Project description & installation guide

Team A1: Code craftsmen

Bachelor in applied sciences

Lucas De Greef
Syan Delbart
Charles Kwakye
Wieland Vandebotermet
Tristan Van Loy
Jen Verboven

Academic year:2023-2024

Campus : Geel

Table of contents

Introduction

This document serves as a comprehensive resource understanding and implementing our solution for gathering, normalizing and visualizing data about water levels in Flanders.

This document is organized into two main sections:

1. **Project description:** Here we delve into the details of the project. More about the objective and the scope of the project, the key features and the chosen technologies can be found here.
2. **Installation guide:** This section contains a step-by-step guide to successfully install, configure and recreate everything you need to make this solution work yourself.

Team members

Jen Verboven

Tristan van Loy

Wieland vandebotermet

Syan Delbart

Lucas De Greef

Charles Nana Kwakye

Project description

In this section, we'll go over the structure of the product we ended up with. Among other things, we'll take a look at the functionalities we implemented, how we retrieved and processed the required data and what technologies we used to achieve the results.

Objective

The objective of this project was to provide local governments of provinces in Flanders with an online platform to monitor the water levels of second grade water streams in Flanders. Using this platform, they would then be able to more accurately track water levels, and thus be able to make more accurate and swift decisions when measures would have to be taken against those water levels.

Scope

In this section, we define what we minimally have to make for our product to be effective, what would be nice to have on top of that and what we will definitely not realize during this project.

Must have

For our minimum viable product, we have to set up infrastructure which collects the necessary data about second grade water streams from the different possible sources discussed in the data flow section. This data should also be stored in a normalized format according to the OSLO standards.

Also, we have to create a dashboard where the retrieved data about second grade water streams is visualized. This dashboard should be structured logically and made easy to understand for non-technical end users.

Should have

On top of the dashboard, it would be of great use to the end users if the graphs of the dashboard are displayed in a nice front-end application which is easier to reach for a larger number of end users.

Furthermore, the options to both add new sensors to the database and to be able to manually insert measurements for any available sensor could be of great use to certain end users.

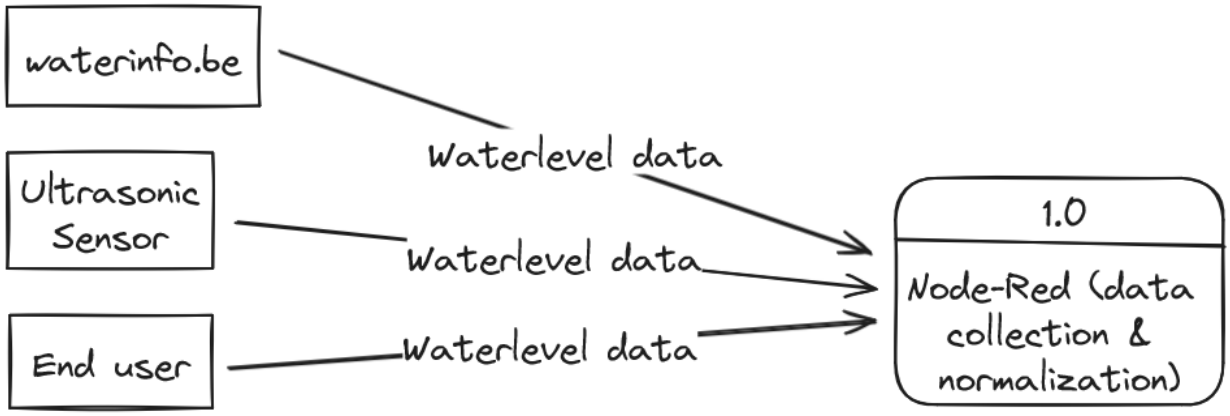
Out-of-scope

The prediction of water levels at certain locations based on actions taken or events happening would be the ultimate goal of this product but was out of scope for this particular project.

Key features

In this section, we go over the features we were or weren't able to implement during the realization phase of the project.

Data collection



In Node-Red, we set up our data collection functionalities so that data from the three different sources visualized above is brought together on the same platform. More detailed information about how this works is given in the section about data flow.

Data normalization

Using functions in plain JavaScript inside of Node-Red, and a predefined application profile for normalizing data about waterbodies defined by the Flemish government, The data we retrieve from waterinfo.be and from sensors is normalized to the OSLO standards.

Data storage

The normalized data described in the data normalization section above is sent to an Amazon S3 bucket in the JSON-LD format. The data is also stored in a PostgreSQL database without the application profile context for further use. This dual-storage approach ensures that we have the normalized data available in the S3 bucket as well as a relational data source which is more accessible and necessary for further processing of the data on platforms like Grafana.

Data visualization

The data stored in the PostgreSQL database is then used to create visualizations in our Grafana dashboard. Here, we made sure to make the created graphs are as clear and easy to understand as possible for any type of end user, while at the same time also providing the necessary insights about the data to be able to effectively reach the objective of the project described earlier in the project description.

The visualizations in our dashboard are updated with fresh data every twenty minutes. We gave our data collection functionalities this interval as the external data source we work with only provides fresh data every fifteen minutes. Because of some slight timing differences between the various stations provided by waterinfo.be, we chose to use this interval to avoid trying to fetch duplicate data records too often.

Technologies used

In this section we'll go over the technologies we used to realize the key features described above

Node-Red

For the backend, we used Node-Red as provided by Cipal Schaubroeck on the MyCSN platform.

Over the course of the realization phase of the project, Node-Red proved to be a very versatile tool for setting up multiple backend functionalities and combining multiple data sources relatively quickly. We were able to combine and transform data retrieved through API calls as well as data from our own proof of concept ultrasonic sensor over an MQTT connection without having to write too much code manually or having to worry about how to make a connection to those data sources in the first place.

We were also able to write our own API endpoints in Node-Red which we would later use to provide our web application with data from our own database.

PostgreSQL

As our data storage, we used a PostgreSQL database as provided by Cipal Schaubroeck on the MyCSN platform.

An SQL database like PostgreSQL was needed as Grafana needs an SQL database to function properly. On top of that, our team members had way more experience using relational databases than they had using NoSQL databases, so this was also our preference.

Grafana

As our data visualization platform, we used Grafana as provided by Cipal Schaubroeck on the MyCSN platform.

With Grafana we were able to provide our web application with various user-friendly visualizations of the data we collected from waterinfo.be and our own ultrasonic sensor.

NextJS

For the front-end, we chose to use NextJS. We made this decision because NextJS is a React-based framework, which all of our group members who specialize in application development are familiar with. We picked NextJS over React itself because NextJS provides some extra built-in functionalities which React does not.

Esp32 microcontroller

Esp32 is a system on chip (SoC) that supports 2 of the mainstream wireless communications: Wi-Fi and Bluetooth. The reason we are using this microcontroller in this project is the good integration with the Arduino framework. This framework gives some built in libraries that makes fast development possible.

MQTT protocol

For sending the data from our own from our prototype sensor we used the MQTT protocol. This protocol is a standard used for IoT devices. It's also easy to implement and use for sending data effectively. We also chose this way of data transmission as it integrates well with Node-Red.

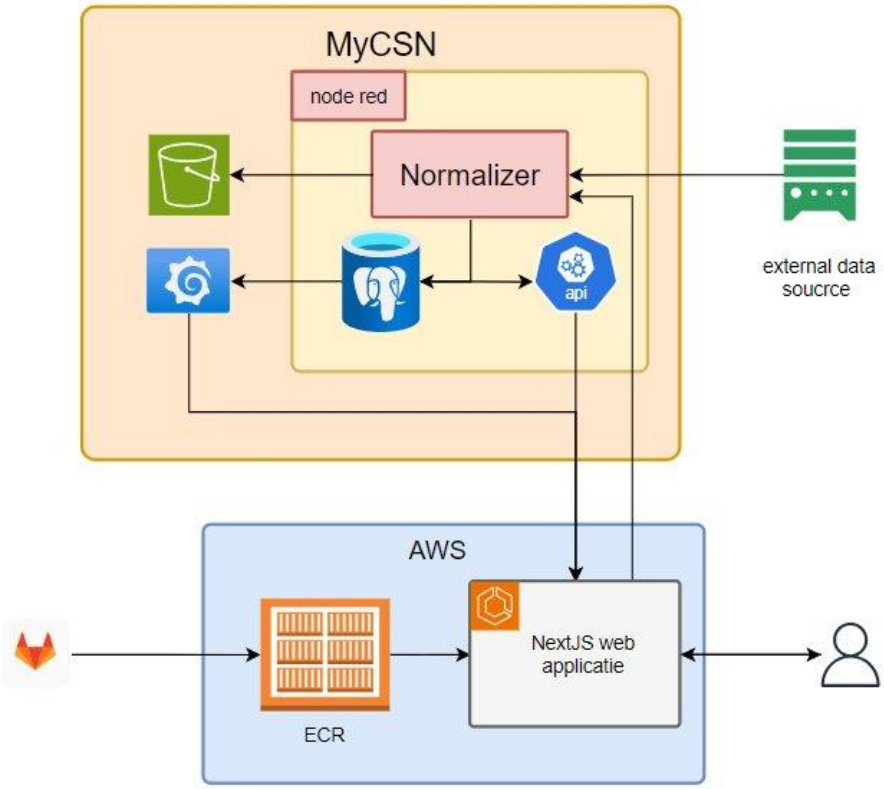
Ultrasonic sensor

Initially, our team was assigned a Meratch NB-IOT sensor for making a proof of concept. We ended up not getting access to its data, so we came up with our own solution for a DIY sensor.

To measure the distance between the prototype device and a surface, we utilized an ultrasonic sensor. The microcontroller computes the data and sends it to the MQTT topic located on Node-Red on the MyCSN platform. For the DIY sensor, the choice was made to do a quick build on a breadboard, because in combination with the esp32 dev kit and ultrasonic sensor the solution is quick and simple to rebuild.

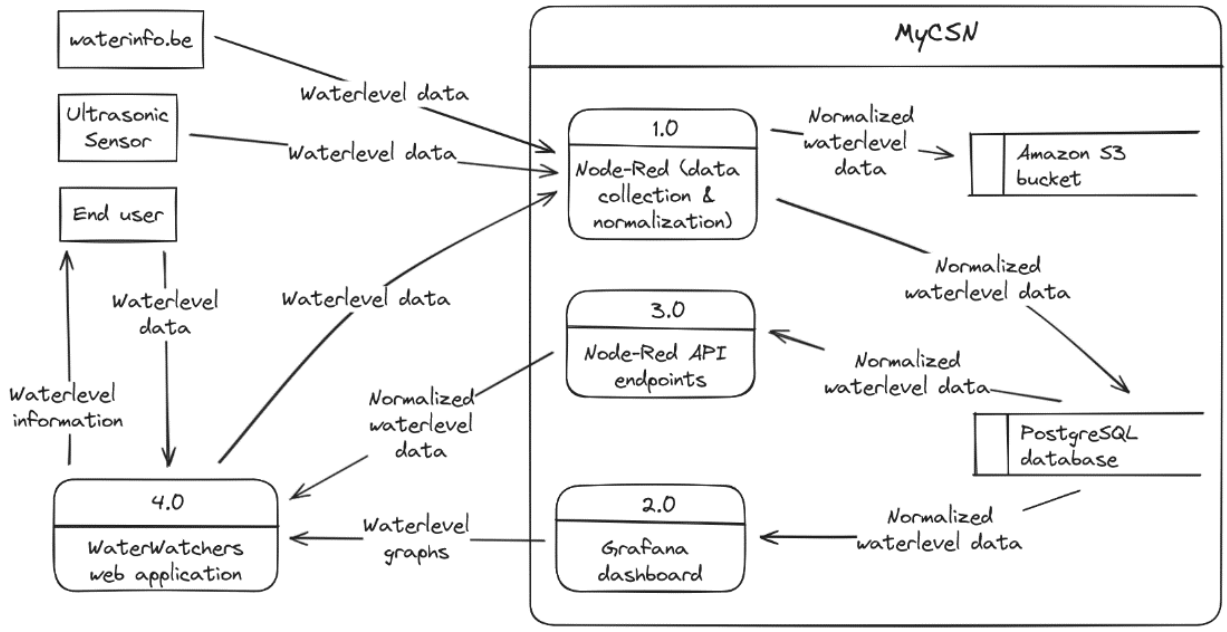
Architecture

Architecture wise a lot was provided for us, the architecture for our web application relies on Amazon Web Services (AWS) for hosting, ensuring scalability, reliability, and performance.



Data flow

In this section, the data flow throughout our product will be discussed.



As visualized in the diagram above, our solution can, but doesn't have to, retrieve data about water levels from three different types of sources. First up, we have waterinfo.be. This can in fact be any type of external data source which can be queried through API calls. The second type is an ultrasonic sensor. The third type is an end user of our web application.

All of this data is initially retrieved in our Node-Red backend. In here, we have set up API calls for retrieving data from waterinfo.be on an interval. For the sensor data we have set up an MQTT connection over which our proof-of-concept ultrasonic sensor sends its data. This sensor has been manually set up to send most of the data needed to make our dashboard/web app work properly. Lastly, end users can manually put in their own measurements for any sensor they added through the web application or for any sensor from waterinfo.be.

Next up, all of this data is processed in Node-Red and added to the PostgreSQL database. From there, the data is either queried by the Grafana dashboard itself, where it's used to make visualizations, or queried via API endpoints in Node-Red. The web application can then query the data in the database through those API endpoints or can embed the graphs created in the Grafana dashboard through the use of *iframes* and Grafana's API. The web application is the final destination in the data flow before it reaches our end users.

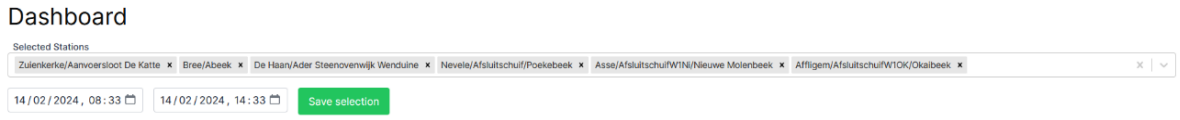
User interaction

Lastly, we developed a web application to make the data and the visualizations of that data more easily accessible to end users and provide them with the option to insert their own data.

On top of providing the users with easier access to the already existing data and graphs, the web application also further extends those functionalities. Through the web application, our end users can also add images of their sensor setups to the database for clarity and personalization, as well as choose to receive push notifications about the water level at certain locations after subscribing to that location.

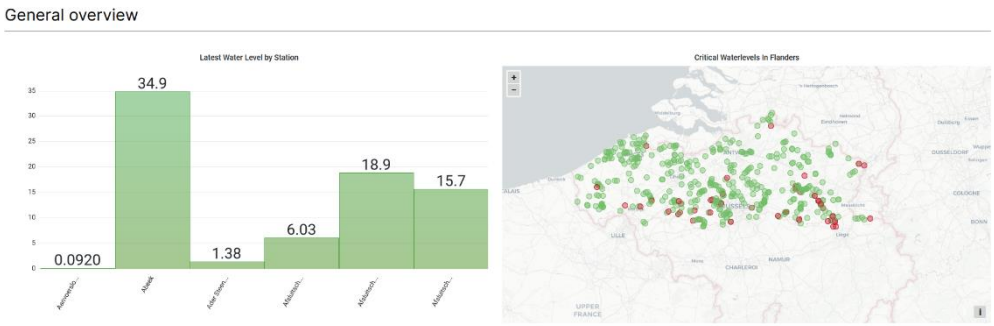
The dashboard homepage

The dashboard is made of the iframes which has embedded Grafana graphs and the select components, which selects locations and the time.

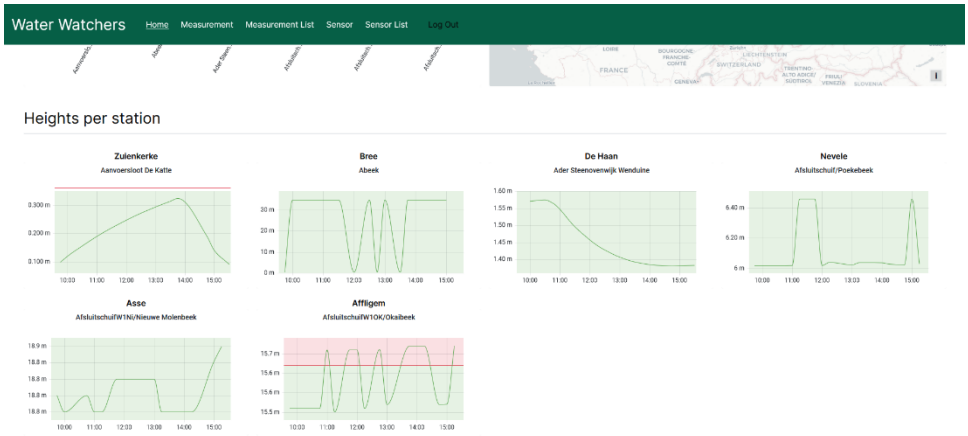


After it we have 2 Grafana graphs which show the latest water levels:

- A bar graph that shows all the selected locations
- And a Map Graph that shows all locations on the map.

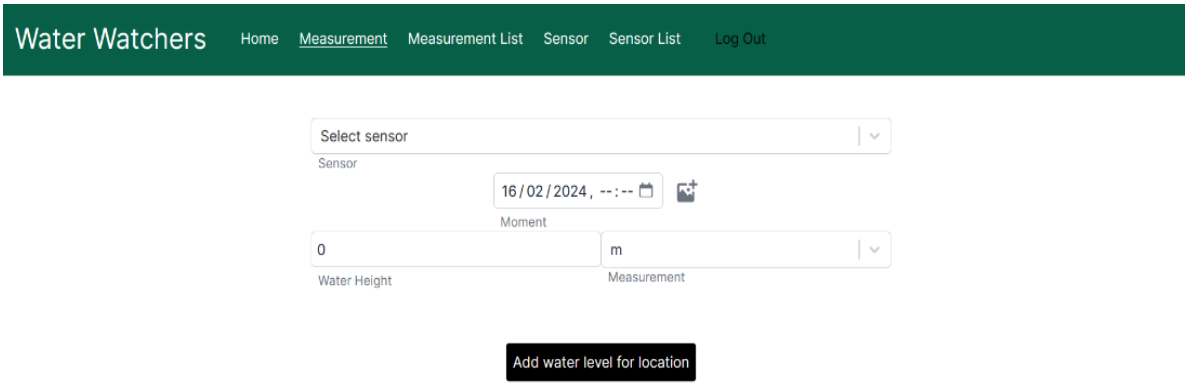


After which we have the individual graphs for each location, that show the latest water levels.

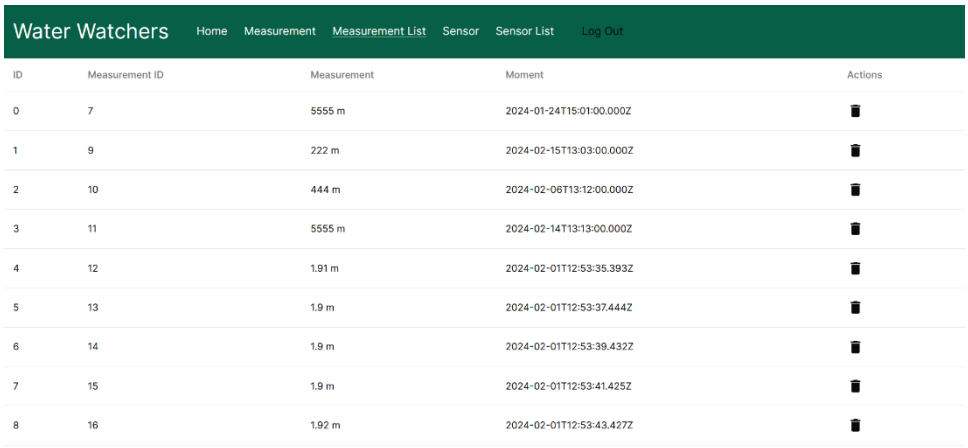


The Measurements

The Measurements tab is to add a warning to a specific chosen sensor and water height. After wich the user will be given a notification if it reaches or falls to the specified height.

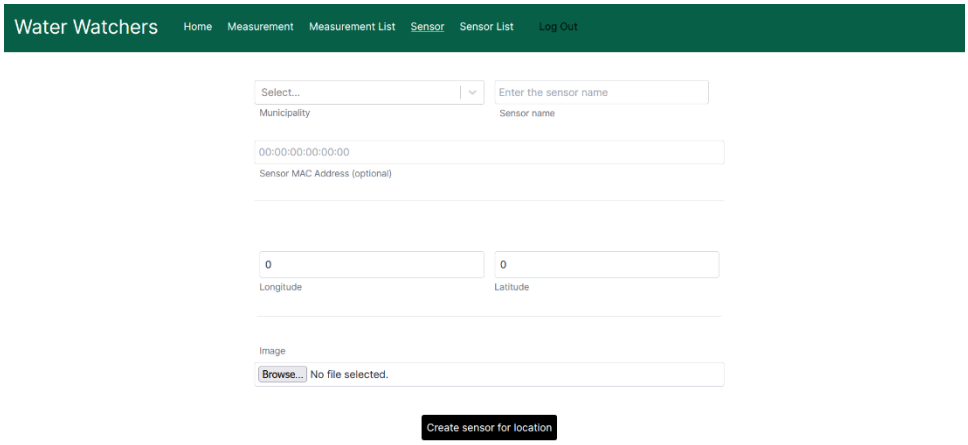


The List shows all sensors with a measurement.

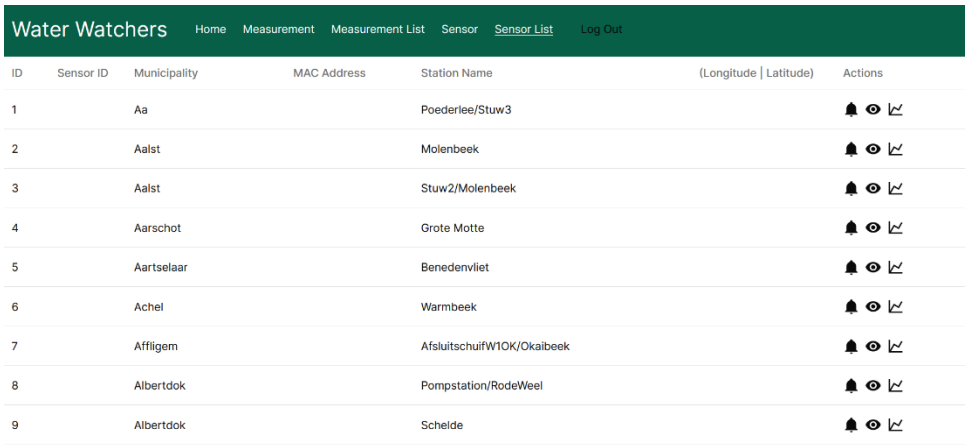


The sensor

The sensor tab is to add sensors, it requires a municipality, name, longitude, latitude and optionally a mac address and image.



The List shows all sensors.



Deployment

To deploy our web application, we decided to go in two different directions. For testing purposes Vercel was used is this is an easy way to host web applications but has its limitations. With these limitations in mind, we decided to build our own infrastructure on AWS. To build this infrastructure Terraform and Gitlab were used, terraform as a way to define our infrastructure as code and Gitlab was used for its easy-to-understand way to build CI/CD pipelines this ensured that there was a certain level of automatization to the setup of the infrasructure as well as the deployment.

Security measures

In the Node-Red backend, we used SQL prepare statements where possible when we insert data into the database. These statements predefine the structure and the data types of the data to be inserted before the query is executed. This way, we can severely limit an attacker's options during SQL injection attacks, as the insert will fail if the predefined parameters aren't met.

Results (benefits for stakeholders)

Through making use of our product, local governments in Flanders will now have a way to more accurately and swiftly analyze water levels of second grade water streams in Flanders. This will allow for quicker and better decision-making and the taking of measures against problems regarding water levels will be facilitated as we provide them with one centralized source of data and less personal communication will be needed to have everyone on the same page.

Installation manual

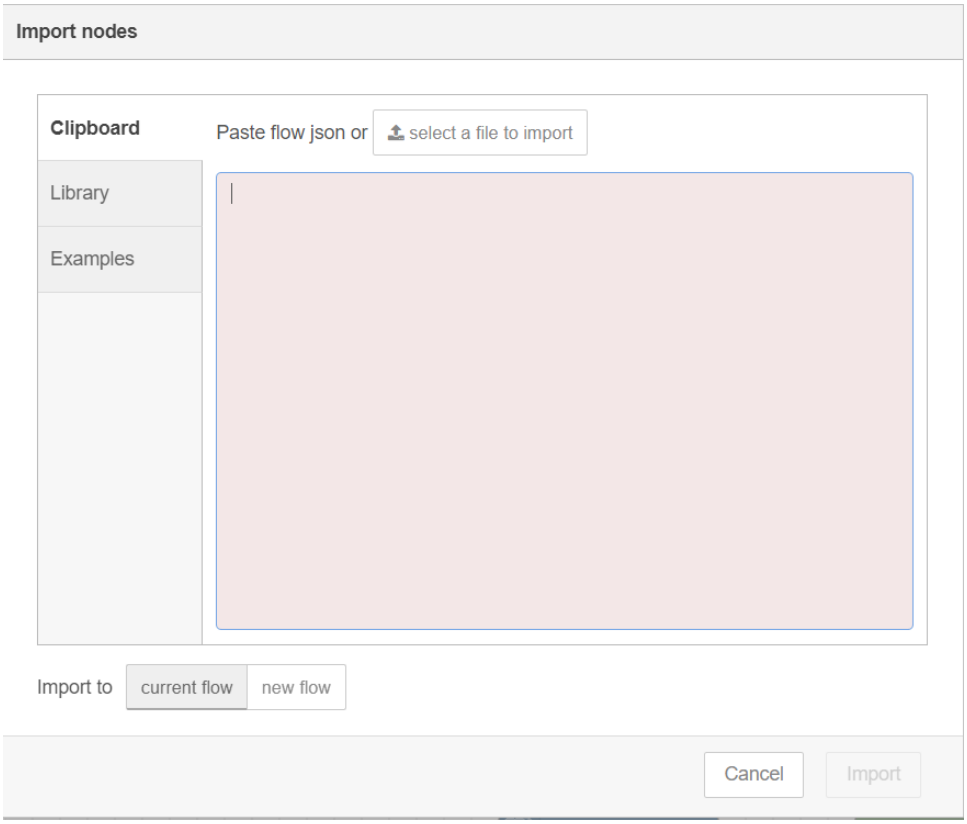
In this section we'll explain the steps one needs to take to recreate our solution.

Node-Red

To start off, create a new Node-Red project. Press *ctrl + i* on windows or *cmd + i* on mac whilst in your new project to import our configuration file added in the project deliverable.

You can either choose to import the file itself by choosing the *"select a file to import"* option, or you can paste the contents of the file in the textbox underneath the "*select a file to import"* button.

For the configuration of your postgreSQL database, start out by double-clicking any one



of the postgreSQL nodes you can find in one of the flows. Beneath is an example node:

The following popup will open:

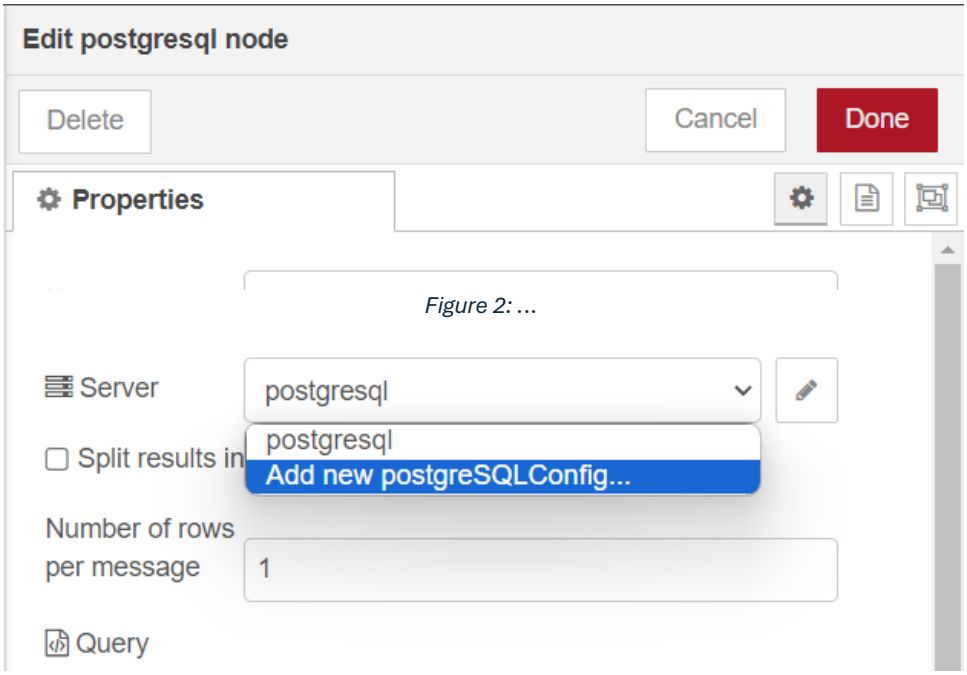
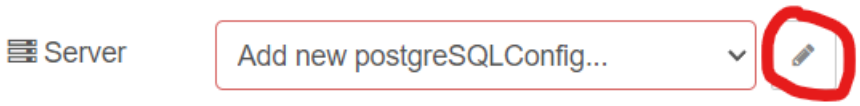


Figure 2: ...

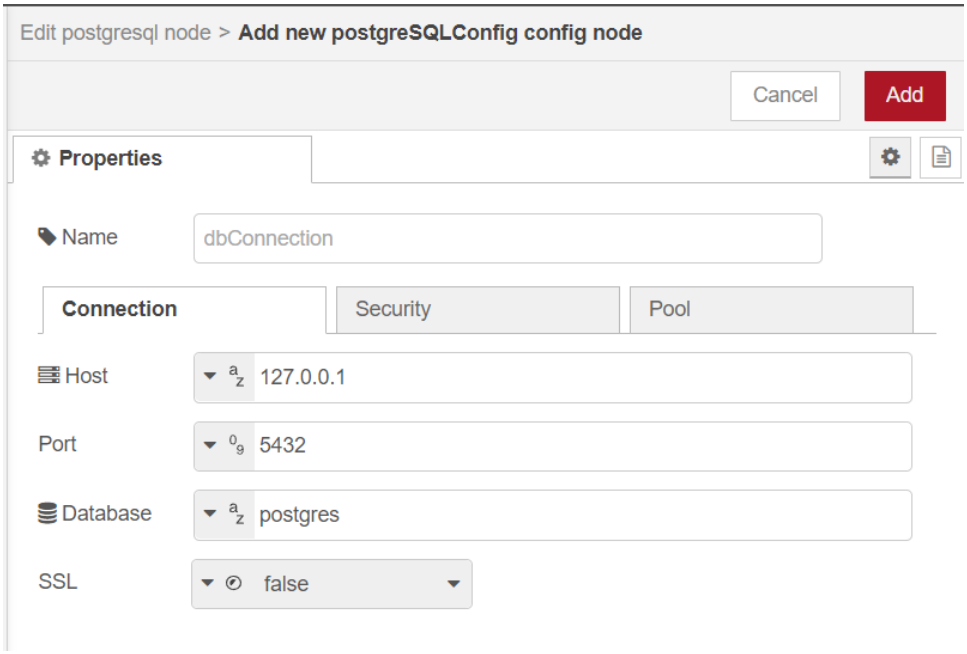
Figure 1: ...

In he

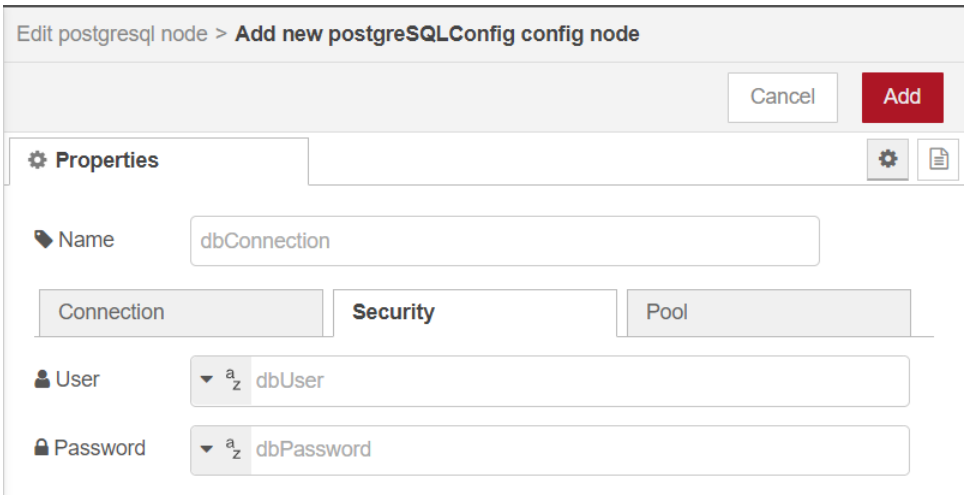
In here, click the *"Server"* dropdown and choose the option *"Add new postgreSQL Config..."*. After making the selection, click the pen icon to start adding a new database connection:



The following popup will open:

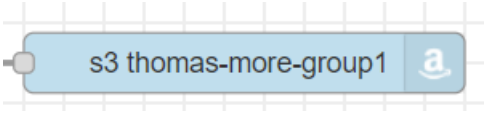


Change the values in the *"Connection"* tab (which you are in by default) to the specific values tied to your postgreSQL database. Then switch over to the *"Security"* tab and fill in your postgreSQL username and password:

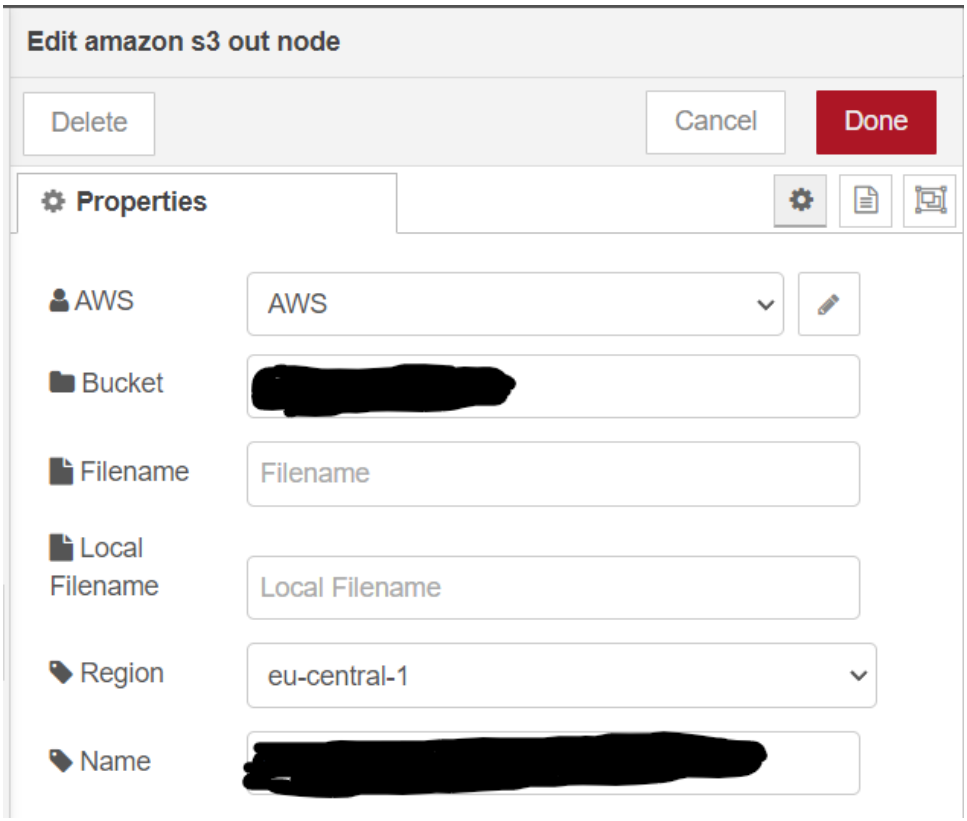


You can give the connection any name you like by filling in the *"Name"* field. When you're done filling everything in, click the red *"Add"* button at the top right and your database connection will be added to the list of connections you can pick from. To finish the configuration of your database, you'll have to **go into every postgreSQL node in all of the flows and change the *"Server"* of the node to the one you just added**.

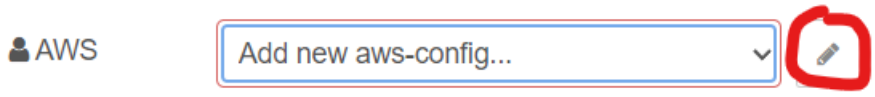
Next up, for the configuration of the AWS S3 bucket, find an S3 node (in the dataflow to s3 tab). It looks like this:



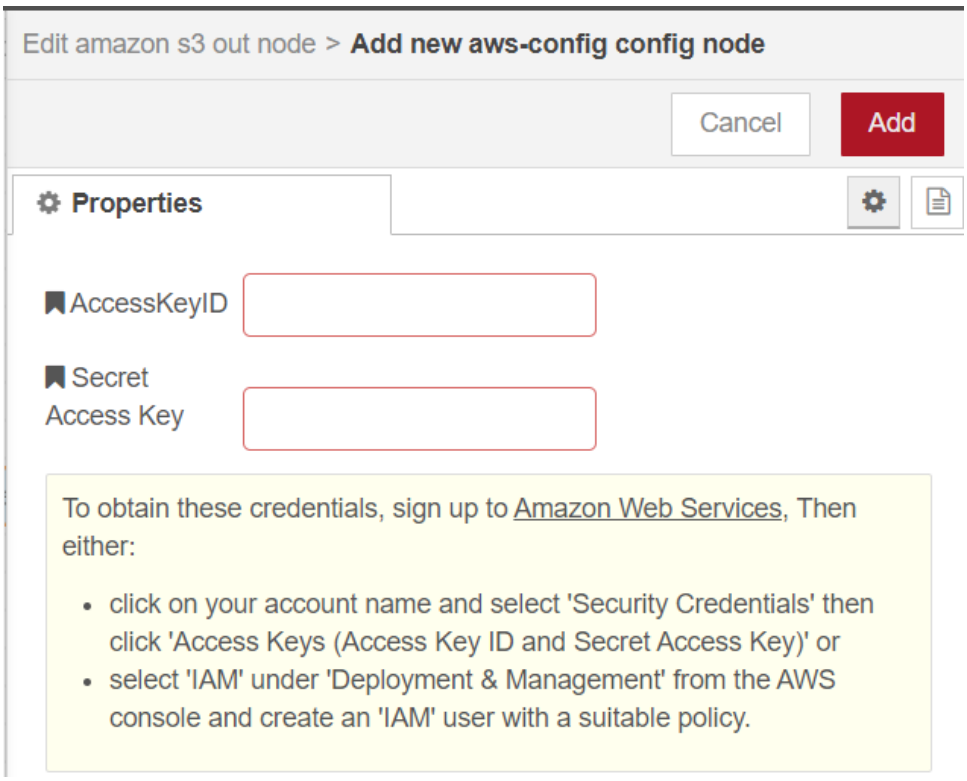
Dubble-click the node and following popup will appear:



In here, fill in the *"Region"*, the name of your bucket in *"Bucket"*. Next up, select that you want to add a new AWS configuration by clicking the option in the dropdown menu and then clicking the pencil next to the dropdown menu:



In the next popup, follow the instructions at the bottom and fill in your AWS credentials. When you're done click *"Add":*

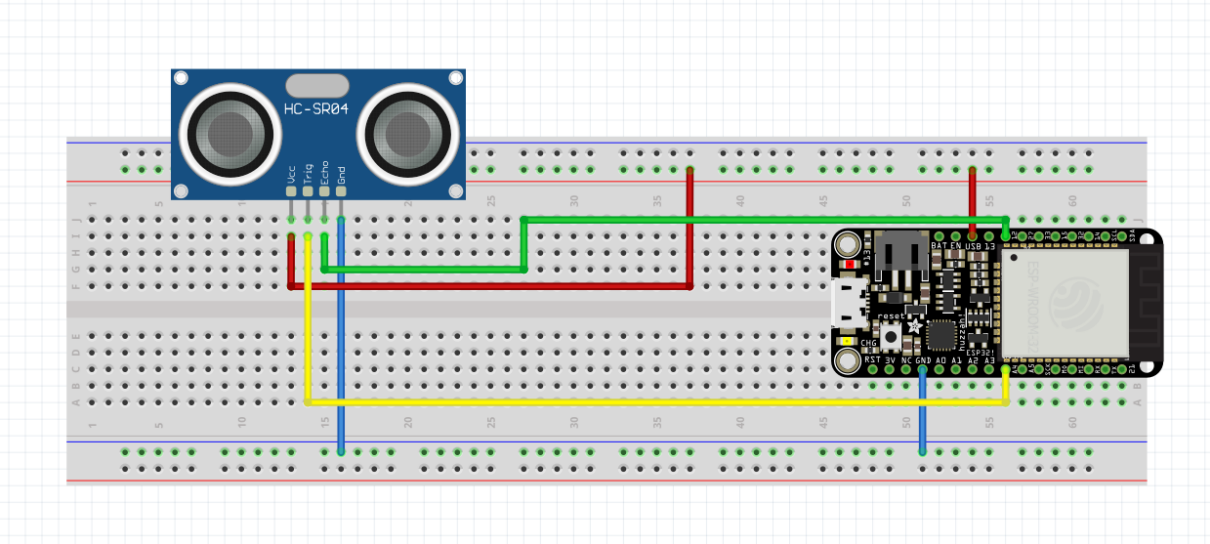


Lastly, select the AWS configuration you just made and from there, data normalized to the OSLO standards will be stored in your S3 bucket.

Assembly DIY sensor

When connecting this ultrasonic sensor, the voltage must come from the esp32 dev kit. There is a pin present called *"USB"*. This pin provides the 5-volt supply to the ultrasonic sensor and should also be connected "*Vcc*" pin.

Also, the "*GND*" pin of the esp32 dev kit must be connected to the "*GND*" pin of the ultrasonic sensor. The *"Trig"* pin of the ultrasonic sensor is connected to "*A4*" pin of the dev kit ("*GPIO36"*). The "*GPIO"* pin 12 of the dev kit is connected to "*Echo"* pin of this sensor.



Code DIY sensor

First, we'll discuss what libraries we used to program the microcontroller and what they are used for.

```
#include <ArduinoJson.h>
#include <ArduinoJson.hpp>
#include <WiFi.h>
#include <PubSubClient.h>
#include <Ultrasonic.h>
#include "main.hpp"
```

The include "*AruidnoJson*" library makes it possible for formatting the sensor data into JSON format. The "*wifi.h*" library makes it possible for connecting the esp32 devkit to a Wi-Fi access point. The "*PubSubClient.h*" header makes it possible for using the MQTT protocol for the esp32. The "*Ultrasonic.h*" library keeps the code cleaner for reading out this sensor. The "*main.hpp*" file is where all the confidential credentials are stored.

Next, we are going to declare which pin of esp32 will be used as "*Trigger*" pin and same goes for the echo pin:

```
const int trigPin = 17;
const int echoPin = 21;

Ultrasonic ultrasone(trigPin, echoPin);
```

The “*wifiClient”* is needed for esp32 to be connected to WIFI and “*PubSubClient”* is used to publish data to an MQTT topic and subscribe to an MQTT topic:

```
WiFiClient espClient;
PubSubClient client(espClient);
```

This function is called “*setup_wifi”*. This will ensure that the esp32 gets connected to an accesspoint:

```
void setup_wifi() {
  delay(100);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}
```

The purpose of the “*reconnect*” function is that if the connection to the MQTT server is broken, it will try to reconnect to it by logging in again.

```
void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    String clientId = "ESP32Client";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqtt_user, mqtt_password)) {
      Serial.println("connected");
      // client.subscribe("group1/#");  // Subscribe to the desired topic
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
```

```
      Serial.print(" - Trying again in 5 seconds");
      delay(5000);
    }
  }
}
```

The “*setup*” function is executed at the beginning of the program. It initiates the “*setup_wifi*” function, and upon its completion, retrieves the MAC address of the ESP32 DevKit. Once obtained, the MAC address is converted into a char array. Finally, the program establishes a connection to the MQTT server:

```
void setup() {
  Serial.begin(9600);
  while (!Serial);
  Serial.println(F("Wifi Test"));

  setup_wifi();

  // Haal het MAC-adres op en sla het op in de MAC-variabele
  String macAddress = WiFi.macAddress();
  macAddress.toCharArray(MAC, 18);

  client.setServer(mqtt_server, 1883);

}
```

The loop function is designed to continuously monitor the connection status with the MQTT server. If the connection is lost, it triggers the reconnect function.

Within the code, a Json document and a payload array are declared, each with a size of 100 characters. Additionally, a "*now*" variable of long integer type is created to store the current runtime in milliseconds.

The code includes a condition where, if the time difference between the "*now*" variable and the “l*astMsg*” is less than 2000 milliseconds, it enters an if statement. Within this statement, the ultrasonic sensor is read, and the data obtained is converted to meters and formatted into a Json payload.

Once the Json payload is prepared, it is sent to the MQTT topic specified in the Node-Red.

```
void loop() {
```

```
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  StaticJsonDocument<100> doc;
  static char payload[800];

  unsigned long now = millis();

  if (now - lastMsg > 2000) {
    lastMsg = now;
    float afstand = ultrasone.read();

    doc["result"] = afstand/100;

    doc["sensor_id"] = 75;
    doc["measurement"]="m";

    serializeJson(doc, payload);
    Serial.println(payload);

    // Stuur de payload om de 5 seconden
    client.publish("…", payload);
    Serial.println("Payload has been sent");
  }
}
```

Infrastructure

This section will cover how to set up the infrastructure for our web application

Vercel

To host on Vercel, all that is needed is an account and the GitHub repository where the app is located. To deploy the site to the internet from the repository, start a new project and follow the steps on the Vercel website.

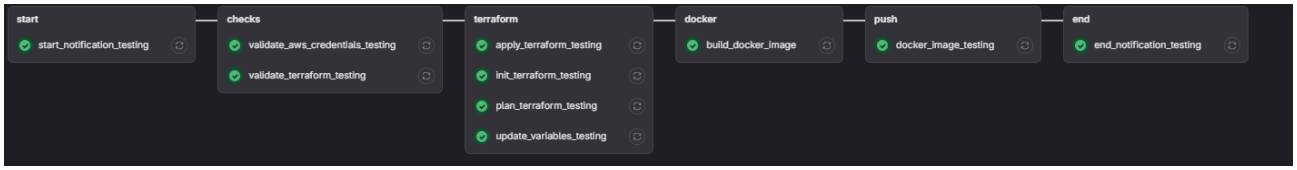
AWS

The deployment on AWS is automated for the most part, however it is essential to do a few checks before the infrastructure can be set up.

Since this works via a GitLab pipeline with environment variables it is important that these are present. Below is a list of all the environment variables that are used in the project:

- ACCESS_TOKEN_INFRASTRUCTURE
- AWS_ACCESS_KEY_ID
- AWS_DEFAULT_REGION
- AWS_SECRET_ACCESS_KEY
- AWS_SESSION_TOKEN
- DISCORD_URL
- DOCKER_HOST
- DOCKER_REGISTRY
- DOCKER_TAG
- ECR_REGISTRY
- IMAGE_TAG
- INFRASTRUCTURE_REGISTRY
- OUTPUT_ECR_NAME
- OUTPUT_ECR_URL
- OUTPUT_ECS_NAME
- OUTPUT_ECS_SERVICE_NAME REPOSITORY_URL_INFRASTRUCTURE
- SCRIPT
- TF_STATE_NAME
- TF_VAR_aws_access_key
- TF_VAR_aws_token
- TF_VAR_ec2_count

If all these variables are filled in correctly the pipeline should run. The pipeline will first send a notification to discord that it has started then it will check if the AWS credentials are valid, if these are correct the infrastructure will be build with Terraform. When the infrastructure is up and running the pipeline can also create and push a Docker image to the ECR, this image will run on the ECS to have a functioning web application



All the code made the infrastructure will be attached to this file.

The Web Application

In this section, we'll cover the steps on how to retrieve and install our web application.

The GitHub repository

Download the repository

https://github.com/syandelbart/watermonitor

The installation

Requires Node.js and NPM

Edit the .env.example into .env.local and fill in the blanks:

```
NODE_RED_API=""
API_USERNAME=""
API_PASSWORD=""
NEXT_PUBLIC_GRAFANA_URL=""
NEXT_PUBLIC_GOOGLE_MAPS_API_KEY=""
NEXT_PUBLIC_VAPID_PUBLIC_KEY=""
VAPID_PRIVATE_KEY=""
NEXT_PUBLIC_APPLICATION_URL=""
```

In the Terminal use the commands:

```
npm install
npm run dev
```

The image

Requires Docker

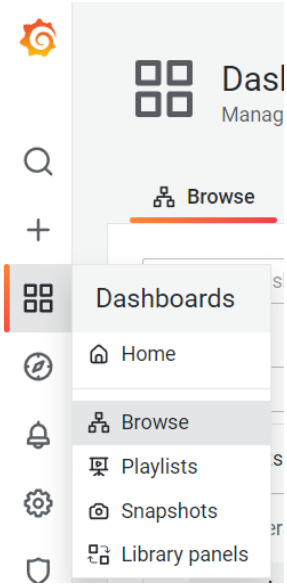
In the Terminal use the commands:

```
Docker run build
```

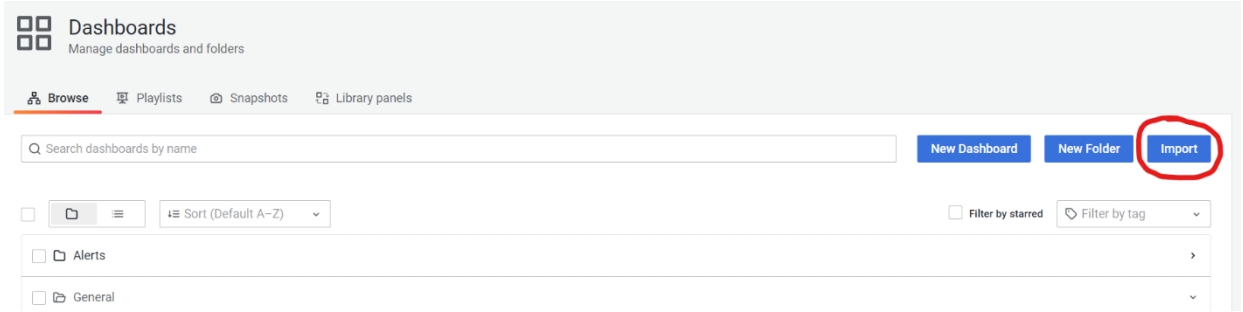
Grafana

To recreate our Grafana dashboard (if needed), you can import the Grafana configuration file we provided in the project deliverable in Grafana yourself.

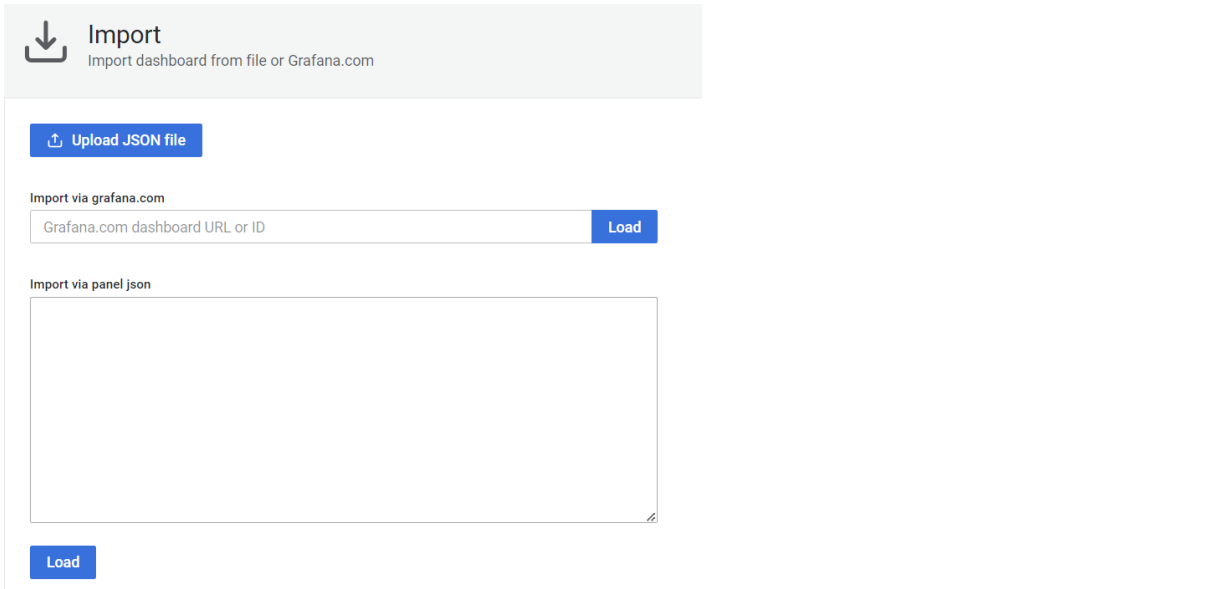
Start by navigating to the *"Browse"* tab:



Next up, you can select the *"import"* option from the menu on the right:



From here, using the JSON configuration file, you can either import the dashboard from that file or paste the contents of the file into the textbox:



Conclusion

Water Monitoring is a project designed to help local governments in Flanders keep track of water levels more effectively. The main goal is to provide a reliable platform for gathering, organizing, and visualizing water level data.

The project encompasses various aspects including the collecting, normalizing, storing and visualizing of data, and user interaction through a web application. Key technologies used include Node-Red, PostgreSQL, Grafana, NextJS, ESP32 microcontroller, MQTT protocol, and a ultrasonic sensor.

Through 4 weeks of meticulous planning and implementation, we have successfully achieved our must and should have project objectives, enabling users to make informed decisions based on accurate and timely data visualized on graphs.

In conclusion, the Water Monitoring project equips local governments with the tools needed to make informed decisions regarding water management, ensuring a more sustainable future for Flanders.