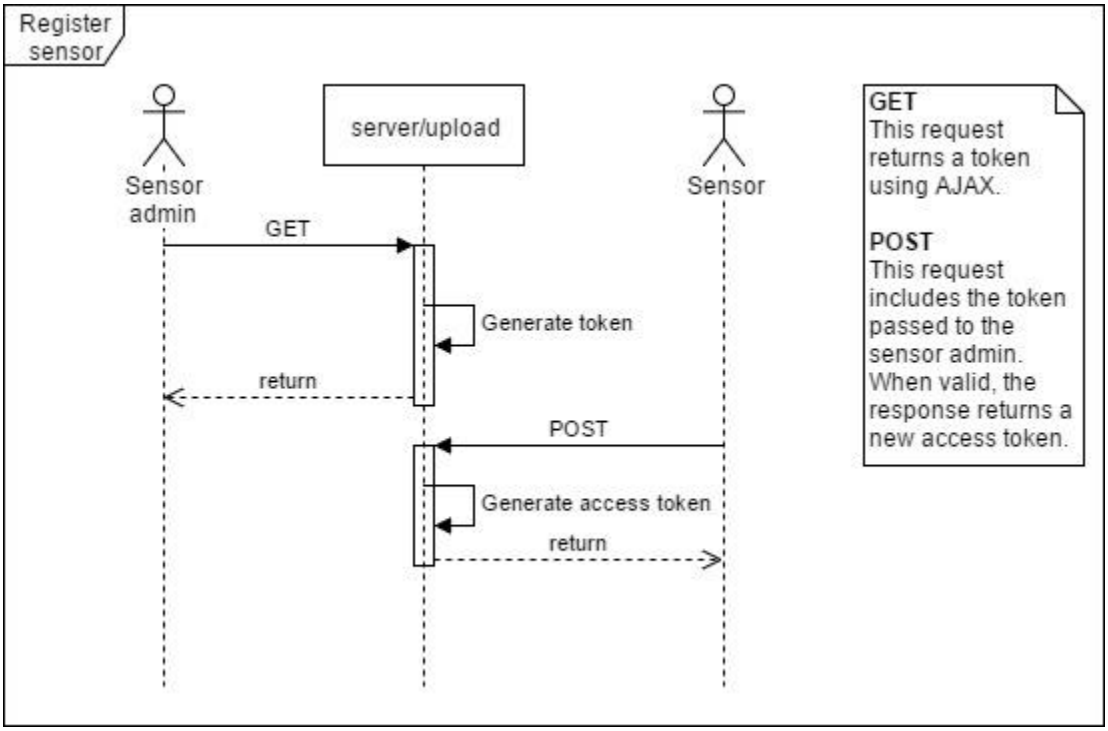


Figure 10 - Process of registering a new sensor

Data format generated token for registering sensor

```
{
   "token":"ABC-123"
}
```

Data format access token

```
<html>
<head></head>
<body>6Y9RBl6UjWk74A7EHPEasiHcWV2aMQNj</body>
</html>
```

Expected return codes for sensor admin

Expected result on success	Error (ineligible)
Status code 200 (OK)	Status code 403 (Forbidden)

Expected return codes for sensor

Expected result on success	Error (invalid code)	Error (timeout elapsed)
Status code 200 (OK)	Status code 400 (Bad Request)	Status code 403 (Forbidden)

Change sensor status

Event No.	Requirement	Request Type	Data Type	URL Path
3	Course guide (Dashboard)	POST	JSON	/edit

As quoted from the course guide: "*In the dashboard the manager can create/remove/stop nodes/instances of the network*". After we read this, we immediately decided that a manager must also be able to start a node/instance. We assumed starting and stopping means disabling a sensors access but not removing its access token so it can be reenabled.

A sensor admin edits a sensor by providing its access token and the action they want to perform. The server handles the operation itself.

Data format

```
{
   "access_token":"6Y9RBl6UjWk74A7EHPEasiHcWV2aMQNj",
   "action": "remove"
}
```

Expected return codes

Expected result on success	Error (corrupt data)	Error (ineligible)
Status code 200 (OK)	Status code 400 (Bad Request)	Status code 403 (Forbidden)

API 3 – Retrieving Data

This API is used whenever a user wants to receive raw data such as visualization data.

Get Historic Data

Event No.	Requirement	Request Type	Data Type	URL Path
4	REQ_7 / REQ_9	GET	JSON	/data?dataset={0}&start={1}&end={2}

When a user wants to see visualizations of data or needs historic data, this function is used. The user makes a request to this URL and specifies the name of the dataset as well as the start and end date as EPOCH timestamp. The result is then returned as a JSON string. Naturally, the available datasets are limited to historic sensor data and not the user info.

Data format

```
{
    "results": [
        {
            "statement_id": 0,
            "series": [
                {
                    "name": "cpu_load_short",
                    "columns": [
                        "time",
                        "value"
                    ],
                    "values": [
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            2
                        ],
                        [
                            "2015-01-29T21:55:43.702900257Z",
                            0.55
                        ],
                        [
                            "2015-06-11T20:46:02Z",
                            0.64
                        ]
                    ]
                }
            ]
        }
    ]
}
```

Note: This is the data format provided by our metric database, InfluxDB.

Expected return codes

Expected result on success	Error (invalid arguments)
Status code 200 (OK)	Status code 400 (Bad Request)

Remaining functionality

This section covers the actions of logging in and registering as a new user. Naturally, both logging in and registering must be done over HTTPS/TLS and with CSRF protection. Also, the way how models are transported from the server to a client will be discussed.

Registering

Event No.	Requirement	Request Type	Data Type	URL Path
5	REQ_7	POST	JSON	/register

A user must be registered and logged in before they can perform any actions. To register, a user needs to fill out a form containing their first and last name, email address and password. A secondary password box will check whether both password entries are the same. When all's good, the form will be sent to the server, where it will be checked again for validity and security. The new user then receives a confirmation email in their inbox with an activation link. The activation link must contain a random verification token that matches with the server's random token. When this link is visited, the user is registered and added to the default user group.

Data format

```
{
    "email": "username@server.org",
    "password": "SomeP@ssw0rd123"
    "role": "data_scientist"
}
```

Expected return codes

Expected result on success	Error (invalid arguments)
Status code 200 (OK)	Status code 400 (Bad Request)

Logging in

Event No.	Requirement	Request Type	Data Type	URL Path
6	REQ_7	POST	JSON	/login

To log in, a user needs to fill out a form containing email address and password. Both are checked with the server and if it matches a user, a login token cookie is placed for the current session.

Data format

Request	Response
{ "email": "username@server.org", "password": "SomeP@ssw0rd123" }	{ "token": "6Y9RBl6UjWk74A7EHPEasiHcWV2aMQNj" }

Expected return codes

Expected result on success	Error (invalid arguments)
Status code 200 (OK)	Status code 400 (Bad Request)