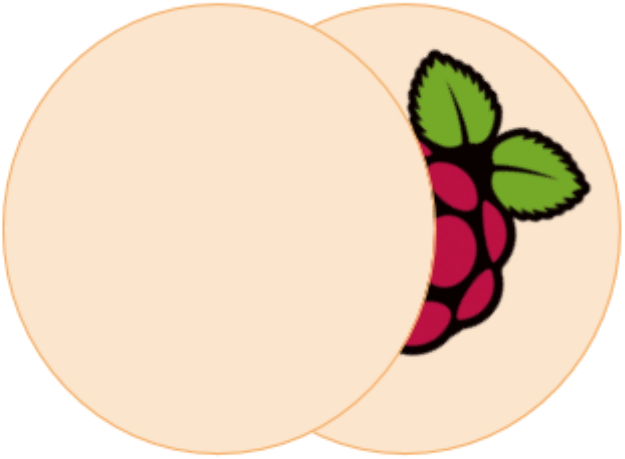
Design Document

Computer Infrastructures



Authors : Robin van Wijk, Roy Voetman, Shaquille Louisa, Robbert Tamminga, René Oun and Joey Marthé Behrens

Client : Lithuanian Maritime Academy
Document Number : DD.0001
Version : 1.3.4

Status : Completed
Document Date : 03-02-2020

I. Revision history

Version	Date	Description	Author(s)
1.0.0	19-01-2020	Document layout	Robin van Wijk, Roy Voetman, Shaquille Louisa, Robbert Tamminga and René Oun
1.1.0	20-01-2020	Chapter 4 and 6	Roy Voetman
1.1.1	20-01-2020	Chapter 6	René Oun
1.2.0	21-01-2020	Chapter 1, 2, and 3	Robin van Wijk and Joey M. Behrens
1.3.0	21-01-2020	Chapter 9	Robbert Tamminga
1.3.1	02-02-2020	Improved chapter 6	Roy Voetman and Shaquille Louisa
1.3.2	03-02-2020	Grammer correction	Robbert Tamminga
1.3.3	03-02-2020	Added code snippets	Roy Voetman
1.3.4	03-02-2020	Chapter 5	Robin van Wijk and René Oun
1.3.5	03-02-2020	Attachments	Robin van Wijk and Joey M. Behrens

Table of contents

1. Introduction

Hexagoons, a company specialising in weather data collecting and processing, was hired by the Lithuanian Maritime Academy to fill their needs for a system that provides them with the ability to look at the current and previous state of the weather without relying on third parties.

The Lithuanian Maritime Academy educates the next generation of qualified seafarers and universal specialists. They make heavy use of simulators in their practical classes to prepare students for real life situations.

The academy has issues with using third party weather data, because they have no control over anything that was done with the data. To make sure their simulators are equipped with accurate information, they would like full control over the systems that provide this data.

Hexagoons is hired by the Lithuanian Maritime Academy to provide the system they are requesting. The system has to be able to process about 2000-2500 weather stations with a longitude of 60 degrees or higher. The collected data should be stored in a safe location. This data should be able to be accessed using a website that translates them into easily understandable graphs. The website is equipped with a login system with three levels of users: Admins, Researchers, and Students.

In the next chapter we will go over the Project Definition. The requirements of the project are discussed in chapter 3. After which we will go into the infrastructure and environment in Chapter 4. Chapter 5 is an in depth look at the protocols used to make this project a reality. In chapter 6 the security measures will be defined and in chapter 7 we will go into the visual representation on the website. Lastly, we will go into the test procedures in chapter 8.

2. Project Definition

This chapter will go into the parties involved with the project itself, as well as give a general description that defines the project.

2.1 Organisation

There are two organisations involved with this project. The first party, Hexagoons, are the ones that actually have to realize the vision of the second party, the Lithuanian Maritime Academy.

2.1.1 Hexagoons

Hexagoons is a company known for their expertise in weather data. They collect data from a vast array of weather stations, and build applications and hardware solutions to incorporate this data for any particular end user.

The Hexagoon team is made up of six dedicated programmers that used to be freelancers working for a wide variety of companies. They started Hexagoons together, because they noticed nobody filled the void that they saw in the market.

As of today, Hexagoons collect data from 8000 different kinds of weather stations from all over the world and even have data of days gone by.

2.1.2 Lithuanian Maritime Academy

The Lithuanian Maritime Academy as its name implies, is an academy that educates the next generation of higher educated qualified seafarers and universal specialists. The academy consists of three departments: Navigation, Maritime Engineering, and Port Economics and Management.

To educate these students they provide modern classes and laboratories. The practical classes take place in several different simulators. Which have a very specific need for the weather data Hexagoons are able to provide.

Students are also taught to interpret data in its simplest form to make sure that they won’t rely on the simulators doing everything for them. Which means that the academy is looking for raw data, as well as the visual representation of said data.

2.2 Project description

The project that was described during the first meeting with the Lithuanian Maritime Academy representatives, is a distributed system that is able to collect, process, store, and provide a visual representation for the weather data the academy is interested in.

The system will consist of a Raspberry Pi 3 that is responsible for processing the data coming from the weather stations. After processing the data, the raspberry pi will store the collected data using an NFS mount on a virtual machine hosted on Microsoft's Azure datacenters. On this virtual machine will also run a web server that acts as the interface to the data, and provides graphs, tables and other visual elements making it easy to interpret the collected data.

To make sure all demands of the Lithuanian Maritime Academy are met, they made some requirements, which we will cover in the next chapter.

3. Requirements

During the meeting with the Lithuanian Maritime Academy representative, we discussed all the requirements they had for the project that we are going to make. Following is a list of requirements that were discussed during the meeting:

- ➔ Only data from weather stations above 60 degrees longitude
- ➔ Specific weather data:
  - ◆ Temperature
  - ◆ Wind
  - ◆ Snow
  - ◆ Weather events:
    - Hurricane
    - Storm
    - Hail
    - Freeze
    - Rain
- ➔ Data should be displayed in graphs
- ➔ Ability to support 25 concurrent users
- ➔ Ability to export weather data in XML format
- ➔ Multiple levels of authentications:
  - ◆ Admin
    - Can manage users, add remove set roles
    - Can manipulate weather data
    - Can export weather data
    - Can view weather data
  - ◆ Researcher
    - Can export weather data
    - Can view weather data
  - ◆ Student
    - Can view weather data
- ➔ Encrypted data traffic from web server to user (SSL)
- ➔ When a measurement is missed, interpolate the measurement between last and next.
- ➔ Website should be designed following the academy’s house style
- ➔ The website can be made in English only

4. Infrastructure & environment

4.1 High level overview of the application

The infrastructure will be distributed over several environments. To provide a high level overview of the application we will briefly consider each component.

- Weather data stations
  - We have a vast array of weather data stations around the world. These stations are all running software that sends the data to a data exchange point.
- Data Exchange Point
  - At a DXP the data of the 10 stations is collected and will be sent over a TCP socket to a central data processor.
- Central data processor
  - The processor will be a Raspberry Pi 3 with the Java Runtime Environment installed. A Java application will parse the incoming data and implement a mathematical function to estimate missing values. The parsed data will then be sent to a server capable of storing all the data.
- Azure Virtual Machine
  - The storage device that will be used is a virtual machine hosted on Microsoft Azure. Next to storing data this computer will also function as a web server to provide users with a graphical representation of the data.

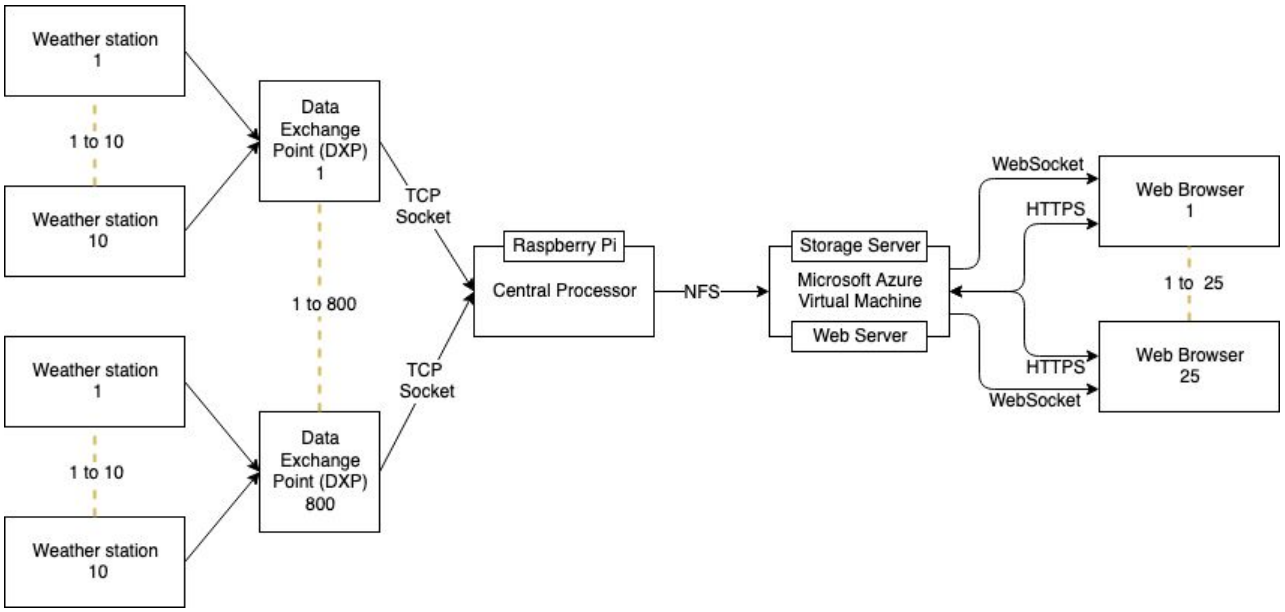


Figure 1.4.1 *Graphical representation of the infrastructure*

4.2 Weather data stations and DXPs

We have 8000 weather data stations around the world. Our weather data stations are grouped in clusters of 10 stations and every cluster has there own data exchange point (DXP). At a DXP the data of the 10 stations is collected and will be sent over a TCP socket to a central data processor. A DXP provides data at a set interval of one second, this means that the processor needs to be capable of processing 800 connections per second. The data will be provided in the XML-format shown in ***Figure 4.2.1***.

It is possible that certain fields will be left empty, this is the case when, due to e.g. corruption of data, an invalid measurement was taken. The central server receiving these empty fields will have to implement a mathematical function to estimate the value.

```
<WEATHERDATA>
   <MEASUREMENT>
       <STN>123456</STN>     <!-- Station number -->
       <DATE>2009-09-13</DATE>
       <TIME>15:59:46</TIME>
       <TEMP>-60.1</TEMP>    <!-- Celsius -->
       <DEWP>-58.1</DEWP>    <!-- Dew point in degrees Celsius -->
       <STP>1034.5</STP>     <!-- Air pressure at station level in millibars -->
       <SLP>1007.6</SLP>     <!-- Air pressure at sea level in millibars -->
       <VISIB>123.7</VISIB> <!-- Visibility in kilometers -->
       <WDSP>10.8</WDSP>     <!-- Wind speed in kilometers per hour -->
       <PRCP>11.28</PRCP>    <!-- Precipitation in centimeters -->
       <SNDP>11.1</SNDP>     <!-- Snow drop in centimeters -->
       <!--
       Events on this day, cumulatively, expressed binary.
       Freeze, Rain, Snow, Hail, Thunderstorm, Tornado / whirlwind
      -->
       <FRSHTT>010101</FRSHTT>
       <CLDC>87.4</CLDC>     <!-- Cloudiness in percentages-->
       <WNDDIR>342</WNDDIR> <!-- Wind direction in degrees -->
   </MEASUREMENT>
</WEATHERDATA>
```

Figure 4.2.1 *XML-format used by the weather stations.*

4.3 Central data processor

A Raspberry Pi (version 3) will be used as the Central data processor. The NFS protocol will be used to communicate to the Azure VM. This decision has been made because it is easy to configure and results in a low overhead and implementation time.

On the Raspberry Pi the following suite of software will be installed:

- The DietPi Operating System
    - DietPi is a Raspbian Lite inspired OS that is easy to configure. The OS is headless which means there will be less overhead compared to a GUI based OS. It's highly optimized for minimal CPU and RAM resource usage.
    - DietPi is faster than Raspbian Lite (system response times is 25 milliseconds compared to 100 milliseconds). In addition, some team members already have experience with DietPi servers.
- OpenJDK HotSpot JVM
    - Open Source
    - Optimized for ARM cores
    - Performance
- Dynamic DNS
    - A dynamic DNS service will be used so that the Pi can always accessed by its domain name without knowing the IP address. This is useful when the Pi is connected to a public network. This also removes the need for configuring a static IP.
- OpenSSH-Server
    - With the SSH protocol we can remotely access the Pi via an encrypted channel. OpenSSH is a tool that implements the SSH protocol and is supported by DietPi by default. Logging in to the Pi will be one with the domain name provided by the dynamic DNS. Authentication of users will be done with SSH keys. This eliminates the need for password logins and removes the vulnerability of brute force password attacks.
- Nfs-common (NFS Client)
    - To send data to the Azure Virtual Machine the NFS (Network File System) protocol will be used. With Nfs-common the Pi can access files from a remote file system as if it is a local file system.

This software running in the JVM needs to be highly optimized to keep up with the provided data in combination with the low resources that the Raspberry Pi offers. A multi-threaded architecture is needed to handle all TCP sockets (semi-)parallel. To provide a solid boilerplate for the program code the producer consumer pattern is used. A reference is made to ***Figure 4.3.1*** to provide further context on how this pattern is used in this situation.

4.3.1 Thread synchronization

Java has an implementation for a Linked Blocking Queue which extends the Abstract Blocking Queue implementation. This queue will be the **only use of thread synchronization** within the Java application. A blocking queue is chosen to provide a thread safe way to let the provider threads communicate with the consumer thread.

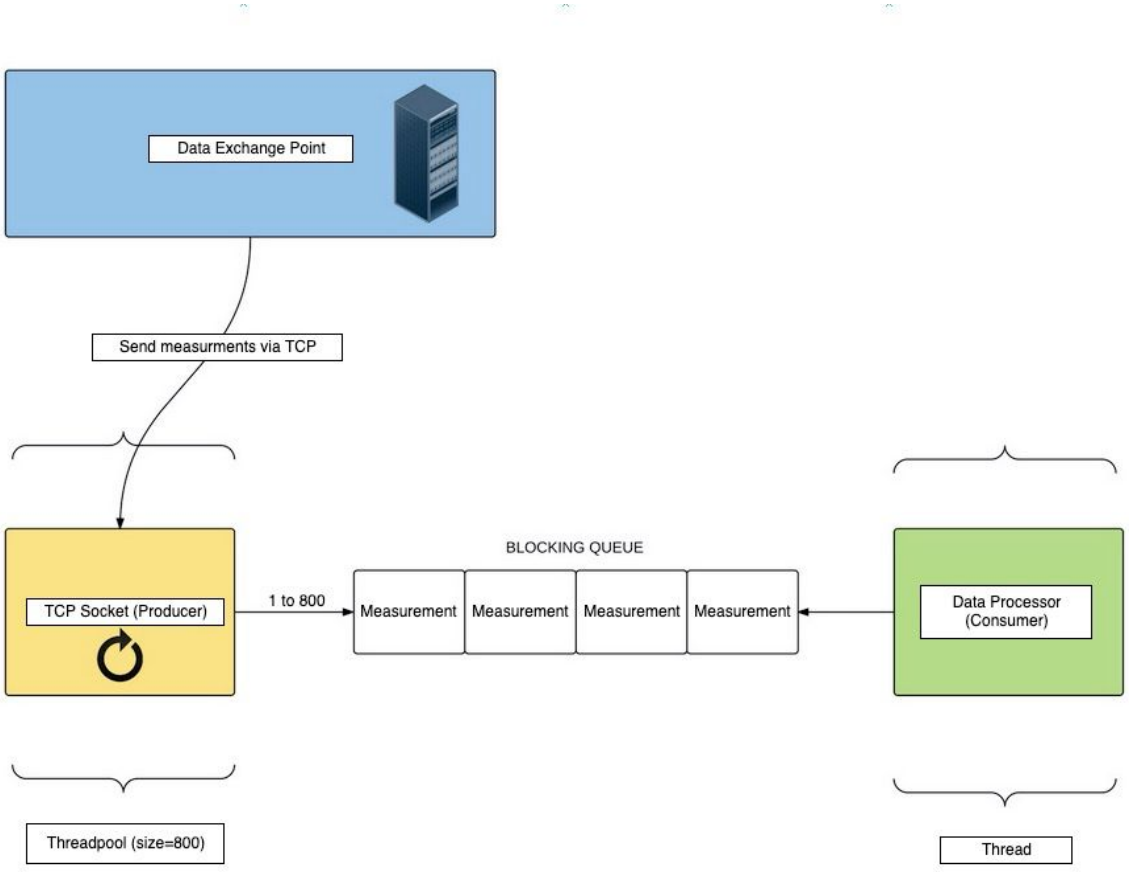


Figure 4.3.1 *High level overview of the data processor software.*

4.4 Azure VM

The Azure Virtual Machine will be used as the storage server and the web server. The web server is required to provide users with a graphical representation of the data. These two servers are on the same computer so the web server can directly access the data that has to be presented to the users.

On the Azure Virtual Machine to following suite of software will be installed:

- Ubuntu OS
    - The pre-installed operating system on this Azure virtual machine is Ubuntu. The decision is made to keep this OS because the necessary configurations have already been done (e.g. ssh support). Changing the OS must also be done via a request to the system management department. If bottlenecks arise at a later stage due to the OS choice, we can always replace the OS. To date, no problems seem to arise here and we do not want to attract extra work.

- Web Server
    - The web server that will be used is NGINX. NGINX does not have out of the box support for handling dynamic content. The website itself will be written in PHP because many of our team members know how to code in PHP. To handle a dynamic language (e.g. PHP) an external processor needs to be configured in NGINX. Although this can complicate configuration only pages with dynamic content will call upon the external processor. In addition, team members already have experience with NGINX servers.
    - A WebSockets server will be built with NodeJS and will be used to provide users with real time updates. NodeJS has its own built in web server and NGINX has a feature called “Reverse Proxy Server”. With a Reverse Proxy Server we can use NGINX as an intermediate application that forwards requests to the NodeJS process.
    - Based on the requested uri NGINX can distinguish then between passing the request to the PHP processor or passing along to the NodeJS process.

- SSH
    - With the SSH protocol we can remotely access the VM via an encrypted channel. The pre-installed operating system on the VM has built in support for SSH. Logging in to the VM will be one with the public IP provided by online Azure Portal. Authentication of users will be done with SSH keys. This eliminates the need for password logins and removes the vulnerability of brute force password attacks.

- Nfs-kernel-server
    - To receive data from the Raspberry Pi the NFS (Network File System) protocol will be used. With Nfs-kernel-server the Pi can upload files to this file system remotely as if it is a local file system.
    - Nfs-kernel-server is a daemon process that is capable of communicating with Nfs-common package on the Raspberry Pi.

5. System configuration

5.1 No-IP

For convenient access to the Raspberry Pi 3 and the Azure machine, we have installed the No-IP packages on both devices. No-IP will redirect the dns name to the IP Address of the machines. This allows us to connect to the machines by using the dynamic dns name.

This installation was also needed for the NFS protocol we use. The IP address of the Azure machine would change regularly and a static IP address is needed for the protocol.

Azure VM	hexagoons.ddns.net
Raspberry Pi 3	hexagoonspi.ddns.net

5.2 NFS

The connection between the Raspberry Pi and the Azure VM is done by mounting the disk that exists on the Azure VM, on the Raspberry Pi. This can be done by using the NFS protocol and software. NFS stands for Network File System, which means a file structure can be accessed using the internet.

The Azure VM is equipped with an *nfs-kernel-server* package which configures the disks to be available on the network. Within the */etc/exports* file we can allow for certain folders and files to be accessed. To ensure only the Raspberry Pi is able to access a particular folder or file, we can whitelist the DDNS ip discussed in the previous paragraph. We register a folder to the NFS server by adding the following line to the */etc/exports* file: */nfsshare hexagoonspi.ddns.net(rw, no_wdelay, sync, no_root_squash)*.

The Raspberry Pi itself is equipped with the *nfs-common* package, which makes sure we have the necessary software to mount a NFS disk. In DietPi we can make use of the *dietpi-drivemanager* command. This will open a UI for our headless version of the DietPi operating system, where we can easily mount a network disk by entering the DDNS ip of the Azure VM.

5.3 DietPi Configuration

To make sure we get the most out of our operating system we made some configuration changes to the DietPi operating system itself. DietPi offers a software tool that provides us easy access to the variables we need to set. The biggest of these variables is the linux governor. We set the governor to use the performance governor, meaning we force the CPU of the Raspberry Pi to run at its maximum clock speed as much as possible.

We also registered our application as a service in the systemctl settings by making a unit configuration file. This means we can start, stop and restart the application using the build-in service manager. When the application is registered as a service we can set certain priorities that we couldn't before. Our application is configured to have close to the maximum available priority on both the CPU and any I/O devices.

6. Protocols

6.1 Storage

6.1.1 Requiring data from DXP

The data that will be sent by the DXPs will be sent through *TCP* to the data processor. A TCP connection is more reliable than UDP in terms of missing data. The central data processor will be listening for connections using a *ServerSocket*. Everytime it makes a connection with a DXP, it will make a new task for that connection to process the data. This connection will be created by using the *Socket* Class.

6.1.1.1 Multi-threading (threadpool)

For requiring data, there will be a threadpool implemented. It could be very hard for a device to process data from 800 different locations, because per location it gets its own task. A threadpool is the solution to divide the workload evenly over the cores of a device, it will also automatically make a queue for all incoming tasks. The built-in Java Class, *Executor*, is able to make and configure the thread pools. We have used a Fixed Thread Pool with 803 nThreads. This is chosen because of the intensive IO operations that the device will execute. In this thread pool are 800 provider threads, one consumer thread, one logger thread and the main thread.

6.1.2 Data extrapolation

The incoming data will be extrapolated, so that missing data will be filled in. For the extrapolation we will use the formula below.

$$new\ value\ = \frac{\sum_{n=1}^{30}}{30}$$

The missing value will be a result of the average of the previous values. Weather changes usually concomitantly, so there won't be much differences between two consecutive values. The implementation of this formula in the code is shown in attachment #5. Every measurement is validated before it is added to the cache. The cache is a LIFO (Last In, First Out) collection so the most recent measurement is put at the end and the first measurement is deleted from the cache, pushing all the other measurements back.

6.1.3 XML-parsing

All incoming data from the DXP are written in a XML-format. This will be converted to a binary file, so it won't take too much space on the storage server. All the messages from the DXP are parsed with the SaxParser that is included in the Java standard library. The abstract DefaultHandler classic overwritten without our custom handler. the handler extract all the data and puts this in a messagequeue. The message queue is then polled by the consumer thread which handles validation and subsequently writes it to the storage server.

6.1.4 Saving/sharing data

All the data that has been processed will be automatically saved on the Azure VM. We are using a NFS-protocol that allows us to have shared directories on the Raspberry Pi and Azure VM. To minimize overhead at the processing unit (raspberry pi) all measurements will be saved into the same file. This file will be using a binary format instead of the provided XML format (see chapter 4.2).

A binary format will enable us to send data with the lowest amount of overhead. By removing the overhead of the XML, internet bandwidth requirements can be minimized. A measurements will be saved in the following binary format with a total of 63 bytes:

Measurement format:

Name:	STN	DATE	TIME	TEMP	DEWP	STP
Byte range:	1 - 4	5 - 14	15 - 22	23 - 26	27 - 30	31 - 34
Byte length	4	10	8	4	4	4

Name:	SLP	VISIB	WDSP	PRCP	SNDP	FRSHTT
Byte range:	35 - 38	39 - 42	44 - 46	47 - 50	51 - 54	55 - 55
Byte length	4	4	4	4	1	1

Name:	CLDC	WNDDIR
Byte range:	56 - 59	60 - 63
Byte length	4	4

Mnemonic definitions:

STN	Weather station identification number.
DATE	The recorded date is formatted into year - month - day.
TIME	The recorded time formatted into hour - minutes.
TEMP	The recorded temperature in degrees celsius.
DEWP	The recorded dew point in degrees celsius.
STP	The recorded air pressure in millibar.
SLP	The recorded air pressure on sea-level in millibars.
VISIB	The recorded visibility in kilometers.
WDSP	The recorded wind speed in kilometers per hour.
PRCP	The recorded precipitation in centimeters.
SNDP	The recorded snow drop in centimeters.
FRSHTT	The recorded events: Frost, Rain, Snow, Hail, Thunderstorms and Tornadoes
CLDC	The recorded cloud coverage in percentages.
WINDDIR	The recorded wind direction in radial degrees.

6.1.5 Querying of stored data

The NodeJS application will be responsible for responding to user queries. A user, once authenticated, will be able to emit data requests. The following data requests can be made:

- Most recent measurement of a station.
- Top 10 stations in a specific category.
- Export data of a station from a user specified start/end date.

As mentioned in chapter 5.1.3 all the measurement will be saved in a single file in a binary format. To increase lookup times we will be dividing the file into sections where one section contains one second of recorded measurements. When dividing the file the following mathematical constant can be used:

- Amount of measurements per seconds (8000)
- Amount of bytes per measurement (63)
- The timestamp (date and time) of the first measurement in the file.

The start/end of a specific section can be determined with the following equations:

$$difference\ in\ seconds\ =\ (searched\ for\ date)\ -\ (first\ measurement\ in\ file)$$

$$start\ of\ section\ =\ measurements\ per\ second\ *\ measurement\ size\ *\ (difference\ in\ seconds)$$

$$end\ of\ section\ =\ start\ of\ section\ +\ (measurements\ per\ second\ *\ measurement\ size)$$

These equations will then be used to calculate the byte offset(s) needed to fulfill a specific data request (i.e. to calculate a top 10 a whole section will have to be iterated through).

6.1.5.1 Code implementation

The equations provided in chapter 6.1.4 are here implemented in NodeJS to search for a specific datetime. Subtracting dates in NodeJS results in the difference in milliseconds, to take this into account the difference is divided by 1000.

```
module.exports.readMeasurementBlock = function(searchedForDate, callback) {
    var difference = (searchedForDate - firstMeasurement) / 1000;
    var measurementsPerSecond = config['measurementsPerSecond'];

    var start = measurementsPerSecond * measurementsSize * difference;
    var end = start + (measurementsPerSecond * measurementsSize);
    readMeasurementsFromFile(start, end, callback);
};
```

A function is created to parse a specific measurement. A measurement variable is generated based on the measurement data from a specific measurement. This measurement can be found by it's index number and the associated buffer.

```
module.exports.readMeasurement = function (buffer, index) {
    var measurement = {};

    measurement['stn'] = buffer.readUIntBE(index, 4);
    measurement['temp'] = (buffer.readFloatBE(index+22)).toFixed( fractionDigits: 2);
    measurement['date'] = buffer.slice(index+4, index+14).toString('ascii');
    measurement['time'] = buffer.slice(index+14, index+22).toString('ascii');
    measurement['wdsp'] = (buffer.readFloatBE(index+42)).toFixed( fractionDigits: 2);
    measurement['sndp'] = (buffer.readFloatBE(index+50)).toFixed( fractionDigits: 2);

    var events = buffer.slice(index+54, index+55)[0];
    measurement['ice'] = (events & parseInt( s: '10000000',  radix: 2)) >> 7;
    measurement['snow'] = events & parseInt( s: '00100000',  radix: 2) >> 5;
    measurement['tornado'] = events & parseInt( s: '00000001',  radix: 2);

    return measurement;
};
```

The order at which measurements are added to the file can not be kept consistent because of the use of multithreading in the Java application. When a specific section has been read this function is used to iterate through the section till the requested stn is found. In the event that the measurement is not found “null” will be returned.

```
module.exports.findMeasurementInBlock = function(stn, buffer) {
    var index = 0;

    while(true) {
        if(index+4 > buffer.length) {
            return null;
        }

        var currentStn = buffer.readUIntBE(index, 4);

        if(currentStn === stn) {
            break;
        }

        index += measurementsSize;
    }

    return index;
};
```

6.2 Website

6.2.1 Communication to webserver

We will use HTTPS for the communication between the webclient and web server (PHP). This provides the web server with better protection than regular HTTP. See chapter 6.3 for more information on this connection. Once a web page is loaded the web client will communicate with the NodeJS application to get real time updates of data.

6.2.2 Authentication on the webapp

When loading the web app by filling in the dns-name into a web client, a login page will be shown to the user. Here researchers and students can authenticate themselves to prevent malicious users from accessing the data.

Next to researcher and student accounts there will be an administrator account. Only accounts with administrator privileges will be able to create new accounts.

6.2.3 Getting data for webapp

The PHP backend of the web app will consult the running NodeJS application via the command line (PHP provides this functionality via the “exec” function). As described in chapter 5.1.4 the NodeJS application will implement the algorithm query the binary file from chapter 5.1.3.

6.2.4 Developing our own framework

As already explained the web application will use both PHP and NodeJS because no framework provides a way to communicate between these two languages we will create our own framework. This framework will enable us to use both languages within the same web application. Third party framework also introduces a lot of overhead which we can minimize to improve page load speeds and server load.

7. Security

7.1 User authentication

Users will authenticate themselves via the web login interface. Here users will be prompted to provide an email address and password. User passwords will be saved as bcrypt hash in the database. This prevents password leaks when e.g. the database security will be compromised.

Only one email address can be coupled to a user account, this has the advantage that the email can be used as a unique identifier. Once a user provided their email address and password, a database lookup will be done for the provided email address. Zero or one result will be returned saying that either the users does not exist (Zero) or does exist (One).

When a user has been found their provided password will be hashed. With a mathematical function provided by the bcrypt algorithm the newly created hash will be compared with the hash in the database. The function will result in either a “match” or “not a match”, if a “match” was returned the user has successfully authenticated itself.

When a user is authenticated the user can use all the functionality corresponding to his/her role until their session is expired. After expiry the user will be promoted to re-authenticate themselves.

7.2 WebSocket authentication

Handling WebSockets will be done with NodeJS. This process doesn’t share the same session data as PHP. This means users will have to be authenticated in a different way.

A WebSocket connection is setup with a “three way handshake” (messages to setup connection). In this handshaking process custom payloads can also be added. This feature will be used to send a by PHP generated token. This token will then be associated with a particular user, these tokens are always created in the same manner. This is to ensure that the NodeJS can reconstruct the token and validate the authenticity.

7.2.1 Token generation

The sha256 algorithm will be used to generate a hash on the following string: {users database id} + {users full name} + {users role}. An example would be to hash: “1 Hexagoons Admin”, this in turn would result in the following sha256 hash: “7c5042d796aa9f2849ac76050ed56bf96e2de797bd6c90f6543e1bf26f888d4d”. NodeJS can then reconstruct this hash by querying the users data in the central database.

PHP already provides an implementation of the sha256 algorithm via the hash function.

```
$_SESSION["user"]["token"] = hash( algo: 'sha256',  data: $id . $name .$role);
```

Before a socket connection is established, NodeJS will run the following code snippet to ensure the authenticity of a token.

```
con.query( sql: "SELECT id, first_name, last_name, role FROM users WHERE id = ?",
    values: [socket.handshake.query.id],  cb: function (err, result) {

    if (result.length !== 1) {
        next(new Error('Authentication error'));
    } else {
        var hash = shajs( algorithm: 'sha256').update(result[0].id
            + result[0].first_name + ' ' + result[0].last_name
            + result[0].role).digest( algorithm: 'hex');

        if(hash === socket.handshake.query.token) {
            next();
        } else {
            next(new Error('Authentication error'));
        }
    }
});
```

7.3 Encryption (TLS/SSL)

An TLS/SSL certificate will be installed on the webserver to provide a secure HTTP connection (HTTPS). HTTPS is an extension on top of the HTTP protocol. This is done by encrypting the HTTP messages via TLS (Transport Layer Security), formerly known as SSL (Secure Sockets Layer). Therefore HTTPS certificates are often referred to as TLS/SSL certificates.

Let's Encrypt (https://letsencrypt.org/) will be used to generate a free and validated TLS/SSL certificate.

7.3.1 How a secure connection is setup

Client	**Server**
Hello The client starts a session with a Hello message. This Hello message contains: **Version** The version number that it supports. For TLS, the version number is 3.1. **Random** The date and time of the client together with a pseudo-random number of 26 bytes. **Session ID (optional)** This is included if the client wants to resume an earlier session. **Encryption suite** This is the list of encryption suites that are supported by the client. An example of an encryption suite is **TLS_RSA_WITH_DES_CBC_SHA**, where TLS is the protocol version, RSA is the algorithm used for key exchange, DES_CBC is the encryption algorithm and SHA is the hash function.	**Hello** As soon as the server receives a hello message from the client, the server responds with its own hello message. This message contains: **Version** The server sends the highest version number that is supported by both client and server. **Random** The server also generates its own random value. It also contains its own date and time. **Session ID** If the client sends an empty session ID to start a new session, the server generates a new session ID. If the client sends a non-zero session ID to resume a previous session, it uses the same session ID sent by the client. If the server is unable or unwilling to resume a previous session, it will generate a new session ID. **Encoding Suite** This is the individual coding suite selected by the server from the coding suites proposed by the client. Server Certificate The server sends its certificate to the client.

	The server certificate contains the public key of the server. The client uses this key to authenticate the server and to encrypt the pre_master secret.
	Server Certificate The server sends its certificate to the client. The server certificate contains the public key of the server. The client uses this key to authenticate the server and to encrypt the pre_master secret.
	Server Key Exchange (optional) The server can create a **temporary** key and send it to the client. This key can be used by the client to encrypt the **Client Key Exchange** message. This step is only necessary if the server certificate does **not** contain a public key.
	Client Certificate Request (optional) The server asks the client to authenticate itself.
	Server Hello Done The server indicates that it is ready and is waiting for a response from the client.
Client Certificate If a **Client Certificate Request** is sent by the server, the client will reply with a **Client Certificate** message. The certificate contains the public key of the client.	
Client Key Exchange After the calculation of the **pre_master secret** this message will be sent. The pre_master secret consists of: the protocol version and the random number. The pre_master secret is encrypted with the public key of the server.	

Generating master secret

In this phase, the client and server separately create a **master secret** of the data exchanged during previous phases.

The client and server use the following to generate the master secret:

- pre_master secret
- Random values from client and server

These two master secrets are never exchanged.

Only the server with the secret key that matches the public key (from its certificate) can decrypt data that is encrypted with its public key and vice versa.

Client Certificate Verify

If a **Client Certificate Request** is sent by the server, the client will now respond with a **Client Certificate Verify** message.

This message contains the hash of all handshake messages that have so far been signed with the public key of the client.

Change Cipher Spec / Client Finished

The client sends a **Change Cipher Spec** message to the server to indicate that all messages that follow the **Client Finished** message are encrypted using the keys and algorithms that have been negotiated.

The **Client Finished** message follows immediately after this message.

Change Cipher Spec / Server Finished

The server also sends a **Change Cipher Spec** message.

This is immediately continued with a **Server Finished** message to indicate that the server is going to encrypt messages using the keys and algorithms that have been negotiated.

Application Data

From now on the HTTP protocol will be used over the now secured **TLS / SSL connection**.

The **Record Protocol** receives the data from HTTP and is responsible for:

- Fragmentation of the data in blocks
- Reconstruction of the data blocks
- Organizing the data blocks
- Compression / Decompression of the data
- Encoding / Decoding of the data.

8. Visual representation

8.1 Interfaces

There are several different interfaces on the web page, on the first page users can login. Logging in allows users to be able to see the content that their roles give them access to. After having logged in the user will be brought to the dashboard page.

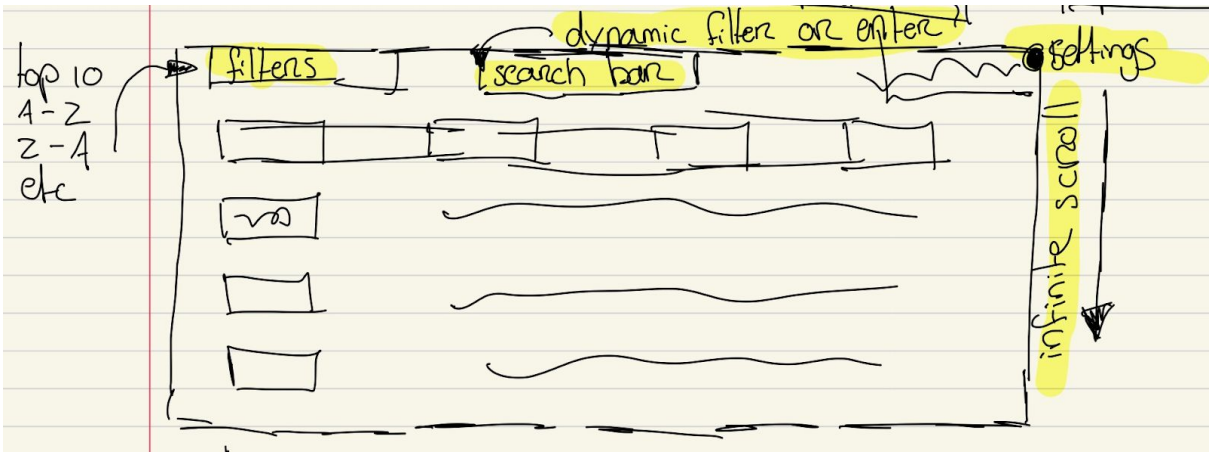


Figure 7.1.1 *The Dashboard page.*

The sketch Figure 7.1.1 represents the layout of the dashboard of the web application. On this page users can find a list of all weather stations. The weather stations can be filtered and sorted. Finding a weather station by name can be done through the search bar In the top-center of this page. The list of weather stations will then be updated based on the user input in this search bar. To find weather stations without knowing the name of the weather stations can be done through the different filter options. The different filter options are located in the sidebar on the left side of this page.

This page needs to be able to display a lot of weather station names. To be able to display all station names, the user can scroll through this page. The stations in this list are grouped by country and sorted by name. To make sure that navigating between the different stations is done efficiently; the page will always scroll to the location of the last opened station.

There are no role specific options on this page, all users can see all the info on this page.

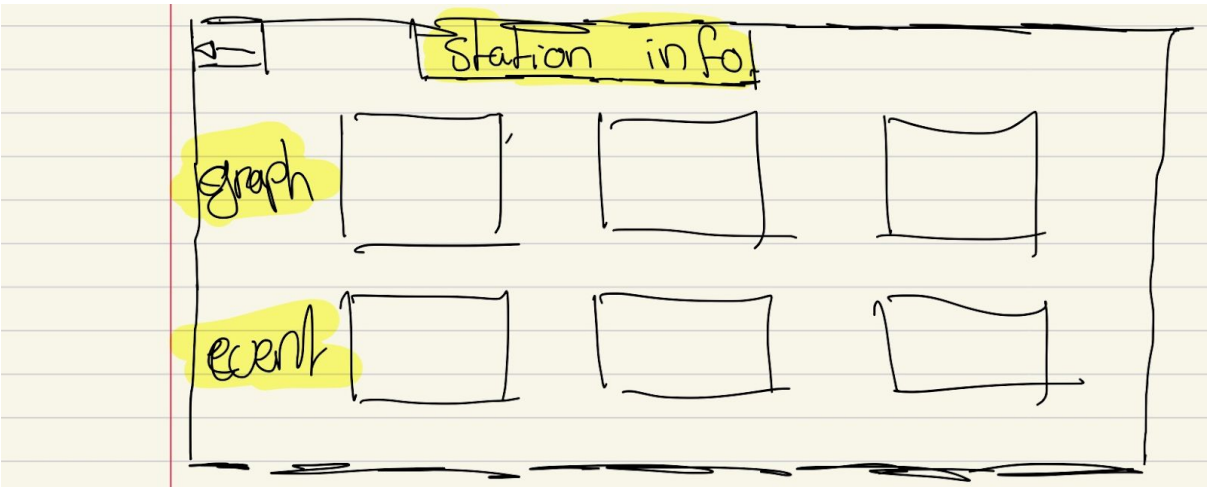


Figure 7.1.2 *Weather Station page.*

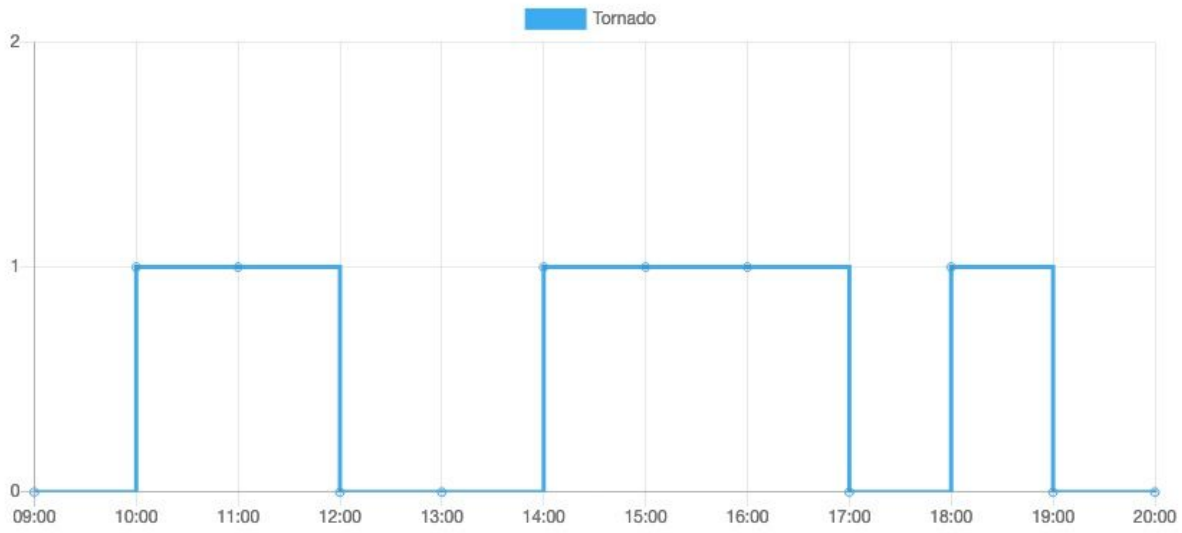
The sketch in Figure 7.1.2 represents the layout of the weather station page.
On this page users can find all weather measurements and location details related to the selected station. There are two types of weather measurements being displayed on this page, graphs and events.

8.2 Graphs

8.2.1 Graphs

Graphs display the weather measurements that are based on data that can be drawn using a line. We will use graphs for displaying temperature for example. The names of the variables will be displayed alongside their relevant axis.

8.2.1 Events



Events display the weather measurements that can not be drawn using a line. We will use events for displaying hurricanes for example. Events are only measured as; true or false. This means that the line used in drawing event graphs will always be drawn near the bottom of the event image or at the top.

9. Test procedure

9.1 Stress tests

Stress testing is a necessity for every application. It ensures that developers are aware of the stability and reliability of the system. Stress testing is done by pressuring the system with heavy workload. In our stress test we pressure the system with a large amount of users and monitor how the system is holding up under this amount of pressure. The goal we want to achieve is to see how it behaves after it endured a failure. By analysing this behaviour we can find out how to make the system more reliable and increase the recovery time if it endures a failure.

One way of stressing the system is by altering settings from the weather generator. This generator has an option of increasing missed values per second. By increasing this value more data will be corrupted when sent. Interpolation will be used to give corrupted data a value of what it could have been, though every time this has to be done more pressure is set on the system. Monitoring this will be of vital importance.

Another method of stressing the system is by increasing the speed of the datastream. Sending more information to the pi will increase the workload. At some point the system can not handle it anymore. Monitoring this performance will be very important to ensure we know the limits.

The website will also be tested on performance. This will be done by loading the website with a large amount of users, in our case 25. We want to make sure the website can hold this amount of pressure. Also, making sure our live graphs will perform without any issues since this is a major factor of the website itself.

9.2 User tests (usability)

On the website there are three different roles a user can have. These roles are admin, researcher, and student. All these roles have different levels of authentication. To ensure that these authentication levels are being enforced on the website this has to be tested. This will be done by logging in as every role and trying to perform tasks the role should not be allowed to do. If a breach of authentication level is detected, this will be reported and fixed to ensure it never happens again. If everything checks out, the test is done.

9.3 Penetration Testing

Since our application is website based, it is of vital importance this website is secure. By performing a penetration test (pen test) we will know if the website is secure, or to find out how we can fix this. We are going to use automated software packets to test our website.

9.4 Documentation and logging

While performing a stress test of the system, a large amount of data has to be monitored and stored for an in depth analysis later on. While the stress test is being performed, we monitor the graphs and store it for later use. This method will be performed for all testing. Once the testing is completed a report will be drafted for an in depth analysis with the developers.

Attachments

#1 Server accept

```
Socket currentConnection;
new Thread(consumer).start();
new Thread(loggerThread).start();

try {
    while (true) {
        currentConnection = serverSocket.accept();
        threadPool.execute(new Provider(currentConnection, queue));
    }
} catch (IOException e) {
    e.printStackTrace();
}

System.out.println("Listening...");
```

The code snippet above, shows how the server listens to incoming connection requests and executes a new task in the threadpool. The threadpool consists of 800 available threads to make sure we don't block other tasks. Every connection is handled by a provider which adds any incoming measurements to the LinkedBlockingQueue.

#2 Provider implementation

```
try {
    //Read the lines in the buffered reader(socket)
    while ((line = bufferedReader.readLine()) != null) {
        //Appen the read line
        sBuilder.append(line.trim());

        //Check if we reached the end of the xml file stream
        if (line.equalsIgnoreCase("</WEATHERDATA>")) {
            //Create a string reader and parse the xml file
            StringReader reader = new StringReader(sBuilder.toString());
            stringInputSource.setCharacterStream(reader);
            parser.parse(stringInputSource, saxHandler);

            //Get the result of the parser using the handler
            ArrayList<Measurement> measurements = saxHandler.getMeasurements();

            //Loop through the measurements that were send
            for(Measurement measurement : measurements){
                //Validate that there are no errors, or fix them if there are
                validateMeasurement(measurement);

                //Manage the cache
                if(measurementCache.containsKey(measurement.stn)){
                    ArrayList<Measurement> cache = measurementCache.get(measurement.stn);
                    if(cache.size() == 30){
                        cache.remove(0);
                    }
                    cache.add(measurement);
                }
                else{
                    measurementCache.put(measurement.stn, new ArrayList<>());
                    measurementCache.get(measurement.stn).add(measurement);
                }
                //Add the measurement to the blocking queue
                queue.put(measurement);
            }

            //Reset the string builder and close the reader
            sBuilder.setLength(0);
            reader.close();
        }
    }
```

After creating a new provider and adding it to the threadpool, we start listening to the socket using a BufferReader. The buffered reader allows us to sometimes miss a cycle as it just stores data within its own buffer. After we read a line we append it using the stringbuilder and parse the xml string sent by the weather stations as soon as we find the end of the file.

To handle errors and missing values, we use extrapolation and validation. To facilitate this feature we must keep a cache of measurements for each station. We can use these measurements in an extrapolation formula.

#3 Single File Consumer implementation

```
public SingleFileConsumer(LinkedBlockingQueue<Measurement> queue) {
    super(queue);
    try {
        File file = new File(filepath + fileName);

        if (!file.exists()) {
            file.createNewFile();
        }

        outputStream = new BufferedOutputStream(new FileOutputStream(file, true));
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}

@Override
public void run() {
    Measurement measurement;
    try {
        while (true) {
            while ((measurement = this.queue.take()) != null) {
                outputStream.write(measurement.getBytes());
                outputStream.flush();
            }
        }
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
}
```

This code snippet shows the implementation of the single file consumer that writes binary data to the NFS disk. When we create the single file consumer we create a file if it doesn't exist yet, and keep the file stream open. We keep this filestream open to prevent us from needing to open and close file streams, which will result in a huge performance loss. Instead, we flush the writer after we write to it.

#4 Measurement class

```java
//This class acts as a struct (like in C/C++), reason being the optimization of function calls
public class Measurement {
    public int stn;
    public String date;
    public String time;
    public Float temp;
    public Float dewp;
    public Float stp;
    public Float slp;
    public Float visib;
    public Float wdsp;
    public Float prcp;
    public Float sndp;
    public Float cldc;
    public int frshtt;
    public Integer wnddir;

    public byte[] getBytes(){
        ByteBuffer buffer = ByteBuffer.allocate(63);
        buffer.putInt(stn);
        buffer.put(date.getBytes());
        buffer.put(time.getBytes());
        buffer.putFloat(temp);
        buffer.putFloat(dewp);
        buffer.putFloat(stp);
        buffer.putFloat(slp);
        buffer.putFloat(visib);
        buffer.putFloat(wdsp);
        buffer.putFloat(prcp);
        buffer.putFloat(sndp);
        buffer.putFloat(cldc);
        buffer.put((byte)frshtt);
        buffer.putInt(wnddir);
        return buffer.array();
    }
}
```

As the first line in the code snippet suggests, this class functions as a struct. A struct is used in C/C++ as a sort of class with for all intents and purposes just variables. Some languages support adding functions as well, however. The reason we use this class as if it was a struct is because of the overhead a function call would create. Normally you would make private variables in Java and make so called “Getter‘ and “Setter” methods to interact with the data of the variables. Every variable in the measurement class is used about 16.000 times each second. Calling all the getters and setters for each variable would create massive overhead.

#5 Data extrapolation (part of provider)

```
private void validateMeasurement(Measurement measurement){
    if (!measurementCache.containsKey(measurement.stn))
        measurementCache.put(measurement.stn, new ArrayList<>( initialCapacity: 30));

    if (measurement.temp == null) {
        float temp = 0;
        for (int x = 0; x < measurementCache.get(measurement.stn).size(); x++) {
            if (measurementCache.get(measurement.stn).get(x).temp != null)
                temp += measurementCache.get(measurement.stn).get(x).temp;
        }
        measurement.temp = temp / measurementCache.size();
    }
    if (measurement.dewp == null) {
        float temp = 0;
        for (int x = 0; x < measurementCache.get(measurement.stn).size(); x++) {
            if (measurementCache.get(measurement.stn).get(x).dewp != null)
                temp += measurementCache.get(measurement.stn).get(x).dewp;
        }
        measurement.dewp = temp / measurementCache.size();
    }
```