UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Developing a data repository for the Climate Adaptive City Enschede

Joeri Planting B.Sc Thesis August - 2018

Supervisor:
ir. ing. R.G.A. Bults
Critical Observer:
ir. J. Scholten

Creative Technology
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede

Abstract

Due to climate change, more frequent and heavier rainfall occurs in the Netherlands. The city of Enschede's sewage system is unable to handle the amount of rainwater in case of heavy rainfall, which causes for streets to flood. The municipality of Enschede is looking for solutions, which resulted in the Smart Rainwater Buffer graduation project. The municipality also wants to monitor air temperature to gain insight in the urban heat island effect in Enschede, which resulted in the graduation project Air Temperature Monitoring. Both projects are brought together under the name Climate Adaptive City Enschede (CAC). This report describes the development of a suitable data repository for the CAC project.

Literature and state of the art research was done in order to gain insight in the characteristics and pros and cons of the different classes of databases that exist, which are (old)SQL, NoSQL, and NewSQL. Some database management systems of each of the classes were compared, with focus on characteristics as data integrity, performance, and geospatial functionalities. The concept for the repository consist of the database and a generic interface. After setting up the requirements, the functional architecture was designed with the data repository consisting of a database, and three interfaces. One for each of the following roles, administrator, producer, consumer, which contain the needed functionalities.

The implementation of the database of the final prototype is done with the database management system PostgreSQL, extended with PostGIS. This combination is a very mature, reliable, and fast database management system. The PostGIS extension offers the most extensive set of geospatial functionality of all database management systems. The interfaces are implemented using the Lumen php micro-framework.

Acknowledgements

First I’d like to thank my fellow students Jeroen Waterink, Thijs Dortman, Laura Kester, and Adam Bako for the collaboration within the Climate Adaptive City Enschede project. I would also like to thank my supervisor Richard Bults and my critical observer Hans Scholten a lot, for their guidance, help, and insights during this graduation project.

Finally, I’d like to thank Hendrik-Jan Teekens from the municipality of Enschede for making the CAC graduation projects available to the University of Twente, as well as for his time and valuable input.

Table of Contents

1. Introduction 1
1.1 Situation 1
1.2 Challenges 1
1.3 Research Questions 1
1.4 Outline 2
2. Background Research 4
2.1 Background 4
2.1.1 Enschede's water problem 4
2.1.2 Urban Heat Island Effect 5
2.1.3 Sub-projects 6
2.2 Previous Work 7
2.2.1 Smart Rainwater Buffer 7
2.2.2 Air Temperature Monitoring 8
2.2.3 Conclusion 9
2.3 Literature Review 9
2.3.1 Introduction 9
2.3.2 Types of databases 10
2.3.3 Geospatial data 12
2.3.4 Conclusion 13
2.4 State Of The Art Review 14
2.4.1 PostGIS 14
2.4.2 MySQL 15
2.4.4 MongoDB 16
2.4.5 VoltDB 17
2.5 Conclusion 17
3. Methods and Techniques 20
3.1 Interviews 20
3.2 Design process for Creative Technology 21
3.2.1 Ideation 22
3.2.2 Specification 22
3.2.3 Realization 22
3.2.4 Evaluation 23
3.3 Stakeholder Analysis 23
3.3.1 Stakeholder roles 24
3.3.2 Stakeholder types 24

3.3.2 Stakeholder prioritization 24
3.4 Requirement Analysis 25
3.4.1 MoSCoW 25
3.5 Scenarios 26
4. Ideation 28
4.1 Stakeholder Identification and Analysis 28
4.1.1 Identification 28
4.1.2 Categorization and prioritization 29
4.2 Preliminary requirements 31
4.2.1 Functional requirements 31
4.2.2 Non-functional requirements 32
4.2.3 The dataset 32
4.3 Concept 35
5. Specification 39
5.1 Requirements 39
5.1.1 Functional requirements 39
5.1.2 Non-functional requirements 42
5.2 Table structure 42
5.2.1 RPS 43
5.2.2 AirT 44
5.2.3 SRB 46
5.2.4 Universal solution 47
5.3 Functional architecture 53
5.3.1 Overview 53
5.3.2 Repository 54
5.3.3 Producer interface 54
5.3.4 Consumer interface 59
5.4.5 Administrator interface 64
5.4 PACT Analysis 64
5.4.1 People 64
5.4.2 Activities 65
5.4.3 Context 66
5.4.4 Technologies 66
5.5 PACT-FICS scenario 66
6. Realization 71
6.1 Hardware 71
6.2 Database management system 71

6.2.1 PostgreSQL & PostGIS 72
6.2.2 PostgreSQL install log 75
6.2.3 PostGIS install log 76
6.2.4 Creating the database 77
6.3 Lumen php micro-framework 80
6.3.1 Installation 80
6.3.2 Routes 81
6.3.3 AdminController.php 86
6.3.4 ProducerController.php 87
6.3.5 ConsumerController.php 92
7. Evaluation 99
7.1 Postman 99
7.2 Functional testing 99
7.2.1 Tests 100
7.3 Performance 108
7.3.1 Admin interface 108
7.3.2 Producer interface 109
7.3.3 Consumer interface 109
7.4 Conclusion 110
8. Conclusion 113
9. Recommendations 116
References 117
Appendix 120
A. PostGIS Function Support Matrix 120
B. web.php, routes specification 130
C. AdminController.php 131
D. ProducerController.php 132
E. ConsumerController.php 136

1. Introduction

In this chapter a short description of the situation will be provided, followed by the challenges and research questions regarding this graduation project. Finally, an outline of the contents of this thesis will be described.

1.1 Situation

Climate change, increase in city density and increase in hardened surfaces cause problems handling water and heat, which are expected to increase over time [1]. The municipalities of Almelo, Hengelo and Enschede as well as the Vechtstromen water authority participate in several projects, brought together under the name Klimaat Actieve Stad (KAS) [2]. With the KAS projects the municipalities and water authority aim to better cope with water and the changing climate, and contribute to an increase in quality of life for its citizens. From the KAS projects of the municipalities and the water authority, the graduation project Climate Adaptive City (CAC) Enschede followed.

The CAC Enschede project can be seen as a so called smart city project. Data handling arises as an important factor in such smart city projects. The Climate Active City Enschede (CAC) graduation project consists of several sub-projects, namely the Smart Rainwater Buffer (SRB), Air Temperature Monitoring (AirT), and the Reference Precipitation System (RPS). These projects will generate large amounts of location based data which has to be processed, stored, and be accessible for visualization in user interfaces.

1.2 Challenges

The goal of this graduation project is to develop a data repository for the CAC Enschede project. Data repository development is highly dependent on the system's requirements. To identify the requirements, close collaboration with all projects generating and querying data and the municipality is necessary.

Each of the projects will have multiple sensor nodes deployed which frequently generate location based data, resulting in a huge amount of data that needs to be stored. The stored data has to be accessible by user interfaces to provide visualizations for end-users. From this, multiple challenges follow regarding how to handle the heterogenic origin of data, the efficiency, costs, and scalability of the database system for potentially big data, concurrent access, and data integrity. Regarding the visualization, challenges follow in querying areas within a spatial based database system and potentially storing additional information on area types.

1.3 Research Questions

The research question that followed the challenges stated in the previous section is:

> *How to develop a suitable data repository for geo-tagged environmental data for the Climate Adaptive City Enschede project?*

The following sub-questions will be used in order to answer the above stated research question. The first, addressing the geospatial aspect of the data, is:

What database management system is most suitable for storing geospatial data?

The second, regarding the performance of the system, is:

How to maintain performance with large amounts of data generating sensor nodes?

1.4 Outline

First a background study will be discussed in chapter 2. This chapter starts with describing the water problem of Enschede and the urban heat island effect. Secondly, previous work on the Smart Rainwater Buffer and Air Temperature Monitoring graduation projects in relation to data storage will be discussed. Thirdly a literature review regarding different types of databases and storage of geospatial data is discussed, after which a state of the art review describes 4 different database management systems and their geospatial functionalities. Chapter 3 describes the methods and techniques used in this graduation project. Chapters 4, 5, 6, and 7 contain the results of the Creative Technology design process, which are the concept, requirements, functional architecture, and the final prototype. Chapter 8 contains the conclusion. Finally, future recommendations are made in chapter 9.

2. Background Research

This chapter introduces the situation in Enschede regarding its water problems and the heat island effect as the context of this research. Thereafter previous work on the SRB and AirT projects will be discussed. Thirdly the conducted literature review as well as a state of the art review is discussed.

2.1 Background

2.1.1 Enschede's water problem

The changing climate has an effect on the frequency and intensity of rainfall in certain periods during the year. Periods of intense and heavy rainfall are alternated with longer periods of draught and heat waves. Enschede faces some problems managing the amount of water in case of heavy rainfall. There are four main reasons for the issues Enschede has with managing heavy rainfall, of which some are visualized in figure 2.1. First of all the city is built on a moraine, causing for a difference in height of approximately 44 meters. Enschede is built over several natural water sources on this moraine, which would naturally dispose of water continuously and gradually. However, due to the increase of hardened surface of the city, a very high volume of water flows during heavy rainfall in a short period of time [3]. Secondly, Enschede has a rich history of textile industry. The decline of this industry in Enschede has caused for factories that existed to close down, causing the groundwater levels to rise. The third reason is that most urban brooks that flowed through Enschede have disappeared over the years, as can be seen in figure 2.2. Finally the city is mainly build on clay ground, which is a poor permeable soil layer.

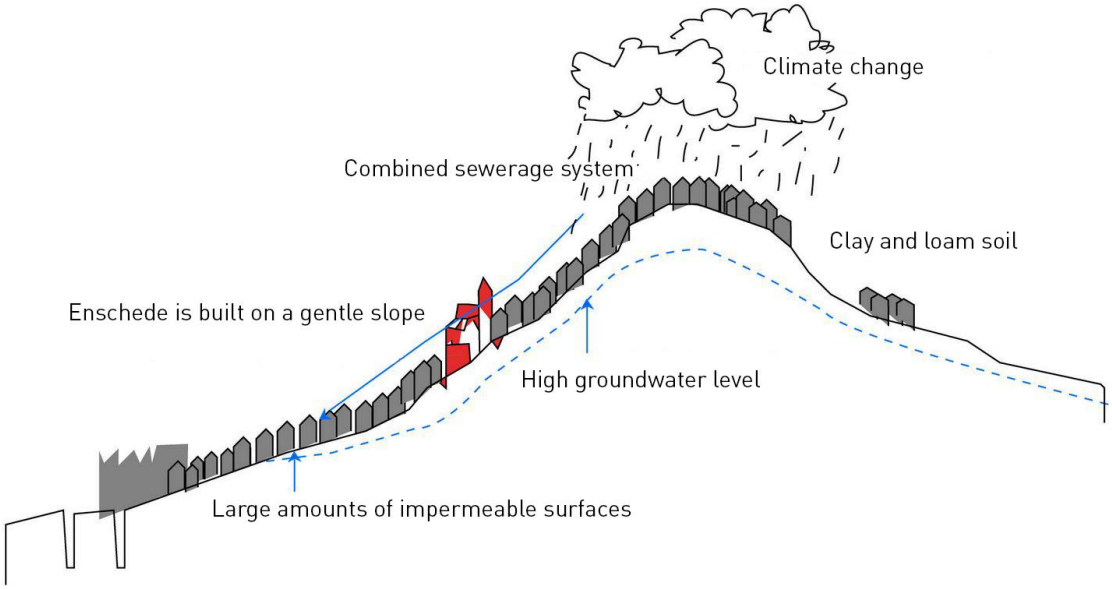

Figure 2.1: Visualization of Enschede its water issues due to its location. Image by Gemeente Enschede, modified by Gelieke Steeghs

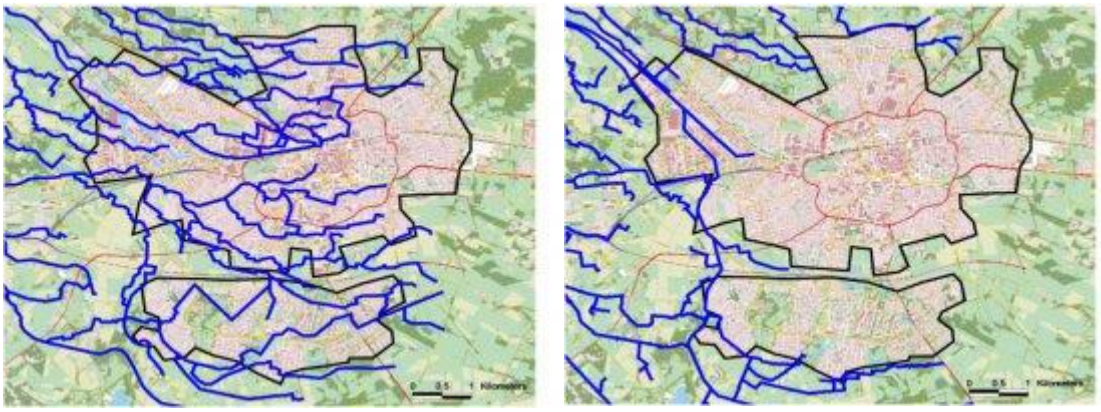

Figure 2.2: Urban brook system in Enschede of 1900 (left) and 2010 (right). Image by Kennisportaal Ruimtelijke Adaptatie.

Already several projects have been deployed to improve the cities capabilities in handling heavy rainfall. Urban brooks are being reconstructed, of which de Roombeek, Beek ‘t Zwering, and de Stadsbeek are examples [4]. Also wadi’s are used as natural water buffers in case of heavy rainfall [5]. A wadi is a lower area for buffering rainwater, but it can be used by citizens in dry periods. Water flows to these lower areas during rainfall and is held there so that is can gradually infiltrate the ground. An example of a wadi can be found in figure 2.3a. Furthermore, green roofs (figure 2.3b) are used to temporarily store rainwater and slow the drainage of the water from the roofs, a reservoir called Kristalbad has been realized which can store approximately 187.000 m^3 water, and a huge water buffer will be constructed underneath de Oldenzaalsestraat.

Figure 2.3a: Example of a wadi *Figure 2.3b, Example of a green roof*

2.1.2 Urban Heat Island Effect

Periods of heat can have severe negative effects on the health and wellbeing of people. The municipality of Enschede wants to monitor city temperature, in order to gain insight in the urban heat

island effect (UHI) [6]. This effect means that the temperature in urban areas is higher than its surrounding areas, as depicted in figure 2.4, and has (potentially) serious consequences for the health and wellbeing of the citizens. Consequences of UHIs are increases in peak energy demand, degradation of air quality, increased thermal stress on residents, strong impact on urban ecosystems, and a significant increase in the level and risk of morbidity or illness caused by heat.

The primary cause for the UHI effect is urbanization. This means the increase of building density and the amount of hardened and heat absorbing surfaces, and the decrease of natural vegetation. Also the rise in city temperature causes for an increase in the use of air conditioning systems, which in turn dissipate heat into the city air. In order to increase knowledge of the UHI effects on Enschede, the municipality wants to monitor temperature throughout the city by deploying several sensors.

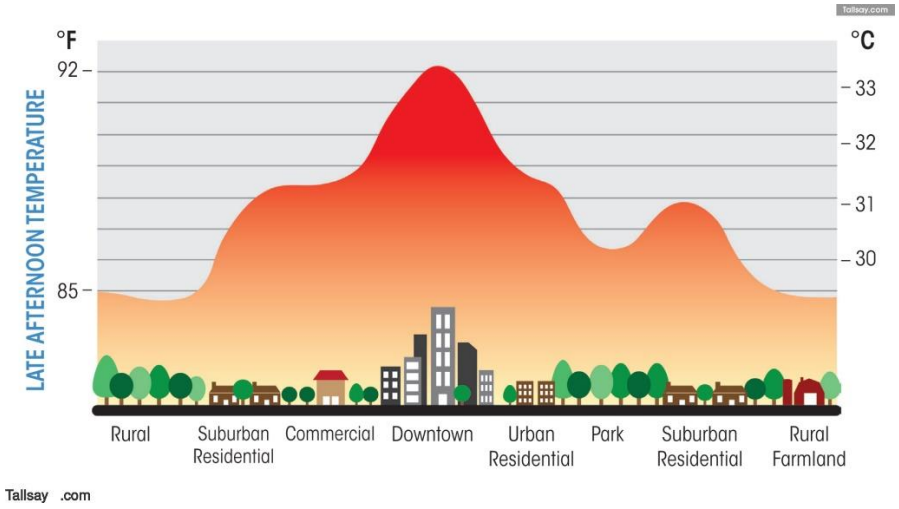

Figure 2.4:Visualization of the urban heat island effect. Image by tallsay.com.

2.1.3 Sub-projects

In this section each of the CAC sub-projects will be explained in short, excluding the data repository.

SRB

The primary goal of the SRB project is to buffer rainwater in case of heavy rainfall to reduce the strain on the sewage system. The buffering will be done by a smart rainwater buffer which in this phase of the project will be owned by citizens of the municipality of Enschede. In order to buffer and dispose rainwater autonomously, the SRB will use sensors to provide the system with the water level and water temperature measurements. The dashboard for the users will contain visualizations of the data produced by one or more SRBs.

AirT

The AirT project consist of multiple sensor nodes deployed in the Enschede, of which the produced data will be used by the municipality of Enschede and researchers of the University of Twente to gain insight in the urban heat island effect in Enschede.

RPS

For the reference precipitation system, a Lambrecht precipitation sensor will be used. The data produced by this sensor will be used to give insight in the smart rainwater barrel performance and for historic precipitation.

2.2 Previous Work

This section covers previous graduation projects regarding the Smart Rainwater Buffer and Air Temperature Monitoring, with focus on data storage.

2.2.1 Smart Rainwater Buffer

In 2016/2017 Felicia Rindt [7] and Gelieke Steeghs [8] worked on the development of a smart rainwater buffering system. As a functional requirement they state that the data generated by the buffer should be store in a central database. They use a Raspberry Pi 3B as server to host the database. A relational database management system has been used, namely MySQL.

The database consists of 7 tables, of which a detailed view can be found in figure 2.5. The user table holds the user's id, name and address. The water buffer table describes a single buffer, holding an id, location, capacity, planned discharge id, datetime heartbeat, update time, default output valve and future volume refill. The discharge table consists of a planned discharge id, start date and time, and the planned discharge amount. The discharge command table contains data on a discharged amount by the citizen. It holds the unique discharge command id, the discharge buffer id, the amount, the discharge status, the discharge creation date and time, and the valve used for the discharge. The waterflow table holds the cumulative output flow for a valve on a single buffer. The buffer information table contains the water level in the buffer on a certain moment in time. Lastly, the event table contains data on the priority, date and time, type of the event, and a message to be displayed in the interface.

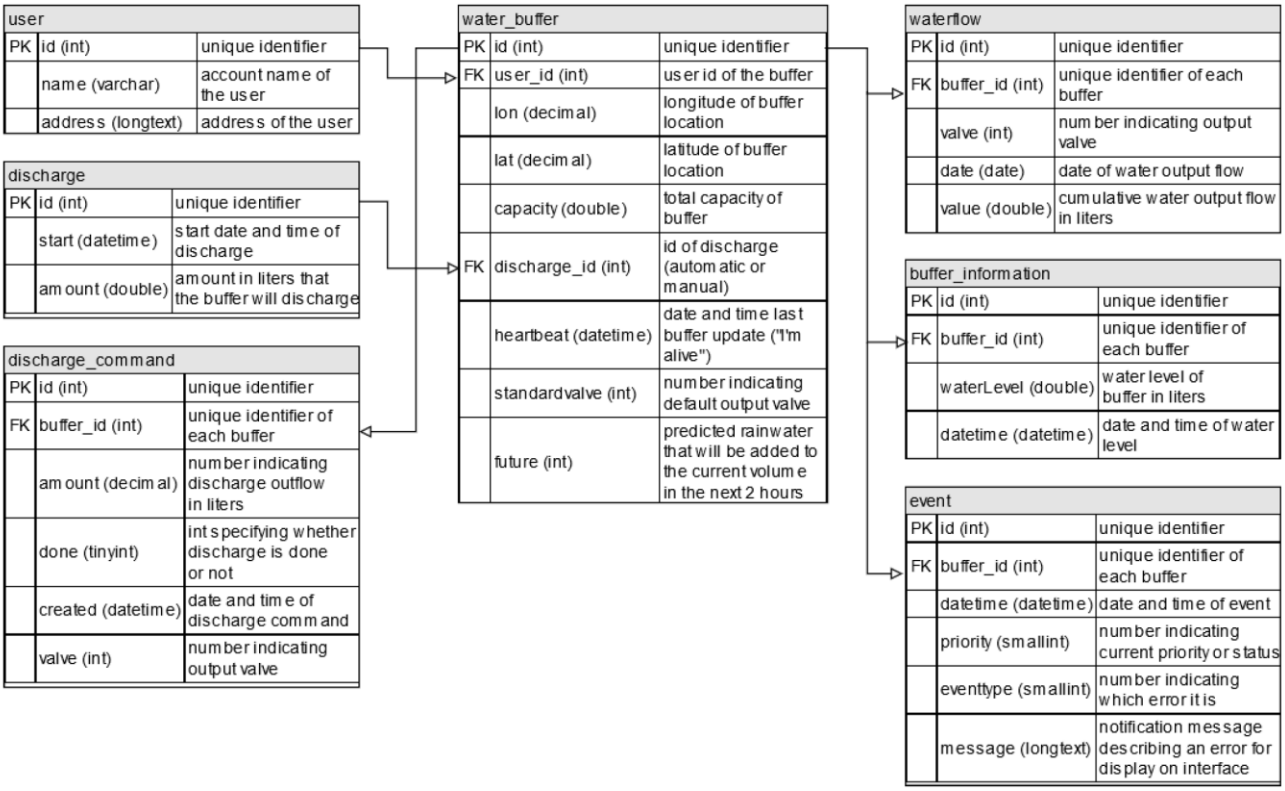

Figure 2.5: Table structure of the current SRB database [6]

2.2.2 Air Temperature Monitoring

For the AirT graduation project, Yoan Latzer [9] and Tom Onderwater [10] have investigated several communication techniques to transfer the sensor data to a central server. They decided to use The Things Network (TTN), which is an open source Internet of Things focused network for low powered devices using the LoRaWAN protocol. A Python script collects the data from an online TTN application using the MQTT protocol and writes the data to the database's main table, the measurement table. A second table holds a list of all deployed sensor nodes, in order to easily return a list of sensors for data formatting. This measurement table contains the following fields:

- **measurement_type**: In their project the only measured data was Temperature, so this field was manually set to Temperature. They do state that other types of measurements could be incorporated in the future.
- **device_id:** A unique identifier for each device, provided by the TTN metadata.
- **value**
- **latitude**
- **longitude**
- **day**
- **month**

- **year**
- **hour**
- **minute**

A relational database management system has been used. Furthermore, a Django server is used to communicate with the database, as can be seen in figure 2.6. The Django framework does not officially support NoSQL databases [11].

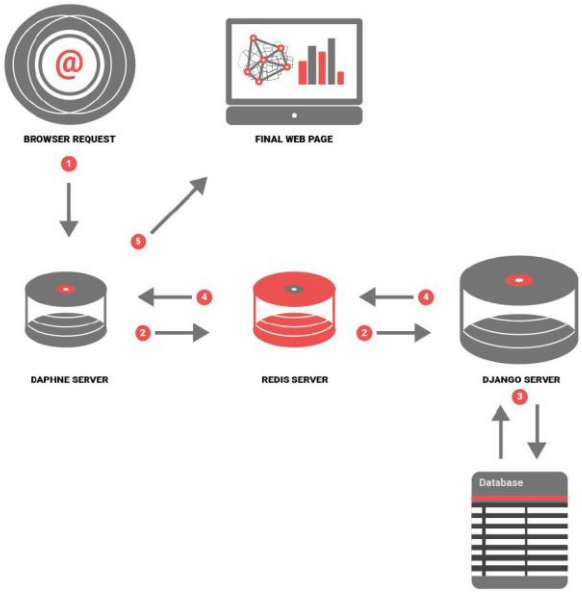

Figure 2.6: Application structure for the air temperature monitoring project [9].

There were some performance issues loading the webpage. Thee page loading delay was caused by a the application requesting the full measurements table data.

2.2.3 Conclusion

Following from previous work it can be concluded that limited research on data storage has been done in the context of the CAC Enschede project.

2.3 Literature Review

2.3.1 Introduction

Data plays an increasingly important role in a large variety of projects, like smart cities, which more and more incorporate internet of things applications and internet connected devices. City management can use collected data to improve overall city management efficiency and for

improvements in various sectors, like water- and sewage management, city temperature monitoring and prediction, traffic reduction, energy saving and improving overall quality of life for its citizens [12]. The increasing amount and heterogeneity of sensor data is accompanied with challenges in managing and storing these large volumes of data.

For the CAC Enschede a data repository has to be developed, which lays at the center of several sub-projects. These projects are the Smart Rainwater Buffer (SRB), the Reference Precipitation System (RPS), and Air Temperature Monitoring (AirT), and also accompanied data visualizations in user interfaces. For the data produced and used by the sub projects, the location of the sensor nodes are of great value. Visualizing UHIs requires location based visualization. For the smart rainwater buffer, the location is very important as well. In order to reduce the strain on the sewage system, it is necessary to know where rain will fall first and in what sequence the buffers should be emptied.

The total number of sensor nodes producing location based data is very likely to increase above 10.000 in the future. Therefore, exploration of possibilities for geo-tagged data processing, storing, and managing is needed. This literature review aims at giving an overview of characteristics, and pros and cons of several database types, database management systems and geospatial possibilities in the context of the CAC project.

The main question to be answered in this literature review is: *what solutions do exist for managing and storing large volumes of geo-tagged date?* This question will be answered by addressing the following subtopics, namely: *what database types exist, what are their characteristics, and what is a possible solution concerning storage of location based data.* Scientific literature will be used to answer the research question and sub questions, postulated in section 1.3.

2.3.2 Types of databases

Since the mid 1960's, data has been primarily stored in relational databases. Because they use SQL as their querying language, they are also known as SQL database systems [13]. Data is becoming more and more important nowadays and finds an increasing amount of possible applications. The internet of things is a fast growing concept. The amount of generated data increases along with the growth in the value of data and internet connected applications. This growth in generation of data has caused for two new classes of database management systems to emerge, non-relational and new-relational data, better known as NoSQL and NewSQL databases respectively [13]. Which type of database is most suitable, is highly dependent on the application requirements [14]. In order to make a good decision based on the application requirements, understanding of the key characteristics of the different types of databases is needed.

SQL

The first and often called traditional database management system is SQL. SQL database management systems have since the start been used to store large amounts of data in fixed schemas [13]. The relational model has been thoroughly studied, is well understood, and has for long provided a high level of consistency and efficiency [15]. Data is stored in tables in a traditional row and column format. Stored data is organized in relations, using keys to link data in different tables together. SQL database management systems adhere to the so called ACID properties. ACID stands for Atomicity, Consistency, Isolation, and Durability. By adhering to these properties, database management systems handle concurrency in transactions, ensuring the integrity of the data. The ACID model is standard, efficient, and reliable and its properties are very important characteristics of a SQL database management system [13]. SQL systems scale vertically. This means when scaling up, the centralized storage and processing capacity have to be increased by upgrading the hardware [15]. The current CAC project has structured data, thus the SQL class of databases could be a good fit.

NoSQL

Scalability and a decrease in performance when the dataset grows very large in SQL systems caused for the need of another solution. NoSQL emerged as a possible system with the main intention to improve scalability and performance issues encountered with traditional SQL systems. Database management systems based on NoSQL are non-relational. These databases have no fixed schemas and can therefore handle a wide variety of data [13]. NoSQL database management systems are also called document base systems, storing data as documents in formats likes XML and JSON [15]. NoSQL systems are not expected to substitute the relational based SQL system, but can outperform a SQL system when a document based system better fits the requirements. However, improvement in performance comes at a cost. R. Sánches et. Al [15] state that non-relational systems do not offer support for join operations and do not fully adhere to all ACID properties. However, S. D. Kuznetsov and A.V. Poskonin [16] state that NoSQL stands for all non-relational database management systems. The term therefore also incorporates earlier non-distributable and ACID compliant systems. Distribution means that these systems scale horizontally. By decentralizing the database, the storage can be distributed over multiple data centers and multiple CPUs can be utilized at the same time, improving query performance [15]. When in the future the CAC project is fully deployed, distributing the database over different municipalities could be a good way to greatly improve and maintain performance, but this introduces an increase in complexity for the unstructured way of storing the data. Furthermore, a drawback of this type of database management system is that is does not offer full ACID compliance, which could compromise the data integrity for the project.

NewSQL

Both SQL and NoSQL have their drawbacks. A system that tries to bring together the best of both worlds is NewSQL. NewSQL, also referred to as the modern relational model, keeps the relational property of traditional relational database management systems. The difference with the traditional system is that NewSQL incorporates NoSQL solutions as well [13] [17] [18], like scalability and high performance. NewSQL is claimed to be a very capable database management systems specifically for the increasing amount of internet of things data [13]. NewSQL databases primarily use SQL querying language, adhere to the ACID properties for transactions, have an architecture offering higher performance than traditional relational database management systems, and have the possibility to run on a large number of nodes without suffering from bottlenecks [17]. Most NewSQL database management systems use in-memory storage of data, resulting in higher performance compared to SQL as well as NoSQL databases [13] [17] [18]. According to K. Grolinger et. al. [19], using NewSQL systems is in general suitable in circumstances where additional scalability and performance is required from traditional database management systems. NewSQL could be a fitting solution in the future when the CAC project grows amongst the entire Vechtstromen district, which will likely require higher performance capabilities from the system than the initial pilot project.

2.3.3 Geospatial data

The data that will be generated by the CAC projects is location based. This means that the data repository solution has to offer support for geo-tagged data and location metadata to indicate the area type of the location. Several approaches can be used to realize storage of geospatial data.

A four layer framework is proposed by S. Luan et. al. [20], consisting of a geography node using R-tree indexing, a logical node, an application node, and a storage node. The geography node describes the location and shape of a place, to express location information more accurately compared to the geometric point based representation [20]. However, this framework utilizes a hybrid SQL - NoSQL approach and introduces an unnecessary amount of complexity to the project.

There exist database management systems specially designed for storing and querying spatial data, namely spatial database management systems. These systems usually are a regular system, extended with spatial capabilities. They also extend the mature querying language SQL, then called *spatial querying language,* and offer spatial querying features [21]. In order to provide efficient querying of spatial data, which is of high importance within the project, spatial indexing is required. R-tree indexing is the most widely used method for managing spatial data objects [22]. An open source SQL database management system, namely PostgreSQL, offers very good spatial information storage support using the PostGIS extension [21]. PostGIS and PostgreSQL use R-tree indexing on top of

GiST (generalized search tree), offering robust spatial indexing. Performance of PostgreSQL compared to the NoSQL database system MongoDB, is sufficient [23]. Both implementations have different overall characteristics, and performance differs depending on the type and amount of simultaneous read/write operations. Again, choice of database management system is highly dependent on system requirements. As said before, Spatial Database Management Systems are usually not an independent software solution, but serve as an extension to existing database management systems like PostgreSQL or MySQL [24].

The PostGIS extension for the PostgreSQL database management system is open source, has the largest user base, and offers the most complete implementation of OGC's Simple Feature Specification (a standard for mostly two dimensional geometrics) of any free and open source database management system [24]. However, in order to make a definite decision, both a clear overview of the requirements for the CAC project has to be acquired, as well as further exploration of spatial possibilities of NoSQL and NewSQL database management systems.

2.3.4 Conclusion

Regarding the question what solutions do exist for managing and storing large volumes of geo-tagged data, an overview of different database management systems was provided as well as possibilities for storing geospatial based data. Caused by an increase in the amount and importance of data and applications generating data, two new types of database management systems have arisen: NoSQL and NewSQL. Database management systems can be categorized in being relational (SQL) and non-relational (NoSQL). The former being based on the traditional and most mature model, the latter offering a new solution to immense volumes of data and performance, but with a tradeoff in one or more of the ACID properties. Next to the traditional relational and non-relational database management systems SQL and NoSQL respectively, a so called modern relational system has emerged. NewSQL offers the possibility for distributed nodes, storing relational data in-memory instead of on-disk. This means an increase in performance capabilities comparable to NoSQL, whilst adhering to the ACID properties and the relation model used in SQL systems.

Since the choice for a database management system is highly dependent on the requirements for system, clearly specifying these requirements in the next phase of the project is needed in order to decide which system to use. Furthermore, sufficient research exists on SQL based solutions for storage of geospatial data, but further exploration of NoSQL and NewSQL geospatial solutions has to be done.

2.4 State Of The Art Review

In this chapter first the Simple Features Access of the Open Geospatial Consortium [25] will be described. Second, a description of different database management systems and their possibilities for storing geospatial data is provided, in order to be able to compare geospatial support and performance of the different systems. As stated in the literature review in section 2.3.3, the PostGIS extension of PostgreSQL offers a very good and reliable solution for storing geospatial data. However, performance comparisons have shown that NoSQL and NewSQL systems are likely to have higher performance than SQL systems, and offer better scalability [23]. Overall the CAC project will be write intense as compared to reads, thus next to support for geo-tagged data storage, performance is an important preliminary requirement as well. Therefor the performance of PostGIS, MySQL, MongoDB, and VoltDB, as well as their geospatial features will be examined.

2.4.1 PostGIS

PostGIS is an extension of PostgreSQL, making it a spatial database management system by adding support for geographical objects and location querying in SQL. It claims to offer a large set of features, of which many are rarely found in other spatial databases [26]. PostGIS offers full support of the Open Geospatial Consortium Simple Features standard.

PostGIS supports geography and geometry objects. Geography is a new datatype. It allows for the storage of data in latitude/longitude pairs and supports long range distance measurements, because no projection on cartesian spatial reference systems is done which often don't represent the entire earth, but it comes at a cost. Most computations on geography are slower, there are fewer functions defined on geography than on geometry, and these functions require more CPU time to execute.

Geometry is the best fit if all data fits in a single spatial reference system representing for instance only the Netherlands, or a lot of spatial processing is required, such as clustering of geometries. If the data is contained in a small area like a municipality, using geometry and appropriate projection is the best solution both in terms of performance and available functionality.

Relevant Functions

The AirT and SRB projects will cover the area of the municipality of Enschede. Eventually, the SRB is expected to cover the area covered by the Vechtstromen water authority. With this in mind, some relevant functions and geometry types of PostGIS are explained below. For a complete list of PostGIS features, see Appendix A.

Since the projects are location based and cover a certain area, representation of an area in the database is useful. A *Polygon* is a geometry type well suited to represent areas. They represent objects

of which the size and shape is important. Examples given are city limits, parks, building footprints, and bodies of water. The concept of polygons is included in most graphics systems.

Collections can be used to group simple geometries into sets. These are useful for modeling real world objects, like the smart rainwater buffers, as spatial objects. Supported collections are the *MultiPoint, MultiLineString, MultiPolygon, and the GeometryCollection,* which are a collection of points, linestrings, polygons, and a heterogeneous collection of any geometry respectively.

Querying collections of data from multiple sensor nodes of a certain area of the city, is a preliminary requirement of the system. Spatial databases have the ability to compare relationships between geometries. Some functions provided by PostGIS include *ST_Equals*, which returns TRUE if two geometries are of the same type and have the same x, y coordinate pair. *ST_Intersects, ST_Disjoint, ST_Crosses, and ST_Overlaps* are functions that check whether two geometries of equal or differing types have space in common. *ST_Within* and *ST_Contains* check whether a geometry is fully contained within another geometry. In order to return geometries within a certain distance of for example a point, the function *ST_DWithin* provides an index-accelerated boolean test; hence calculation of an actual buffer is not necessary. Spatial Joins are supported and provide the functionality to combine information from different tables, using the spatial relationship as key.

Ensuring performance when the dataset grows is very important. Searching the table rows in sequential order becomes an issue with tables exceeding a few thousand rows. Indexing the dataset is a solution to improve performance. Spatial indexing is claimed to be one of the greatest assets of PostGIS. PostGIS uses R-tree indexing on top of GiST indexing. R-tree indexing means breaking up data into rectangles, which in turn break up in sub-rectangles, which again break up in sub-rectangles, etc. GiST, or Generalized Search Tree, indexing, breaks up data into things on one side, things that overlap, and things that are inside. Spatial indexes greatly improve spatial query performance of the spatial database.

2.4.2 MySQL

Compared to the PostgreSQL – PostGIS combination, MySQL offers a less complete set of Simple Features and the documentation is less extensive. The supported spatial data types are geometry, point, linestring, polygon, multipoint, multilinestring, multipolygon, and geometrycollection [27]. Spatial indexes are implemented using R-tree indexing only. MySQL does not offer support for rasters which are very useful for the creation of heatmaps. Also topology, geocoding, address standardization, and aggregate functions are not supported. Geocoding is the transformation of a description of a location, like a coordinate pair or an address, to an actual place on the earth's surface. Some aggregate functions are, taking a few PostGIS functions as example, constructing an array of geometries, creating a

Linestring from point geometries, returning the union of geometries, and creating a polygon GeometryCollection from linework of a set of geometries.

Another drawback of using MySQL as a spatial database management system, is space operation. For example, the PostGIS function *ST_Contains* will return all geometry types within another geometry, like points within a polygon. *MBR_Contains* on the other hand, a comparable MySQL function, only supports geometry within the minimum bounding box of another geometry, like a polygon. See figure 2.7 for a visual representation of this difference.

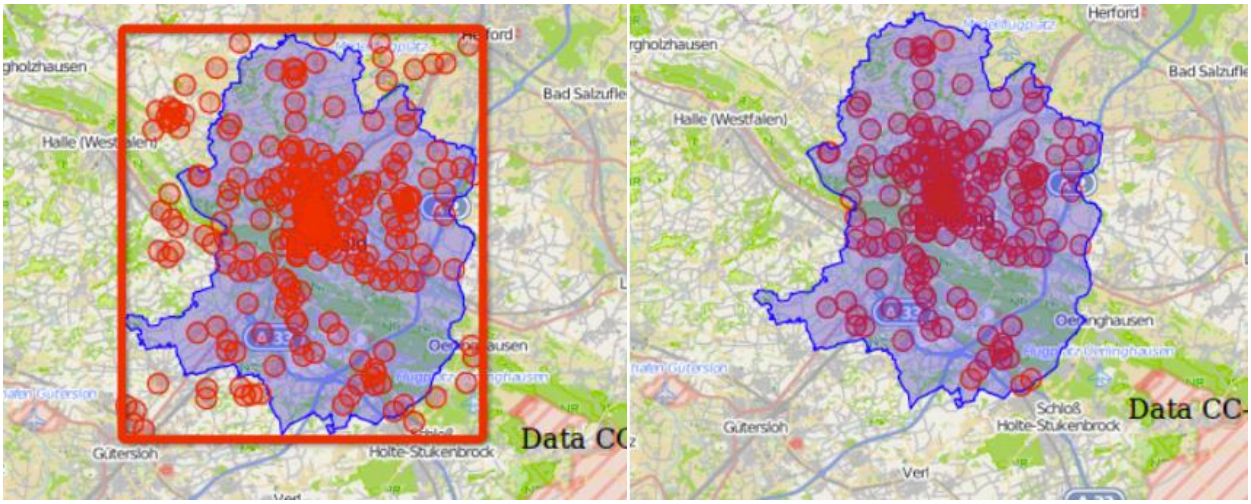

Figure 2.7: Differences in spatial querying functions of PostGIS and MySQL

2.4.4 MongoDB

As previously stated in the literature review, MongoDB belongs to the non-relational database class, also called NoSQL. The main difference between SQL and NoSQL is adherence to the ACID properties, which ensure transactions to fully take place or not take place at all, handle concurrent access, and ensure durability of the data in case of a crash or power loss. As said before, SQL systems store structured data in a predefined table based schema, using keys to define relations between tables. NoSQL systems allow for the storage of unstructured data. It does so by storing key-value pairs in JSON like documents. Similar documents can be stored in a collection, which is comparable to a table in SQL. Due to the more flexible data structure of NoSQL, it is easier to make mistakes. Next to the lack of ACID adherence, NoSQL has no equivalent for the JOIN functionality of SQL systems. With SQL, JOIN offers the possibility to query related data using a single SQL statement. Taking the SRB project as example, obtaining the water level of all smart rainwater buffers in SQL would be possible in a single SQL statement, combining the *water_buffer* and *buffer_information* tables of figure 2.5 using JOIN. In order to obtain the same result in a NoSQL system, the retrieval of all water_buffer documents as well as the buffer_information documents is needed. Then all documents have to be manually linked in program logic. Concerning data integrity, SQL's fixed schema will ensure that all

buffer_information entries are related to a water_buffer; i.e. the deletion of a water_buffer is not permitted if one or more buffer_information entries are associated with the water_buffer. Such integrity rules are not available to NoSQL systems, since they allow storage of any data regardless of other documents. NoSQL systems trade-off the lesser data integrity with higher scalability and performance, due to the denormalized way of storing data.

In MongoDB geospatial data is stored in GeoSON objects. Supported objects include Point, Linestring, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection [28]. Geospatial queries on GeoSON objects are calculated on a sphere. It is possible to use two dimensional indexing with MongoDB, allowing querying of flat surface data. The list of supported query operators is rather short, especially when compared to PostGIS. Assuming two dimensional non-spherical indexing, the provided query operations are *$geoWithin, $near,* and *$nearSphere.* Respectively this means the selection of geometries within a bounding GeoSON geometry, returning geospatial objects ordered by the distance to a point, and returning geospatial objects ordered by the distance to a point on a sphere.

2.4.5 VoltDB

VoltDB belongs to the relational database class. It is a NewSQL database management system. As stated in the literature review of section 2.3. NewSQL aims at combining the best of both worlds into a new type of database. The best of both worlds meaning the structured relational storage schema, SQL language, and ACID adherence of SQL, and the scalability and performance of NoSQL systems.

Regarding the support for geospatial data, VoltDB lacks behind. Especially when compared to PostGIS. VoltDB supports only two geospatial datatypes, namely GEOGRAPHY and GEOGRAPHY_POINT [29]. The GEOGRAPHY_POINT represents a single point on earth which is defined by the latitude/longitude pair. The GEOGRAPHY datatype represents a bounded region of the earth and is defined by one or more polygons. Provided functions are AREA(), CENTROID(), CONTAINS(), and DISTANCE(). Respectively they return the area of a region, the center point of a region, whether a region contains a certain point, and the distance between a point an a region or two points.

2.5 Conclusion

The previous research regarding the data storage of the SRB and AirT projects was certainly limited. Therefor a literature and state of the art review was conducted to elaborate on the different possibilities that exist for the storage of geospatial data.

In the conducted literature review in section 2.3, an overview of different database types and characteristics was provided. The conclusion drawn from this literature review is that of all open source and free database management systems, the PostgreSQL extension PostGIS provides the most complete list of features regarding geospatial support. However being a SQL type of database, some issues arise regarding scalability and performance as compared to NoSQL and NewSQL database systems.

In the state of the art review of section 2.4, several database management systems of different database classes were investigated. In the relational class regarding the SQL database type, the PostgreSQL extension PostGIS and MySQL were discussed. Regarding the non-relational NoSQL database type, MongoDB was discussed. Finally, the relational NewSQL database management system VoltDB was discussed. The state of the art review showed similar results as compared to the literature review. PostGIS had by far the most extensive set of features supporting geospatial data. PostGIS being completely free and open source is a big plus, as well as its extensive documentation and large user community. MySQL, MongoDB and VoltDB all offer community as well as enterprise solutions, and have a less rich set of supported geospatial features.

Each discussed system has its advantages and disadvantages, mainly regarding data integrity, geospatial data support, scalability, and performance. As was concluded from the literature review, choosing a database type is highly dependent on the system requirements. Further research is needed in order to determine a suitable data repository solution for the Climate Adaptive City Enschede graduation project, offering a combination of sufficient geospatial support and performance.

3. Methods and Techniques

This chapter describes the methods and techniques used for this bachelor thesis, in context with the research subject.

3.1 Interviews

Interviews with stakeholders will be used to determine and verify the (preliminary) requirements. Interviewing can be done by applying different types of interviews. Four of these types of interviews are explained below [30].

Structured interviews

With structured interviews, the questions are created prior to the interview and are the same for each respondent. The questions are mostly close ended and there is usually not much room for variation in responses. The interviewer has a neutral role, acting casual and friendly, and does not insert his or her own opinion.

Semi-structured interviews

Semi-structured interviews take place in formal setting. The difference with a structured interview is that the interviewer develops and follows an interview guide with topics that have to be addressed in the interview, usually in a particular order. The interviewer is allowed to deviate from the guide when he or she feels it is appropriate.

Unstructured interviews

Again this type of interview takes place in a formal setting, with both the interviewer and respondents being aware that an interview is taking place. There is no structured interview guide, but the interviewer does a plan regarding the focus and goal of the interview. The questions are usually open ended and have low control over the respondents answers, allowing the respondents to open up.

Informal interviews

The interviewer talks with respondents informally, without the use of an interview guide. The conversation has to be remembered by the interviewer, and he or she can make notes to help recall the conversation. Informal interviews allow respondents to speak freely and openly.

3.2 Design process for Creative Technology

The bachelor program Creative Technology aims to teach students to develop new and innovative products. It is a multidisciplinary program with the goal to produce engineers that are able to act as a "bridge" in multidisciplinary teams, which means being able to speak the language of different engineering disciplines, such as for example industrial design, interaction design, electrical engineering, and computer science. According to A. Mader and W. Eggink [31], design within the Creative Technology field lies between user centered design and classical engineering design approaches. They propose a Creative Technology specific design method, consisting of four phases, namely ideation, specification, realization, and evaluation as can be seen in figure 3.1. In the following sections each phase will be explained in relation to their use in this thesis.

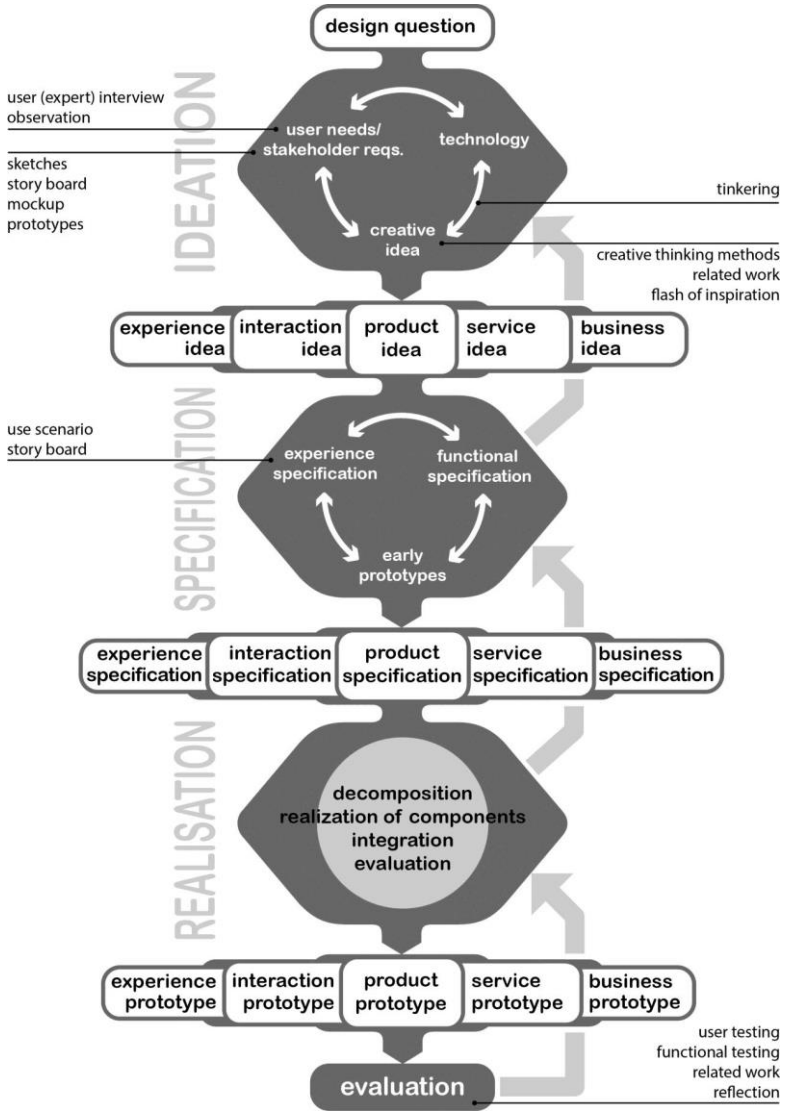

Figure 3.1 The Creative Technology design process [31]

3.2.1 Ideation

The Creative Technology design process starts with a design question, which in this case consists of the research questions stated in chapter 1. In this first phase of the design process, user needs/stakeholder requirements and technology can be starting points or motivational forces. The ideation phase in context of this graduation project will have technology as the starting point, primarily using multiple existing technologies as source of inspiration. Preliminary project requirements will be elicited from the stakeholders by the use of (informal) interviews and brainstorm sessions. These methods will are explained in section. The inspiration from looking at existing technologies and the preliminary requirements will be used to come up with a more elaborated project idea at the end of the ideation phase. Looking at the possible results of the ideation phase, an *experience idea, interaction idea, product idea, service idea, and business idea* as can be seen in figure 3.1, a *service idea* is likely to be the result. This is because this graduation project will provide data storage, data access, and geospatial analysis functionality as a service to the SRB and AirT projects rather than being an experience, interaction, product, or business idea.

3.2.2 Specification

Starting with where the ideation phase ends, a creative idea which in the case of this graduation project will be a service idea, the specification phase further explores the influence that user experience and functional specifications have on each other by using multiple prototypes. Short evaluation and feedback loops are used in order to determine shortcomings and strengths of prototypes, after which prototypes can be discarded, improved, or combined into new prototypes. By using the short evaluation and feedback loops it is also possible that new functional requirements arise, which can then possibly result in new prototypes. Driving factor in this phase is the user experience. Prototypes are often reduced to one or a few parts of the future product, each part being responsible for a certain part of the user experience. At the end of this phase the requirements for this graduation project will be final. In order to finalize the preliminary requirements, they need to be verified with the stakeholders of this project by the use of (informal) interviews and brainstorm sessions. The specification phase results in a complete service specification with which the realization phase will be started.

3.2.3 Realization

The realization phase starts with the conclusion of the specification phase, which is a complete service specification. This specification will be decomposed into subcomponents by looking at different the different roles that exist in the prototype. Each subcomponent can then be further analyzed by decomposing its functional requirements into three levels, each describing the component’s

functionality in more detail. The separate components are then realized and integrated into one prototype. Some functional testing will have to be done in this phase to validate whether the end service of a prototype meets the project's functional specifications. This will be done by writing applications simulating components of the stakeholder's projects. More extensive functional testing will be done in the evaluation phase.

3.2.4 Evaluation

Evaluation is the final phase of the Creative Technology design process. Although some functional testing has usually already taken place in the realization phase, further functional testing will be done in the evaluation phase in order to check whether all functional requirements are met. Also user testing can be used to check whether the user requirements and the intended experience are satisfied by the design decisions that were made during the project. Since there are no direct end users involved in this project, the main focus will be on testing the functional requirements. The testing done in this phase will be more extensive as compared to the functional testing in the realization phase. Next to testing the prototypes functionality, also the performance and storage capacity requirements will be tested. This will be done by writing applications simulating multiple data producing and consuming clients and analyzing measurements done on the system's performance and the amount of produced data.

3.3 Stakeholder Analysis

The approach for analysis of the stakeholders proposed by Sharp et al. [32] will be used, together with a power versus interest grid [33]. Sharp et al. [32] state multiple definitions for the term stakeholder, of which the following is believed to best describe the term in context of this graduation project:

"System stakeholders are people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements". [34]

First the stakeholders for this project will be identified by brainstorming, after which categorization will be done. H. Sharp et al. [32] distinguish between two types of stakeholders, namely baseline- and satellite stakeholders. For this project the focus will be on the baseline stakeholders, since they interact with the system directly. Baseline stakeholders will be divided into two meta group roles, namely that of producer or consumer. Also, baseline stakeholders will be assigned one or more of four types, namely users, developers, legislators, and decision-makers. After that, individual roles within the meta role groups and baseline stakeholder types will be assigned. Finally, each stakeholder will be placed

on a power versus interest grid, to determine which stakeholders to manage closely and which to just keep up to date.

3.3.1 Stakeholder roles

As mentioned in the previous section, the baseline stakeholders will be divided into the meta role group of either producer or consumer. Producer stakeholders produce information and provide this to the system. Consumer stakeholders retrieve, process, and use information from the system.

3.3.2 Stakeholder types

The types that can be assigned to a baseline stakeholder are that of user, developer, legislator, and decision-maker. Users are defined as the people, groups, or companies who interact with and control the system directly, and those who will use the products of the system such as information. Developers are stakeholders in the requirement process as well as the users, but have a different role in the requirement specification and the system itself as compared to the users. Roles within the developers baseline group could be for example analysts, designers, programmers, and project managers. Two examples of the legislator baseline type are professional bodies and government agencies, and finally the decision-makers are development team managers, user managers and financial controllers. The legislators and decision-makers can affect development and operation of the system by guidelines for operation, such as guidelines for costs, performance, security, and privacy.

3.3.2 Stakeholder prioritization

Next to categorization, each of the stakeholders will be placed on a power versus interest grid, of which an empty example can be seen in figure 3.2. The y-axis indicates the power a stakeholder has, ranging from low power to high power from bottom to top. The power indicates the amount of influence a stakeholder has on the development of the system. The x-axis indicates the stakeholder's interest in the development of the system, ranging from low interest to high interest from left to right. The diagram is divided into 4 equal sized sections. The sections are:

- Monitor (minimum effort) –*low power and high interest*
- Keep satisfied – *high power, low interest*
- Keep informed – *low power, high interest*
- Manage closely – *high power and high interest*

Depending on the combination of the power and interest, a stakeholder is placed in one of the four diagram sections.

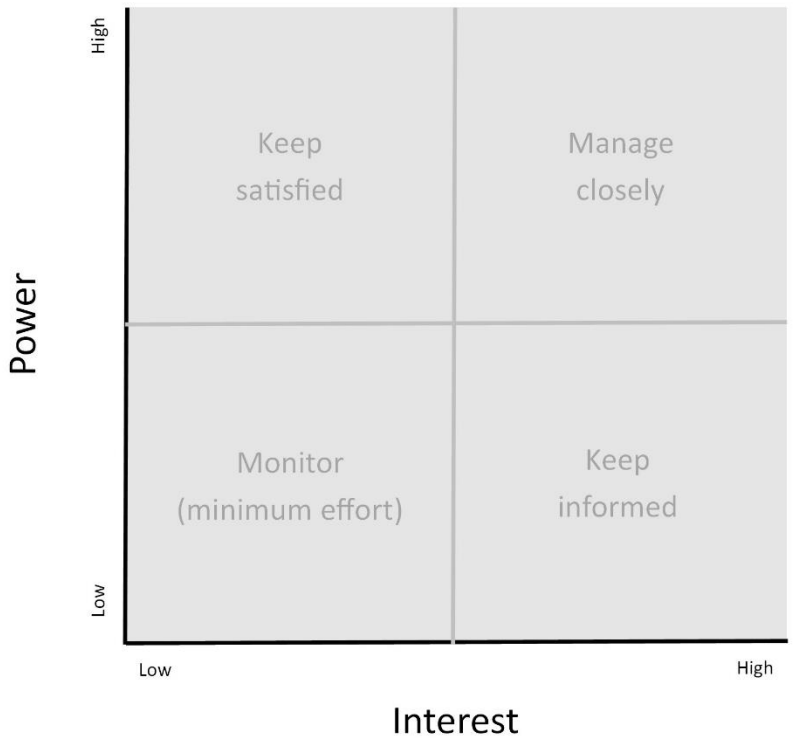

Figure 3.2, An empty power versus interest grid [33]

3.4 Requirement Analysis

The requirements will be determined by informal interviews with the stakeholders (see section 3.1). The requirements will first be categorized in functional and non-functional requirements. After that further categorization will be done using the MoSCoW method, which is explained in section 3.4.1. Functional requirements will describe functionality and behavior that the system should provide, and the non-functional requirements will specify the quality attributes of the system. The MoSCoW method is explained in the next section.

3.4.1 MoSCoW

In order to categorize the (non-)functional requirements, the MoSCoW method will be used. Categorization will be done by specifying the requirements as must have, should have, could have, and won't have requirements. All must have requirements must be included in the final solution. They are the most critical requirements that must be implemented within the current timeframe of the project. Should have requirements are important as well, but not as critical within the current project timeframe as must have requirements. These requirements should be implemented as much as possible after the solutions meets the must have requirements. A could have requirement is desirable, has lower priority than a should have requirement, and should only be implemented if it fits within the current timeframe and available resources of the project. Lastly, a won't have requirement is identified as the least

important requirement. Won't have requirements will either not be implemented at all, or possibly reconsidered for future work.

3.5 Scenarios

Scenarios will be constructed to describe how each of the clients will interact with the system, and to further specify their needs and the preliminary requirements. In order to construct these scenarios from both a user's as well as a designer's perspective, the PACT framework together with FICS will be used.

PACT Analysis

PACT analysis can be used to describe the user's perspective and to structure the construction of scenarios by identifying activities of people in different contexts and using different technologies. PACT stands for People, Activities, Context, and Technologies [35].

People

The people part of the PACT analysis for this project will consist of the different roles that the stakeholders have.

Activities

Each individual role of the stakeholders is accompanied with specific activities in interacting with the project. These activities are very important in constructing useful scenarios that describe the user's perspective.

Context

The context will describe the context in which the activities are performed.

Technologies

The technologies part focusses on the in- and output of data, the content, and communication.

FICS

FICS will be used to describe the designer's perspective, and stands for functions and events, interactions and usability issues, content and structure, and style and aesthetics.

4. Ideation

This chapter describes the ideation phase of this graduation project.

4.1 Stakeholder Identification and Analysis

4.1.1 Identification

First all stakeholders have been identified by brainstorming and drawing a scheme of the total CAC project, which can be seen in figure 4.1. As said in chapter 1, the CAC project consists of the following (graduation) projects excluding the data repository:

- SRB
- AirT
- RPS

Descriptions of each of the projects can be found in section 2.1.3.

The SRB project consists of a team of three, namely Jeroen Waterink, Thijs Dortman, and Sefora Tunc. Sefora will not be considered to be a baseline stakeholder for this project since she focusses on the D.I.Y. assembly instructions of the SRB and doesn't interact with the project directly. The other two, Jeroen and Thijs, focus on the design of a modular smart rainwater buffer and a user dashboard respectively. The other project, AirT, consists of a team of 2, Laura Kester and Adam Bako, of which both are considered baseline stakeholders within this graduation project. Laura Kester focusses on the design of a sensor module and Adam Bako deals with the visualization of the sensor data. Thijs works on the RPS as well, deploying a Lambrecht precipitation sensor.

Three other stakeholders exist within this graduation projects. These are the municipality of Enschede, the Vechtstromen water authority, and the University of Twente. They don't interact with system directly, and hence they're not considered to be baseline stakeholders within this project.

As has been said in section 3.3.1, the baseline stakeholders will be divided into the grouped role of either producer or consumer. Within the grouped role, the baseline stakeholders are assigned individual roles. The projects of Jeroen and Laura will both produce data. Therefore Jeroen and Laura will both be assigned the grouped role of producer. Their individual roles are that of SRB developer and AirT developer respectively. Thijs and Adam's projects will consume data and are therefore assigned the grouped role of consumer. Their individual roles will be that of SRB dashboard developer and AirT dashboard developer respectively. Thijs will also have the role of RPS developer, since he works on the RPS as well.

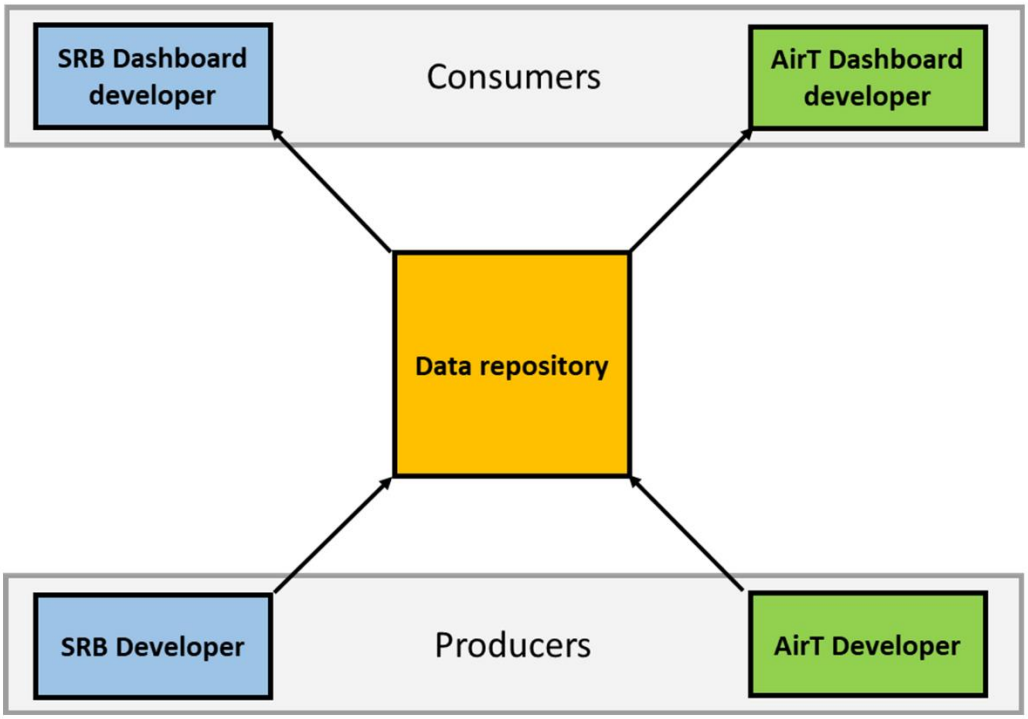

Figure 4.1 The CAC project from the data repository's perspective

In figure 4.1, the baseline stakeholders are grouped per project and per meta role group. On the left and indicated by blue are the SRB stakeholders and on the right and indicated by green are the AirT stakeholders. As said before, the RPS stakeholder is indicated by purple. Next to the grouping by project and specific individual roles, these stakeholders are grouped by their meta role in the CAC project, indicated by the grey boxes.

4.1.2 Categorization and prioritization

To give a clear image of the types of stakeholders, the baseline stakeholders are categorized using the baseline stakeholder types: users, developers, legislators, and decision makers (see section 3.3.2). Table 4.4 contains a list with this categorization. Next to categorizing the stakeholders, they are placed on a power vs. interest grid to give an indication of the power and interest each of the stakeholders have in the project and how close each of the stakeholders have to be managed.

Table 4.4 Stakeholder categorization

Stakeholder	**Type**
SRB developer	User – developer
SRB dashboard developer	User – developer
AirT developer	User – developer
AirT dashboard developer	User – developer
RPS developer	User – developer

On the power vs. interest grid in figure 4.3, the baseline stakeholders are indicated by the bold text. Although the municipality of Enschede, the Vechtstromen water authority, and the University of Twente are not considered baseline stakeholders in this project, they are included in the power vs. interest grid to give an indication of their power and interest in this project.

The municipality of Enschede and the University of Twente are placed on the border of the *keep satisfied* and *monitor* boxes of the grid. Due to their indirect influence via the baseline stakeholders, their interest in this specific project is considered low and their power medium. The Vechtstromen water authority cooperates with the municipality of Enschede, has low power and interest and is placed in the *monitor* box.

All baseline stakeholders are placed in the *manage closely* box. A top-down approach will be used to determine the dataset that is needed by the consumers (see section 4.2.3). This means that the dataset that needs to be produced by the producers will be determined by the dataset needed by the consumers. Therefor the SRB and AirT dashboard developers are placed higher on the power axis of the grid than the SRB and AirT developers. Their interest in this project is considered to be equal, since both producers and consumers will have more or less equal dependency on the data repository. The SRB dashboard developer is placed a bit higher on the power axis than the AirT dashboard developer, because it's expected that the SRB project will require more from the repository's functionality than the AirT project.

Finally, the RPS developer is considered to have less power and interest than the SRB and AirT projects, since this is an additional non-graduation project which will have lower priority for the RPS and SRB baseline stakeholder than the development of the SRB dashboard.

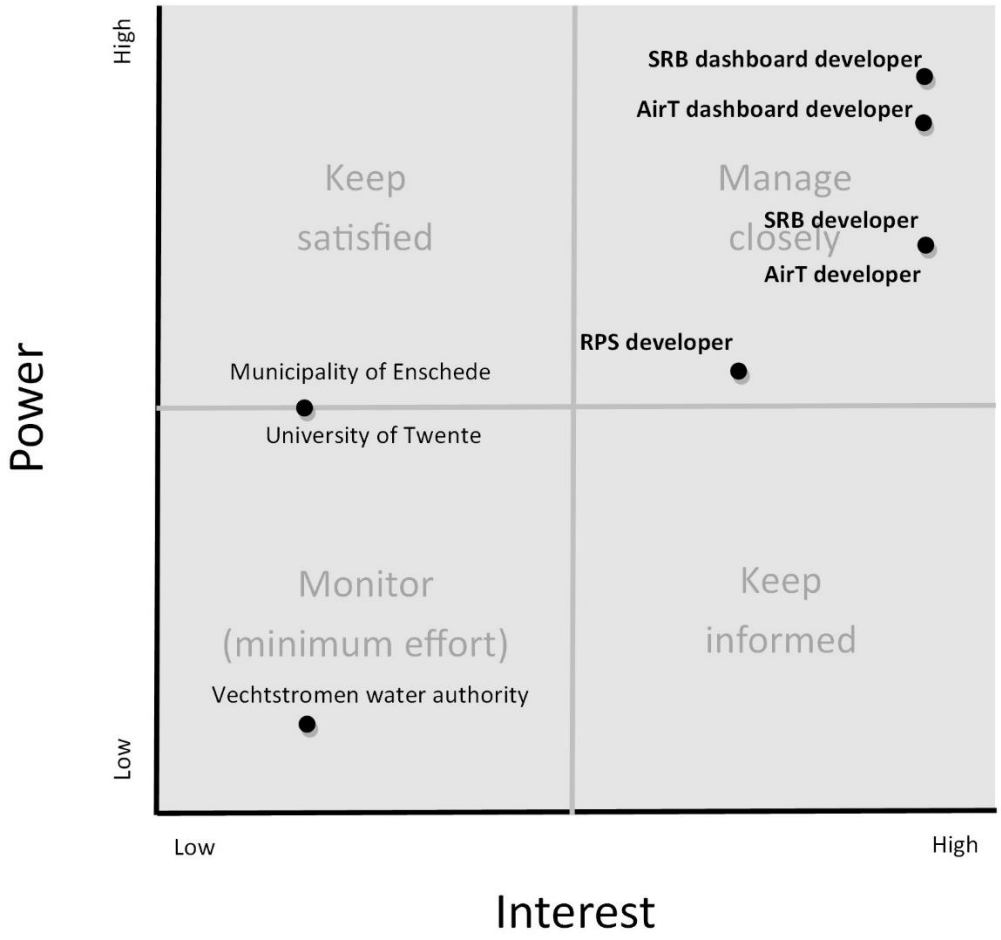

Figure 4.3, Stakeholder power vs. interest grid

4.2 Preliminary requirements

The requirements listed in this section are preliminary, since further specification and prioritization of the requirements will be done in the specification phase in chapter 5. The preliminary requirements listed in this section will be categorized in functional and non-functional requirements.

4.2.1 Functional requirements

- The system must be able to store geo-tagged data.
- The system must support multiple types of sensor system nodes (SRB, AirT, RPS).
- The system must be able to store sensor system node specific characteristics.
- The system should be able to provide a .csv formatted download of all raw data.
- The system must be application independent. (support multiple different visualization applications)
- The system could provide location metadata.

- The system must be able to provide location based data.
- The system must be able to provide specific sensor system node data.
- The system should be able to provide street level data.
- The system should be able to provide selected area data.
- The system should be able to provide clustered sensor system nodes data.
- The system should be able to provide time based data.
- The system must be able to store sensor system specific events.

4.2.2 Non-functional requirements

- The system must maintain performance with large amounts of stored data.
- The system must support clients producing data every 5 minutes minimum.
- The system should require low maintenance.
- The system must shield clients from complexity.
- The system must be reliable.
- The system must maintain data integrity.
- The system must handle concurrency.
- The system should be low cost.
- The system must be flexible.
- The system must run on a hardware platform

4.2.3 The dataset

A top-down approach is used in order to determine the dataset needed by the consumers for the visualizations. By informal interviews and brainstorming with the consumers (with the producers present as well), the preliminary required data was identified and communicated to the producers. See figure 4.4 for a visual representation of the used approach. The results from the informal interviews and brainstorm sesions, the required datasets, are listed below.

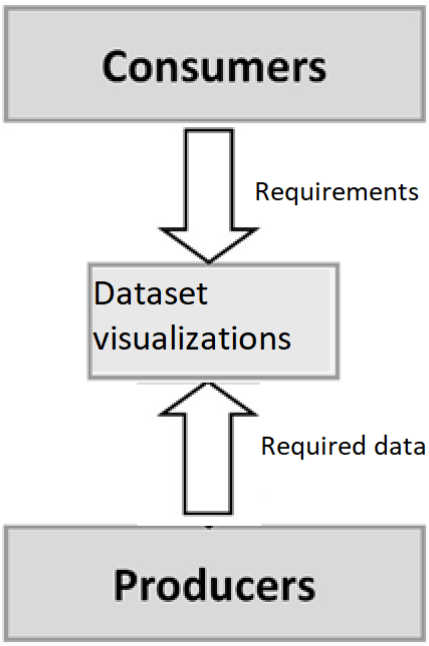

Figure 4.4, Top-down approach to determine data requirements

SRB data

The preliminary dataset needed by the SRB dashboard project is shown in table 4.5. SRB specific characteristics are needed for calculations and visualizations in the SRB dashboard application: location and buffer capacity. The current SRB fill level will be used to inform SRB users how much water is available in their SRB for private usage. The historic amount of buffered rainwater is needed to compare a SRB user's buffering performance with other SRB users buffering performances. The planned discharges are used to visualize what is expected to happen with the buffered rainwater in a user's SRB, along with the accompanied predicted precipitation. Regular discharges are used to visualize what actually happened regarding discharges, again with the accompanied precipitation. The manual discharges will indicate when buffered water has been drained manually using the water tap. For legionella and frost warnings, the water temperature is needed. The historic precipitation will be produced by the RPS project.

Table 4.5 Data and units needed by the SRB dashboard project

Data type	**Unit**
Location	Latitude - longitude
Buffer capacity	Liters
Current fill level	Liters
Historic amount of buffered rainwater	Liters per time period
(Manual) discharge	Date, time, desired fill level in liters, precipitation in mm/h
Planned discharge	Date, time, desired fill level in liters, precipitation in mm/h
Precipitation predictions for SRB location	mm/h
Historic precipitation for users location	Per day for past month Per month for past 2 years
Water temperature	Degrees Celsius

AirT data

The preliminary dataset needed by the AirT dashboard project can be found in table 4.6. Again some sensor system characteristics are important for calculations and visualization: the location and the skyview factor. This skyview factor is the percentage of visible sky for the AirT location. This means sky not blocked by for example buildings or trees. This factor will be used to calculate solar radiation. The data that will be visualized is the air temperature, wind speed, and humidity.

Table 4.6 Data and units needed by the AirT dashboard project

Data type	**Unit**
Location	Latitude - longitude
Air Temperature	Degrees Celsius
Wind speed	m/s
Humidity	Percentages
Skyview factor	Percentages

4.3 Concept

For this project, in order to provide database storage and access and geospatial functionality to the CAC sub-projects, a custom client interface will be developed. This interface will be used to abstract between the different layers of the total project. It will provide the clients with general functionality in order to insert and retrieve data from the database, of which a visual representation can be found in figure 4.4. The interface will be generic, offering producers input possibilities to register a new sensor system node, insert sensor system measurements, and add events such as a rainwater buffer discharge. The consumers will have the possibility to retrieve point based, street level based, and clustered data as well as a .csv formatted download of stored data. The meaning of point based data, street level data, and clustered data is explained in more detail below.

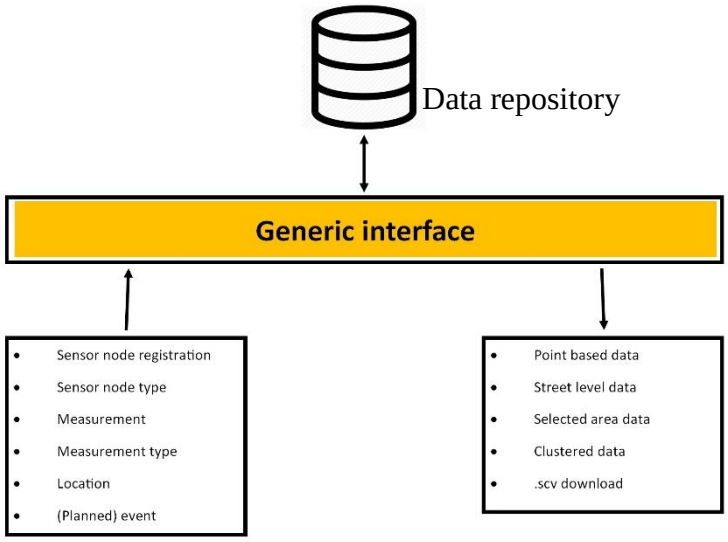

Figure 4.4, Database system structure and provided functionality

Point based data

Client applications, such as the SRB and AirT dashboard applications, will be able to retrieve data for a certain point location by making a point location request to the generic interface. If multiple sensor system nodes exist on that particular location, selection of one or more sensor system nodes will be possible.

Street level data

For privacy reasons with the SRB dashboard application, users will only be able to see the performance of other SRB users at street level. This was decided with the SRB stakeholders and the municipality

during an informal interview. The system will be able to calculate and provide averages for multiple sensor system nodes on street level, represented by a geometry of the type *linestring*.

Area selection

Consumer applications will be provided the functionality to select an area on a map, returning data for all sensor system nodes contained within the selected area.

Clustered data

In the early stages of the CAC project, it is expected that an insufficient number of SRBs will be deployed to meet the privacy requirements by providing street level data only. There is a requirement for a function to cluster sensor system nodes together and specify a minimum amount of sensor system nodes per cluster.

Following the preliminary requirements and the grouped roles of producer and consumer that were assigned to the projects that will interact with the repository, three generic interfaces will be developed: a consumer interface, a producer interface, and an administrator interface. Both the producer and consumer interfaces will be universal instead of sensor system specific. Sensor system specific functionality of the producers and consumers, if required, will be implemented by the producers and/or consumers themselves. The interfaces shield producers, consumers, and administrators from complexity of the repository side programming by offering functions that can be called, and to improve maintainability of the repository. A scheme of this high level architecture can be found in figure 4.5. The producers are the SRB, AirT, and RPS developers. The consumers are the SRB and AirT dashboard developers. If required, for example for data formatting, client specific interfaces will be needed to be developed by the students working on the specific projects themselves.

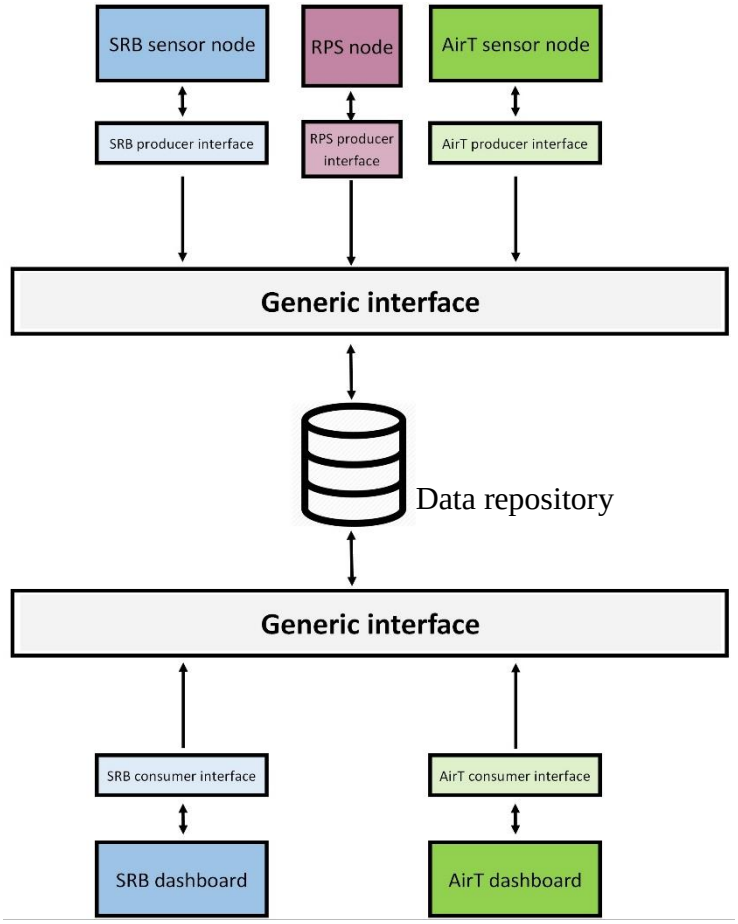

Figure 4.5, High level architecture

5. Specification

In this chapter the preliminary functional and non-functional requirements listed in chapter 4 will be further specified and prioritized by the use of the MoSCoW method explained in chapter 3. After that the database table structure will be explained. Finally, the final concept idea in chapter 4 will be further described by a 3 level deep analysis of the functional architecture.

5.1 Requirements

In this section the system requirements will be prioritized using the MoSCoW method. Determining the priority of each of the system's requirements is done by verification in informal interviews with the project stakeholders. Since a good specified requirement only describes one single function or attribute, some preliminary requirements from chapter 4 are split up in multiple separate requirements. Also, by describing each requirement as a single functionality or attribute, the requirements can be prioritized more precisely.

5.1.1 Functional requirements

The prioritized functional requirements can be found in table 5.1. As mentioned above, some of the preliminary requirements of chapter 4 have been split up in several requirements, in order to describe only one single system functionality or attribute. The system must at least support SRB, AirT, and RPS nodes along with the types of measurements that have been determined by and verified with the stakeholders. This means for example that supporting those three types of sensor system nodes are prioritized as *must have* requirements, and the more general requirement *be sensor type independent* is prioritized as *should have* requirement.

All RPS requirements are considered could have requirements, since the RPS developer's priority lies with the development of the SRB dashboard and thus minimizes the time spend on the RPS. Both street level data and area selection will not be implemented by the SRB dashboard developer and the AirT dashboard developer, so these requirements are prioritized as *won't haves.* The alternative to the street level data, the clustered nodes data, is dropped by the SRB dashboard developer and does not apply to the AirT dashboard developer. The SRB dashboard developer's stakeholders don't consider this functionality useful in this stage of the SRB project. Instead, the SRB dashboard developer will provide comparison of an SRB user's buffered amount of rainwater with the city's total amount of buffered rainwater for last month. Calculation of this data will be implemented in the consumer interface. This is considered extra work and therefor prioritized as a could have requirement.

Table 5.1. Prioritized functional system requirements

The system must
be able to store geo-tagged data.
support multiple data producing sensor system nodes.
support multiple data consuming applications.
support SRB nodes.
support SRB dashboard application.
support AirT nodes.
support AirT dashboard application.
be able to store SRB node (historic) location.
be able to store SRB legionella protection setting**.**
be able to store SRB frost protection setting.
be able to store SRB buffer capacity.
be able to store SRB fill level.
be able to store SRB water temperature.
be able to store SRB planned discharge.
be able to store SRB autonomous discharges.
be able to store SRB manual discharges.
be able to store AirT node (historic) location.
be able to store AirT air temperature.
be able to store AirT humidity.
be able to store AirT wind speed.
be able to provide a .csv formatted download of all stored AirT data.
be able to provide current location of SRBs.
be able to provide current location of AirTs.
be able to provide historic locations of SRBs.
be able to provide historic locations of AirTs.
be able to provide point location data to the SRB dashboard application.
be able to provide sensor system specific data to the SRB dashboard application.
be able to provide time selection data to the SRB dashboard application.
be able to provide point location data to the AirT dashboard application.

be able to provide time selection data to the AirT dashboard application.
be able to provide the data to the AirT dashboard application in json format.
be able to provide the data to the SRB dashboard application in json format.
The system should
be application independent.
be sensor system type independent.
be measurement type independent.
be able to store sensor system specific events.
be able to store sensor system specific characteristics.
The system could
be able to store an SRB user's username and password
be able to provide last month's single SRB buffered rainwater amount to the SRB dashboard application.
be able to provide last month's total SRB buffered rainwater amount to the SRB dashboard application.
be able to provide clustered sensor system nodes data to the SRB dashboard application.
Be able to support RPS nodes.
be able to store RPS node (historic) location.
be able to store RPS precipitation measurements.
be able to store precipitation prediction data belonging to a discharge event.
be able to provide a .csv formatted download of all stored SRB data.
be able to provide a .csv formatted download of all stored RPS data.
offer sensor system type registration.
offer measurement type registration.
offer characteristic type registration.
offer event type registration.
The system won't
be able to provide street level averaged data.
be able to provide averaged sensor system node data by area selection.

5.1.2 Non-functional requirements

In this section the preliminary non-functional requirements are prioritized using the MoSCoW method. Table 5.2 shows the prioritized non-functional requirements.

Table 5.2. Prioritized non-functional system requirements

The system must
maintain performance with large amounts of stored data.
require low maintenance.
shield clients from complexity.
be reliable.
run on a single hardware platform.
maintain data integrity.
handle transaction concurrency.
support multiple client connections.
support frequent data input.
support frequent data output.
The system should
be flexible.
The system could
be low cost.
The system won't
be scalable.

5.2 Table structure

The table structure for the database will be explained in this section. Data can be either a fact or a dimension. A fact is a numerical measure for which the dimension(s) provide the context [36]. A fact is described in one table row per combination of dimensions. First the fact(s) and dimension(s) will be determined by determining which questions the data should answer, so that a star schema can be drawn to visualize the fact(s) and dimension(s). This star schema will provide the basis upon which the table structure will be designed. This section will start with a separate analysis for each of the subprojects, after which found similarities will lead to a universal table schema.

5.2.1 RPS

Star schema

The RPS will produce location and time based precipitation data, in order to provide (historic) precipitation measurements that can be used by the SRB project to determine the SRB buffering performance to some extent. The question that needs to be answered with this data is: *What is the historic precipitation for a certain RPS node?* The facts, dimensions and their attributes following this question can be found in table 5.3. The fact in this case is the *precipitation* and the dimension is the *RPS node*. There will be one row of precipitation data per combination of RPS node and time. The attributes of *precipitation* are *amount* and *time.* The attribute for the *RPS node* is *location.* The associated star schema can be found in figure 5.1.

Table 5.3. Fact, dimensions, and attributes for the RPS.

Fact	Dimension

• **Precipitation**
  ○ Amount
  ○ Time
	• **RPS node**
	  ○ location

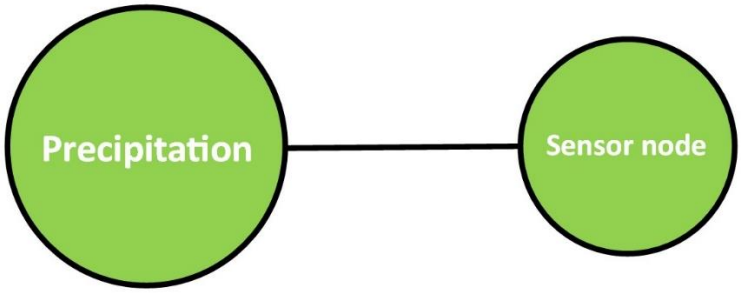

Figure 5.1. Star schema for the RPS

Table structure

For the RPS project the table structure can be found in figure 5.2. The schema consists of the fact table *precipitation* and the dimension table *RPS_node*. The *location* attribute of the *RPS_node* dimension is a separate table, since the location of a RPS node can change and thus requires a timestamp. When the RPS node location has changed, the historic RPS node location has to be preserved. The precipitation table contains time based precipitation measurement values for individual RPS nodes, linked by the RPS_nodeid which uniquely identifies each RPS node. The RPS_node table contains a unique identifier for all deployed RPS nodes.

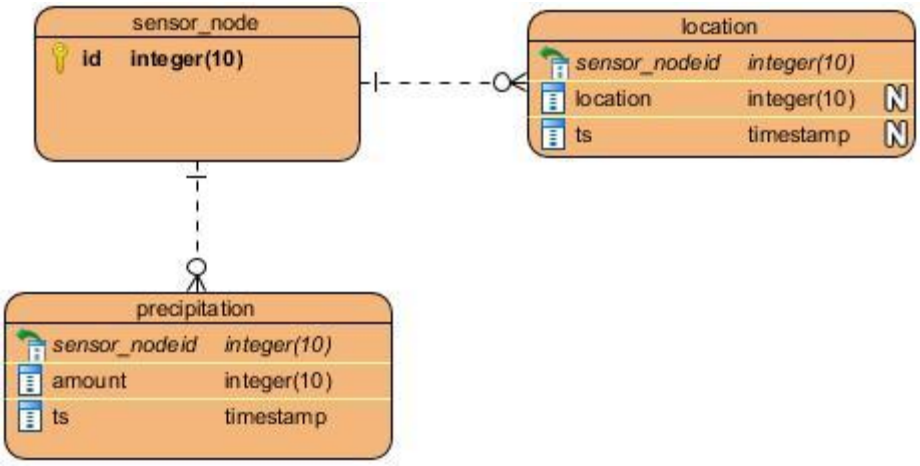

Figure 5.2. Table structure for the RPS

5.2.2 AirT

Star schema

The AirT project consists of multiple deployed AirT sensor system nodes. Each of these nodes will produce location and time based environmental measurements: air temperature, humidity, and wind speed, and possibly solar radiation. In order to determine the solar radiation, a one-time per AirT node location skyview factor will be used. The question that needs to be answered is: *What are the (historic) measured values for a certain AirT sensor node?* The fact, dimensions, and attributes can be found in table 5.4. The fact in this case is a *measurement*, containing the attributes *air temperature, humidity, wind speed,* and *time.* The solar radiation was prioritized and verified with the AirT stakeholders as a could have requirement, since at this stage of the project solar radiation will not yet be calculated. The dimension is *AirT node*. The attributes for the *AirT node* is *location.* The star schema for the AirT project can be found in figure 5.3.

Table 5.4. Fact, dimensions, and attributes for the AirT.

Fact	Dimension

• **Measurement**
  ○ Air temperature
  ○ Humidity
  ○ Windspeed
  ○ Time
	• **AirT node**
	  ○ Location

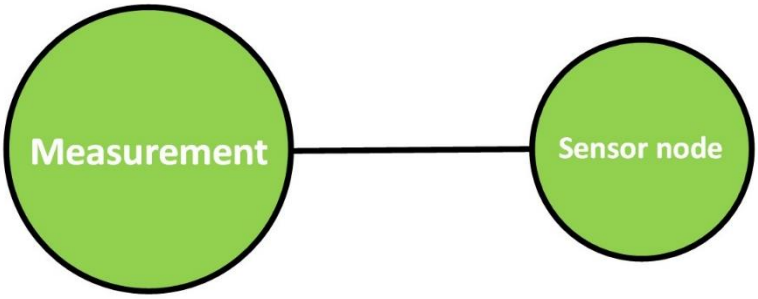

Figure 5.3. Start schema for the AirT & SRB

Table structure

For the AirT project the table structure can be found in figure 5.4. For the AirT nodes the historic location is of importance as well. Therefor here the attribute *location* is a separate table again, linked to an AirT node by the AirT node's unique identifier and containing the latitude, longitude and timestamp of the location. The fact table *measurement* contains the attributes *temperature*, *humidity*, *windspeed*, and the timestamp *ts*. Each measurement in this table is linked to an *AirT_node* by the use of the AirT node's unique identifier.

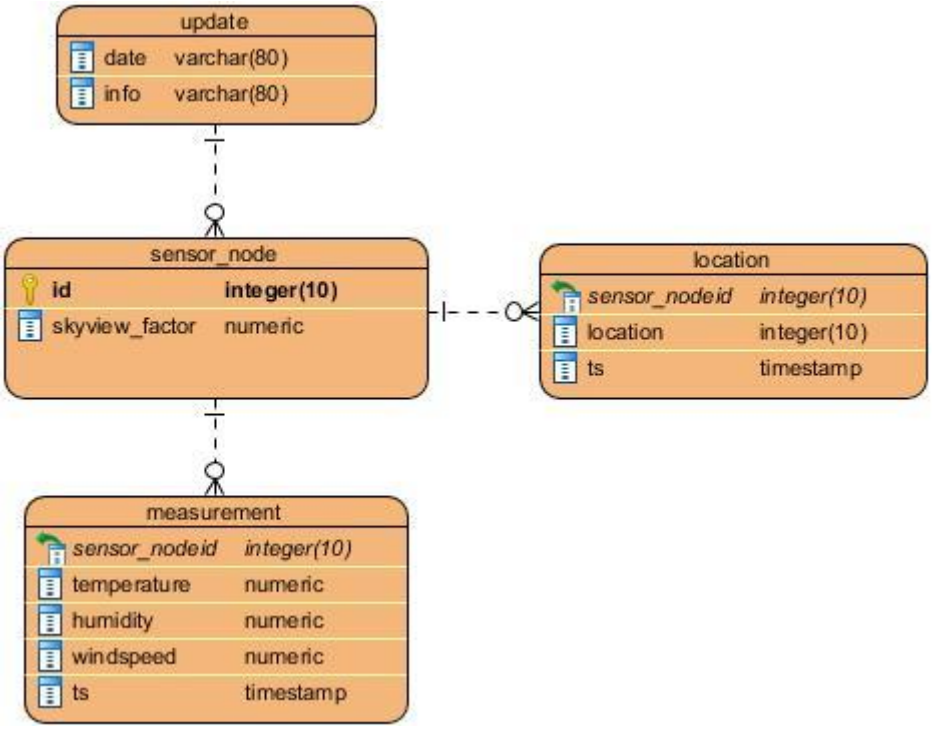

Figure 5.4. Table structure for the AirT

5.2.3 SRB

Star schema

The SRB nodes produce location and time based measurements as well. The measurements provided by each SRB node are the *buffer fill level* and *water temperature.* The question that needs to be answered with this data is: *What are the (historic) measured values for (multiple) SRB nodes?* The fact, dimensions, and attributes can be found in table 5.5. Again, the fact for the SRB is *measurement*, with attributes *fill level* and *water temperature.* The dimensions are the *SRB node* and *time.* The *SRB node* dimension has attributes *location*, *capacity*, *planned discharge*, *manual discharge*, and *discharge.* The star schema for the SRB has the same structure as the AirT and can be found in figure 5.2.

Table 5.5. Fact, dimensions, and attributes for the SRB.

Fact	Dimension

• **Measurement**
  ○ Fill level
  ○ Water temperature
  ○ Time
	• **SRB node**
	  ○ Location
	  ○ Capacity
	  ○ planned discharge
	  ○ manual discharge
	  ○ discharge

Table structure

For the SRB project the table structure can be found in figure 5.5. The attribute *location* is again a separate table. The location is linked to a SRB by the SRB_node's unique identifier. Additional to the table structure of the AirT, tables for planned, manual, and regular discharges were added.

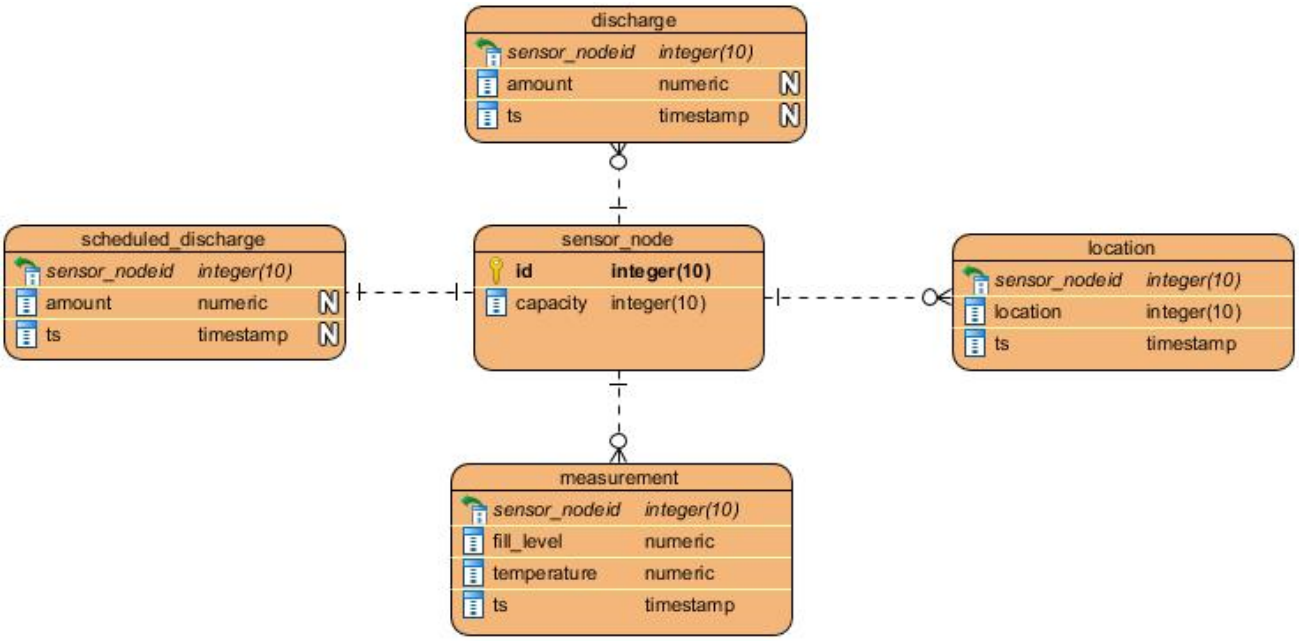

Figure 5.5. Table structure for the SRB

5.2.4 Universal solution

As said in section 4.3, the producer and consumer interfaces will be universal. This means that any type of sensor system or application can use the interface's functionalities. In order to support the universal nature of the interfaces, the underlying database structure should be universal as well. By looking at the similarities of the individual table structures of the three pre-mentioned projects, a universal solution was designed and verified with the stakeholders. The idea behind the solution will be explained by explaining separate parts of the total table structure. Finally, the total table structure will be discussed by putting all separate parts together.

Sensor systems

In order to design a universal table structure for the database, the table structure should be sensor system type independent. Instead of having separate tables for the SRB, AirT, and RPS types of sensorsystems, the table structure has 1 sensor systems table and a sensor system types table (see figure 5.6). Let's first take a look at the *sensorsystems* table. In this table, each registered sensor system has an *id* and a *sensorsystem_type_id*. The id is used to uniquely identify individual sensor systems, and the sensorsystem_type_id is linked to a sensor system type in the *sensorsystem_types* table. This table contains all types of sensor systems that are allowed to register. The table as columns for the id and the type in human readably text, for instance the srb, airt, and rps. Sensor systems can register using the producer interface, as long as the type of sensor system exists in the sensorsystem_types table. Whenever a new type of sensor system has to be added to the repository, the only thing that has to be

done is for the administrator to add the new type to the sensorsystem_types table using the administrator interface. From that moment on sensor systems of the newly added type can be registered in the sensorsystems table. The circle symbol on the sensorsystems table side of the dashed line, indicates that a sensorsystem_type can have zero or more sensorsystems. The perpendicular dash on the sensorsystem_types table side of the dashed line, indicates that each registered sensor system can have only one sensor system type.

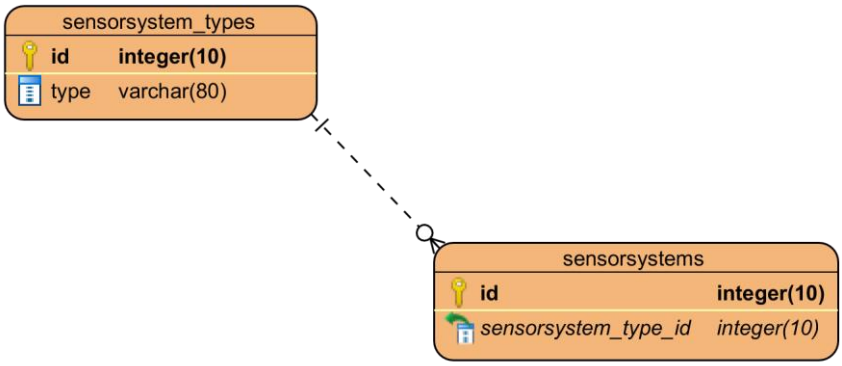

Figure 5.6, sensorsystems and sensorsystem_types tables

Locations

Next to the fact that each registered sensor system has a in the sensorsystem_types table defined type, it also has a location. The *locations* table is shown in figure 5.7. The table has columns *sensorsystem_id, location, ts,* and *precision.* The sensorsystem_id is a foreign key, linking a location to a certain sensorsystem. The location is the location of the sensorsystem. The ts stands for timestamp, which is required since sensor systems can change location, and historic locations have to be preserved. The precision column is nullable. It is for sensor systems optional to add a value for the precision of the location. The location is the latitude longitude pair of the sensor system's location on earth. Each sensor system can have multiple locations, and each combination of location and timestamp can have one sensor system.

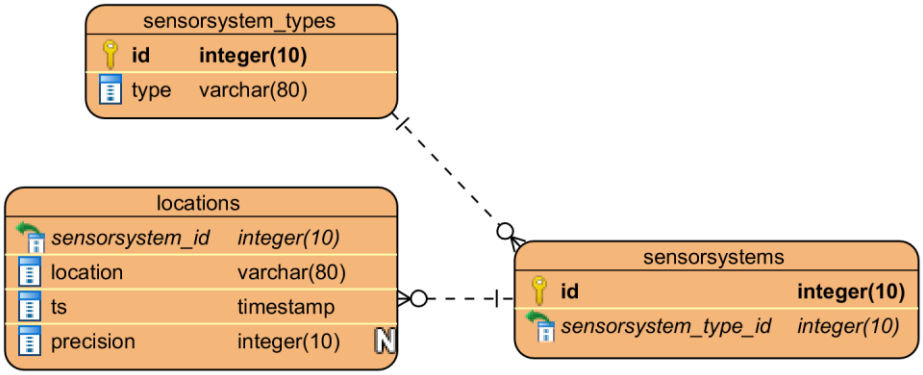

Figure 5.7, sensorsystems, sensorsystem_types, and locations tables

Characteristics

Recall the universal solution of the sensorsystems and sensorsystem_types tables: a sensor system can only register when the type of sensor system is defined in the sensorsystem_types table, and adding a new type of sensor system is as simple as adding just one row in the sensorsystem_types table. The same idea is used for the characteristics. Taking an SRB as example, a characteristic could be the SRB capacity. So when an SRB node registers itself, assuming the type SRB is defined in the sensorsystem_types table and thus registration is allowed, the characteristic capacity could be included in the registration. Looking at the tables *characteristics* and *characteristic_types*, it is clear that the table pair uses the same principle as the sensorsystems and sensorsystem_types tables. A characteristic is linked to a *sensorsystem_id*, a *characteristic_type*, and has a *value*. A characteristic type has an *id*, a *type* in human readable text, and is linked to a *sensorsystem_type_id*. The latter means that characteristics are sensor system type specific, for example the characteristic capacity belongs to the sensor system type SRB. When an SRB wants to register with a capacity of 900 liters, the characteristic type capacity has to be defined in the characteristic_types table, linked to the sensorsystem_type_id belonging to the type SRB. Again, when a sensor system acquires a new type of characteristic, all that needs to be done is for the administrator to add the new type to the characteristic_types table.

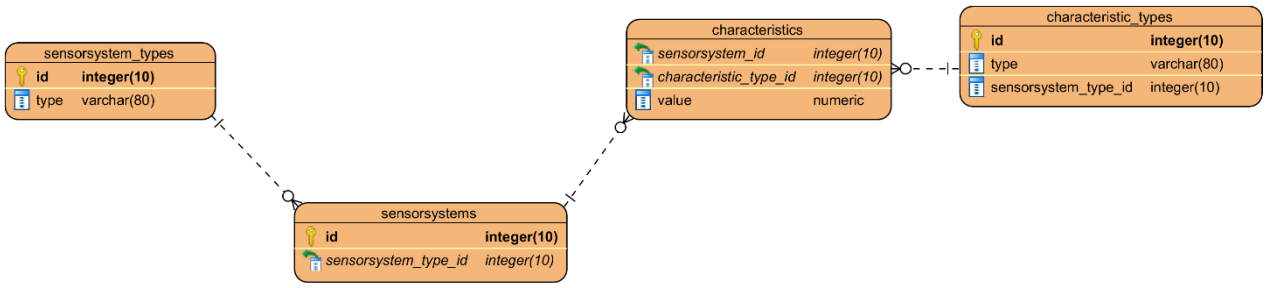

Figure 5.8, sensorsystems, sensorsystem_types, characteristics, and characteristic_types table

Measurements

All projects produce measurements of different types. The SRB produces fill level and water temperature measurements. The AirT produces air temperature, humidity, and wind speed measurements. The RPS produces precipitation measurements. Having all these different types of measurements as columns in one measurements table would cause for the table to contain a large number of empty fields. Also this would not be a universal solution, since the measurements table has to be altered when a sensor system would expand with a sensor measuring a not yet in the table existing phenomenon. Instead of having columns for each type of measurement in a single table, the same universal idea is used for measurements (see figure 5.9). In the *measurements* table, each row represents a measurement of a single type indicated by the *measurement_type_id*, linked to a single sensor system by the *sensorsystem_id*, with accompanied value and timestamp. The *measurement_types* table contains all types of measurements that are allowed, linked to a certain type of sensor system. Taking the SRB again as example, measurements types will be the fill level and water temperature. As long as the measurement types are defined in the measurement_types table and linked to the type of sensor system, a registered sensor system can register measurements of the defined measurement types. Adding a new type of measurement for a certain sensor system type is as easy as adding one row in the measurement_types table, which will be done by the repository administrator.

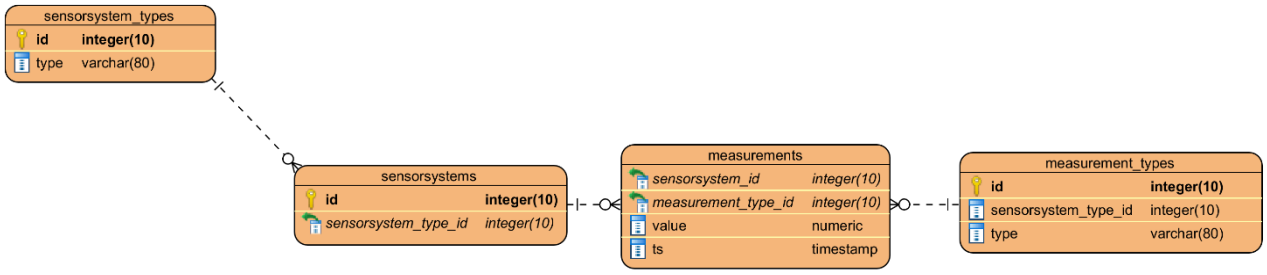

Figure 5.9, tables for the measurements and measurement types

Events

Again, the same idea is used for the *events* and *event_types* tables. For a sensor system to register an event, the type of the event has to exist in the event_types table. Events can be for instance, taking the SRB again as example, manual, autonomous, or planned discharges. See figure 5.10 for the events and event types tables.

The events table contains columns with foreign key *sensorsystem_id*, which links an event to a sensor system, foreign key *event_type_id*, which links an event to an event type, two columns for values belonging to an event, and a column for the timestamp. An event can thus have two values, which resulted from the requirement that precipitation prediction data belonging to a SRB discharge

event should be stored as well as the desired capacity of the SRB. The second value column, *value2*, is nullable since some types of events could also require just one value to be stored.

The event_types table contains the columns *id, type,* and *sensorsystem_type_id.* The id is used to uniquely identify event types. The type column contains the type of event in human readable text. The sensorsystem_type_id column links specific events to a specific type of sensor system. The event type manual discharge for example belongs to the sensor system type SRB, so only a SRB can register events of the type manual discharge. New event types can be added to the table by the repository administrator via the administrator interface.

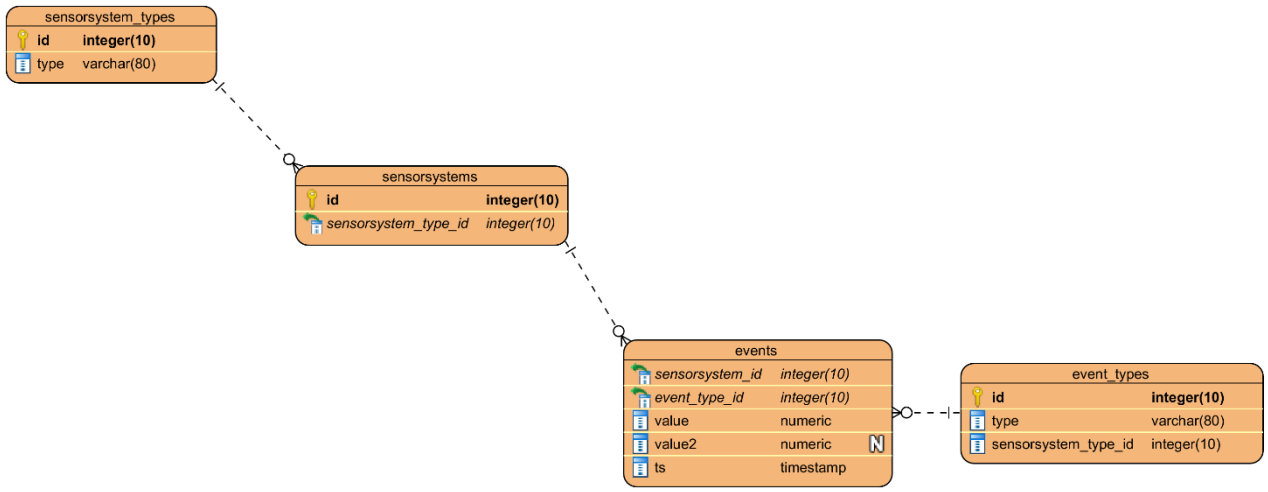

Figure 5.10, tables for the events and event types

Users

The requirement for the storing of SRB owners usernames and password followed from the privacy issues explained in section 4.3. An individual SRB owner can only see other SRB owner's buffering performance in a merged manner. Individual data can only be accessed by the users themselves. When a new SRB node registers, a username and password will be linked to the SRB node in the *users* table (see figure 5.11). This table contains the columns *id, username, password,* and *sensorsystem_id.* The id uniquely identifies a user, the username is the user's username, the password is the user's password, and the sensorsystem_id is a foreign key linking a user to a sensorsystem.

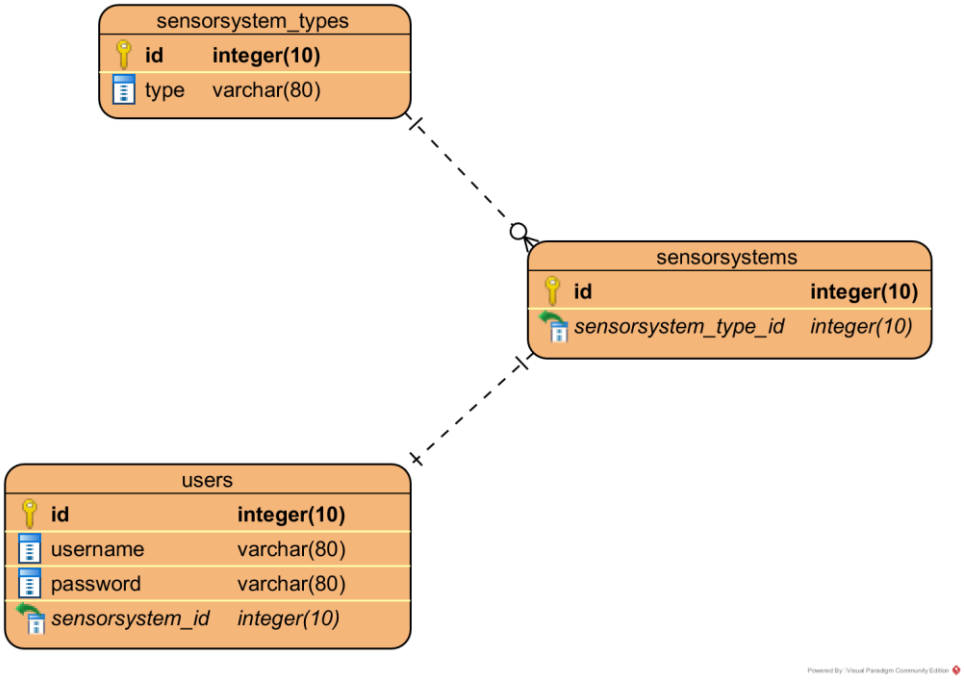

Figure 5.11, the users table.

Total table structure

Putting together the tables *sensorsystems* and *sensorsystem_types* for the sensor systems, *locations* for the locations of the sensor systems, *characteristics* and *characteristic_types* for the sensor system characteristics, *measurements* and *measurement_types* for sensor system measurements, *events* and *event_types* for sensor system events, and *users* for sensor system users, brings us to the total table structure of figure 5.12. The total table structure will be further elaborated on by examples in section 5.4.

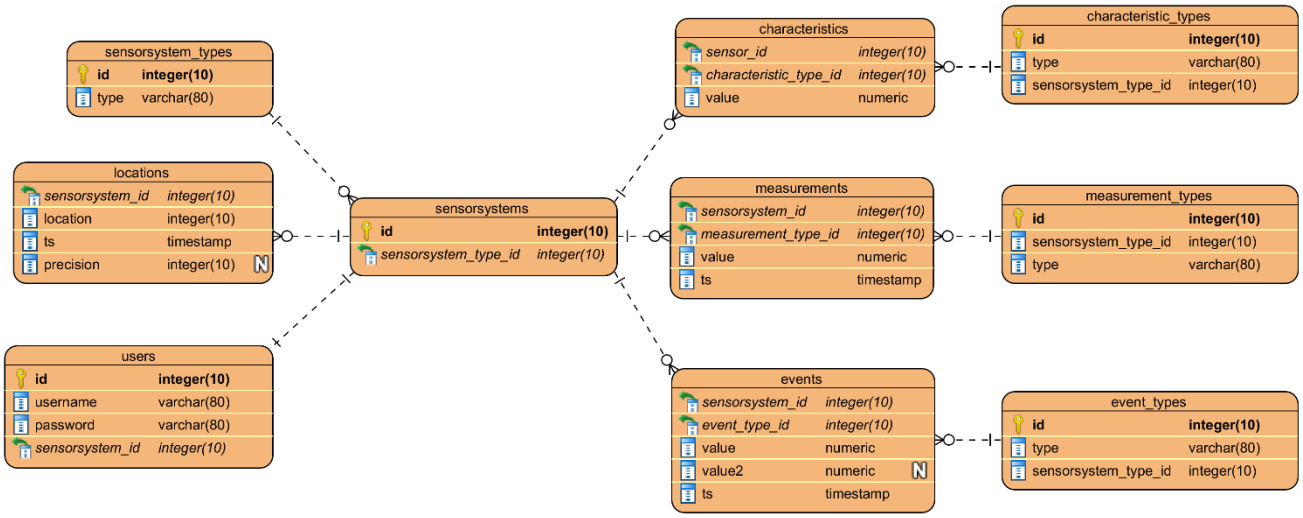

Figure 5.12, the total table structure

5.3 Functional architecture

In this section the functional architecture will be discussed by a three level deep decomposition of the project. Starting with level 0, an overview of the repository's functional architecture will be discussed. Level 1 will describe the repository's functional architecture in more detail, and finally in level 2 the interface's default functional architecture will be discussed in detail. Some extra stakeholder specific functionalities have been implemented, which will be discussed in chapter 6.

In the figures in the following sections, white block arrows indicate data transfer between producer, consumer, administrator, and their corresponding interfaces. The black filled arrow indicate http response status codes in case of the producer and administrator, and data requests in case of the consumer.

5.3.1 Overview

The functional architecture at level 0 (figure 5.13) simply consists of the repository represented by a black box along with everything that goes in and out of the repository. In the middle is the data repository, represented by the grey colored box. The required input functionality is listed on the left. This includes registration of a user, registration of a sensor system, registration of sensor system's characteristics, measurements, and events, and registration of sensor system types, characteristic types, event types, and measurement types. On the right the required output functionality is listed. This includes obtaining the sensor system id belonging to a certain user, the current and historic location(s) of all or a single sensor system, sensor system characteristic(s), measured data for a certain sensor system, registered events, last month's amount of buffered rainwater for a single SRB as well the total amount of buffered rainwater for last month, and the possibility to download a .csv formatted file of all raw data.

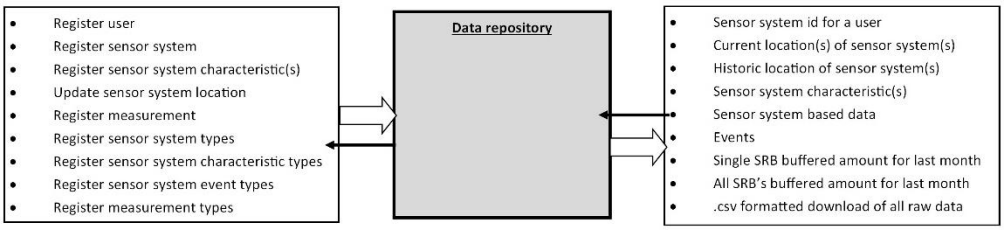

Figure 5.13, Data repository decomposition level 0

5.3.2 Repository

In the previous section, all input functionality is grouped together. However, if we take a closer look we can see that the bottom five points in the list don't belong to the producers role. These are functionalities provided in the administrator interface (see section 4.3 and 5.2.4). Therefor at decomposition level 1, the black box representing the data repository is show with more detail in figure 5.14. In this figure the required input and output functionalities are split up in parts belonging to the roles of producer, consumer, and administrator. Here it is clear that the role specific functionality will be implemented in three separate interfaces. These interfaces will translate requests made to an interface into SQL commands for inserting into and retrieving data from the database, which lies at the center of the repository, indicated by the blue box in figure 5.14.

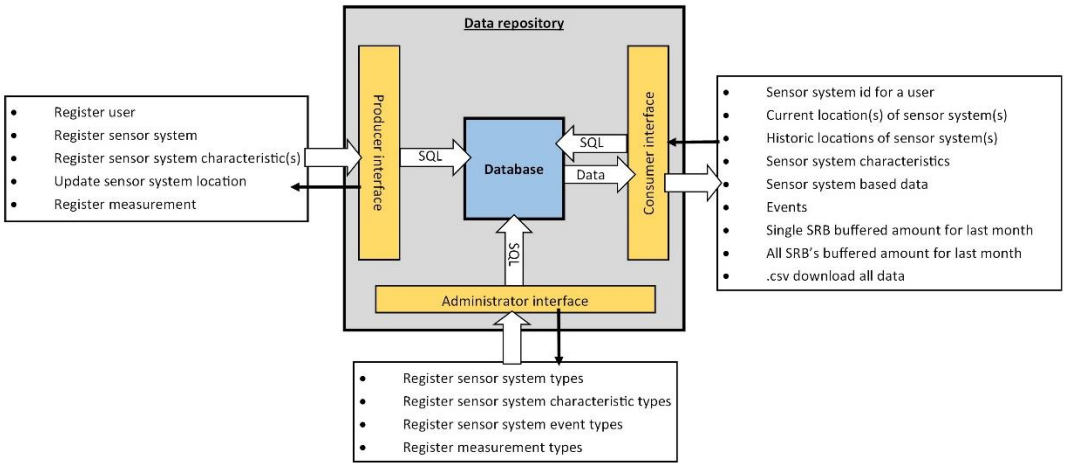

Figure 5.14, Data repository decomposition level 1

5.3.3 Producer interface

The producer interface provides the functionality to register a user, register a sensor system, register sensor system characteristics, update the sensor system's location, and register measurements. All these functionalities will be described below.

User and sensor system registration

The sensor system registration lies at the basis of this part of the producer interface's functionality. This basis is extended with the possibility to register a sensor system along with a user. First the basis will be described. The basic registration of a sensor system is visualized in figure 5.15. Here a producer requests registration of a sensor system and provides the interface with the sensor system's serial number, the type of sensor system, and the sensor system's location. The interface will check the validity of the request, by checking whether the serial number provided is unique and whether the type

of sensor system is registered in the sensorsystem_types table (see section 5.2.4). If the request is valid, the interface will use SQL commands to insert the data into the database. In both cases the producer will receive a response, informing whether the made request was successful.

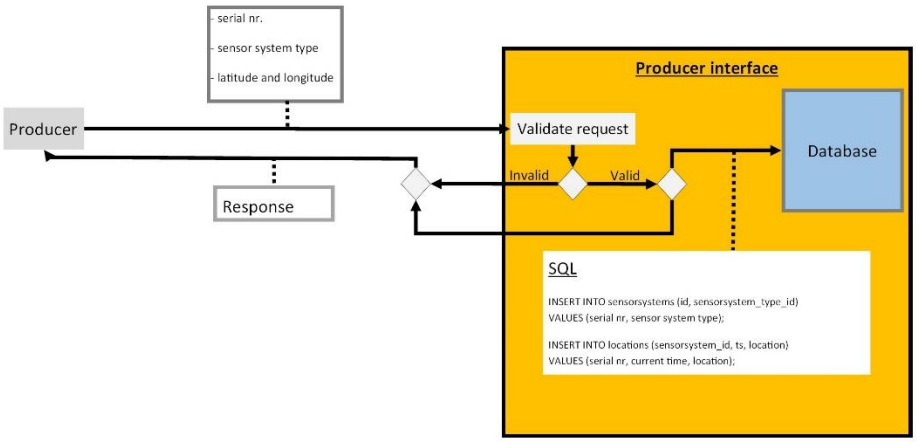

Figure 5.15, Registration of a sensor system

Additionally to the basic functionality of sensor system registration, a producer can add a value representing the precision of the latitude longitude pair. When the precision of the location is provided in the sensor system registration request, the precision will be inserted into the locations table's column *precision* as well (see figure 5.16). Again, in both the cases where the request was valid or invalid, the producer will receive a response, informing whether the made request was successful.

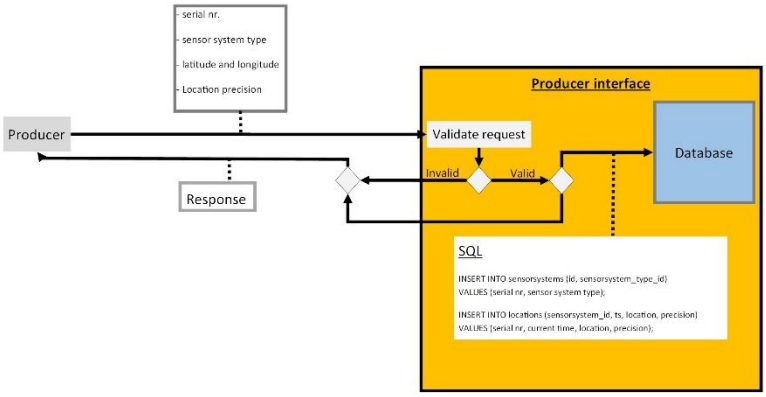

Figure 5.16, Registration of a sensor system with location precision

For registration of a sensor system with a user (figure 5.17), a producer can provide a username and password in the request. If the serial number and user are unique and the sensor system type is registered in the sensorsystem_types table, the request is valid and the password will be hashed. Finally the request will be translated into three SQL insert commands, registering the sensor system in the sensor systems table, it's location in the locations table, and the corresponding user in the users table.

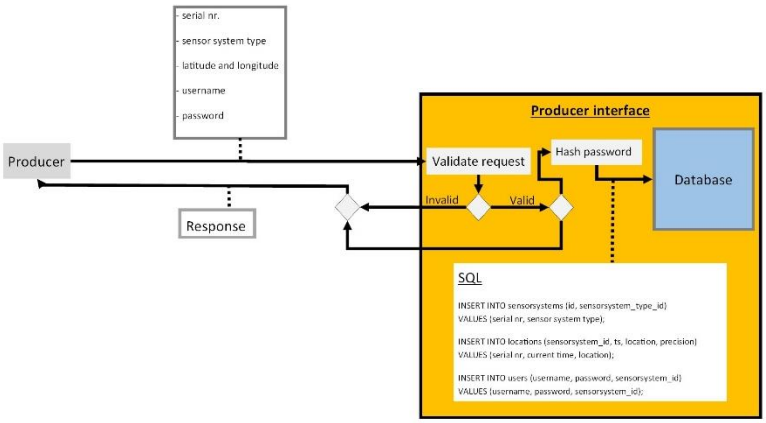

Figure 5.17, Registration of a sensor system with a user

Registration of sensor system characteristics

A producer might want to register certain sensor system characteristics, such as the example from section 5.2.4, the SRB capacity. Figure 5.18 shows the process of making a request for registration of a sensor system characteristic. The producer provides the sensor system's serial number and type, the type of characteristic, and the characteristic's value. The request will then be validated, by checking whether the serial number exists in the sensorsystems table, whether the sensor system's serial number is registered with the requested sensor system type, and whether the characteristic type with corresponding sensor system type exists in the characteristic_types table. When the request was valid, the characteristic will be inserted into the characteristics table. From that moment on, the registered sensor system has a registered characteristic. Multiple different characteristics can be registered, as long as the sensor system's serial number is registered in the sensorsystems table, and the characteristic type is registered with corresponding sensor system type in the characteristic_types table.

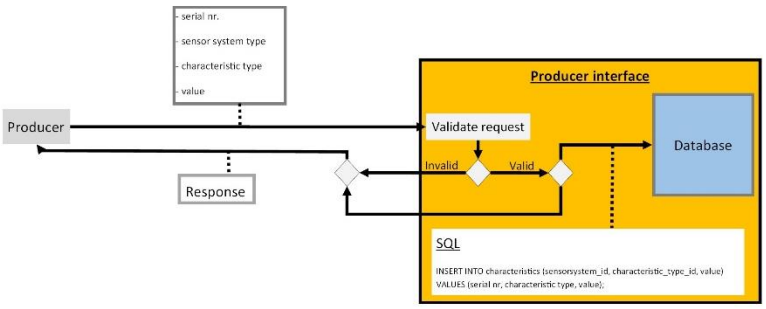

Figure 5.18, Registration of sensor system characteristics

Update sensor system's location

When a sensor system is moved, its location changes. For instance when the municipality of Enschede decides to relocate one of the AirT nodes. A producer can update a sensor system's location by sending

a location update request to the producers interface (see figure 5.19). The producer provides the sensor system's serial number and type, the new latitude longitude pair, and optionally the precision indicated by blue. The same validation occurs as with the sensor system and characteristic registration. In order to update a sensor system's location, the sensor system's serial number has to exist in the sensorsystems table and the provided sensor system type has to correspond with the type that is the sensor system's serial number is registered with. When the request is valid, the new location will be inserted into the locations table.

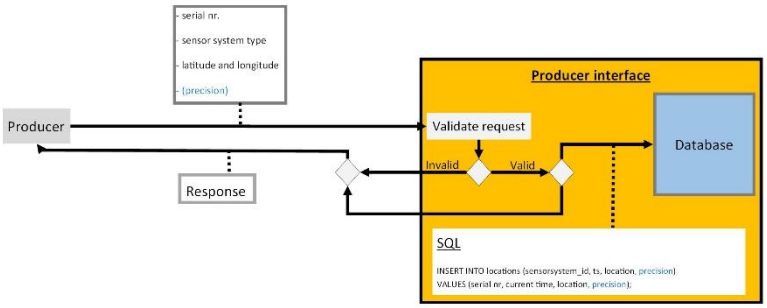

Figure 5.19, Updating a sensor system's location

Registration of events

For the SRB project, it is a requirement that manual, autonomous, and planned discharges are stored in the data repository's database. These three types of SRB discharges are unified under the name *events*, in order to keep the solution universal. Other types of sensor systems could in the future also require for certain events to be stored, for example a planned update or maintenance of which the information can then be communicated via for instance visualizations or notifications in a user dashboard. The process of event registration is shown in figure 5.10. A producer provides the interface with the sensor system's serial number and type, the type of event, the value, and the timestamp. Optionally (indicated by blue in figure 5.20), a producer can add a second value to an event. This can for example be precipitation prediction data corresponding with the planned or autonomous discharge of a SRB.

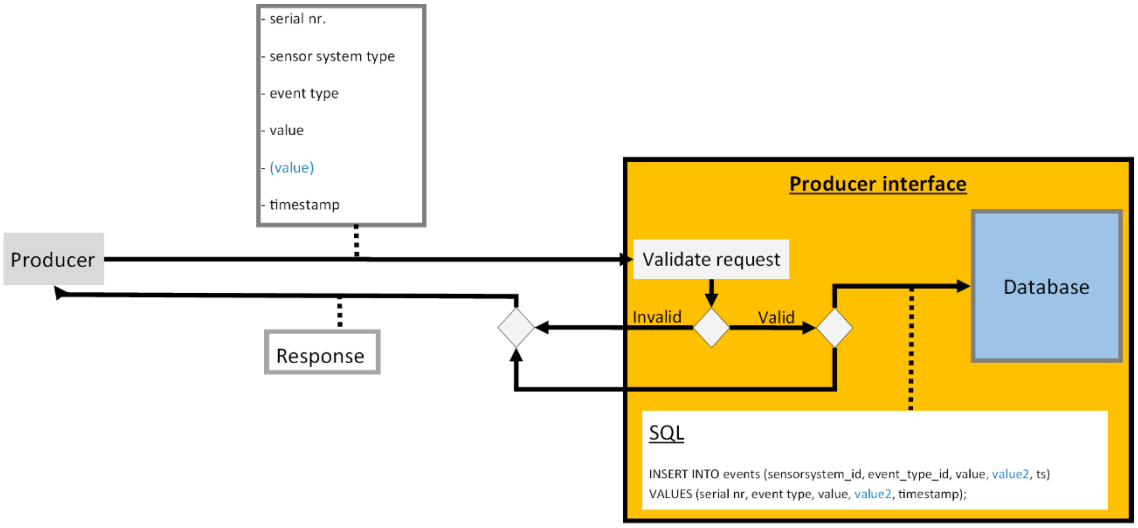

Figure 5.20, Registration of an event

Registration of measurements

Sensor systems, such as for example an SRB and AirT, possibly have multiple sensors measuring different (environmental) phenomenon. Therefor, I chose to provide producers with the possibility to register up to four types of measurements and their corresponding values in one request. Figure 5.21 shows a visual representation of the measurement registration process for a single type of measurement and figure 5.22 for two types of measurements. Registering three or four types of measurement values in one request, works the same as registering two types of measurement values. First the measurement type has to be specified, after which the corresponding value follows. When the request is valid, which means that the sensor system's serial number is registered with the same type as send in the request, and the measurement types are registered with the corresponding sensor system type in the measurement_types table, the measurements are inserted into the measurements table.

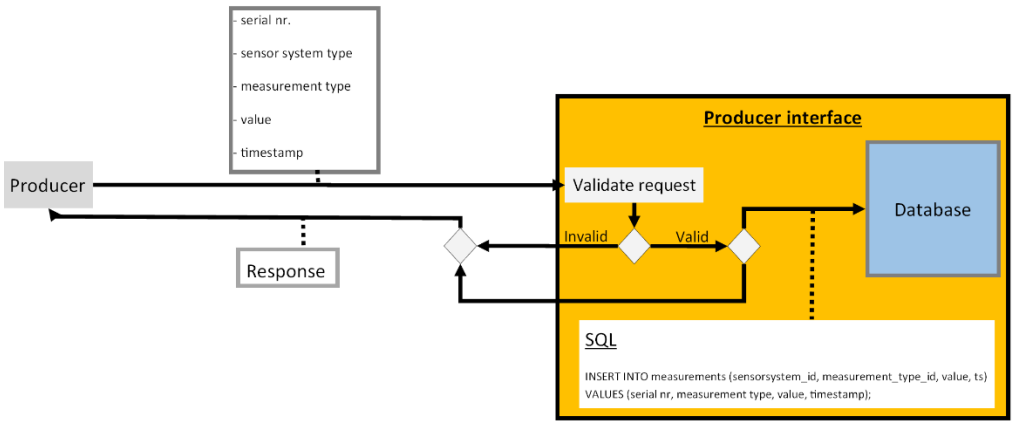

Figure 5.21, Registration of a single type of measurement

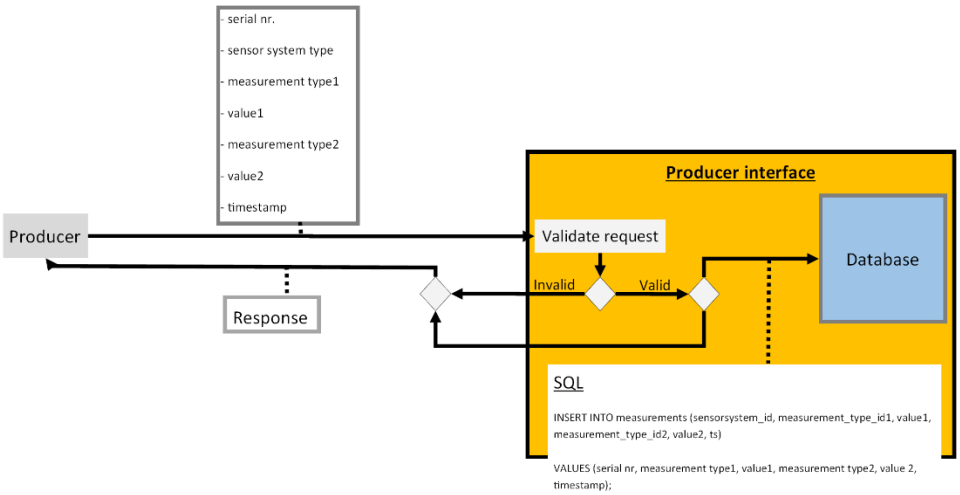

Figure 5.22, Registration of two types of measurements

5.3.4 Consumer interface

The consumer interface provides consumers with the functionality to retrieve the sensor system's id belonging to a certain user, the current location of one or all sensor systems, the historic locations of one or all sensor systems, sensor system's characteristics, sensor system's measurements, events, and the possibility to download a .csv formatted file of all raw AirT data. Some extra functionality was implemented for the SRB dashboard developer, namely the possibility to retrieve last month's amount of buffered rainwater for a single SRB as well as for all SRBs, which will be further elaborated on in chapter 6.

Sensor system id for a user

In the SRB user dashboard that will be developed by the SRB dashboard developer, individual SRB data can only be accessed by its owners. Therefor logging in with a username and password is needed in order to retrieve individual SRB data. When the username and password match, the corresponding sensor system id is returned to the dashboard application. The scheme of this functionality can be seen in figure 5.23. When the request made is valid, the username exists and the password matches, the result of the SQL query will be returned to the consumer, which is the sensor system's id and type.

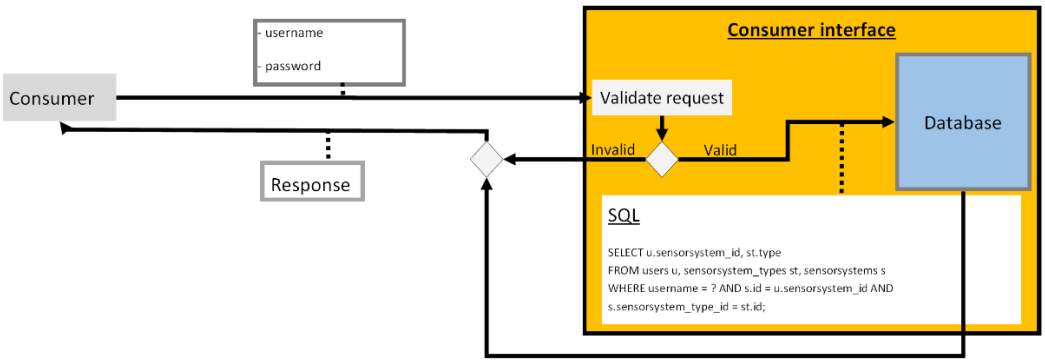

Figure 5.23, Retrieving the sensor system id belonging to a user

Current location(s) of sensor system(s)

For visualization on for instance a map, a consumer has to be able to retrieve the current location of one or more sensor systems. Measurements within the same timeframe as the sensor system's location can be matched linked to that location. The scheme of retrieving a single sensor system's current location can be found in figure 5.24. Figure 5.25 shows the process of retrieving all sensor systems of a single type's current locations. The same validation as described before happen again, checking whether the sensorsystem_type exists, and whether the sensor system's serial number is registered with the same type as send in the request. If the request is valid, the result will be returned tot the consumer.

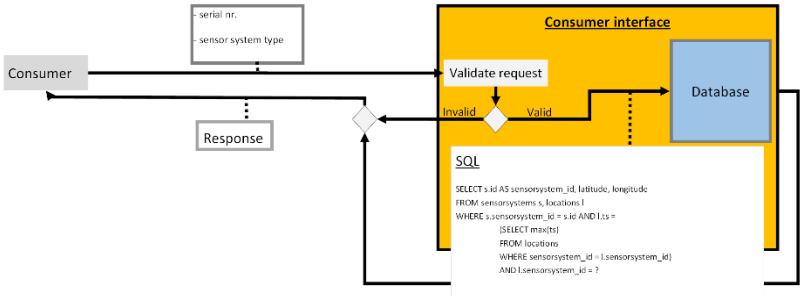

Figure 5.24, Retrieving a single sensor system's current location

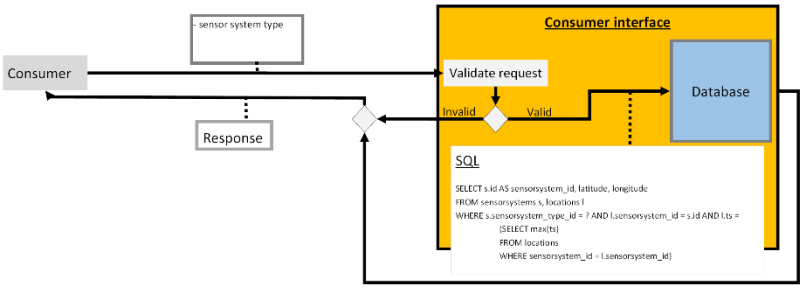

Figure 5.25, Retrieving all sensor systems of a single type's current locations

Historic locations of sensor system

In the request for sensor system's historic locations (figure 5.26), a consumer should provide the interface with the sensor system's serial number and type. After validating the existence of the serial number and corresponding sensor system type in the sensorsystems table, database is queried and the result is returned to the consumer.

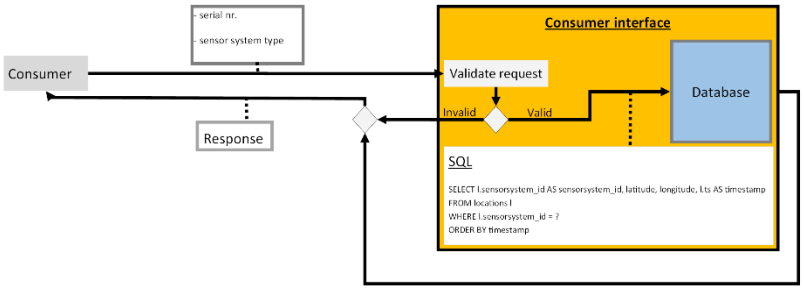

Figure 5.26, Retrieving historic locations of a sensor system

Sensor system characteristics

Sensor systems are provided the possibility to store sensor system specific characteristics in the data repository. As said in section 4.2.3, a SRB has for example the characteristic capacity. This characteristic can be important for calculations, determining buffering performance, and visualizations in the SRB dashboard. A visual representation of requesting sensor system's characteristics can be found in figure 5.27. The consumer provides the interface with the sensor system's serial number and type, after which the both are validated. If the supplied parameters are valid, the database is queried and the result is returned to the consumer.

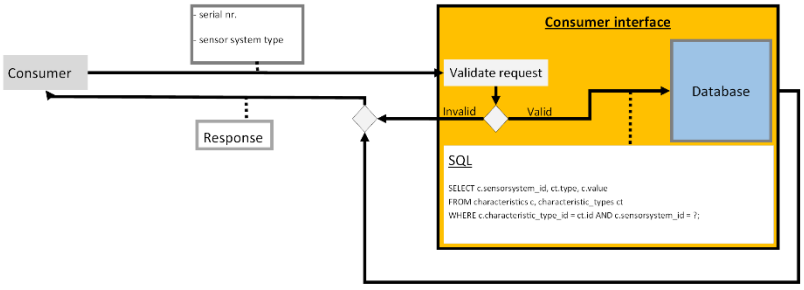

Figure 5.27, Retrieving sensor system characteristics

Sensor system's measurements

Consumers are provided the functionality to retrieve all measurements for a single sensor system (figure 5.28), all current day measurements for a single sensor system (figure 5.29), all measurements in a specified time frame for a single sensor system (figure 5.30), and all measurements in a specified

time frame for all sensor systems of a single type (figure 5.31). Again the same validation as explained multiple times before occurs prior to querying the database for the result, and will from now on only be explained when validation deviates from the previous mentioned validation of the in the request send parameters (existence of the sensor system's serial number and type in the sensorsystems table).

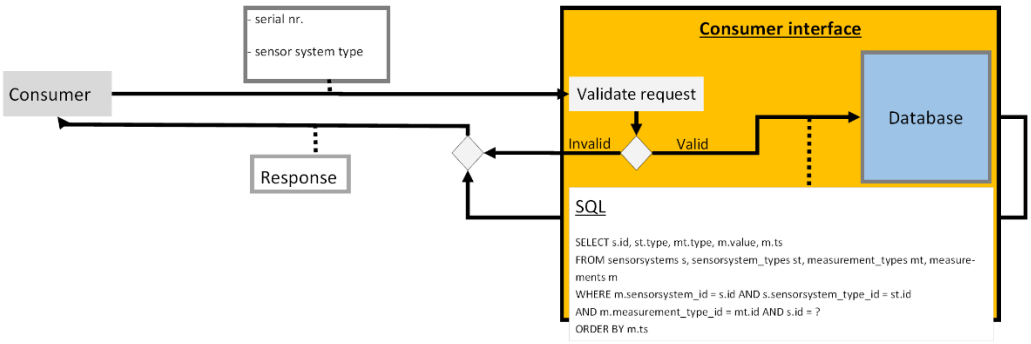

Figure 5.28, Retrieving all measurements for a single sensor system

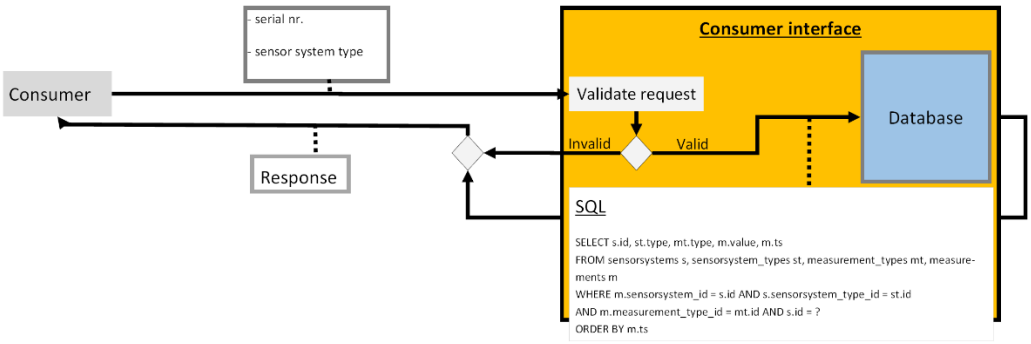

Figure 5.29, Retrieving all current day measurements for a single sensor system

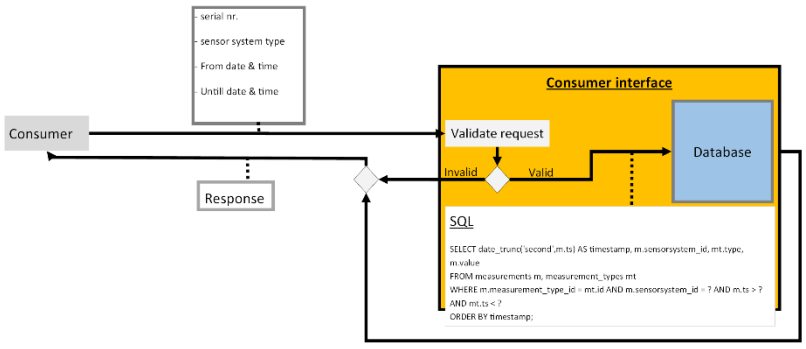

Figure 5.30, Retrieving all measurements in a specified time frame for a single sensor system

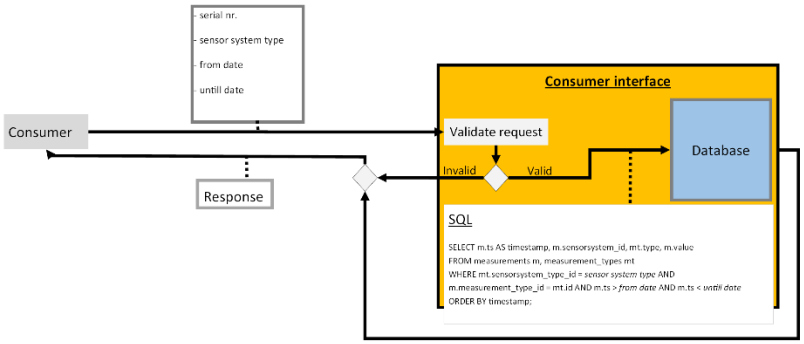

Figure 5.31, Retrieving all measurements in a specified time frame for all sensor systems of a single type

Events

Consumers are provided the functionality to request all events for a single sensor system (figure 5.32), all events for a single sensor system within a specified timeframe (figure 5.33), and specific events within a specified time frame (figure 5.34).

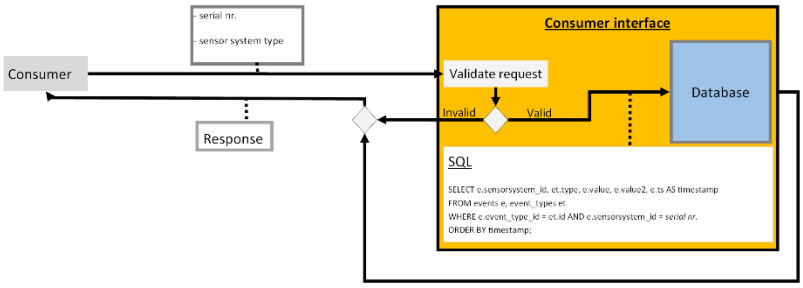

Figure 5.32, Retrieving all events for a single sensor system

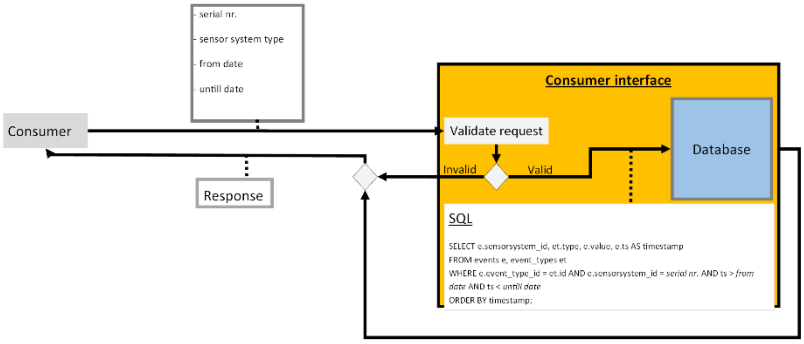

Figure 5.33, Retrieving all events within a specified time frame for a single sensor system

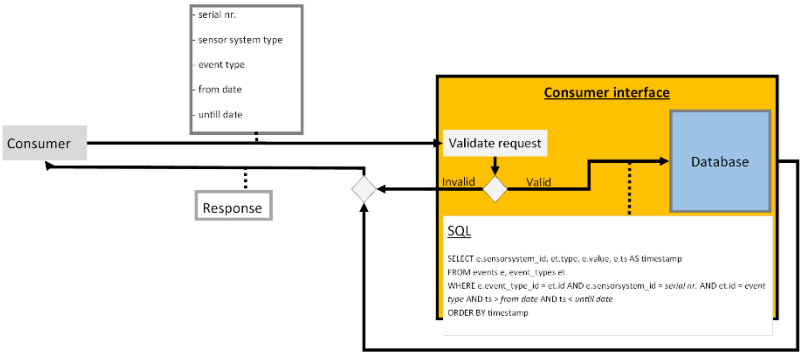

Figure 5.34, Retrieving all events of a specified type within a specified time frame for a single sensor system

5.4.5 Administrator interface

For the administrator interface, the implemented functionality consists of registration of a new sensor system type, a new characteristic type, a new event type, and a new measurement type. A visual representation can be found in figure 5.35. All four provided functionalities are shown in a single figure.

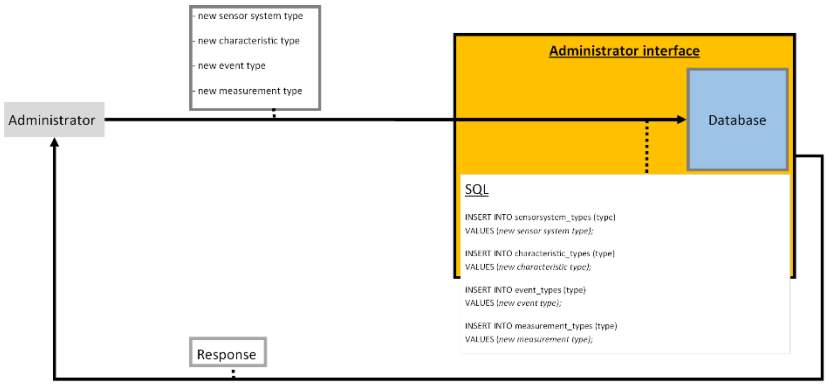

Figure 5.35, All administrator interface functionality

5.4 PACT Analysis

In order to describe the system users perspective and structure the construction of scenarios, a PACT analysis will be used.

5.4.1 People

Three types of ‘people’, for which in the context of this project roles would be a better fitting name, can be identified. These roles are:

- Administrator

- Producer
- Consumer

Each role has its own characteristics and requirements. Each role will be described in short below.

Administrator

The table structure and interfaces will be universal. This means that any type of sensor system can be add to the system, as well as any type of measurement, characteristic, and event. The data repository's administrator will have control over registration of new types of sensor systems, measurements, characteristics, and events.

Producer

A sensor system produces data which has to be stored in the data repository, and can therefor be seen as a producer. A producer is likely to need to be able to register itself, register its user, register measurements, register characteristics, and register events.

Consumer

A consumer consumes data from the data repository. A consumer can be for instance a visualization application, which visualizes sensor system data like measurements and events based on sensor system locations and within a certain time frame.

5.4.2 Activities

The data repository provides several functions in each of the interfaces. The implemented interfaces should accommodate the three described roles with functions to perform the activities described below.

Administrator

The administrator's activities are registering new types of sensor systems, new types of sensor system characteristics, new types of sensor system measurements, and new types of sensor system events.

Producer

Producer activities are registering a sensor system with location, registering a sensor system with location and location precision, and registering a sensor system with location, username and password. All given that the type of sensor system has been registered in the repository by the repository administrator. When a sensor system has been registered, a producer can update its location, and register sensor system characteristics, measurements, and events, given that the types of

characteristics, measurements, and events have been registered in the repository by the repository's administrator.

Consumer

A consumer, for example a visualization application, needs to be able to retrieve current and historic locations of sensor systems and sensor system characteristics. Also (specific or all) events registered by sensor systems should be made available to a consumer, with the possibility to specify a time frame in for the requested events. Finally, requesting measurements for a sensor system. For measurements its also needed to be able to specify a time frame. Additionally for the SRB consumer, the last month's amount of buffered rainwater of a single as well as all SRBs can be requested.

5.4.3 Context

The context is the same for all three roles. In order to be able to use the functionality provided by each of the interface, administrators, producers, and consumer need to implement the possibility to make http POST and GET request to interface end points in their sensor systems and/or applications. The only difference lies in the end points that need to specified in the http requests.

5.4.4 Technologies

The technologies used for each of the three roles is the same. Producers and administrators can input data using the producers and administrators interfaces respectively. Consumers can output data using the consumers interface. The API server runs on a single machine, and has a connection with the database. Administrators, producers, and consumers only have access to the corresponding interfaces, which will translate the requests into SQL queries to the database.

5.5 PACT-FICS scenario

The PACT analysis of section 5.4 is used together with FICS, to construct a scenario describing both the user's (or role's) perspective as well as the designer's perspective of this project.

Dominiek is 22 year old third year student Creative Technology and works on the development of a new type of sensor system to be add to the CAC project. She works together with 21 year old Heinrich, who is also a third year Creative Technology student working on the development of a visualization application for the data produced by Dominiek's sensor system.

Heinrich's visualization application needs to provide users with location based fine dust measurements and traffic flow measurements. For every sensor system location he needs the amount of cars that pass per 10 minutes as well as fine dust measurements. This data should be provided by

Dominiek's developed sensor system, be stored in the CAC data repository, and made available to Heinrich's visualization application. Before the development of the sensor system was completed, the relevant database tables contain the data shown in figure 5.36 and 5.37:

sensorsystem_types	
id	**Type**
1	srb
2	airt

Figure 5.36, The sensorsystem_types table prior to registration of a new sensor system type

measurement_types		
id	**sensorsystem_type_id**	**type**
1	1	Fill_level
2	1	water_temperature
3	2	air_temperature
4	2	humidity
5	2	wind_speed

Figure 5.37, The measurement_types table prior to registration of new measurement types

Dominiek and Heinrich's project is called Fine Dust Monitoring (FDM). Dominiek finished the development of her prototype and contacts the repository administrator. She requests for the type of sensor system to be added to the repository, as well as the new types of measurements accompanied with the new sensor system type. The repository administrator first registers the new sensor system

type using the administrator interface, with the type *fdm*. The sensorsystem_types table now contains the new sensor system type (see figure 5.38).

sensorsystem_types	
id	**Type**
1	srb
2	airt
3	fdm

Figure 5.38, The sensorsystem_types table with the new sensor system type

Now the new type of sensor system is registered, Dominiek can register fdm sensor systems. However, she can not yet register measurements, since no measurement types have yet been registered for the fdm sensor system type. In order to be able to register measurements, the measurement types have to be specified in the measurement_types table with corresponding sensor system type. The administrator adds the types *fine_dust* and *passed_cars* to the measurement_types table (see figure 5.39). From this moment on Dominiek and Heinrich can both communicate with the producer and consumer interfaces using http POST and GET requests in order to insert and retrieve fine dust monitoring data.

measurement_types		
id	**sensorsystem_type_id**	**type**
1	1	Fill_level
2	1	water_temperature
3	2	air_temperature
4	2	humidity
5	2	wind_speed
6	3	fine_dust
7	3	passed_cars

6. Realization

This chapter describes the hardware and software that was used to develop the prototype, the database management system installation log, the interface software installation log, the functionality that was implemented as extra service to the stakeholders, and the prototype code.

6.1 Hardware

The prototype, consisting of the database management system and the interfaces, runs on a single machine. This is a DELL OptiPlex 7050 with the following specifications:

- Operating system: Ubuntu 18.04
- Memory: 7,7 GiB
- Processor: Intel® Core™ i7-7700 CPU @ 3,60 GHz x8
- Graphics: Intel HD Graphics 630 (Kaby Lake GT2)
- Hard Disk: 2,0 TB WDC WD20EZRZ-00z5hb0
- SSD: Samsung PM961 256 GB

6.2 Database management system

In order to determine which database management system should be used, the options and their characteristics described in chapter 2 were presented to and discussed with the stakeholders. The outcome showed a clear preference for the use of the PostgreSQL database management system, extended with spatial objects and functionality by PostGIS. There are three reasons for the choice for PostgreSQL with PostGIS. The first is that both PostgreSQL and PostGIS are completely free and open source software. The discussed alternatives MySQL, MongoDB, and VoltDB are not. Next to a free version, MySQL offers enterprise editions as well. MongoDB and VoltDB both offer monthly subscriptions in order to use their services. The second reason is that PostgreSQL and PostGIS offer an immense set of functionality, not equaled by any of the other database systems. The third and final reason is that PostgreSQL and PostGIS are very mature, extremely well documented (see appendix A), and have a very large supporting community. See sections 2.3, 2.4, and 2.5 for background research regarding PostgreSQL and PostGIS. The PostgreSQL tutorial in the PostgreSQL documentation [37] gives three options to access the database:

- Enter, edit, and execute SQL commands in psql, the PostgreSQL interactive terminal program.
- By using an existing graphical frontend tool. pgAdmin or an office suite with ODBC or JDBC support are mentioned examples.
- Writing a custom client interface.

For this project the third option will be implemented (see section 6.3).

6.2.1 PostgreSQL & PostGIS

PostgreSQL is an open source object relational database system, with a strong reputation for reliability, data integrity, feature robustness, extensibility, and performance [37]. It also offers advanced GiST indexing (see section 2.3.3). The most relevant aspect of the choice for PostgreSQL is the PostGIS extension (see section 2.3.3 and 2.4). PostGIS offers the most extensive set of geospatial functionality. PostGIS extends PostgreSQL with geometry and geography object, as well as geometry and geography functions. Some relevant examples are explained below. For this project the geometry object types are used, since more functionality is available for geometry objects as compared to geography objects and performance is higher. Especially when flat earth projection is used. This means that instead of a round earth representation, locations are projected on a cartesian special reference system like the Dutch Rijksdriehoekscoördinaten (RD-coordinates system) [38] (see figure 6.1), which will be used in the representation of locations in the data repository. The city Amersfoort used to be where the x and y axis intersected, but for practical reasons the axes shifted such that each locations in the Netherlands yields a positive value for the x and y axis. Some relevant examples of PostGIS functions are elaborated on below.

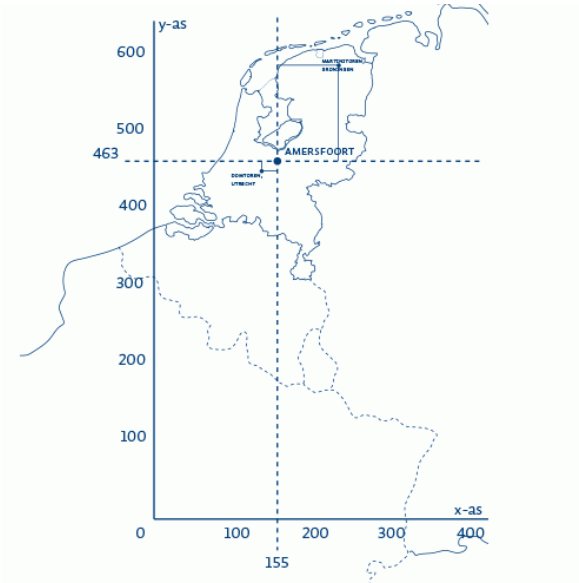

Figure 6.1, Visual representation of the RD coordinate system

Area selection

With this functionality, PostGIS has an important advantage over other database management systems. Selecting geometries such as points within another geometry such as a polygon, is more precise (see

figure 6.2). Instead of using a minimum bounding box around the polygon, indicated by the red square in the left image, which returns some false results, PostGIS returns only geometries that are actually inside the polygon geometry, which is the blue colored area in the picture.

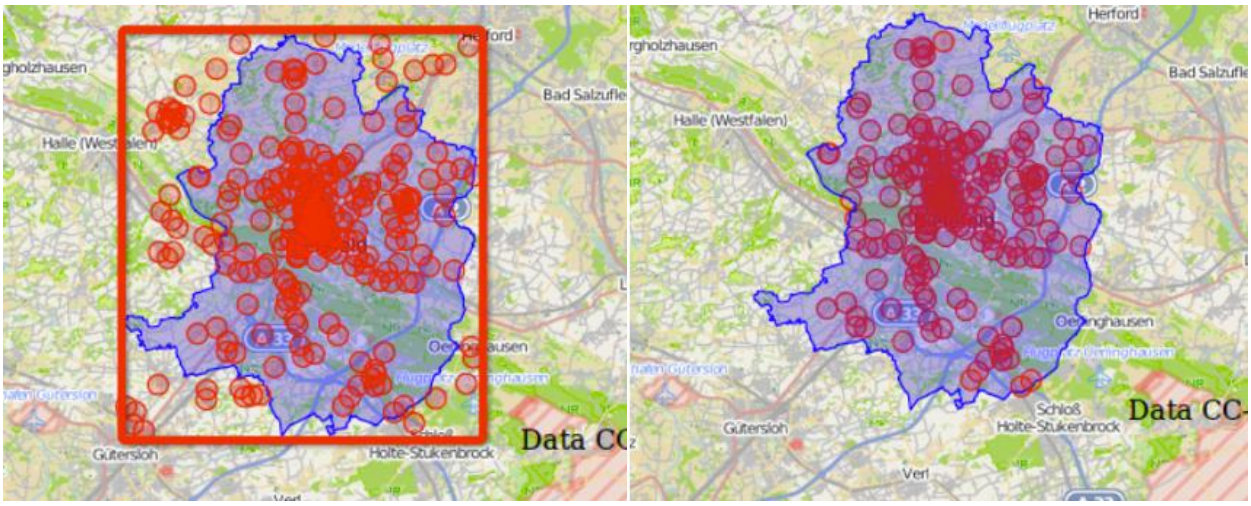

Figure 6.2: Differences in spatial querying functions.

This property of this PostGIS function can be very useful when for example in the future SRB data will be made available to other on a postal code are level, like the postal code areas in Enschede shown in figure 6.3.

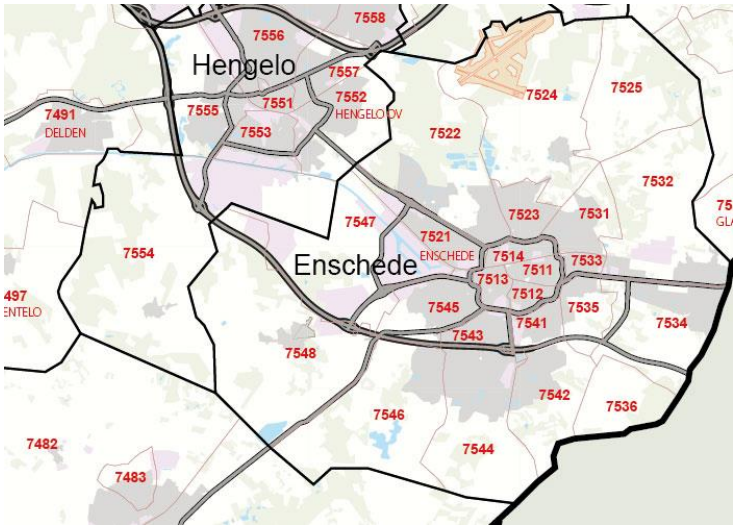

Figure 6.3, Enschede's postal code areas

Geometry clustering

As said in section 5.1.1, the SRB dashboard developer's requirement for the system to provide clustered averaged SRB data was reprioritized into a could have requirement. At this time of the project

this functionality will not be implemented, but it can be very useful for future work. For example the PostGIS function ST_ClusterDBScan (figure 6.4) can cluster geometries based on the distance between their centroids, the arithmetic mean position of all the points in a geometry. The minimum amount of geometries per cluster can be specified using this function, which can be very useful for anonymization of SRB data.

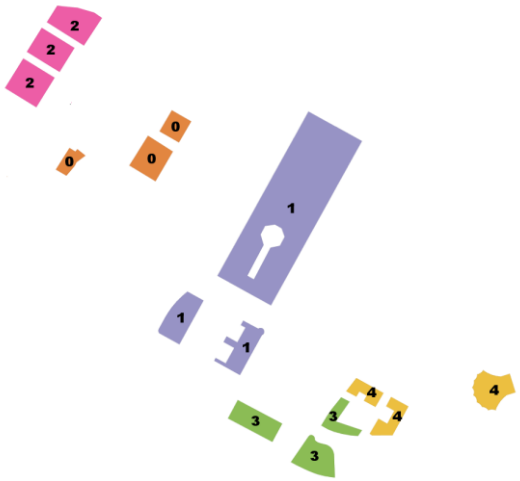

Figure 6.4, Visual representation of the ST_ClusterDBScan function. Clusters are assigned a cid (cluster id)

Subdividing areas

ST_SubDivide is a function that can subdivide an area, for example a polygon geometry shape, into multiple subareas, such as show in figure 6.5. This can possibly be useful in the future as well for anonymization of for example SRB data.

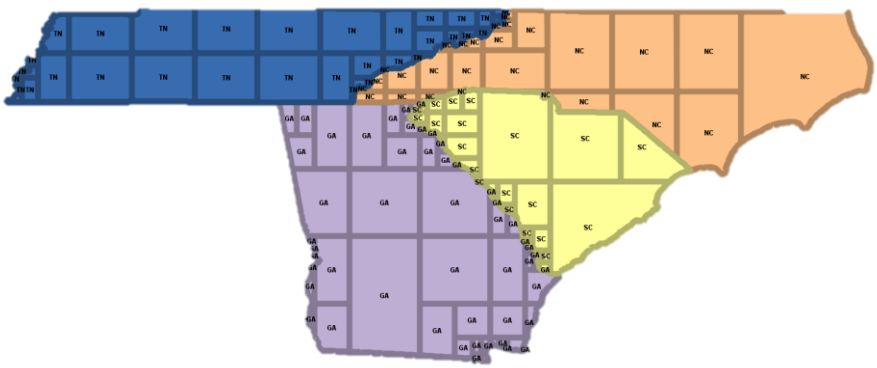

Figure 6.5, Visual representation of the ST_SubDivide function

Street level averaged data

Again for possible future anonymization of for example SRB data, is averaging data on street level (see figure 6.6). In this timeframe of the project this functionality will not be implemented, and is prioritized as a could have requirement (see section 5.1.1). With this function it would be possible to create geometries of the type *linestring*, representing streets. SRB data other than that of the SRB user's own SRB can then made available only on street level. Data of SRBs of which the locations intersect or lie within a certain range of the linestring representing a street, can then be averaged.

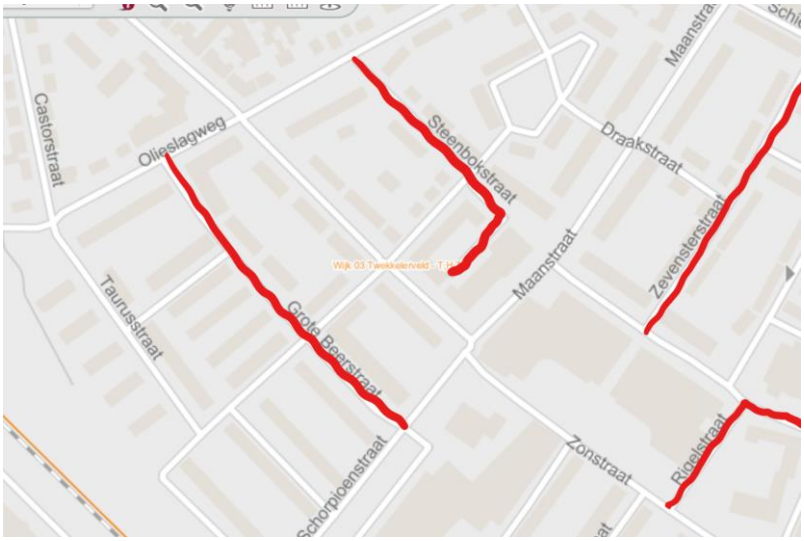

Figure 6.6, Visual representation of linestring geometry objects representing streets.

6.2.2 PostgreSQL install log

This section contains the log of the PostgreSQL installation on the server machine, running Ubuntu 18.04. The PostgreSQL version that will be used is PostgreSQL 10.4, which is available as package in Ubuntu's default repositories.

```
sudo apt update
sudo apt upgrade
sudo apt install build-essential
sudo apt-get install postgresql-10
```

After installation, the directory in which the database will be stored has to be created and directory permissions for the postgres user which automatically exists after installation of PostgreSQL have to be set. After that, log in as user postgres with password postgres.

```
root# mkdir /media/hd/pgsql
root# chown postgres /media/hd/pgsql
root# su postgres
```

Next, the database will be initialized by the first command. The second command starts the database server and the third command stops the database server.

```
/usr/lib/postgresql/10/bin/pg_ctl -D /media/hd/pgsql/data initdb
/usr/lib/postgresql/10/bin/pg_ctl -D /media/hd/pgsql/data -l logfile start
/usr/lib/postgresql/10/bin/pg_ctl -D /media/hd/pgsql/data stop
```

Configure the database server settings for remote access.
Modify `/media/hd/pgsql/data/postgresql.conf`
Replace at the listen_addresses ‘localhost’ with ‘*’.

I encountered some issues with starting the database server. In order to fix this, modify the file in the same folder, named `pg_hba.conf`. Add the following two lines at the end:

```
host  all   all   0.0.0.0/0  md5
host  all   all   ::/0       md5
```

Test with the following command

```
psql -h 107.170.158.89 -U postgres
```

In case issues arise with attempting connect to the database with the default postgres password, this can be fixed by changing the password. Log in on linux as postgres user and enter the following commands and enter the new password:

```
psql postgres
\password postgres
```

6.2.3 PostGIS install log

First verify what version of Ubuntu is running, by

```
sudo lsb_release -a
```

Add repository to sources.list

```
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt bionic-pgdg
main" >> /etc/apt/sources.list'
```

Add keys

```
wget --quiet -O - http://apt.postgresql.org/pub/repos/apt/ACCC4CF8.asc |
sudo apt-key add -
sudo apt update
sudo apt upgrade
```

Install PostGIS 2.4

```
sudo apt install postgresql-10-postgis-2.4

sudo apt postgresql-10-postgis-scripts
sudo apt install postgis
```

6.2.4 Creating the database

Log in as user postgres. Then create the database using the createdb command with the desired database name. Open the terminal-based front end to be able to type in queries to the database by using the psql command together with the database name. After that, create the postgis and tablefunc extensions. The tablefunc extensions add the functionality to create pivot tables using PostgreSQL's crosstab function.

```
createdb cac
psql cac
CREATE EXTENSION postgis;
CREATE EXTENSION tablefunc;
```

Now the tables need to be created. Enter the following commands to create the tables (see section 5.2.4 for the table structure). The type SMALLSERIAL is an auto incrementing value of the type small int. Primary keys can be specified by adding *primary key* after a column specification. Foreign keys can be specified by adding *references [table name]([column name])* after a column specification. The geometry column is added to the locations table by *AddGeometryColumn ('[table name]', '[column name]', [SRID], '[geometry type]', [dimension]);*. The SRID (spatial reference system id) that is used below is the Dutch RD-coordinate system.

```
CREATE TABLE sensorsystem_types (
id SMALLSERIAL primary key,
type varchar(80)
);

CREATE TABLE sensorsystems (
id INTEGER primary key,
sensorsystem_type_id SMALLINT references sensorsystem_types(id)
);

CREATE TABLE locations (
sensorsystem_id INTEGER references sensorsystems(id),
ts TIMESTAMP
);

SELECT AddGeometryColumn ('locations', 'location', 28992, 'POINT', 2);

CREATE TABLE measurement_types (
id SMALLSERIAL primary key,
sensorsystem_type_id SMALLINT references sensorsystem_types(id),
type varchar(80)
);

CREATE TABLE measurements (
sensorsystem_id INTEGER references sensorsystems(id),
measurement_type_id SMALLINT references measurement_types(id),
value numeric,
ts TIMESTAMP
);

CREATE TABLE characteristic_types (
id SMALLSERIAL primary key,
sensorsystem_type_id SMALLINT references sensorsystem_types(id),
type varchar(80)
);

CREATE TABLE characteristics (
```

```
sensorsystem_id INTEGER references sensorsystems(id),
characteristic_type_id SMALLINT references characteristic_types(id),
value numeric
);

CREATE TABLE event_types (
id SMALLSERIAL primary key,
sensorsystem_type_id SMALLINT references sensorsystem_types(id),
type varchar(80)
);

CREATE TABLE events (
sensorsystem_id INTEGER references sensorsystems(id),
event_type_id SMALLINT references event_types(id),
value numeric,
precipitation numeric,
ts TIMESTAMP
);
```

Since at this moment the interfaces have not been implemented yet, the sensor system types, characteristic types, event types, and measurement types have to be inserted manually by writing SQL commands in the terminal-based front end psql.

```
INSERT INTO sensorsystem_types (type)
VALUES ('srb'),('airt'),('rps');

INSERT INTO characteristic_types (sensorsystem_type_id, type)
VALUES (1,'capacity'), (2,'skyview_factor');

INSERT INTO event_types (sensorsystem_type_id, type)
VALUES (1,'discharge'), (1,'manual_discharge'), (1,'planned_discharge);

INSERT INTO measurement_types (sensorsystem_type_id, type)
VALUES (1,'fill_level'), (1,'water_temperature'), (2,'air_temperature'),
(2,'humidity'), (2,'wind_speed');
```

6.3 Lumen php micro-framework

Lumen is a php micro-framework by Laravel [39]. It's an elegant solution for building Laravel based and fast APIs. It claims on the website to be one of the fastest micro-frameworks available. Using Lumen, three RESTful APIs will be developed, one for consumers, one for producers, and one for administrators. The consumer interface provides the requested data to consumers in json format, which is a requirement for the project. A RESTful API uses HTTP requests to GET, PUT, POST, and DELETE data. With Lumen it is possible to make use of Models and Controllers, however the for this project required queries are to complex for this approach. Therefor models will not be used, and raw queries will be implemented in the Controller files functions. This section starts with the installation log, after which the routes that have been set up are discussed. Finally, the controller files and implementation of the functionalities will be discussed.

6.3.1 Installation

The lumen framework has a few system requirements:

- PHP >= 7.1.3
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension

To list the PHP extensions, enter the following command:

```
php -m
```

Enter the following commands to install the missing PHP extensions:

```
sudo apt-get install php-mbstring
sudo apt-get install php-pgsql
sudo systemctl restart apache2
```

Lumen uses composer [40] to manage its dependencies. Install composer by running the following script:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php') ===
'544e09ee996cdf60ece3804abc52599c22b1f40f4323403c44d44fdfdd586475ca9813a8
58088ffbc1f233e9b180f061') { echo 'Installer verified'; } else { echo
'Installer corrupt'; unlink('composer-setup.php'); } echo PHP_EOL;"
```

```
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Get the Lumen installer

```
composer global require “laravel/lumen-installer”
```

Install the following extension

```
sudo apt-get install php-xml
```

Install Lumen, with name cacapi. The directory cacapi will be created, in which lumen will be installed.

```
composer create-project –prefer-dist laravel/lumen cacapi
```

Serving the application, using port 23450

```
php -S 0.0.0.0:23450 -t cacapi/public
```

Configure environment. Rename the file .env.example to .env, using the following command:

```
mv .env.example .env
```

Set environment variables. Open the .env file and fill in the following values:

```
DB_CONNECTION=pgsql
DB_HOST=130.89.12.84
DB_PORT=5432
DB_DATABASE=cac
DB_USERNAME=postgres
DB_PASSWORD=postgres
```

Uncomment the lines `//app->withEloquent` and `//$app->withFacades();` in the file `cacapi/bootstrap/app.php`

6.3.2 Routes

Setting up routes for the RESTful API can be done in the *web.php* file in the directory */cacapi/routes*. Router groups are created, one for each role. Each router group has its own prefix. The prefixes are *cons* for consumers, *prod* for producers, and *admin* for administrators. These routes are the endpoints in the APIs.

Admin routes

For the administrators interface with prefix *admin*, four routes have been developed. These routes with example POST requests are:

- newsensorsystemtype/{type}
  http://130.89.12.84:23450/admin/newsensorsystemtype/srb
  Inserts a new type of sensor system (srb) into the sensorsystem_types table
- newcharacteristictype/{type}/{sensorsystem_type_id}
  http://130.89.12.84:23450/admin/newcharacteristictype/capacity/1
  Inserts a new type of srb characteristic (capacity) into the sensorsystem_types table
- neweventtype/{type}/{sensorsystem_type_id}
  http://130.89.12.84:23450/admin/neweventtype/discharge/1
  Inserts a new type of srb event (discharge) into the event_types table
- newmeasurementtype/{type}/{sensorsystem_type_id}
  http://130.89.12.84:23450/admin/newmeasurementtype/fill_level/1
  Inserts a new type of srb measurement (fill_level) into the measurement_types table

Routes are implemented as shown in figure 6.7. The Admin routes are used once as example. The consumer and producer routes are implemented in the same way and will not be explained in detail. The complete code can be found in appendix B.

On line 19, the router group and corresponding prefix are specified. Lines 21 – 24 describe functionality specific routes within the admin router group. The type of request is specified in each route, which in all four cases is a post request. The first part of a route specifies the route's address, and behind the slash and in brackets the required parameters. The second part of the route, after the comma, specifies which function of which controller file has to be called. In case of the route specified on line 21 in figure 6.7, the function *newSensorsystemType* of the *AdminController* will be used. The parameters specified behind the slash and in brackets will be passed on to the in the controller specified function. The controllers and their functions will be discussed in sections 6.3.3 – 6.3.5.

```
18 // ADMIN ROUTES
19 $router->group(['prefix' => 'admin'], function () use ($router) {
20
21         $router->post('newsensorsystemtype/{type}', ['uses' => 'AdminController@newSensorsystemType']);
22         $router->post('newcharacteristictype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newCharacteristicType']);
23         $router->post('neweventtype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newEventType']);
24         $router->post('newmeasurementtype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newMeasurementType']);
25 });
```

Figure 6.7, The administrator interface routes

One route address can contain multiple end points by specifying a varying number of parameters that can be passed to a route. See figure 6.8 for an example. These two routes belong to the consumer interface, and are used to retrieve all data for a single sensor system in a specified time period and all measurements for all sensor systems in a specified time period respectively. Highlighted by yellow are the routes addresses, which are identical. However, the number of parameters is different and therefor two different functions of the consumer controller can be called (highlighted with green).

```
// Get all measurements in specified period for single sensorystem
$router->get('datemeasurements/{sensorsystem_id}/{sensorsystem_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getMeasurementsPeriod']);

// Get all measurements in specified period for all sensor systems of single type
$router->get('datemeasurements/{sensorsystem_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getAllMeasurementsPeriod']);
```

Figure 6.8, Example of one route address with a different number of parameters

Producer routes

For the producers interface with prefix *prod*, thirteen routes have been developed. These routes with example POST requests are:

- registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{username}/{password}
  http://130.89.12.84:23450/prod/registersensorsystem/1/1/52,239154/6,850667/Joeri/mypassword
  Registers a sensor system with id = 1, type = 1, lat = 52,239154, long = 6,850667, userame = Joeri, and password = mypassword
- registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}
  http://130.89.12.84:23450/prod/registersensorsystem/1/1/52,239154/6,850667
  Registers a sensor system with id = 1, type = 1, lat = 52,239154, long = 6,850667
- registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{precision}
  http://130.89.12.84:23450/prod/registersensorsystem/1/1/52,239154/6,850667/100
  Registers a sensor system with id = 1, type = 1, lat = 52,239154, long = 6,850667, and precision = 100
- registercharacteristic/{sensorsystem_id}/{sensorsystem_type_id}/{characteristic_type_id}/{characteristic_value}
  http://130.89.12.84:23450/prod/registercharacteristic/1/1/1/500

Registers a characteristic with sensor system id = 1, sensor system type = 1, characteristic type =1, and value = 500

- updatelocation/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}
  http://130.89.12.84:23450/prod/updatelocation/1/1/52,239154/6,850667
  Updates the location of sensor system with id = 1, type = 1
- updatelocation/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{precision}
  http://130.89.12.84:23450/prod/updatelocation/1/1/52,239154/6,850667/100
  Updates the location of sensor system with id = 1, type = 1, and with precision 100
- registerevent/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{value}/{ts}
  http://130.89.12.84:23450/prod/registerevent /1/1/1/130,5/2018-06-12:14:29:15
  Registers an event of type = 1, value = 130,5 and timestamp = 2016-06-12:14:29:15 for sensor system id = 1 and sensor system type = 1
- registerevent/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{value}/{precipitation}/{ts}
  http://130.89.12.84:23450/prod/registerevent /1/1/1/130,5/6/2018-06-12:14:29:15
  Registers an event of type = 1, value = 130,5, precipitation = 6, and timestamp = 2016-06-12:14:29:15 for sensor system id = 1 and sensor system type = 1
- registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id}/{value}/{ts}
  http://130.89.12.84:23450/prod/registermeasurement /1/1/1/200/2018-06-12:14:29:15
  Registers a measurement of type = 1, value = 200, timestamp 2018-06-12:14:29:15, for sensor system id = 1 and sensor system type = 1
- registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{ts}
  http://130.89.12.84:23450/prod/registermeasurement /1/1/1/200/2/23/2018-06-12:14:29:15
  Same as previous, but with 1 additional type of measurement (2 total)
- registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{measurement_type_id3}/{value3}/{ts}
  http://130.89.12.84:23450/prod/registermeasurement /1/1/1/200/2/23/3/44/2018-06-12:14:29:15
  Same as previous, but with 1 additional type of measurement (3 total)
- registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{measurement_type_id3}/{value3}/{measureme

nt_type_id4}/{value4}/{ts}

http://130.89.12.84:23450/prod/registermeasurement /1/1/1/200/2/23/3/44/4/9,21/2018-06-12:14:29:15

Same as previous, but with 1 additional type of measurement (4 total)

- deleteplanneddischarges/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/prod/deleteplanneddischarges/1/1

  Deletes all planned discharges for a single sensor system

Consumer routes

For the consumers interface with prefix *cons*, fourteen routes have been developed. These routes with example GET requests are:

- id/{username}/{password}

  http://130.89.12.84:23450/cons/id/Joeri/mypassword

  Requests the sensor system id for user = Joeri and password = mypassword

- locations/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/locations/1

  Requests current locations of all sensor systems with type = 1

- location/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/location/1/1

  Request current location of sensor system id = 1 and type = 1

- locations/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/locations/1/1

  Request all locations of sensor system id = 1 and type = 1

- characteristics/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/characteristics/1/1

  Request characteristics of sensor system id = 1 and type 1

- measurements/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/measurements/1/1

  Request all measurements of sensor system id = 1 and type = 1

- daymeasurements/{sensorsystem_id}/{sensorsystem_type_id}

  http://130.89.12.84:23450/cons/daymeasurements/1/1

  Request all measurements of current day of sensor system id = 1 and type = 1

- datemeasurements/{sensorsystem_id}/{sensorsystem_type_id}/{date1}/{date2}
  http://130.89.12.84:23450/cons/datemeasurements/1/1/2018-01-01:00:00:00/2018-05-05:23:59:59
  Request all measurements for sensor system id = 1, type = 1, within the specified timeframe
- datemeasurements/{sensorsystem_type_id}/{date1}/{date2}
  http://130.89.12.84:23450/cons/datemeasurements/1/2018-01-01:00:00:00/2018-05-05:23:59:59
  Request all measurements for all sensor systems of type = 1 within the specified timeframe
- buffered/{sensorsystem_id}/{sensorsystem_type_id}
  http://130.89.12.84:23450/cons/buffered/1/1
  Request last month's amount of buffered rainwater for sensor system id = 1 and type = 1
- buffered/{sensorsystem_type_id}
  http://130.89.12.84:23450/cons/buffered/1
  Request last month's total amount of buffered rainwater for all sensor systems of type = 1
- events/{sensorsystem_id}/{sensorsystem_type_id}
  http://130.89.12.84:23450/cons/events/1/1
  Request all events for sensor system id = 1 and type = 1
- events/{sensorsystem_id}/{sensorsystem_type_id}/{date1}/{date2}
  http://130.89.12.84:23450/cons/events/1/1/2018-01-01:00:00:00/2018-05-05:23:59:59
  Request all events for sensor system id = 1 and type = 1 within the specified timeframe
- events/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{date1}/{date2}
  http://130.89.12.84:23450/cons/events/1/1/1/2018-01-01:00:00:00/2018-05-05:23:59:59
  Request all events of event type = 1 for sensor system id = 1 and type = 1 within the specified timeframe

6.3.3 AdminController.php

The AdminController.php file contains the implementation of the functions referenced to by the admin routes (see section 6.3.2 for examples). The complete code can be found in appendix C. The admin controller functions are shown in figure 6.9. The routes pass the in the http post request specified parameters to the functions, after which the values are used in the SQL insert statements.

```
11          public function newSensorsystemType($type){
12                  DB::insert("
13                          INSERT INTO sensorsystem_types (type)
14                          VALUES (?)", [$type]);
15          }
16
17          public function newCharacteristicType($type, $sensorsystem_type_id){
18                  DB::insert("
19                          INSERT INTO characteristic_types (type, sensorsystem_type_id)
20                          VALUES (?,?)", [$type, $sensorsystem_type_id]);
21          }
22
23          public function newEventType($type, $sensorsystem_type_id){
24                  DB::insert("
25                          INSERT INTO event_types (type, sensorsystem_type_id)
26                          VALUES (?,?)", [$type, $sensorsystem_type_id]);
27          }
28
29          public function newMeasurementType($type, $sensorsystem_type_id){
30                  DB::insert("
31                          INSERT INTO measurement_types (sensorsystem_type_id, type)
32                          VALUES (?,?)", [$sensorsystem_type_id, $type]);
33          }
```

Figure 6.9, The admin controller's functions

6.3.4 ProducerController.php

The producer functions will be described in this section. The complete code can be found in appendix D.

Validation functions

The following validation functions are used to verify the in the request specified parameters prior to making changes in the database. The three functions in figure 6.10 are used to verify whether a sensor system id is already registered, whether the type of sensor system is registered by the administrator in the sensorsystem_types table, and whether a requested sensor system id and sensor system type id match respectively.

```
15          // Check if sensorsystem id exists
16          public function sensorsystemIdExists($id){
17                  return DB::select("SELECT count(id) FROM sensorsystems WHERE id = ?", [$id])[0]->count == 1;
18          }
19
20          // Check if sensorsystem type id exists
21          public function sensorsystemTypeExists($type_id){
22                  return DB::select("SELECT count(id) FROM sensorsystem_types WHERE id = ?", [$type_id])[0]->count == 1;
23          }
24
25          // Validate sensorsystem type with sensorsystem id
26          public function validateSensorsystem($id, $type_id){
27                  if(!$this->sensorsystemIdExists($id)) return -1;
28                  if(!$this->sensorsystemTypeExists($type_id)) return -2;
29                  if(DB::select("SELECT s.sensorsystem_type_id
30                                  FROM sensorsystems s
31                                  WHERE s.id = ?", [$id])[0]->sensorsystem_type_id != $type_id) return -3;
32                  return 1;
33          }
```

Figure 6.10, Sensor system validation functions

The function in figure 6.11 is used to check whether a user exists in the users table.

```
35          // Check if user exists
36          public function userExists($username){
37          return DB::select("SELECT EXISTS (SELECT true FROM users WHERE username = ?)", [$username])[0]->exists;
38          }
```

Figure 6.11, User validation function

Validation functions for characteristics, events, and measurements can be found in figure 6.12. The existence of a characteristic type that is requested for registration, as well as validation whether the characteristic type corresponds with the sensor system type and whether the characteristic with corresponding value has already been registered for the sensor system id is checked in the function called *characteristicExists*.

The same goes for the measurements and events. The first check that is made in the *eventTypeExists* function is whether the type of event exists in the event_types table. After that, the combination of event type and sensor system type are validated by checking whether these match in the event_types table.

```
40          // Check characteristic
41          public function characteristicExists($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id){
42                  if(DB::select("SELECT count(id) FROM characteristic_types WHERE id = ?", [$characteristic_type_id])[0]->count != 1){
43                          return -1;
44                  } else if(DB::select("SELECT sensorsystem_type_id FROM characteristic_types WHERE id = ?", [$characteristic_type_id])[0]->sensorsystem_type_id != $s
    ensorsystem_type_id){
45                          return -2;
46                  } else if(DB::select("SELECT characteristic_type_id FROM characteristics WHERE sensorsystem_id = ?", [$sensorsystem_id]) != NULL){
47                          return -3;
48                  } else {
49                          return 1;
50                  }
51          }
52
53          // Check events
54          public function eventTypeExists($sensorsystem_type_id, $event_type_id){
55                  if(DB::select("SELECT count(id) FROM event_types WHERE id = ?", [$event_type_id])[0]->count != 1){
56                          return -1;
57                  } else if(DB::select("SELECT sensorsystem_type_id FROM event_types WHERE id = ?", [$event_type_id])[0]->sensorsystem_type_id != $sensorsystem_type_i
    d){
58                          return -2;
59                  } else {
60                          return 1;
61                  }
62          }
63
64          // Validate measurement types
65          public function validateMeasurementType($sensorsystem_type_id, $measurement_type_id){
66
67                  return ($sensorsystem_type_id == DB::select("SELECT sensorsystem_type_id FROM measurement_types WHERE id = ?",[$measurement_type_id])[0]->sensorsyst
    em_type_id);
68          }
```

Figure 6.12, Characteristic, event, and measurement validation functions

Sensor system registration functions

Three functions regarding the registrations of a sensor system have been implemented. These are the basic registration, registration with added value for location precision, and registration with a user.

The basic registration function can be seen in figure 6.13. This function requires the sensor system id, the sensor system type, and the latitude and longitude of the sensor system's location. First the uniqueness of the to be registered sensor system's id is validated, by calling the function *sensorsystemIdExists*. If the sensor system id already exists in the database, a response message and

http status code is returned. If the to be registered sensor system id is unique, the existence of the to be registered type of the sensor system is validated. If the sensor system type exists, the sensor system and its location will be inserted into the corresponding database tables, and a response message and http status code is returned. If the sensor system type doesn't exist, nothing will be inserted into the database and a response message and http status code will be returned.

The SQL statement on line 82 contains some functions that require elaboration in order to understand what is going on. The *now()* function is used to insert the timestamp of the moment of insertion of the new sensor system. The *ST_GeomFromText* creates a geometry object of the type *point*, with SRID 4326. The function *ST_Transform* is used to project the SRID 4326 onto the cartesian special reference system with SRID 28992, which is the Dutch RD-coordinate system (see section 6.2.1).

```
72      // Register new sensorsystem
73      public function registerSensorsystem($sensorsystem_id, $sensorsystem_type_id, $lat, $lon)
74      {
75              if($this->sensorsystemIdExists($sensorsystem_id)){
76                      return (new Response("sensorsystem_id " .$sensorsystem_id ." already exists", 400));
77              } else if($this->sensorsystemTypeExists($sensorsystem_type_id)){
78                      DB::insert("INSERT INTO sensorsystems (id, sensorsystem_type_id) VALUES (?, ?)", [$sensorsystem_id, $sensorsystem_type_id]);
79                      $lon = str_replace(',','.', $lon);
80                      $lat = str_replace(',','.', $lat);
81                      $point = "POINT(" .$lon ." " .$lat .")";
82                      DB::insert("INSERT INTO locations (sensorsystem_id, ts, location) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992))",[$sensorsy
stem_id, $point]);
83                      return (new Response("Sensor system registration successful!", 200));
84              } else {
85                      return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist", 400));
86              }
87      }
```

Figure 6.13, Basic sensor system registration function

The basic registration function is extended with the possibility to add a value for the precision of the latitude longitude pair. This function can be found in the complete code in appendix D. The only difference here is one extra value in the *INSERT INTO locations* statement on line 82 of figure 6.14.

It's also possible to register a sensor system with a username and password. In order to do so, the two lines in figure 6.14 are added to the basic sensor system registration function as well as a call to the *userExists* validation function prior to inserting data into the database. Again, for the complete code see appendix D. The password will be hashed using the sha256 hash algorithm, before it will be stored in the database.

```
117                     $hashedPassword = hash('sha256', $password);
118                     DB::insert("INSERT INTO users (username, password, sensorsystem_id) VALUES (?, ? ,?)", [$username, $hashedPassword, $sensorsystem_id]);
```

Figure 6.14, Registration with username and password

Characteristic registration functions

For the registration of sensor system characteristics, the producer provides the function in figure 6.15 with the *sensorsystem_id*, *sensorsystem_type_id*, *characteristic_type_id*, and the *characteristic_value*. First the call to the function *validateSensorsystem* validates the existence of the sensor system id with corresponding type. If no conflicts arise, the function *characteristicExists* validates the existence of the characteristic type with matching sensor system type, and checks whether or not the characteristic has already been registered with accompanied value. Again, if all validations pass, the characteristic data will be inserted into the database tables. As can be seen in figure 6.15, again all outcomes of validation and successful data insertion return response messages and http status codes.

```
128         //Register sensor system characteristics
129         public function registerSensorsystemCharacteristic($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id, $characteristic_value)
130         {
131                 $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
132                 switch ($sensorsystemValidationCode){
133                 case -1:
134                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
135                         break;
136                 case -2:
137                         return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
138                         break;
139                 case -3:
140                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
141                         break;
142                 }
143
144                 $characteristicValidationCode = $this->characteristicExists($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id);
145                 switch($characteristicValidationCode){
146                 case 1:
147                         $characteristic_value = str_replace(',','.', $characteristic_value);
148                         DB::insert("
149                                 INSERT INTO characteristics (sensorsystem_id, characteristic_type_id, value)
150                                 VALUES (?, ?, ?)", [$sensorsystem_id, $characteristic_type_id, $characteristic_value]);
151                         return (new Response("sensorsystem characteristic registration successful!", 200));
152                         break;
153                 case -1:
154                         return (new Response("Characteristic type doesn't exist",400));
155                         break;
156                 case -2:
157                         return (new Response("Characteristic type not registered for sensor system type",400));
158                 case -3:
159                         return (new Response("Characteristic already registered",400));
160                         break;
161                 }
162         }
```

Figure 6.15, Characteristic registration function

Location update functions

Regarding updating a sensor system's location, two functions are implemented. One for updating the location without a value for location precision, and one for updating the location with a value for location precision. The same validation on lines 131-142 of figure 6.15 is used in both location update functions. See appendix D for the *updateSensorsystemLocations* and *updateSensorsystemLocationWithLocationPrecision* function's code.

Event registration functions

Again the same validation on lines 131-142 of figure 6.15 is used in the functions *newEvent* and *newEventWithPrecipitation*. Additionally, the event type specified in the request is validated by the validation function *eventTypeExists* (see figure 6.16). This functions verifies the existence of the

requested event type in the database, as well as whether the event type corresponds with the in the request specified sensor system type. The difference of the *newEventWithPrecipitation* function with the function shown in figure 6.16, is one added function parameter, and one extra value in the *INSERT INTO events* statement on line 244. For the complete code, see appendix D.

```
219         public function newEvent($sensorsystem_id, $sensorsystem_type_id, $event_type_id, $value, $ts)
220         {
221                 $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
222                 switch ($sensorsystemValidationCode){
223                 case -1:
224                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
225                         break;
226                 case -2:
227                         return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
228                         break;
229                 case -3:
230                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
231                         break;
232                 }
233
234                 $eventValidationCode = $this->eventTypeExists($sensorsystem_type_id, $event_type_id);
235                 switch ($eventValidationCode){
236                 case -1:
237                         return (new Response("Event type doesn't exist",400));
238                         break;
239                 case -2:
240                         return (new Response("Event type not registered for the sensor system type",400));
241                         break;
242                 case 1:
243                         $value = str_replace(',','.', $value);
244                         DB::insert("INSERT INTO events (sensorsystem_id, event_type_id, value, ts) VALUES (?, ?, ?, ?)",[$sensorsystem_id, $event_type_id, $value, $
    ts]);
245                         return (new Response("Event registration successful!",200));
246                 }
247         }
```

Figure 6.16, The new event registration function

Measurement registration functions

A measurement post request can contain up to four measurements (see section 5.3.3). Four functions have been implemented, namely *oneNewMeasurement*, *twoNewMeasurements*, *threeNewMeasurements*, and *fourNewMeasurements*. In order to explain the measurements functions, the function for registering four measurements will be explained. The code of this function can be found in figure 6.17. First, the sensor system and sensor system type are validated again using the *validateSensorsystem* function. After that, all measurement types in the request are put into an array. In the foreach loop, the *validateMeasurement_type* function is called for each measurement type in the array. If all specified measurement types pass validation, the measurements are inserted into the measurements table.

```
369         public function fourNewMeasurements($sensorsystem_id, $sensorsystem_type_id, $measurement_type_id1, $value1, $measurement_type_id2, $value2, $measurement_ty
    pe_id3, $value3, $measurement_type_id4, $value4, $ts)
370         {
371                 $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
372                 switch ($sensorsystemValidationCode){
373                 case -1:
374                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
375                         break;
376                 case -2:
377                         return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
378                         break;
379                 case -3:
380                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
381                         break;
382                 }
383
384                 $requested_types = array((int)$measurement_type_id1, (int)$measurement_type_id2, (int)$measurement_type_id3, (int)$measurement_type_id4);
385                 foreach ($requested_types as $measurement_type){
386                         if(!$this->validateMeasurementType($sensorsystem_type_id, $measurement_type)){
387                                 return (new Response("Measurement type " .$measurement_type ." not registered with sensor system type " .$sensorsystem_type_id,400))
    ;
388                         }
389                 }
390
391                 $value1 = str_replace(',','.', $value1);
392                 $value2 = str_replace(',','.', $value2);
393                 $value3 = str_replace(',','.', $value3);
394                 $value4 = str_replace(',','.', $value4);
395
396                 DB::insert("INSERT INTO measurements(sensorsystem_id, measurement_type_id, value, ts) VALUES (?, ?, ?, ?), (?, ?, ?, ?), (?, ?, ?, ?), (?, ?, ?, ?)"
    ,[$sensorsystem_id, $measurement_type_id1, $value1, $ts, $sensorsystem_id, $measurement_type_id2, $value2, $ts, $sensorsystem_id, $measurement_type_id3, $value3, $t
    s, $sensorsystem_id, $measurement_type_id4, $value4, $ts]);
397
398 return (new Response("Measurements registration successful!",200));
399         }
```

Figure 6.17, Function for registration of four measurements

6.3.5 ConsumerController.php

In this section, crucial parts of the consumer functions will be described. The complete code can be found in appendix E.

Validation functions

The same validation functions as in the producers interface are used (see section 6.3.4), except for the characteristics, events, and measurement validation functions.

User's sensor system id

When a SRB user logs in in the SRB dashboard application, the user's SRB sensor system id can be requested at the consumer interface. The function *getUserSensorsystemId* in figure 6.18 takes the username and password as parameters, after which the validation function *userExists* is called to check whether the user exists in the users table. If the provided password is correct, the user's SRB sensor system id will be returned.

```
43          // Get sensorsystem id for user
44          public function getUserSensorsystemId($username, $password){
45
46                  if ($this->userExists($username)){
47                          if (strcmp(hash('sha256', $password), DB::select("SELECT password FROM users WHERE username = ?", [$username])[0]->password) == 0){
48                                  return (new Response(DB::select("
49                                          SELECT u.sensorsystem_id, st.type
50                                          FROM users u, sensorsystem_types st, sensorsystems s
51                                          WHERE username = ? AND s.id = u.sensorsystem_id AND s.sensorsystem_type_id = st.id", [$username]), 200));
52                          } else {
53                                  return (new Response("password incorrect", 400));
54                          }
55                          return "no";
56                  } else {
57                          return (new Response("user doesn't exist", 400));
58                  }
59          }
```

Figure 6.18, Function for retrieving a user's sensor system id

Locations

Regarding the retrieval of sensor system locations, three functions have been implemented. The first, *getAllSensorsystemsCurrentLocations*, returns all current locations of all sensor systems of a single sensor system type (see figure 6.19). It takes the sensor system type as parameter and validates its existence in the database. On line 68, the sensor system's locations are transformed from the RD-coordinate system, to SRID 4326 using the *ST_Transform* function, after which the functions *ST_YMax* and *ST_XMax* extract the latitude and longitude respectively from the point geometry object. The sub query starting on line 71, selects the most recent stored locations using the *max()* function.

```
63      // Current locations of all sensorsystems of specifiied sensorsystem type
64      public function getAllSensorsystemsCurrentLocations($sensorsystem_type_id)
65      {
66              if($this->sensorsystemTypeExists($sensorsystem_type_id)){
67                      return (new Response(DB::select("
68                      SELECT s.id AS sensorsystem_id, ST_YMax(ST_Transform(l.location,4326)) AS latitude, ST_XMax(ST_Transform(l.location,4326)) AS longitude
69                      FROM sensorsystems s, locations l
70                      WHERE s.sensorsystem_type_id = ? AND l.sensorsystem_id = s.id AND l.ts = (
71                              SELECT max(ts)
72                              FROM locations
73                              WHERE sensorsystem_id = l.sensorsystem_id)", [$sensorsystem_type_id]), 200));
74              } else return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
75      }
```

Figure 6.19, Function for retrieving all locations of all sensor systems of a single sensor system type

The second function regarding sensor system location is *getSingleSensorsystemCurrentLocation*, which returns the current location of a single sensor system. The only differences in the code shown in figure 6.19, are that the function takes an extra parameter for the sensor system id, calls the *validateSensorsystem* function, and has an extra statement in the WHERE clause starting on line 70 (see figure 6.20).

```
FROM sensorsystems s, locations l
WHERE l.sensorsystem_id = s.id AND l.ts = (
        SELECT max(ts)
        FROM locations
        WHERE sensorsystem_id = l.sensorsystem_id)
        AND l.sensorsystem_id = ?", [$sensorsystem_id]), 200));
```

Figure 6.20, Difference with the function in figure 6.19, heighted with yellow

The final function regarding sensor system locations is *getSingleSensorsystemAllLocations*, which returns the current as well as the historic locations of a single sensor system. The function takes two parameters, for the sensor system id and sensor system type id. These are validated again using the function *validateSensorsystem*. The raw SQL is quite similar to the previous described location functions, and can be found below in figure 6.21.

```
return (new Response(DB::select("
SELECT l.sensorsystem_id AS sensorsystem_id, ST_YMax(ST_Transform(l.location,4326)) AS latitude, ST_XMax(ST_Transform(l.location,4326))
AS longitude, l.ts AS timestamp
FROM locations l
WHERE l.sensorsystem_id = ?
ORDER BY timestamp", [$sensorsystem_id]), 200));
```

Figure 6.21, Raw SQL for retrieval of all locations of a single sensor system

Characteristics

One function regarding characteristics has been implemented in the consumers interface, namely *getSingleSensorsystemCharacteristics* (see figure 6.22). The function takes two parameters: the sensor system id and the sensor system type id. Again, the both are validated using the *validateSensorsystem* function. When the sensor system send in the request is valid, the validation function returns '1', after which *case 1* of the switch statement on line 141 will be executed. Another switch statement will execute code depending on the sensor system type id. This is because some extra functionality has been implemented for the SRB and AirT dashboard developers next to the default functionality.

On line 143, a string is stored in the variable *$crosstab*. This string contains SQL, which will be used in the crosstab function on line 151 and 160. The crosstab function was added by creating the tablefunc extension for PostgreSQL (see section 6.2.4). Using this function, pivot tables can be created. This implemented functionality is added as extra service to the SRB and AirT dashboard developers. By the use of the crosstab function, each different characteristic type will be represented in the result by its own column instead of having a characteristic_type column with separate rows for each type of characteristic (see tables 6.1 and 6.2 for an example). The SQL in the default option, which will be used when a new sensor system type is added to the project, produces results as shown in table 6.2.

```
136         // All characteristics of single sensorsystem
137         public function getSingleSensorsystemCharacteristics($sensorsystem_id, $sensorsystem_type_id)
138         {
139                 $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
140
141                 switch ($validationCode){
142                 case 1:
143                         $crosstab = "
144                                 SELECT c.sensorsystem_id, ct.type, c.value
145                                 FROM characteristics c, characteristic_types ct
146                                 WHERE c.characteristic_type_id = ct.id AND c.sensorsystem_id = " .$sensorsystem_id ." order by 1";
147
148                         switch ($sensorsystem_type_id) {
149                         case 1:
150                                 return (new Response(DB::select("
151                                         SELECT * FROM crosstab (?) AS (
152                                                 sensorsystem_id INTEGER,
153                                                 capacity numeric,
154                                                 waterquality_guard_setting numeric,
155                                                 freeze_guard_setting numeric,
156                                                 water_collection_area numeric)",[$crosstab]), 200));
157                                 break;
158                         case 2:
159                                 return (new Response(DB::select("
160                                         SELECT * FROM crosstab (?) AS (
161                                                 sensorsystem_id INTEGER,
162                                                 skyview_factor numeric)",[$crosstab]), 200));
163                                 break;
164                         default:
165                                 return (new Response(DB::select("
166                                         SELECT c.sensorsystem_id, ct.type, c.value
167                                         FROM characteristics c, characteristic_types ct
168                                         WHERE c.characteristic_type_id = ct.id AND c.sensorsystem_id = ?", [$sensorsystem_id]), 200));
169                         }
170                         break;
171                 case -1:
172                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
173                         break;
174                 case -2:
175                         return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
176                         break;
177                 case -3:
178                         return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
179                         break;
180                 }
181         }
```

Figure 6.22, Function for retrieval of sensor system characteristics

Table 6.1, Example of the query's result using the crosstab() function

sensorsystem_id	capacity	water_quality_guard_setting	freeze_guard_setting	water_collection_area
1	500	1	1	20

Table 6.2, Example of the query's result without using the crosstab() function

sensorsystem_id	characteristic type	value
1	capacity	500
1	water_quality_guard_setting	1
1	freeze_guard_setting	1
1	water_collection_area	20

Measurements

Four functions regarding retrieval of sensor system measurements have been implemented. The first, *getAllMeasurements*, provides the consumer with all measurements of a single sensor system. It takes parameters for the sensor system id and the sensor system type, which will be verified using the same verification functions explained numerous times before. The same extra functionality as with the characteristic function is implemented in all measurement functions as well. The second function, *getDayMeasurements*, provides the consumer with all measurements of a single sensor system on the

current day. The third function, *getMeasurementsPeriod*, takes two extra parameters: *date1* and *date2*. The function provides the consumer with all measurements of a single sensor system within the specified time period. Finally, the function *getAllMeasurementsPeriod* takes parameters for the sensor system type id, and two dates specifying the timeframe. This function returns all measurements for all sensor systems of a single type within the specified timeframe. See appendix E for the complete code.

Last month's amount of buffered rainwater

Two functions have been implemented as extra service for the SRB dashboard developer. The first, *getBuffered*, provides the consumer with last month's amount of buffered rainwater for a single sensor system (see figure 6.23). This function sums the increase in the SRB's fill level for last month.

```
445         // Get buffered amount single sensor
446         public function getBuffered($sensorsystem_id, $sensorsystem_type_id)
447         {
448
449                 return DB::select("
450                         SELECT sum(increase) AS buffered
451                         FROM (
452                                 SELECT value - lag(value) OVER() AS increase
453                                 FROM measurements
454                                 WHERE measurement_type_id = 1
455                                 AND sensorsystem_id = ?
456                                 AND ts >= date_trunc('month', current_date - interval '1 MONTH')
457                                 AND ts < date_trunc('month', current_date)) AS t
458                         WHERE increase > 0", [$sensorsystem_id]);
459         }
```

Figure 6.23, Function providing last month's amount of buffered rainwater for a single sensor system

The second function, *getBufferedTotal*, provides the consumer with last month's amount of buffered rainwater for all SRBs (see figure 6.24). First, all sensor system ids of the type SRB are stored in the variable *$sensorsystemids.* In the foreach loop, the same query as in figure 6.23 is run for every SRB and the results are summed. The summed result is stored in an object, which then is encoded as json and returned to the consumer.

```
462         // Get total buffered amount
463         public function getBufferedTotal($sensorsystem_type_id)
464         {
465                 $sensorsystemids =  DB::select("SELECT * from sensorsystems WHERE sensorsystem_type_id = 1");
466                 $sum = 0.0;
467
468                 foreach ($sensorsystemids AS $ssid){
469                         $sum += DB::select("
470                                 SELECT sum(increase) AS buffered
471                                 FROM (
472                                         SELECT value - lag(value) OVER() AS increase
473                                         FROM measurements
474                                         WHERE measurement_type_id = 1 AND sensorsystem_id = ?
475                                         AND ts >= date_trunc('month', current_date - interval '1 MONTH')
476                                         AND ts < date_trunc('month', current_date)) AS t
477                                 WHERE increase > 0", [$ssid->id])[0]->buffered;
478         }
479                 $result = new \stdClass();
480                 $result->total_buffered = $sum;
481                 $myJSON = json_encode($result);
482                 return response()->json($myJSON);
483         }
```

Figure 6.24, Function providing last month's total amount of buffered rainwater.

7. Evaluation

In this chapter the functional testing will be described.

7.1 Postman

Postman [41] will be used to simulate administrator, producer, and consumer projects. Using The interfaces will be tested by making HTTP requests using the Postman application. With Postman, collections of requests can be made. The collections of request are shown on the left side of the screenshot in figure 7.1. For each route that has been implemented (see section 6.3.2) a request has been added to a collection in Postman. Requests are grouped per role and type. For example, all consumer routes concerning measurements are grouped in the collection called CONS measurements.

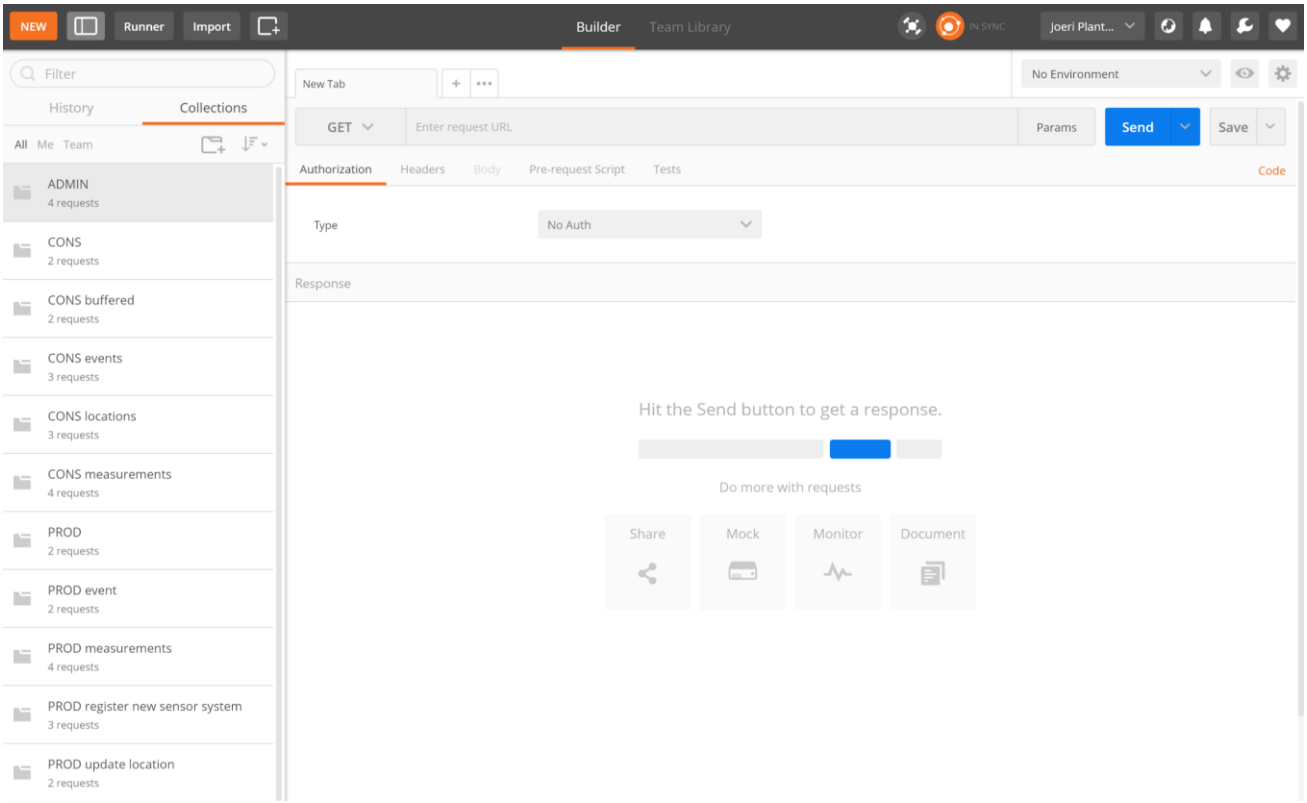

Figure 7.1, Screenshot of the Postman environment

7.2 Functional testing

This section describes the test process used for the functional testing using Postman. First test data will be listed with corresponding requests, expected result, and actual result. The tests will be grouped per Postman collection.

The test will start with the *ADMIN* collection, testing the insertion of new sensor system types, new characteristic types, new event types, and new measurement types. After that, test data will be produced and POST requests will be made to the producers interface. Producing the test data will start with the *PROD register new sensor system* collection, after which test characteristics, test measurements, test events, and updated test locations will be registered. When testing the producer interface is completed successfully, the consumer interface will be tested, starting with requesting the sensor system id for a user. After that the characteristics, locations, events, and measurement routes will be tested.

7.2.1 Tests

In this section the test data is listed, http request are set up, and expected and actual test results are listed. Everything will be grouped per Postman collection. All requests have prefix *http://130.89.12.84:23450/*. Parameters should be added to the end of the request.

ADMIN

The following administrator interface functions will be tested:

- Register new sensor system type
  - Parameters: *sst_test*
  - Request: *admin/newsensorsystemtype/*
  - Expected result: *Status: 200 OK*
  - Result: *Status: 200 OK*

- Register new characteristic type
  - Parameters: c*t_test/3*
  - Request: *admin/newcharacteristictype/*
  - Expected result: *Status: 200 OK*
  - Result: *Status: 200 OK*

- Register new event type
  - Parameters: *et_test/3*
  - Request: *admin/neweventtype/*
  - Expected result: *Status: 200 OK*
  - Result: *Status : 200 OK*

- Register new measurement type
  - Parameters: *mt_test/3*
  - Request: *admin/newmeasurementtype/*
  - Expected result: *Status: 200 OK*
  - Result: *Status: 200 OK*

All tests for the administrator interface have passed.

PROD register new sensor system

- Register new sensor system

    - Parameters: *999/3/52,225781/6,915926*
    - Request: *prod/registersensorsystem/*
    - Expected result: *Sensor system registration successful! Status: 200 OK*
    - Result: *Sensor system registration successful! Status: 200 OK*

- Register new sensor system with user
    - Parameters: *998/3/52,225781/6,915926/user/pw*
    - Request: *prod/registersensorsystem/*
    - Expected result: *Sensor system registration successful! Status: 200 OK*
    - Result: *Sensor system registration successful! Status: 200 OK*

- Register new sensor system with location precision
    - Parameters: 997/3/*52,225781/6,915926/100*
    - Request: *prod/registersensorsystem/*
    - Expected result: *Sensor system registration successful! Status: 200 OK*
    - Result: *Sensor system registration successful! Status: 200 OK*

PROD update location

- Update location
    - Parameters: *999/3/52,181175/6,905999*
    - Request: *prod/updatelocation/*
    - Expected result: *Sensor system location update successful Status: 200 OK*
    - Result: *Sensor system location update successful Status: 200 OK*

- Update location with precision
    - Parameters: *999/3/52,181175/6,905999/33*
    - Request: *prod/updatelocation/*
    - Expected result: *Sensor system location update successful Status: 200 OK*
    - Result: *Sensor system location update successful Status: 200 OK*

PROD

- Register characteristics
    - Parameters:
        - *999/3/6/100*
        - *998/3/6/200*
        - *997/3/6/300*
    - Request:
        - *prod/registercharacteristic/*
    - Expected results:
        - *sensorsystem characteristic registration successful! Status: 200 OK*
    - Results:
        - *sensorsystem characteristic registration successful! Status: 200 OK*

PROD event

- Register event
    - Parameters:
        - *999/3/4/100/now()*
        - *998/3/4/200/now()*
        - *997/3/4/300/now()*
    - Requests:
        - *prod/registerevent/*
    - Expected results:
        - *Event registration successful! Status: 200 OK*
    - Results:

        - *Event registration successful! Status: 200 OK*

- Register event with second value
    - Parameters:
        - *999/3/4/1/2/now()*
        - *998/3/4/3/4/now()*
        - *997/3/4/5/6/now()*
    - Request:
        - *prod/registerevent/*
    - Expected results:
        - *Event registration successful! Status: 200 OK*
    - Results:
        - *Event registration successful! Status: 200 OK*

PROD measurements

- Insert one measurement
    - Parameters:
        - *999/3/6/999/now()*
    - Request:
        - *prod/registermeasurement/*
    - Expected result:
        - *Measurement registration successful! Status: 200 OK*
    - Result:
        - *Measurement registration successful! Status: 200 OK*

- Insert two measurements
    - Parameters
        - *998/3/6/998/6/997/now()*
    - Request:
        - *prod/registermeasurement/*
    - Expected result:
        - *Measurement registration successful! Status: 200 OK*
    - Result:
        - *Measurement registration successful! Status: 200 OK*

- Insert three measurements
    - Parameters
        - *997/3/6/996/6/995/6/994/now()*
    - Request:
        - *prod/registermeasurement/*
    - Expected result:
        - *Measurement registration successful! Status: 200 OK*
    - Result:
        - *Measurement registration successful! Status: 200 OK*

- Register four measurements
    - Parameters
        - *999/3/6/1/6/2/6/3/6/4,5/now()*
    - Request:
        - *prod/registermeasurement/*
    - Expected result:
        - *Measurement registration successful! Status: 200 OK*
    - Result:
        - *Measurement registration successful! Status: 200 OK*

All tests for the producers interface have passed.

CONS

- Get sensor system id for user
  - Parameters:
    - *user/pw*
  - Request:
    - cons/id/
  - Expected results:
    - *sensorsystem_id:* *998*
      type: *sst_test*
  - Results:
    - *sensorsystem_id:* *998*
      type: *sst_test*

- Get characteristics of sensor system
  - Parameters:
    - *999/3*
    - *998/3*
    - *997/3*
  - Request:
    - *cons/characteristics/*
  - Expected results:
    - *sensorsystem_id:* *999*
      type: *ct_test*
      value: *100*
    - *sensorsystem_id:* *998*
      type: *ct_test*
      value: *200*
    - *sensorsystem_id:* *997*
      type: *ct_test*
      value: *300*
  - Results:
    - *sensorsystem_id:* *999*
      type: *ct_test*
      value: *100*
    - *sensorsystem_id:* *998*
      type: *ct_test*
      value: *200*
    - *sensorsystem_id:* *997*
      type: *ct_test*
      value: *300*

CONS locations

- Current single sensor location
  - Parameters:
    - *999/3*
  - Request:
    - *cons/location/*
  - Expected result:
    - *sensorsystem_id:* *999*
      latitude: *52.181175*
      longitude: *6.905999*

- Results:
  - *sensorsystem_id:* *999*
    latitude: *57.2286740056091*
    longitude: *6.8550050031981*

- All single sensor system locations
  - Parameters:
    - *999/3*
  - Request:
    - *cons/locations*
  - Expected results:
    - *sensorsystem_id:* *999*
      latitude: *52.225781*
      longitude: *6.915926*
      timestamp: *unknown*
    - *sensorsystem_id:* *999*
      latitude: *52.181175*
      longitude: *6.905999*
      timestamp: *unknown*
    - *sensorsystem_id:* *999*
      latitude: *52.181175*
      longitude: *6.905999*
      timestamp: *unknown*
  - Results:
    - *sensorsystem_id:* *999*
      latitude: *52.2257810040113*
      longitude: *6.91592600329726*
      timestamp: *2018-06-29 21:53:39.578025*
    - *sensorsystem_id:* *999*
      latitude: *52.1811750039849*
      longitude: *6.90599900328681*
      timestamp: *2018-06-29 21:54:22.511096*
    - *sensorsystem_id:* *999*
      latitude: *52.1811750039849*
      longitude: *6.90599900328681*
      timestamp: *2018-06-29 22:00:04.061523*

- All current locations of a single type of sensor system
  - Parameters:
    - 3
  - Request:
    - cons/locations/
  - Expected results:
    - *sensorsystem_id:* *999*
      latitude: *52.181175*
      longitude: *6.905999*
    - *sensorsystem_id:* *998*
      latitude: *52.225781*
      longitude: *6.915926*

        - *sensorsystem_id:* *997*
          latitude: *52.225781*
          longitude: *6.915926*
    - Results:
        - *sensorsystem_id:* *999*
          latitude: *57.2286740056091*
          longitude: *6.8550050031981*
        - *sensorsystem_id:* *998*
          latitude: *52.2257810040113*
          longitude: *6.91592600329726*
        - *sensorsystem_id:* *997*
          latitude: *52.2257810040113*
          longitude: *6.91592600329726*

CONS events

- Get all events of a single sensor system
    - Parameters:
        - *999/3*
    - Request:
        - cons/events
    - Expected results:
        - *sensorsystem_id:* *999*
          event_type: *et_test*
          value: *100*
          precipitation: *null*
          timestamp: *unknown*
        - *sensorsystem_id:* *999*
          event_type: *et_test*
          value: *1*
          precipitation: *2*
          timestamp: *unknown*
    - Results:
        - *sensorsystem_id:* *999*
          event_type: *et_test*
          value: *100*
          precipitation: *null*
          timestamp: *2018-06-29 22:05:04.635241*
        - *sensorsystem_id:* *999*
          event_type: *et_test*
          value: *1*
          precipitation: *2*
          timestamp: *2018-06-29 22:06:49.009815*

- All events for a single sensor system within specified time period
    - Parameters:
        - *999/3/2018-06-01/now()*
    - Request:
        - cons/events
    - Expected results:
        - *sensorsystem_id:* *999*
          event_type: *et_test*

value: *100*
timestamp: *unknown*
- *sensorsystem_id:* *999*
event_type: *et_test*
value: *1*
timestamp: *unknown*

- Results:
  - *sensorsystem_id:* *999*
event_type: *et_test*
value: *100*
timestamp: *2018-06-29 22:05:04.635241*
  - *sensorsystem_id:* *999*
event_type: *et_test*
value: *1*
timestamp: *2018-06-29 22:06:49.009815*

- Get specific event for single sensor system within time period
  - Parameters:
    - *999/3/4/2018-06-01/now()*
  - Request:
    - cons/events
  - Expected results:
    - *sensorsystem_id:* *999*
event_type: *et_test*
value: *100*
timestamp: *unknown*
    - *sensorsystem_id:* *999*
event_type: *et_test*
value: *1*
timestamp: *unknown*
  - Results:
    - *sensorsystem_id:* *999*
event_type: *et_test*
value: *100*
timestamp: *2018-06-29 22:05:04.635241*
    - *sensorsystem_id:* *999*
event_type: *et_test*
value: *1*
timestamp: *2018-06-29 22:06:49.009815*

CONS measurements

- Get all measurements for a single sensor system
  - Parameters:
    - *998/3*
  - Request:
    - cons/measurements
  - Expected results:
    - *id:* *998*
event_type: *mt_test*

value: *998*
ts: *unknown*
- *id:* *998*
event_type: *mt_test*
value: *997*
ts: *unknown*

- Results:
  - *id:* *998*
event_type: *mt_test*
value: *998*
ts: *2018-06-29 22:09:03.761232*
  - *id:* *998*
event_type: *mt_test*
value: *997*
ts: *2018-06-29 22:09:34.011818*

- All measurements current day for a single sensor system
  - Parameters:
    - *998/3*
  - Request:
    - cons/daymeasurements
  - Expected results:
  - Results:

- All measurements for a single sensor system in time period
  - Parameters:
    - *998/3/2018-06-28/2018-06-30*
  - Request:
    - cons/datemeasurements
  - Expected results:
    - *id:* *998*
event_type: *mt_test*
value: *998*
ts: *unknown*
    - *id:* *998*
event_type: *mt_test*
value: *997*
ts: *unknown*
  - Results:
    - *id:* *998*
event_type: *mt_test*
value: *998*
ts: *2018-06-29 22:09:03.761232*
    - *id:* *998*
event_type: *mt_test*
value: *997*
ts: *2018-06-29 22:09:34.011818*

CONS buffered amount

In order to test the consumer interface functions for last month's total and single sensor system's buffered amount of rainwater, three test sensor systems of the type SRB were used. The following fill level data was inserted into the database, with all values from old to new and in liters:

- testsrb1: 100, 50.54, 311.76, 200
    - Expected total buffered: *211.76*
    - Result: *211.76*
- testsrb2: 20.4, 102.89, 300, 99
    - Expected total buffered: *279.6*
    - Result: *279.6*
- testsrb3: 450, 100, 25, 250.3
    - Expected total buffered: *225.3*
    - Result: *225.3*

The buffered amount functions work as expected.

7.3 Performance

In order to test the performance of the system, Postman is used again. The routes and their response time are listed below, as well as the number of rows in the tables used by each route.

7.3.1 Admin interface

In this section the admin routes and their response time are listed, as well the number of rows in each if the tables accessed by the routes. The accessed tables are listed below the route name. The number of rows in each table are indicated by the number in brackets behind the table name. The response time is indicated by number in brackets after the route name.

Register new sensor system type (86 ms)
sensorsystem_types (3)

Register new characteristic type (90 ms)
characteristic_types (6)

Register new event type (569 ms)
event_types(4)

Register new measurement type (194 ms)
event_types(6)

7.3.2 Producer interface

In this section the producers routes and their response time are listed, as well the number of rows in each if the tables accessed by the routes. The accessed tables are listed below the route name. The number of rows in each table are indicated by the number in brackets behind the table name. The response time is indicated by number in brackets behind the route name.

Sensor system registration (107 ms)
sensorsystems(7)
locations(47)

Sensor system registration with user (589 ms)
sensorsystems(7)
locations(47)
users(2)

Update location (614 ms)
locations(48)

register characteristic (580 ms)
characteristics(4)

Register event (94 ms)
events(4573)

Register four measurements (110 ms)
measurements(186177)

7.3.3 Consumer interface

In this section the comsumers routes and their response time are listed, as well the number of rows in each if the tables accessed by the routes. The accessed tables are listed below the route name. The number of rows in each table are indicated by the number in brackets behind the table name. The response time is indicated by number in brackets after the route name.

Get sensor system id for user (94 ms)
sensorsystems(9)
users(3)

Get characteristics of sensor system (83 ms)
characteristic_types(9)
characteristics(5)

Get all current sensor system locations of a single type (64 ms)

sensorsystems(9)
locations(50)

Get all events of single sensor system (413 ms)
events (4574)
event_types (5)

Get all measurements today single sensor system (464 ms)
measurements (186211)
measurement_types (7)

Get all measurements single sensor system (8981 ms)
sensorsystems(186211)
locations(7)

Executing the same query used in the route returning all measurements of a single sensor system using the terminal-based front end psql, yields almost immediate results. The interface appears to have a negative effect on the performance.

7.4 Conclusion

The interface's functionality was tested using Postman. The tests were successful, however some deviations occurred in the values for the latitude and longitude of sensor system locations. The functions in which the deviations in latitude and longitude appeared are:

- Current single sensor location
- All single sensor system locations
- All current locations of a single type of sensor system

At this moment it is not yet clear what causes these deviations. Expected is that is has to do with the projection of the location's latitude and longitude on the cartesian RD-coordinates spatial reference system.

Regarding the performance of the implemented interfaces using the Lumen framework, it can be concluded that at this stage of the project the performance is good. However, the more results a request delivers, the larger the difference in response time as compared to executing the same query using the PostgreSQL terminal-based front end psql. Retrieving all measurements for a single sensor system took 8981 ms using the consumer interface. Executing the same query using psql, showed almost instantaneous results. When the number of rows in the database tables increases, the difference in response time between the interface's functions and executing the same queries psql is expected to increase. Improving performance of the database can be done by for instance building indexes for large tables and tweaking the database settings such as the *max_connections, shared_buffers,* and *work_mem*

[42]. The API seems to be affecting the performance in a negative way. Further research has to be done on why this happens and on how to improve API performance.

8. Conclusion

In this chapter the answers to the research questions stated in section 1.3 will be provided, as well as other conclusions drawn from this graduation project. The research question and sub-questions used to answer the research question stated in section 1.3 are:

- *How to develop a suitable data repository for geo-tagged environmental data for the Climate Adaptive City Enschede project?*
    - *What database management system is most suitable for storing geospatial data?*
    - *How to maintain performance with large amounts of data generating sensor nodes?*

In order to provide an answer to these questions, background research was done. This background research included looking at previous graduation projects, a literature research on different classes of databases and several database management systems and their geospatial functionalities, and a state of art research on multiple different database management systems from different database classes.

Regarding the first question, *what database management system is most suitable for storing geospatial data*, the conclusion can be draw that PostgreSQL in combination with the PostgreSQL extension PostGIS is the most suitable database management system for storing geospatial data. PostgreSQL is a very mature, reliable, fast, well documented object-relational database management system, and has a very large supporting community. Furthermore, it is free and open source. Most other database management systems offer enterprise editions next to free version, which could result is not all beneficial aspects and functionality will be available in the free versions. The PostGIS PostgreSQL extension is free and open source as well, and is just as well documented as PostgreSQL. PostGIS offers the most extensive set of geospatial functionalities of all available database management systems, as well as some advanced extras such as GiST indexing (see section 2.3.3 and 2.4.1).

The second question is *how to maintain performance with large amounts of data generating sensor nodes?* In the literature and state of the art review in chapter 2 of this repost, it was pointed out that the database classes NoSQL and NewSQL usually have higher performance. However, this doesn't mean that (Old)SQL systems have low performance. PostgreSQL is a very mature database management system of the SQL class of databases, and has proven to be a very reliable and high performance database management system. The database settings can be tweaked and advance indexing, such as GiST indexing, can be used on large tables to further improve database performance. The test showed some significant difference in response time in using the interface's functions and

executing the same queries using the terminal-based front end psql. Further research has to be done on the improvement of API performance.

Finally, the main research question, *how to develop a suitable data repository for geo-tagged environmental data for the Climate Adaptive City Enschede project,* can be answered. The list of requirements that followed from brainstorming and informal interviews with project's stakeholders, has lead to the database table structure and the functional architecture of the system. The data repository contains three interfaces, an administrator, a consumer, and a producer interface. The interfaces are generic, providing universal functionalities to sensor systems producing data and (visualization) applications consuming data. Some project specific extra functionality was implemented on top of the basic universal functionality. Underlying the generic interfaces, lies the table structure of the database. The table structure was to designed to be universal as well. It is possible for any type of sensor system to produce any type of measurements, store any type of sensor system characteristics, and store any type sensor system events. If in the future a new type of sensor systems is added to the CAC projects, this type with its accompanied measurement, event, and characteristic types simply has be added by the repository administrator. Also, when a already existing sensor system expands with a new type of sensor producing a new type of measurement, the only thing that has to be done is for the administrator to add the new type of measurement to the system.

9. Recommendations

This chapter contains future recommendations for further development of the system. First of all, when in the future more SRB sensor systems are deployed, data should be anonymized on a maximum zoom in of street level clustered nodes. Individual SRB data is only available to the SRB's owners. At this stage of the project anonymization is done by providing only the last month's total amount of buffered rainwater of all SRBs, and providing individual data only to the SRB owner. The functionality to provide data averaged on street level clustered sensor systems should be implemented in the future, as well as possibly the functionality to cluster sensor systems together based on the distances between them and a specified minimum amount of sensor systems per cluster. Also future research has to be conducted on data repository security, as for this stage of the project no serious security measures have been implemented except for the hashing of user passwords. For example using API client authentication. At this stage of the project anyone in possession of the http request hyperlink can make POST and GET requests to one of the repository's interfaces. Finally, research has to be done on further improving database and API performance. The tests showed significant differences in interface response time and the PostgreSQL terminal-based front end psql, executing the same queries. The database performs very well at this stage of the project. The interfaces perform sufficient as well, however retrieving all measurements of a sensor system has a relative high response time. Improving database performance should be done by creating indexes when tables grow to contain a large amount of rows, and tweaking database setting such as the number of connections and available work memory. Expected is that in the current implementation of the system, the bottleneck will we the implementation of the interfaces. Further research has to be done on how to improve API performance.

References

[1] "Klimaatactieve steden - Natuur en Milieu Overijssel," [Online]. Available: https://www.natuurenmilieuoverijssel.nl/wat-wij-bieden/klimaatactieve-steden/. [Accessed 01 Apr 2018].

[2] "KlimaatActieve Stad (KAS) - Vechtstromen," [Online]. Available: https://www.vechtstromen.nl/over/klimaat/klimaatactieve/. [Accessed 01 Apr 2018].

[3] "Water in Enschede: feiten, cijfers en trends," Gemeente Enschede, 2012.

[4] "Infographic over de Enschedese waterhuishouding," Gemeente Enschede, [Online]. Available: https://www.enschede.nl/file/5278. [Accessed Apr 2018].

[5] "Wadi Het Bijvank," Gemeente Enschede, [Online]. Available: https://www.enschede.nl/duurzame-daad/wadi-het-bijvank. [Accessed Apr 2018].

[6] E. Sharifi and S. Lehmann, "Correlation analysis of surface temperature of rooftops, streerscapes and urban heat island effect," *Journal of Urban and Environmental engineering,* pp. 3-11, 2015.

[7] F. Rindt, "Developing A Smart Rainwater Buffering System For The Citizens Of Enschede," *Bachelor thesis Creative Technology,* July 2017.

[8] G. Steeghs, "Developing A Smart Rainwater Buffering System For The Municipality Of Enschede," *Bachelor thesis Creative Technology,* July 2017.

[9] Y. M. E. Latzer, "Air Temperature Visualization Of Public Spaces In Enschede," *Bachelor thesis Creative Technology,* March 2018.

[10] T. Onderwater, "Developing A Sensor Network For Real Time Temperature Monitoring In Enschede," *Bachelor thesis Creative Technology,* February 2018.

[11] "FAQ: Databases and models, Django documentation," [Online]. Available: https://docs.djangoproject.com/en/2.0/faq/models/#does-django-support-nosql-databases. [Accessed Apr 2018].

[12] J. Jin, J. Gubbi, S. Marusic and M. Palaniswami, "An Information Framework for Creating a Smart City Through Internet of Things," *IEEE Internet of Things Journal,* no. 1, pp. 112-121, 2014.

[13] H. Fatima and K. Wasnik, "Comparison of SQL, NoSQL and NewSQL databases for internet of things," *IEEE Bombay Section Symposium (IBSS),* pp. 1-6, 2016.

[14] S. Duggirala, "NewSQL Databases and Scalable In-Memory Analytics," in *Advances in Computers*, Elsevier, 2018, pp. 49-76.

[15] R. Sánchez-de-Madariaga, A. Muñoz, R. Lozano-Rubí, P. Serrano-Balazote, A. L. Castro, O. Moreno and M. Pascual, "Examining database persistence of ISO/EN 13606 standardized electronic health record extracts: relational vs. NoSQL approaches," *BMC Medical Informatics and Decision Making*, pp. 17-123, 2017.

[16] S. D. Kuznetsov and A. V. Poskonin, "NoSQL data management systems," *Programming and Computer Software,* no. 40, pp. 323-332, 2014.

[17] K. Kaur and M. Sachdeva, "Performance evaluation of NewSQL databases," in *International Conference on Inventive Systems and Control (ICISC)*, Coimbatore, 2017.

[18] A. Pavlo and M. Aslett, "What's Really New with NewSQL?," *ACM SIGMOD Record archive,* no. 45, pp. 45-55, 2016.

[19] K. Grolinger and W. A. Higashino, "Data management in cloud environments: NoSQL and NewSQL data stores," *Journal of Cloud Computing*, pp. 2-22, 2013.

[20] S. Luan, H. Cai, F. Bu and L. Jiang, "A Four-Layer Flexible Spatial Data Framework towards IoT Application," in *IEEE 12th International Conference on e-Business Engineering*, Beijing, 2015.

[21] N. Pant, M. Fouladgar, R. Elmasri and K. Jitkajornwanich, "A Survey of Spatio-Temporal Database Research," *Intelligent Information and Database Systems*, pp. 115-126, 2018.

[22] A. Xavier and A. A. Arivazhagan, "Investigation and Visualization of Query Determine Spatial Pattern in GIS," *Indonesian Journal of Electrical Engineering and Computer Science*, no. 9, pp. 552-554, 2018.

[23] J. S. van der Veen, B. van der Waaij and R. J. Meijer, "Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, Honolulu, 2012.

[24] S. Steiniger and A. J. S. Hunter, "Free and Open Source GIS Software for Building a Spatial Data Infrastructure," in *Geospatial Free and Open Source Software in the 21st Century*, Springer, Berlin, Heidelberg, 2012, pp. 247-261.

[25] "Simple Feature Access - Part 1: Common Architecture | OGC," [Online]. Available: http://www.opengeospatial.org/standards/sfa. [Accessed 05 2018].

[26] "PostGIS - Spatial and Geographic Objects for PostgreSQL," [Online]. Available: https://postgis.net/. [Accessed Apr 2018].

[27] "MySQL," [Online]. Available: https://www.mysql.com/. [Accessed Apr 2018].

[28] "MongoDB for GIANT Ideas," [Online]. Available: https://www.mongodb.com/. [Accessed Apr 2018].

[29] "In Memory Database, VoltDB," [Online]. Available: https://www.voltdb.com/. [Accessed Apr 2018].

[30] "RWJF - Qualitative Research Guidelines | Interviewing | Interviewing," RWJF, 2008. [Online]. Available: http://www.qualres.org/HomeInte-3595.html. [Accessed Juni 2018].

[31] A. Mader and W. Eggink, "A Design Process For Creative Technology," in *INTERNATIONAL CONFERENCE ON ENGINEERING AND PRODUCT DESIGN EDUCATION*, Enschede, 2014.

[32] H. Sharp, A. Finkelstein and G. Galal, "Stakeholder identification in the requirements engineering process," in *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99*, Florence, 1999.

[33] J. M. Bryson, "What to do when stakeholders matter: stakeholder identification and analysis techniques," *Public Management Review*, no. 6, pp. 21-53, 2007.

[34] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, Wiley Publishing, 1998.

[35] D. Benyon and C. Macaulay, "Scenarios and the HCI-SE design problem," *Interacting with Computers*, no. 14, pp. 397-405, 2002.

[36] C. S. Jensen, T. B. Pedersen and C. Thomsen, Multidimensional Databases and DataWarehousing, Morgan & Claypool, 2010.

[37] "PostgreSQL: Documentation: 10: Part I. Tutorial," PostgreSQL, [Online]. Available: https://www.postgresql.org/docs/10/static/tutorial.html. [Accessed May 2018].

[38] "Rijksdriehoeksstelsel," Kadaster, [Online]. Available: https://www.kadaster.nl/rijksdriehoeksstelsel. [Accessed 7 2018].

[39] "Lumen - PHP Micro-Framework By Laravel," [Online]. Available: https://lumen.laravel.com/. [Accessed 7 2018].

[40] "Composer," [Online]. Available: https://getcomposer.org/. [Accessed 7 2018].

[41] "Postman | API Developer Environment," [Online]. Available: https://www.getpostman.com/. [Accessed 07 2018].

[42] "Tuning Your PostgreSQL Server - PostgreSQL wiki," [Online]. Available: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server. [Accessed 07 2018].

Appendix

A. PostGIS Function Support Matrix

Below is an alphabetical listing of spatial specific functions in PostGIS and the kinds of spatial types they work with or OGC/SQL compliance they try to conform to.

- A ✓ means the function works with the type or subtype natively.
- A  means it works but with a transform cast built-in using cast to geometry, transform to a "best srid" spatial ref and then cast back. Results may not be as expected for large areas or areas at poles and may accumulate floating point junk.
- A  means the function works with the type because of a auto-cast to another such as to box3d rather than direct type support.
- A  means the function only available if PostGIS compiled with SFCGAL support.
- A  means the function support is provided by SFCGAL if PostGIS compiled with SFCGAL support, otherwise GEOS/built-in support.
- geom - Basic 2D geometry support (x,y).
- geog - Basic 2D geography support (x,y).
- 2.5D - basic 2D geometries in 3 D/4D space (has Z or M coord).
- PS - Polyhedral surfaces
- T - Triangles and Triangulated Irregular Network surfaces (TIN)

Function	***geom***	***geog***	***2.5D***	***Curves***	***SQL MM***	***PS***	***T***
Box2D	✓			✓		✓	✓
Box3D	✓		✓	✓		✓	✓
Find_SRID							
GeometryType	✓		✓	✓		✓	✓
ST_3DArea							
ST_3DClosestPoint	✓		✓			✓	
ST_3DDFullyWithin	✓		✓			✓	
ST_3DDWithin	✓		✓		✓	✓	
ST_3DDifference							
ST_3DDistance	✓						
ST_3DExtent	✓		✓	✓		✓	✓
ST_3DIntersection							

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_3DIntersects	✓		✓		✓	✓	✓
ST_3DLength	✓		✓				
ST_3DLongestLine	✓		✓			✓	
ST_3DMakeBox	✓		✓				
ST_3DMaxDistance	✓		✓			✓	
ST_3DPerimeter	✓		✓				
ST_3DShortestLine	✓		✓			✓	
ST_3DUnion	✓		✓			✓	✓
ST_Accum	✓		✓	✓		✓	✓
ST_AddMeasure	✓		✓				
ST_AddPoint	✓		✓				
ST_Affine	✓		✓	✓		✓	✓
ST_ApproximateMedialAxis	✓		✓			✓	✓
ST_Area	✓	✓			✓	✓	
ST_AsBinary	✓	✓	✓	✓	✓	✓	✓
ST_AsEWKB	✓		✓	✓		✓	✓
ST_AsEWKT	✓	✓	✓	✓		✓	✓
ST_AsEncodedPolyline	✓						
ST_AsGML	✓	✓	✓			✓	✓
ST_AsGeoJSON	✓	✓	✓				
ST_AsGeobuf							
ST_AsHEXEWKB	✓		✓	✓			
ST_AsKML	✓	✓	✓				
ST_AsLatLonText	✓						
ST_AsMVT							
ST_AsMVTGeom	✓						
ST_AsSVG	✓	✓					
ST_AsTWKB	✓						
ST_AsText	✓	✓		✓	✓		
ST_AsX3D	✓		✓			✓	✓
ST_Azimuth	✓	✓					
ST_BdMPolyFromText	✓						

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_BdPolyFromText	✓						
ST_Boundary	✓		✓		✓		✓
ST_BoundingDiagonal	✓		✓				
ST_Box2dFromGeoHash							
ST_Buffer	✓				✓		
ST_BuildArea	✓						
ST_CPAWithin	✓		✓				
ST_Centroid	✓	✓			✓		
ST_ClipByBox2D	✓						
ST_ClosestPoint	✓						
ST_ClosestPointOfApproach	✓		✓				
ST_ClusterDBSCAN	✓						
ST_ClusterIntersecting	✓						
ST_ClusterKMeans	✓						
ST_ClusterWithin	✓						
ST_Collect	✓		✓	✓			
ST_CollectionExtract	✓						
ST_CollectionHomogenize	✓						
ST_ConcaveHull	✓						
ST_Contains	✓				✓		
ST_ContainsProperly	✓						
ST_ConvexHull	✓		✓		✓		
ST_CoordDim	✓		✓	✓	✓	✓	✓
ST_CoveredBy	✓	✓					
ST_Covers	✓	✓					
ST_Crosses	✓				✓		
ST_CurveToLine	✓		✓	✓	✓		
ST_DFullyWithin	✓						
ST_DWithin	✓	✓					
ST_DelaunayTriangles	✓		✓				✓
ST_Difference	✓		✓		✓		
ST_Dimension	✓				✓	✓	✓
ST_Disjoint	✓				✓		

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_Distance	✓	✓		✓			
ST_DistanceCPA	✓		✓				
ST_DistanceSphere	✓						
ST_DistanceSpheroid	✓						
ST_Dump	✓		✓	✓		✓	✓
ST_DumpPoints	✓		✓	✓		✓	✓
ST_DumpRings	✓		✓				
ST_EndPoint	✓		✓	✓	✓		
ST_Envelope	✓				✓		
ST_Equals	✓				✓		
ST_EstimatedExtent				✓			
ST_Expand	✓					✓	✓
ST_Extent	✓					✓	✓
ST_ExteriorRing	✓		✓		✓		
ST_Extrude							
ST_FlipCoordinates	✓		✓	✓		✓	✓
ST_Force2D	✓		✓	✓		✓	
ST_ForceCurve	✓		✓	✓			
ST_ForceLHR							
ST_ForcePolygonCCW	✓		✓				
ST_ForcePolygonCW	✓		✓				
ST_ForceRHR	✓		✓			✓	
ST_ForceSFS	✓		✓	✓		✓	✓
ST_Force3D	✓		✓	✓		✓	
ST_Force3DM	✓			✓			
ST_Force3DZ	✓		✓	✓		✓	
ST_Force4D	✓		✓	✓			
ST_ForceCollection	✓		✓	✓		✓	
ST_FrechetDistance	✓						
ST_GMLToSQL	✓				✓	✓	
ST_GeneratePoints	✓						
ST_GeoHash	✓			✓			
ST_GeogFromText		✓					

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_GeogFromWKB		✓		✓			
ST_GeographyFromText		✓					
ST_GeomCollFromText	✓				✓		
ST_GeomFromEWKB	✓		✓	✓		✓	✓
ST_GeomFromEWKT	✓		✓	✓		✓	✓
ST_GeomFromGML	✓		✓			✓	✓
ST_GeomFromGeoHash	✓						
ST_GeomFromGeoJSON	✓		✓				
ST_GeomFromKML	✓		✓				
ST_GeomFromTWKB	✓						
ST_GeomFromText	✓			✓	✓		
ST_GeomFromWKB	✓			✓	✓		
ST_GeometricMedian	✓		✓				
ST_GeometryFromText	✓				✓		
ST_GeometryN	✓		✓	✓	✓	✓	✓
ST_GeometryType	✓		✓		✓	✓	
|>>	✓						
<<|	✓						
~	✓						
@	✓						
=	✓	✓		✓		✓	
<<	✓						
|&>	✓						
&<|	✓			✓		✓	
&<	✓						
&>	✓						
>>	✓						
~=	✓					✓	
ST_HasArc	✓		✓	✓			
ST_HausdorffDistance	✓						
ST_InteriorRingN	✓		✓		✓		
ST_InterpolatePoint	✓		✓				
ST_Intersection	✓						

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_Intersects	✓	✓					
ST_IsClosed	✓		✓	✓	✓	✓	
ST_IsCollection	✓		✓	✓			
ST_IsEmpty	✓			✓	✓		
ST_IsPlanar							
ST_IsPolygonCCW	✓		✓				
ST_IsPolygonCW	✓		✓				
ST_IsRing	✓				✓		
ST_IsSimple	✓		✓		✓		
ST_IsSolid							
ST_IsValid	✓				✓		
ST_IsValidDetail	✓						
ST_IsValidReason	✓						
ST_IsValidTrajectory	✓		✓				
ST_Length	✓	✓					
ST_Length2D	✓						
ST_Length2D_Spheroid	✓						
ST_LengthSpheroid	✓		✓				
ST_LineCrossingDirection	✓						
ST_LineFromEncodedPolyline	✓						
ST_LineFromMultiPoint	✓		✓				
ST_LineFromText	✓				✓		
ST_LineFromWKB	✓				✓		
ST_LineInterpolatePoint	✓		✓				
ST_LineLocatePoint	✓						
ST_LineMerge	✓						
ST_LineSubstring	✓		✓				
ST_LineToCurve	✓		✓	✓			
ST_LinestringFromWKB	✓				✓		
ST_LocateAlong	✓						
ST_LocateBetween	✓						
ST_LocateBetweenElevations	✓		✓				

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_LongestLine	✓						
ST_M	✓		✓		✓		
ST_MLineFromText	✓				✓		
ST_MPointFromText	✓				✓		
ST_MPolyFromText	✓				✓		
ST_MakeBox2D	✓						
ST_MakeEnvelope	✓						
ST_MakeLine	✓		✓				
ST_MakePoint	✓		✓				
ST_MakePointM	✓						
ST_MakePolygon	✓		✓				
ST_MakeSolid							
ST_MakeValid	✓		✓				
ST_MaxDistance	✓						
ST_MemSize	✓		✓	✓		✓	✓
ST_MemUnion	✓		✓				
ST_MinimumBoundingCircle	✓						
ST_MinimumBoundingRadius	✓						
ST_MinimumClearance	✓						
ST_MinimumClearanceLine	✓						
ST_MinkowskiSum							
ST_Multi	✓						
ST_NDims	✓		✓				
ST_NPoints	✓		✓	✓		✓	
ST_NRings	✓		✓	✓			
ST_Node	✓		✓				
ST_Normalize	✓						
ST_NumGeometries	✓		✓		✓	✓	✓
ST_NumInteriorRing	✓						
ST_NumInteriorRings	✓				✓		
ST_NumPatches	✓		✓		✓	✓	
ST_NumPoints	✓				✓		
ST_OffsetCurve	✓						

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_OrderingEquals	✓				✓		
ST_Orientation							
ST_Overlaps	✓				✓		
ST_PatchN	✓		✓		✓	✓	
ST_Perimeter	✓	✓			✓		
ST_Perimeter2D	✓						
ST_Point	✓				✓		
ST_PointFromGeoHash							
ST_PointFromText	✓				✓		
ST_PointFromWKB	✓		✓	✓	✓		
ST_PointN	✓		✓	✓	✓		
ST_PointOnSurface	✓		✓		✓		
ST_PointInsideCircle	✓						
ST_Points	✓		✓	✓			
ST_Polygon	✓		✓		✓		
ST_PolygonFromText	✓				✓		
ST_Polygonize	✓						
ST_Project		✓					
ST_Relate	✓				✓		
ST_RelateMatch							
ST_RemovePoint	✓		✓				
ST_RemoveRepeatedPoints	✓		✓			✓	
ST_Reverse	✓		✓			✓	
ST_Rotate	✓		✓	✓		✓	✓
ST_RotateX	✓		✓			✓	✓
ST_RotateY	✓		✓			✓	✓
ST_RotateZ	✓		✓	✓		✓	✓
ST_SRID	✓			✓	✓		
ST_Scale	✓		✓	✓		✓	✓
ST_Segmentize	✓	✓					
ST_SetEffectiveArea	✓						
ST_SetPoint	✓		✓				
ST_SetSRID	✓			✓			

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_SharedPaths	✓						
ST_ShiftLongitude	✓		✓			✓	✓
ST_ShortestLine	✓						
ST_Simplify	✓						
ST_SimplifyPreserveTopology	✓						
ST_SimplifyVW	✓						
ST_Snap	✓						
ST_SnapToGrid	✓		✓				
ST_Split	✓						
ST_StartPoint	✓		✓	✓	✓		
ST_StraightSkeleton							
ST_Subdivide	✓						
ST_Summary	✓	✓		✓		✓	✓
ST_SwapOrdinates	✓		✓	✓		✓	✓
ST_SymDifference	✓		✓		✓		
ST_Tesselate							
ST_Touches	✓				✓		
ST_TransScale	✓		✓	✓			
ST_Transform	✓			✓	✓	✓	
ST_Translate	✓		✓	✓			
ST_UnaryUnion	✓		✓				
ST_Union	✓				✓		
ST_Volume							
ST_VoronoiLines	✓						
ST_VoronoiPolygons	✓						
ST_WKBToSQL	✓				✓		
ST_WKTToSQL	✓				✓		
ST_Within	✓				✓		
ST_WrapX	✓		✓				
ST_X	✓		✓		✓		
ST_XMax			✓	✓			
ST_XMin			✓	✓			
ST_Y	✓		✓		✓		

Function	*geom*	*geog*	*2.5D*	*Curves*	*SQL MM*	*PS*	*T*
ST_YMax	✓		✓	✓			
ST_YMin	✓		✓	✓			
ST_Z	✓		✓		✓		
ST_ZMax	✓		✓	✓			
ST_ZMin	✓		✓	✓			
ST_Zmflag	✓		✓	✓			
~(box2df,box2df)	✓			✓		✓	
~(box2df,geometry)	✓			✓		✓	
~(geometry,box2df)	✓			✓		✓	
<#>	✓						
<<#>>	✓						
<<->>	✓						
|=|	✓						
<->	✓	✓					
&&	✓	✓		✓		✓	
&&&	✓		✓	✓		✓	✓
@(box2df,box2df)	✓			✓		✓	
@(box2df,geometry)	✓			✓		✓	
@(geometry,box2df)	✓			✓		✓	
&&(box2df,box2df)	✓			✓		✓	
&&(box2df,geometry)	✓			✓		✓	
&&(geometry,box2df)	✓			✓		✓	
&&&(geometry,gidx)	✓		✓	✓		✓	✓
&&&(gidx,geometry)	✓		✓	✓		✓	✓
&&&(gidx,gidx)			✓	✓		✓	✓
postgis.backend							
postgis.enable_outdb_rasters							
postgis.gdal_datapath							
postgis.gdal_enabled_drivers							
postgis_sfcgal_versio							

B. web.php, routes specification

```php
<?php

/*
|--------------------------------------------------------------------------
| Application Routes
|--------------------------------------------------------------------------
|
| Here is where you can register all of the routes for an application.
| It is a breeze. Simply tell Lumen the URIs it should respond to
| and give it the Closure to call when that URI is requested.
|
*/

$router->get('/', function () use ($router) {
    return $router->app->version();
});

// ADMIN ROUTES
$router->group(['prefix' => 'admin'], function () use ($router) {

        $router->post('newsensorsystemtype/{type}', ['uses' => 'AdminController@newSensorsystemType']);
        $router->post('newcharacteristictype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newCharacteristicType']);
        $router->post('neweventtype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newEventType']);
        $router->post('newmeasurementtype/{type}/{sensorsystem_type_id}', ['uses' => 'AdminController@newMeasurementType']);
});

// PRODUCER ROUTES
$router->group(['prefix' => 'prod'], function () use ($router) {

        // Register sensorsystem with user
        $router->post('registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{username}/{password}', ['uses' => 'ProducerController@registerSens
orsystemWithUser']);

        // Register new sensor
        $router->post('registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}', ['uses' => 'ProducerController@registerSensorsystem']);

        // register new sensor with location precision
        $router->post('registersensorsystem/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{precision}', ['uses' => 'ProducerController@registerSensorsystemWi
thLocationPrecision']);



        // Register characteristic
        $router->post('registercharacteristic/{sensorsystem_id}/{sensorsystem_type_id}/{characteristic_type_id}/{characteristic_value}', ['uses' => 'ProducerControl
ler@registerSensorsystemCharacteristic']);

        // Update sensor location
        $router->post('updatelocation/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}', ['uses' => 'ProducerController@updateSensorsystemLocation']);

        // Update sensor location with precision
        $router->post('updatelocation/{sensorsystem_id}/{sensorsystem_type_id}/{lat}/{lon}/{precision}', ['uses' => 'ProducerController@updateSensorsystemLocationWi
thLocationPrecision']);

        // Register event
        $router->post('registerevent/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{value}/{ts}', ['uses' => 'ProducerController@newEvent']);

        // Register event with precipitation
        $router->post('registerevent/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{value}/{precipitation}/{ts}', ['uses' => 'ProducerController@newEvent
WithPrecipitation']);

        // Insert measurement
        $router->post('registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id}/{value}/{ts}', ['uses' => 'ProducerController@oneNewMeasur
ement']);
        $router->post('registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{ts}', ['uses' =
> 'ProducerController@twoNewMeasurements']);
        $router->post('registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{measurement_typ
e_id3}/{value3}/{ts}', ['uses' => 'ProducerController@threeNewMeasurements']);
        $router->post('registermeasurement/{sensorsystem_id}/{sensorsystem_type_id}/{measurement_type_id1}/{value1}/{measurement_type_id2}/{value2}/{measurement_typ
e_id3}/{value3}/{measurement_type_id4}/{value4}/{ts}', ['uses' => 'ProducerController@fourNewMeasurements']);

        $router->delete('deleteplanneddischarges/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ProducerController@deletePlannedDischarges']);
});

// CONSUMER ROUTES
//$router->group(['prefix' => 'cons'], function () use ($router) {
$router->group(['prefix' => 'cons'], function () use ($router) {

        // Get sensorsystem_id from user & password
        $router->get('id/{username}/{password}', ['uses' => 'ConsumerController@getUserSensorsystemId']);

        // Get current location of all sensor nodes of specified sensor node type
        $router->get('locations/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getAllSensorsystemsCurrentLocations']);

        // Get current location of single sensor node
        $router->get('location/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getSingleSensorsystemCurrentLocation']);

        // Get all locations of single sensor node
        $router->get('locations/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getSingleSensorsystemAllLocations']);

        // Get characteristics of single sensor node
        $router->get('characteristics/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getSingleSensorsystemCharacteristics']);

        // Get all measurements of single sensor node
        $router->get('measurements/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getAllMeasurements']);

        // Get all measurements of current day
        $router->get('daymeasurements/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getDayMeasurements']);


        // Get all measurements in specified period for single sensorystem
        $router->get('datemeasurements/{sensorsystem_id}/{sensorsystem_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getMeasurementsPeriod']);

        // Get all measurements in specified period for all sensor systems of single type
        $router->get('datemeasurements/{sensorsystem_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getAllMeasurementsPeriod']);


        // Get buffered amount of single sensor
        $router->get('buffered/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getBuffered']);

        // Get total buffered amount
        $router->get('buffered/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getBufferedTotal']);

        // Get events
        $router->get('events/{sensorsystem_id}/{sensorsystem_type_id}', ['uses' => 'ConsumerController@getSingleSensorsystemEvents']);

        // Get events from date till date
        $router->get('events/{sensorsystem_id}/{sensorsystem_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getEventsPeriod']);

        // Get specific event time period
        $router->get('events/{sensorsystem_id}/{sensorsystem_type_id}/{event_type_id}/{date1}/{date2}', ['uses' => 'ConsumerController@getSpecificEventsPeriod']);
});
```

C. AdminController.php

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class AdminController extends Controller
{

	public function newSensorsystemType($type){
		DB::insert("
			INSERT INTO sensorsystem_types (type)
			VALUES (?)", [$type]);
	}

	public function newCharacteristicType($type, $sensorsystem_type_id){
		DB::insert("
			INSERT INTO characteristic_types (type, sensorsystem_type_id)
			VALUES (?,?)", [$type, $sensorsystem_type_id]);
	}

	public function newEventType($type, $sensorsystem_type_id){
		DB::insert("
			INSERT INTO event_types (type, sensorsystem_type_id)
			VALUES (?,?)", [$type, $sensorsystem_type_id]);
	}

	public function newMeasurementType($type, $sensorsystem_type_id){
		DB::insert("
			INSERT INTO measurement_types (sensorsystem_type_id, type)
			VALUES (?,?)", [$sensorsystem_type_id, $type]);
	}
}
```

D. ProducerController.php

```php
<?php

namespace App\Http\Controllers;

//use App\NewSensor;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;
use Illuminate\Http\Response;

class ProducerController extends Controller
{

// VALIDATION FUNCTIONS

        // Check if sensorsystem id exists
        public function sensorsystemIdExists($id){
                return DB::select("SELECT count(id) FROM sensorsystems WHERE id = ?", [$id])[0]->count == 1;
        }

        // Check if sensorsystem type id exists
        public function sensorsystemTypeExists($type_id){
                return DB::select("SELECT count(id) FROM sensorsystem_types WHERE id = ?", [$type_id])[0]->count == 1;
        }

        // Validate sensorsystem type with sensorsystem id
        public function validateSensorsystem($id, $type_id){
                if(!$this->sensorsystemIdExists($id)) return -1;
                if(!$this->sensorsystemTypeExists($type_id)) return -2;
                if(DB::select("SELECT s.sensorsystem_type_id
                                FROM sensorsystems s
                                WHERE s.id = ?", [$id])[0]->sensorsystem_type_id != $type_id) return -3;
                return 1;
        }

        // Check if user exists
        public function userExists($username){
        return DB::select("SELECT EXISTS (SELECT true FROM users WHERE username = ?)", [$username])[0]->exists;
        }

        // Check characteristic
        public function characteristicExists($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id){
                if(DB::select("SELECT count(id) FROM characteristic_types WHERE id = ?", [$characteristic_type_id])[0]->count != 1){
                        return -1;
                } else if(DB::select("SELECT sensorsystem_type_id FROM characteristic_types WHERE id = ?", [$characteristic_type_id])[0]->sensorsystem_type_id != $s
ensorsystem_type_id){
                        return -2;
                } else if(DB::select("SELECT characteristic_type_id FROM characteristics WHERE sensorsystem_id = ?", [$sensorsystem_id]) != NULL){
                        return -3;
                } else {
                        return 1;
                }
        }

        // Check events
        public function eventTypeExists($sensorsystem_type_id, $event_type_id){
                if(DB::select("SELECT count(id) FROM event_types WHERE id = ?", [$event_type_id])[0]->count != 1){
                        return -1;
                } else if(DB::select("SELECT sensorsystem_type_id FROM event_types WHERE id = ?", [$event_type_id])[0]->sensorsystem_type_id != $sensorsystem_type_i
d){
                        return -2;
                } else {
                        return 1;
                }
        }

        // Validate measurement types
        public function validateMeasurementType($sensorsystem_type_id, $measurement_type_id){

                return ($sensorsystem_type_id == DB::select("SELECT sensorsystem_type_id FROM measurement_types WHERE id = ?",[$measurement_type_id])[0]->sensorsyst
em_type_id);
        }
// SENSOR SYSTEM REGISTRATION FUNCTIONS

        // Register new sensorsystem
        public function registerSensorsystem($sensorsystem_id, $sensorsystem_type_id, $lat, $lon)
        {
                if($this->sensorsystemIdExists($sensorsystem_id)){
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." already exists", 400));
                } else if($this->sensorsystemTypeExists($sensorsystem_type_id)){
                        DB::insert("INSERT INTO sensorsystems (id, sensorsystem_type_id) VALUES (?, ?)", [$sensorsystem_id, $sensorsystem_type_id]);
                        $lon = str_replace(',','.', $lon);
                        $lat = str_replace(',','.', $lat);
                        $point = "POINT(" .$lon ." " .$lat .")";
                        DB::insert("INSERT INTO locations (sensorsystem_id, ts, location) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992))",[$sensorsy
stem_id, $point]);
                        return (new Response("Sensor system registration successful!", 200));
                } else {
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist", 400));
                }
        }

        // Register new sensorsystem with location precision
        public function registerSensorsystemWithLocationPrecision($sensorsystem_id, $sensorsystem_type_id, $lat, $lon, $precision)
        {
                if($this->sensorsystemIdExists($sensorsystem_id)){
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." already exists", 400));
                } else if($this->sensorsystemTypeExists($sensorsystem_type_id)){
                DB::insert("INSERT INTO sensorsystems (id, sensorsystem_type_id) VALUES (?, ?)", [$sensorsystem_id, $sensorsystem_type_id]);
                $lon = str_replace(',','.', $lon);
                $lat = str_replace(',','.', $lat);
                $point = "POINT(" .$lon ." " .$lat .")";
                $precision = str_replace(',','.', $precision);
                DB::insert("INSERT INTO locations (sensorsystem_id, ts, location, precision) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992), ?)",[$se
nsorsystem_id, $point, $precision]);
                        return (new Response("Sensor system registration successful!", 200));
                } else {
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist", 400));
```

```
                }
        }

        // Register new sensorsystem with username and password
        public function registerSensorsystemWithUser($sensorsystem_id, $sensorsystem_type_id, $lat, $lon, $username, $password)
        {
                if($this->sensorsystemIdExists($sensorsystem_id)){
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." already exists", 400));
                } else if($this->sensorsystemTypeExists($sensorsystem_type_id) && !$this->userExists($username)){
                        DB::insert("INSERT INTO sensorsystems (id, sensorsystem_type_id) VALUES (?, ?)", [$sensorsystem_id, $sensorsystem_type_id]);
                        $lon = str_replace(',','.', $lon);
                        $lat = str_replace(',','.', $lat);
                        $point = "POINT(" .$lon ." " .$lat .")";
                        DB::insert("INSERT INTO locations (sensorsystem_id, ts, location) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992))",[$sensorsystem_id, $point]);
                        $hashedPassword = hash('sha256', $password);
                        DB::insert("INSERT INTO users (username, password, sensorsystem_id) VALUES (?, ? ,?)", [$username, $hashedPassword, $sensorsystem_id]);
        return (new Response("Sensor system registration successful!", 200));
                } else {
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist", 400));
                }
        }


//CHARACTERISTICS

        //Register sensor system characteristics
        public function registerSensorsystemCharacteristic($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id, $characteristic_value)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $characteristicValidationCode = $this->characteristicExists($sensorsystem_id, $sensorsystem_type_id, $characteristic_type_id);
                switch($characteristicValidationCode){
                case 1:
                        $characteristic_value = str_replace(',','.', $characteristic_value);
                        DB::insert("
                                INSERT INTO characteristics (sensorsystem_id, characteristic_type_id, value)
                                VALUES (?, ?, ?)", [$sensorsystem_id, $characteristic_type_id, $characteristic_value]);
                        return (new Response("sensorsystem characteristic registration successful!", 200));
                        break;
                case -1:
                        return (new Response("Characteristic type doesn't exist",400));
                        break;
                case -2:
                        return (new Response("Characteristic type not registered for sensor system type",400));
                case -3:
                        return (new Response("Characteristic already registered",400));
                        break;
                }
        }


// LOCATIONS

        // Update existing sensor location
        public function updateSensorsystemLocation($sensorsystem_id, $sensorsystem_type_id, $lat, $lon)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

        $lon = str_replace(',','.', $lon);
        $lat = str_replace(',','.', $lat);
        $point = "POINT(" .$lon . " " .$lat .")";
        DB::insert("INSERT INTO locations (sensorsystem_id, ts, location) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992))",[$sensorsystem_id,$point])
;
        return (new Response("Sensor system location update successful",200));
        }

        // Update existing sensor location with precision
        public function updateSensorsystemLocationWithLocationPrecision($sensorsystem_id, $sensorsystem_type_id, $lat, $lon, $precision)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

        $lon = str_replace(',','.', $lon);
        $lat = str_replace(',','.', $lat);
        $point = "POINT(" .$lon . " " .$lat .")";
        $precision = str_replace(',','.', $precision);
```

```
        DB::insert("INSERT INTO locations (sensorsystem_id, ts, location, precision) VALUES (?, now(), ST_Transform(ST_GeomFromText(?,4326),28992), ?)",[$sensorsystem_id,$point,$precision]);

        return (new Response("Sensor system location update successful",200));
        }


//EVENTS

        //
        public function newEvent($sensorsystem_id, $sensorsystem_type_id, $event_type_id, $value, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $eventValidationCode = $this->eventTypeExists($sensorsystem_type_id, $event_type_id);
                switch ($eventValidationCode){
                case -1:
                        return (new Response("Event type doesn't exist",400));
                        break;
                case -2:
                        return (new Response("Event type not registered for the sensor system type",400));
                        break;
                case 1:
                        $value = str_replace(',','.', $value);
                        DB::insert("INSERT INTO events (sensorsystem_id, event_type_id, value, ts) VALUES (?, ?, ?, ?)",[$sensorsystem_id, $event_type_id, $value, $ts]);
                        return (new Response("Event registration successful!",200));
                }
        }

        public function newEventWithPrecipitation($sensorsystem_id, $sensorsystem_type_id, $event_type_id, $value, $precipitation, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $eventValidationCode = $this->eventTypeExists($sensorsystem_type_id, $event_type_id);
                switch ($eventValidationCode){
                case -1:
                        return (new Response("Event type doesn't exist",400));
                        break;
                case -2:
                        return (new Response("Event type not registered for the sensor system type",400));
                        break;
                case 1:
                        $value = str_replace(',','.', $value);
                        $precipitation = str_replace(',','.', $precipitation);
                        DB::insert("INSERT INTO events (sensorsystem_id, event_type_id, value, ts, precipitation) VALUES (?, ?, ?, ?, ?)", [$sensorsystem_id, $event_type_id, $value, $ts, $precipitation]);
                        return (new Response("Event registration successful!",200));
                }
        }


// MEASUREMENTS

        // Register 1 measurement type
        public function oneNewMeasurement($sensorsystem_id, $sensorsystem_type_id, $measurement_type_id, $value, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                if ($this->validateMeasurementType($sensorsystem_type_id, $measurement_type_id)){
                $value = str_replace(',','.', $value);
                DB::insert("INSERT INTO measurements(sensorsystem_id, measurement_type_id, value, ts) VALUES (?, ?, ?, ?)",[$sensorsystem_id, $measurement_type_id, $value, $ts]);
                return (new Response("Measurement registration successful!", 200));
                } else return (new Response("Measurement type not registered with sensor system type",400));
        }

        // Register 2 measurement types
        public function twoNewMeasurements($sensorsystem_id, $sensorsystem_type_id, $measurement_type_id1, $value1, $measurement_type_id2, $value2, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
```

```
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $requested_types =  array((int)$measurement_type_id1, (int)$measurement_type_id2);
                foreach ($requested_types as $measurement_type){
                        if(!$this->validateMeasurementType($sensorsystem_type_id, $measurement_type)){
                                return (new Response("Measurement type " .$measurement_type ." not registered with sensor system type " .$sensorsystem_type_id,400))
;
                        }
                }
                $value1 = str_replace(',','.', $value1);
                $value2 = str_replace(',','.', $value2);

                DB::insert("INSERT INTO measurements(sensorsystem_id, measurement_type_id, value, ts) VALUES (?, ?, ?, ?), (?, ?, ?, ?)",[$sensorsystem_id, $measure
ment_type_id1, $value1, $ts, $sensorsystem_id, $measurement_type_id2, $value2, $ts]);

                return(new Response("Measurements registration successful!",200));

        }


        public function threeNewMeasurements($sensorsystem_id, $sensorsystem_type_id, $measurement_type_id1, $value1, $measurement_type_id2, $value2, $measurement_t
ype_id3, $value3, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $requested_types =  array((int)$measurement_type_id1, (int)$measurement_type_id2, (int)$measurement_type_id3);
                foreach ($requested_types as $measurement_type){
                        if(!$this->validateMeasurementType($sensorsystem_type_id, $measurement_type)){
                                return (new Response("Measurement type " .$measurement_type ." not registered with sensor system type " .$sensorsystem_type_id,400))
;
                        }
                }

                $value1 = str_replace(',','.', $value1);
                $value2 = str_replace(',','.', $value2);
                $value3 = str_replace(',','.', $value3);

                DB::insert("INSERT INTO measurements(sensorsystem_id, measurement_type_id, value, ts) VALUES (?, ?, ?, ?), (?, ?, ?, ?), (?, ?, ?, ?)",[$sensorsyste
m_id, $measurement_type_id1, $value1, $ts, $sensorsystem_id, $measurement_type_id2, $value2, $ts, $sensorsystem_id, $measurement_type_id3, $value3, $ts]);
        return (new Response("Measurements registration successful!",200));
        }


        public function fourNewMeasurements($sensorsystem_id, $sensorsystem_type_id, $measurement_type_id1, $value1, $measurement_type_id2, $value2, $measurement_ty
pe_id3, $value3, $measurement_type_id4, $value4, $ts)
        {
                $sensorsystemValidationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
                switch ($sensorsystemValidationCode){
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered",400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }

                $requested_types = array((int)$measurement_type_id1, (int)$measurement_type_id2, (int)$measurement_type_id3, (int)$measurement_type_id4);
                foreach ($requested_types as $measurement_type){
                        if(!$this->validateMeasurementType($sensorsystem_type_id, $measurement_type)){
                                return (new Response("Measurement type " .$measurement_type ." not registered with sensor system type " .$sensorsystem_type_id,400))
;
                        }
                }

                $value1 = str_replace(',','.', $value1);
                $value2 = str_replace(',','.', $value2);
                $value3 = str_replace(',','.', $value3);
                $value4 = str_replace(',','.', $value4);

                DB::insert("INSERT INTO measurements(sensorsystem_id, measurement_type_id, value, ts) VALUES (?, ?, ?, ?), (?, ?, ?, ?), (?, ?, ?, ?), (?, ?, ?, ?)"
,[$sensorsystem_id, $measurement_type_id1, $value1, $ts, $sensorsystem_id, $measurement_type_id2, $value2, $ts, $sensorsystem_id, $measurement_type_id3, $value3, $t
s, $sensorsystem_id, $measurement_type_id4, $value4, $ts]);

return (new Response("Measurements registration successful!",200));
        }


        public function deletePlannedDischarges($sensorsystem_id, $sensorsystem_type_id)
        {
                DB::delete("DELETE FROM events WHERE sensorsystem_id = ? AND event_type_id = 3", [$sensorsystem_id]);
                return (new Response("Planned discharges deleted",200));
        }
}
```

E. ConsumerController.php

```php
<?php

namespace App\Http\Controllers;


use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;
use Illuminate\Http\Response;

class ConsumerController extends Controller
{

// VALIDATION FUNCTIONS

        // Validate sensorsystem type with sensorsystem id
        public function validateSensorsystem($id, $type_id){
                if(!$this->sensorsystemIdExists($id)) return -1;
                if(!$this->sensorsystemTypeExists($type_id)) return -2;
                if(DB::select("SELECT s.sensorsystem_type_id
                                FROM sensorsystems s
                                WHERE s.id = ?", [$id])[0]->sensorsystem_type_id != $type_id) return -3;
                return 1;
        }

        // Check if sensorsystem id exists
        public function sensorsystemIdExists($id){
                return DB::select("SELECT count(id) FROM sensorsystems WHERE id = ?", [$id])[0]->count == 1;
        }

        // Check if sensorsystem type id exists
        public function sensorsystemTypeExists($type_id){
                return DB::select("SELECT count(id) FROM sensorsystem_types WHERE id = ?", [$type_id])[0]->count == 1;
        }

        // Check if user exists
        public function userExists($username){
        return DB::select("SELECT EXISTS (SELECT true FROM users WHERE username = ?)", [$username])[0]->exists;
        }

// ID FOR USER

        // Get sensorsystem id for user
        public function getUserSensorsystemId($username, $password){

                if ($this->userExists($username)){
                        if (strcmp(hash('sha256', $password), DB::select("SELECT password FROM users WHERE username = ?", [$username])[0]->password) == 0){
                                return (new Response(DB::select("
                                        SELECT u.sensorsystem_id, st.type
                                        FROM users u, sensorsystem_types st, sensorsystems s
                                        WHERE username = ? AND s.id = u.sensorsystem_id AND s.sensorsystem_type_id = st.id", [$username]), 200));
                        } else {
                                return (new Response("password incorrect", 400));
                        }
                        return "no";
                } else {
                        return (new Response("user doesn't exist", 400));
                }
        }

// LOCATIONS ----------------------------------

        // Current locations of all sensorsystems of specifified sensorsystem type
        public function getAllSensorsystemsCurrentLocations($sensorsystem_type_id)
        {
                if($this->sensorsystemTypeExists($sensorsystem_type_id)){
                        return (new Response(DB::select("
                        SELECT s.id AS sensorsystem_id, ST_YMax(ST_Transform(l.location,4326)) AS latitude, ST_XMax(ST_Transform(l.location,4326)) AS longitude
                        FROM sensorsystems s, locations l
                        WHERE s.sensorsystem_type_id = ? AND l.sensorsystem_id = s.id AND l.ts = (
                                SELECT max(ts)
                                FROM locations
                                WHERE sensorsystem_id = l.sensorsystem_id)", [$sensorsystem_type_id]), 200));
                } else return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
        }

        // Current location of single sensorsystem
        public function getSingleSensorsystemCurrentLocation($sensorsystem_id, $sensorsystem_type_id)
        {
        $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch ($validationCode){
                case 1:
                        return (new Response(DB::select("
                                SELECT s.id AS sensorsystem_id, ST_YMax(ST_Transform(l.location,4326)) AS latitude, ST_XMax(ST_Transform(l.location,4326)) AS longitude
                                FROM sensorsystems s, locations l
                                WHERE l.sensorsystem_id = s.id AND l.ts = (
                                        SELECT max(ts)
                                        FROM locations
                                        WHERE sensorsystem_id = l.sensorsystem_id)
                                        AND l.sensorsystem_id = ?", [$sensorsystem_id]), 200));
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }

        // All locations of single sensorsystem
        public function getSingleSensorsystemAllLocations($sensorsystem_id, $sensorsystem_type_id)
        {
```

```
        $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch ($validationCode){
                case 1:
                        return (new Response(DB::select("
                        SELECT l.sensorsystem_id AS sensorsystem_id, ST_YMax(ST_Transform(l.location,4326)) AS latitude, ST_XMax(ST_Transform(l.location,4326))
                        AS longitude, l.ts AS timestamp
                        FROM locations l
                        WHERE l.sensorsystem_id = ?
                        ORDER BY timestamp", [$sensorsystem_id]), 200));
                        break;

                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;

                }
        }


// CHARACTERISTICS ---------------------------------

        // All characteristics of single sensorsystem
        public function getSingleSensorsystemCharacteristics($sensorsystem_id, $sensorsystem_type_id)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch ($validationCode){
                case 1:
                        $crosstab = "
                                SELECT c.sensorsystem_id, ct.type, c.value
                                FROM characteristics c, characteristic_types ct
                                WHERE c.characteristic_type_id = ct.id AND c.sensorsystem_id = " .$sensorsystem_id ." order by 1";

                        switch ($sensorsystem_type_id) {
                        case 1:
                                return (new Response(DB::select("
                                        SELECT * FROM crosstab (?) AS (
                                                sensorsystem_id INTEGER,
                                                capacity numeric,
                                                waterquality_guard_setting numeric,
                                                freeze_guard_setting numeric,
                                                water_collection_area numeric)",[$crosstab]), 200));
                                break;
                        case 2:
                                return (new Response(DB::select("
                                        SELECT * FROM crosstab (?) AS (
                                                sensorsystem_id INTEGER,
                                                skyview_factor numeric)",[$crosstab]), 200));
                                break;
                        default:
                                return (new Response(DB::select("
                                        SELECT c.sensorsystem_id, ct.type, c.value
                                        FROM characteristics c, characteristic_types ct
                                        WHERE c.characteristic_type_id = ct.id AND c.sensorsystem_id = ?", [$sensorsystem_id]), 200));
                        }
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }

// MEASUREMENTS ---------------------------------

        // All measurements of single sensorsystem
        public function getAllMeasurements($sensorsystem_id, $sensorsystem_type_id)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch ($validationCode){
                case 1:

                        switch ($sensorsystem_type_id){
                        case 1:
                                $crosstab = "
                                        SELECT m.ts, mt.type, m.value
                                        FROM measurements m, measurement_types mt
                                        WHERE m.sensorsystem_id = " .$sensorsystem_id ." AND m.measurement_type_id = mt.id order by 1,2";
                                return (new Response(DB::select("
                                        SELECT date_trunc('second',timestamp) AS timestamp, fill_level, temperature
                                        FROM crosstab (?) AS (timestamp TIMESTAMP, fill_level numeric, temperature numeric)
                                        ORDER BY timestamp",[$crosstab]), 200));
                                break;

                        case 2:
                                $crosstab = "
                                        SELECT m.ts, m.sensorsystem_id, mt.type, m.value
                                        FROM measurements m, measurement_types mt
                                        WHERE m.sensorsystem_id = " .$sensorsystem_id ." AND m.measurement_type_id = mt.id AND mt.sensorsystem_type_id = 2 ORDER BY
1";
                                $crosstab2 = "
                                        SELECT DISTINCT mt.type
                                        FROM measurement_types mt
```

```
                                        WHERE mt.sensorsystem_type_id = 2 ORDER BY 1";

                                return (new Response(DB::select("
                                        SELECT DISTINCT date_trunc('second',ct.ts) AS timestamp, ct.sensorsystem_id,
                                                ct.air_temperature, ct.humidity, ct.wind_speed
                                        FROM crosstab(?,?) AS ct(
                                                ts TIMESTAMP,
                                        sensorsystem_id INTEGER,
                                                air_temperature numeric,
                                                humidity numeric,
                                                wind_speed numeric),
                                        sensorsystems st, measurements m
                                        WHERE m.sensorsystem_id = st.id AND st.sensorsystem_type_id = 2
                                        AND m.sensorsystem_id = ?
                                        ORDER BY timestamp",[$crosstab, $crosstab2, $sensorsystem_id]), 200));
                                break;
                        default:
                                return (new Response(DB::select("
                                        SELECT s.id, st.type, mt.type, m.value, m.ts
                                        FROM sensorsystems s, sensorsystem_types st, measurement_types mt, measurements m
                                        WHERE m.sensorsystem_id = s.id AND s.sensorsystem_type_id = st.id
                                                AND m.measurement_type_id = mt.id AND s.id = ?
                                        ORDER BY m.ts", [$sensorsystem_id]),200));
                }
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }

        // All measurements for current day
        public function getDayMeasurements($sensorsystem_id, $sensorsystem_type_id)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch ($validationCode){
                case 1:
                        switch ($sensorsystem_type_id){
                        case 1:
                                $crosstab = "
                                        SELECT m.ts, mt.type, m.value
                                        FROM measurements m, measurement_types mt
                                        WHERE m.sensorsystem_id = " .$sensorsystem_id ." AND m.measurement_type_id = mt.id order by 1,2";

                                return (new Response(DB::select("
                                        SELECT date_trunc('second',ts) AS timestamp, fill_level, temperature
                                        FROM crosstab (?) AS (ts TIMESTAMP, fill_level numeric, temperature numeric)
                                        WHERE EXTRACT(DAY FROM ts) = EXTRACT(DAY FROM now())
                                        ORDER BY timestamp",[$crosstab]), 200));
                                break;

                        case 2:
                                $crosstab = "
                                        SELECT m.ts, m.sensorsystem_id, mt.type, m.value
                                        FROM measurements m, measurement_types mt
                                        WHERE m.sensorsystem_id = " .$sensorsystem_id ." AND m.measurement_type_id = mt.id AND mt.sensorsystem_type_id = 2 ORDER BY
1";
                                $crosstab2 = "
                                        SELECT DISTINCT mt.type
                                        FROM measurement_types mt
                                        WHERE mt.sensorsystem_type_id = 2 ORDER BY 1";

                                return (new Response(DB::select("
                                        SELECT DISTINCT date_trunc('second',ct.ts) AS timestamp, ct.sensorsystem_id,
                                                ct.air_temperature, ct.humidity, ct.wind_speed
                                        FROM crosstab(?,?) AS ct(
                                                ts TIMESTAMP,
                                                sensorsystem_id INTEGER,
                                                air_temperature numeric,
                                                humidity numeric,
                                                wind_speed numeric),
                                        sensorsystems st, measurements m
                                        WHERE m.sensorsystem_id = st.id AND st.sensorsystem_type_id = 2
                                                AND m.sensorsystem_id = ? AND EXTRACT(DAY FROM ct.ts) = EXTRACT(DAY FROM now())
                                                ORDER BY timestamp",[$crosstab, $crosstab2, $sensorsystem_id]), 200));
                                break;
                        default:
                                return (new Response(DB::select("
                                        SELECT s.id, st.type, mt.type, m.value, m.ts
                                        FROM sensorsystems s, sensorsystem_types st, measurement_types mt, measurements m
                                        WHERE m.sensorsystem_id = s.id AND s.sensorsystem_type_id = st.id
                                                AND m.measurement_type_id = mt.id AND s.id = ? AND EXTRACT(DAY FROM ts) = EXTRACT (DAY FROM now())
                                                ORDER BY m.ts", [$sensorsystem_id]),200));
                        }
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }

        // All measurements in specified period
        public function getMeasurementsPeriod($sensorsystem_id, $sensorsystem_type_id, $date1, $date2)
        {
```

```
318             $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);
319
320             switch ($validationCode){
321             case 1:
322                     switch ($sensorsystem_type_id){
323                     case 1:
324                             $crosstab = "
325                                     SELECT m.ts, mt.type, m.value
326                                     FROM measurements m, measurement_types mt
327                                     WHERE  m.sensorsystem_id = " .$sensorsystem_id ."  AND m.measurement_type_id = mt.id order by 1,2";
328
329                     return (new Response(DB::select("
330                             SELECT date_trunc('second',ts) AS timestamp, fill_level, temperature
331                             FROM crosstab (?) AS (ts TIMESTAMP, fill_level numeric, temperature numeric)
332                             WHERE ts > ? AND ts < ?
333                             ORDER BY timestamp",[$crosstab, $date1, $date2]), 200));
334                     break;
335                     case 2:
336                             $crosstab = "
337                                     SELECT m.ts, m.sensorsystem_id, mt.type, m.value
338                                     FROM measurements m, measurement_types mt
339                                     WHERE m.sensorsystem_id = " .$sensorsystem_id ." AND m.measurement_type_id = mt.id AND mt.sensorsystem_type_id = 2 ORDER BY
1";
340                             $crosstab2 = "
341                                     SELECT DISTINCT mt.type
342                                     FROM measurement_types mt
343                                     WHERE mt.sensorsystem_type_id = 2 ORDER BY 1";
344
345                             return (new Response(DB::select("
346                                     SELECT DISTINCT date_trunc('second',ct.ts) AS timestamp, ct.sensorsystem_id,
347                                                     ct.air_temperature, ct.humidity, ct.wind_speed
348                                     FROM crosstab(?,?) AS ct(
349                                          ts TIMESTAMP,
350                                          sensorsystem_id INTEGER,
351                                          air_temperature numeric,
352                                          humidity numeric,
353                                          wind_speed numeric),
354                                          sensorsystems st, measurements m
355                                     WHERE m.sensorsystem_id = st.id AND st.sensorsystem_type_id = 2
356                                           AND m.sensorsystem_id = ? AND ct.ts > ? AND ct.ts < ?
357                                           ORDER BY timestamp",[$crosstab, $crosstab2, $sensorsystem_id, $date1, $date2]), 200));
358                             break;
359                     default:
360                             return (new Response(DB::select("
361                                     SELECT date_trunc('second',m.ts) AS timestamp, m.sensorsystem_id, mt.type, m.value
362                                     FROM measurements m, measurement_types mt
363                                     WHERE m.measurement_type_id = mt.id AND m.sensorsystem_id = ? AND m.ts > ? AND m.ts < ?
364                                     ORDER BY timestamp",[$sensorsystem_id, $date1, $date2]), 200));
365
366                     }
367             case -1:
368                     return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
369                     break;
370             case -2:
371                     return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
372                     break;
373             case -3:
374                     return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
375                     break;
376             }
377     }
378
379     // ALl measurements all sensorsystems time period
380     public function getAllMeasurementsPeriod($sensorsystem_type_id, $date1, $date2)
381     {
382             if($this->sensorsystemTypeExists($sensorsystem_type_id)){
383                     switch ($sensorsystem_type_id){
384                     case 1:
385                     $crosstab = "
386                             SELECT m.ts, m.sensorsystem_id, mt.type, m.value
387                             FROM measurements m, measurement_types mt
388                             WHERE m.measurement_type_id = mt.id AND mt.sensorsystem_type_id = " .$sensorsystem_type_id ." AND m.sensorsystem_id > 0 ORDER BY 1";
389
390                     $crosstab2 = "
391                             SELECT DISTINCT mt.type
392                             FROM measurement_types mt
393                             WHERE mt.sensorsystem_type_id = " .$sensorsystem_type_id ." ORDER BY 1";
394
395             return DB::select("
396                     SELECT DISTINCT ct.ts AS timestamp, ct.sensorsystem_id, ct.fill_level, ct.temperature
397                     FROM crosstab(?,?) AS ct(
398                             ts TIMESTAMP,
399                             sensorsystem_id INTEGER,
400                             fill_level numeric,
401                             temperature numeric),
402                     sensorsystems st, measurements m
403                     WHERE m.sensorsystem_id = st.id AND st.sensorsystem_type_id = ? AND ct.ts > ? AND ct.ts < ?
404                     ORDER BY timestamp",[$crosstab, $crosstab2, $sensorsystem_type_id, $date1, $date2]);
405
406                             break;
407                     case 2:
408                     $crosstab = "
409                             SELECT m.ts, m.sensorsystem_id, mt.type, m.value
410                             FROM measurements m, measurement_types mt
411                             WHERE m.measurement_type_id = mt.id AND mt.sensorsystem_type_id = " .$sensorsystem_type_id ." AND m.sensorsystem_id > 0 ORDER BY 1";
412
413                     $crosstab2 = "
414                             SELECT DISTINCT mt.type
415                             FROM measurement_types mt
416                             WHERE mt.sensorsystem_type_id = " .$sensorsystem_type_id ." ORDER BY 1";
417
418             return DB::select("
419                     SELECT DISTINCT ct.ts AS timestamp, ct.sensorsystem_id, ct.air_temperature, ct.humidity, ct.wind_speed
420                     FROM crosstab(?,?) AS ct(
421                             ts TIMESTAMP,
422                             sensorsystem_id INTEGER,
```

```
                                        air_temperature numeric,
                                        humidity numeric,
                                        wind_speed numeric),
                                sensorsystems st, measurements m
                        WHERE m.sensorsystem_id = st.id AND st.sensorsystem_type_id = ? AND ct.ts > ? AND ct.ts < ?
                        ORDER BY timestamp",[$crosstab, $crosstab2, $sensorsystem_type_id, $date1, $date2]);
                                break;

                        default:
                        return DB::select("
                                SELECT m.ts AS timestamp, m.sensorsystem_id, mt.type, m.value
                                FROM measurements m, measurement_types mt
                                WHERE mt.sensorsystem_type_id = ? AND m.measurement_type_id = mt.id AND m.ts > ? AND m.ts < ?
                                ORDER BY timestamp",[$sensorsystem_type_id, $date1, $date2]);

                        }
                } else return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));

        }

// BUFFERED AMOUNT

        // Get buffered amount single sensor
        public function getBuffered($sensorsystem_id, $sensorsystem_type_id)
        {

                return DB::select("
                        SELECT sum(increase) AS buffered
                        FROM (
                                SELECT value - lag(value) OVER() AS increase
                                FROM measurements
                                WHERE measurement_type_id = 1
                                AND sensorsystem_id = ?
                                AND ts >= date_trunc('month', current_date - interval '1 MONTH')
                                AND ts < date_trunc('month', current_date)) AS t
                        WHERE increase > 0", [$sensorsystem_id]);
        }


        // Get total buffered amount
        public function getBufferedTotal($sensorsystem_type_id)
        {
                $sensorsystemids =  DB::select("SELECT * from sensorsystems WHERE sensorsystem_type_id = 1");
                $sum = 0.0;

                foreach ($sensorsystemids AS $ssid){
                        $sum += DB::select("
                                SELECT sum(increase) AS buffered
                                FROM (
                                        SELECT value - lag(value) OVER() AS increase
                                        FROM measurements
                                        WHERE measurement_type_id = 1 AND sensorsystem_id = ?
                                        AND ts >= date_trunc('month', current_date - interval '1 MONTH')
                                        AND ts < date_trunc('month', current_date)) AS t
                                WHERE increase > 0", [$ssid->id])[0]->buffered;
                }
                $result = new \stdClass();
                $result->total_buffered = $sum;
                $myJSON = json_encode($result);
                return response()->json($myJSON);
        }


// EVENTS -------------------------------------


        // Get all events single sensorsystem
        public function getSingleSensorsystemEvents($sensorsystem_id, $sensorsystem_type_id)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch($validationCode){
                case 1:
                        return (new Response(DB::select("
                        SELECT e.sensorsystem_id AS sensorsystem_id, et.type AS event_type, e.value AS value, e.precipitation AS precipitation, e.ts AS timestamp
                        FROM events e, event_types et
                        WHERE e.event_type_id = et.id AND e.sensorsystem_id = ?
                        ORDER BY timestamp", [$sensorsystem_id]), 200));
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }

        // Get events time period
        public function getEventsPeriod($sensorsystem_id, $sensorsystem_type_id, $date1, $date2)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch($validationCode){
                case 1:
                        return (new Response(DB::select("
                        SELECT e.sensorsystem_id AS sensorsystem_id, et.type AS event_type, e.value AS value, e.ts AS timestamp
                        FROM events e, event_types et
                        WHERE e.event_type_id = et.id AND e.sensorsystem_id = ? AND ts > ? AND ts < ?
                        ORDER BY timestamp", [$sensorsystem_id, $date1, $date2]), 200));
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
```

```
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;

                }
        }

        // Get specific events time period
        public function getSpecificEventsPeriod($sensorsystem_id, $sensorsystem_type_id, $event_type_id, $date1, $date2)
        {
                $validationCode = $this->validateSensorsystem($sensorsystem_id, $sensorsystem_type_id);

                switch($validationCode){
                case 1:
                        return (new Response(DB::select("
                        SELECT e.sensorsystem_id AS sensorsystem_id, et.type AS event_type, e.value AS value, e.ts AS timestamp
                        FROM events e, event_types et
                        WHERE e.event_type_id = et.id AND e.sensorsystem_id = ? AND et.id = ? AND ts > ? AND ts < ?
                        ORDER BY timestamp", [$sensorsystem_id, $event_type_id, $date1, $date2]), 200));
                        break;
                case -1:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered", 400));
                        break;
                case -2:
                        return (new Response("sensorsystem_type_id " .$sensorsystem_type_id ." doesn't exist",400));
                        break;
                case -3:
                        return (new Response("sensorsystem_id " .$sensorsystem_id ." not registered with type_id " .$sensorsystem_type_id, 400));
                        break;
                }
        }



}
```