

BY

Team 3 Strategic Think Thank

Natan Van Braeckel, Joppe Dierckx, Emre Ekici,

Leander Van Bael, Matieu Du Jardin, Finn De Peuter



This document contains all the necessary information needed to set up the environment we created in Node-Red and how the web-app functions.

Table of contents

[1. How to install the Node-Red environment 3](#_Toc872733717)

[Main flow 3](#_Toc1195019534)

[Add province to station subflow 3](#_Toc1182922166)

[API endpoints 3](#_Toc1842156758)

[AlarmWaterNiveau Flow 5](#_Toc135569713)

[Fill warning levels Flow 9](#_Toc925639384)

[Subscribe to MQTT flow 10](#_Toc1163945150)

[2. How to add new endpoints in Node-Red 10](#_Toc1596976175)

[3. WaterHub DIY Sensor 11](#_Toc972926531)

[Code 13](#_Toc241428169)

[4. The postgresql structure 17](#_Toc232253181)

[5. Web application 18](#_Toc1486624035)

[Homepage 18](#_Toc732715824)

[Station dashboard 19](#_Toc352405297)

[Authentication 20](#_Toc1126180951)

[Comparing stations 21](#_Toc1781830571)

[Adding a station 22](#_Toc318284019)

[Adding a measurement 23](#_Toc1349872884)

[6. The Grafana dashboard 24](#_Toc1336525224)

[Homepage 24](#_Toc1574670017)

[Province 25](#_Toc1187663782)

[7. CI/CD pipeline 26](#_Toc555989568)

[Components of the Pipeline: 26](#_Toc68304981)

[The gitlab-ci.yml file 27](#_Toc1492094568)

[Stages: 27](#_Toc2084803106)

[Variables: 27](#_Toc941095743)

[Image: 27](#_Toc1519721296)

[Code: 27](#_Toc274722583)

[8. Dockerfile 30](#_Toc988996524)

[Build stage (builder): 30](#_Toc1980673374)

[Production stage (nginx): 30](#_Toc869801981)

[Nginx default.conf 30](#_Toc1578515750)

[9. Infrastructure 31](#_Toc282176357)

[Schema 31](#_Toc61156676)

[10. Terraform 32](#_Toc570902115)

[aws\_acm.tf 32](#_Toc1969153683)

[aws\_ecs.tf 32](#_Toc1557583877)

[aws\_loadbalancer.tf 32](#_Toc1469986984)

[aws\_secretsmanager\_secret.tf 32](#_Toc2049579733)

[aws\_vpc.tf 32](#_Toc1937850891)

[terraform.tf 32](#_Toc2032005118)

# How to install the Node-Red environment

For the Node-Red environment you need to log in to the right credentials of your Node-Red instance. After you need to import the flows provided in the files. In the upcoming information we will go over what each flow does and where you can change the things you need.

There are a few extra nodes that need to be installed on the palette manager so the node-red flows can be used:

* node-red-contrib-postgresql
* node-red-contrib-mqtt-broker
* node-red-contrib-nodemailer-adapter
* node-red-node-aws

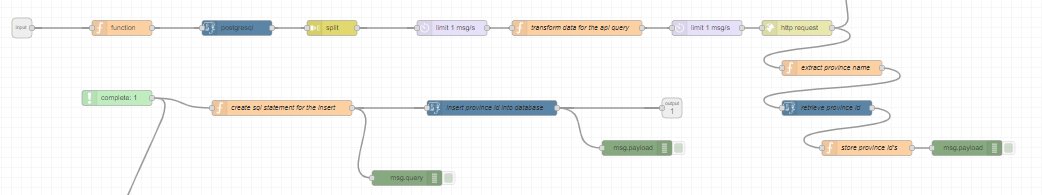
## Main flow

The main flow is the flow that retrieves the information from waterinfo.be and sends it to the database. The flow starts off with a call to the waterinfo api and then transforms the payload into a format so it can insert new stations into the database. After this is completed, the flow checks the station ID and uses these to insert the measurements table with the right station ID attached. If there were new stations inserted, the flow will move to the “add province to stations” subflow which will call an api with the longitude and latitude of the station and receive the correct province to add to the station table.

Afbeelding met tekst, schermopname, diagram, Lettertype

Automatisch gegenereerde beschrijving

### Add province to station subflow

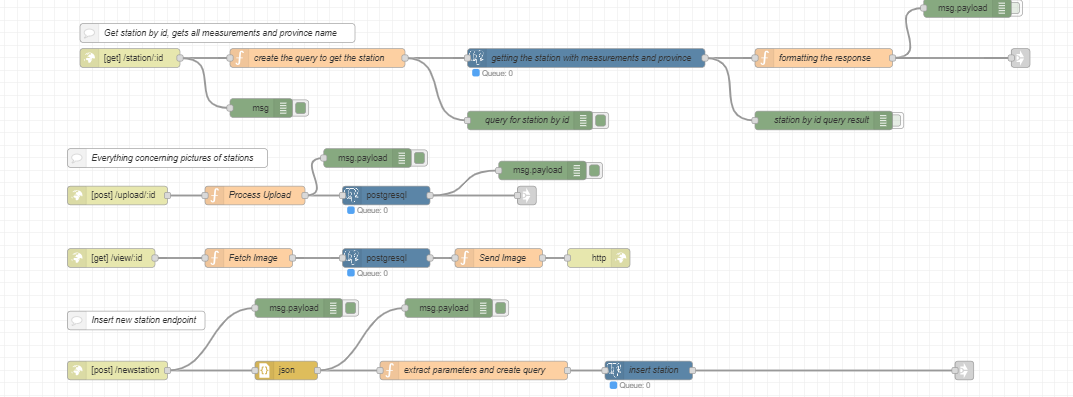


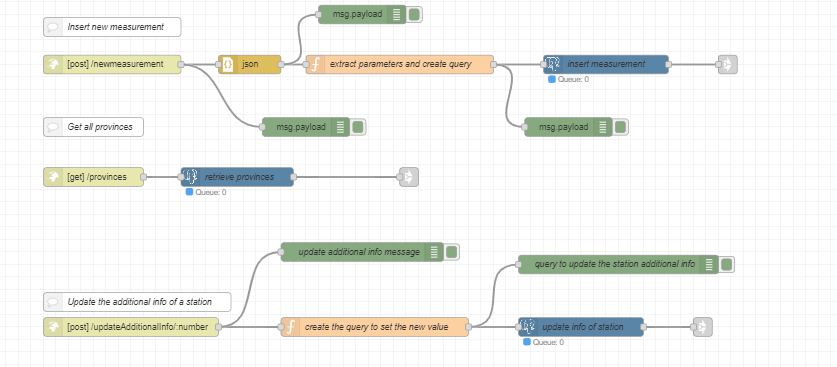
## API endpoints

This flow contains all the needed endpoints that the React-webapp needs to run on. They are set up in this way that there is no CORS (Cross Origin Resource Sharing) error. We will go into further detail on creating endpoints further down in the document.

These are the endpoints:

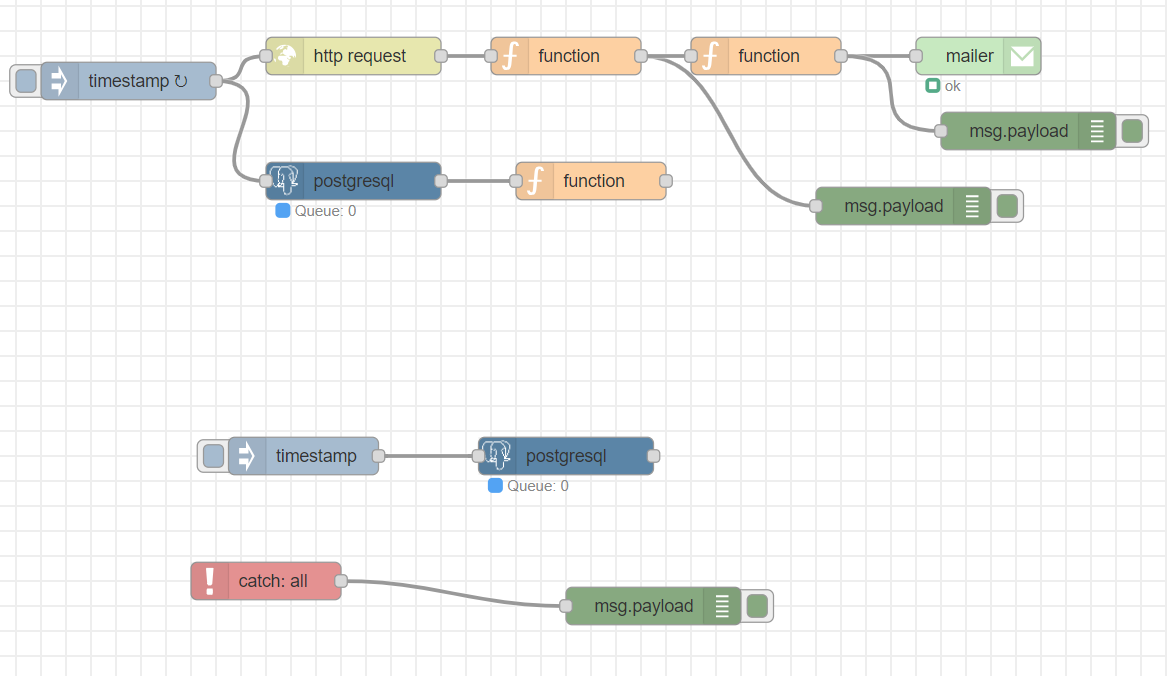
* GET /stations = retrieve all stations from the database
* POST /updateWarning/:id = update the warninglevel value for a station
* GET /station/:id = get the information on a specific station
* POST /upload/:id = upload a photo to a specific station
* GET /view/:id = view the photo of a specific station
* POST /newstation => post a new station
* POST /newmeasurement => post a new manual measurement
* GET /provinces = retrieve all the provinces
* POST /updateAdditionalInfo/:number = update the information of a specific station





All these flows come together at the end in the same function node where the headers are added.

## AlarmWaterNiveau Flow



This flow makes sure you receive an email on a set e-mailadres when the water level goes above the warning level. The timestamp node triggers every X hour the http request and postgresql node. You can change the interval by changing it in the timestamp node.

The postgresql node connects to the database and retrieves the Stationnumber and the warningvalue for the stations.

The function node that follows creates an object called ‘warningValues’ where the ‘station\_no’ is used as a key and the ‘warning\_value’ as a value. This object is saved in the global context of the Node-Red instance to be used in the future. The code in the function node is as follows:

// Assuming msg.payload contains the PostgreSQL response with warning values

var warningValues = {};

if (msg.payload) {

msg.payload.forEach(function (item) {

warningValues[item.station\_no] = item.warning\_value;

});

}

// Store warning values in global context

context.global.warningValues = warningValues;

return msg;

The HTTP Request node is used to retrieve the data from the API with a GET request. The function node after this HTTP request node has the following code:

// Assuming msg.payload contains the API response

var apiResponse = msg.payload;

// Check if the response is an array and has at least one element

if (Array.isArray(apiResponse) && apiResponse.length > 0) {

// Create a new array with objects containing all values

var enrichedData = apiResponse.map(function(item) {

return {

ts\_id: item.ts\_id,

timestamp: item.timestamp,

ts\_value: item.ts\_value,

station\_latitude: item.station\_latitude,

station\_longitude: item.station\_longitude,

station\_name: item.station\_name,

parametertype\_name: item.parametertype\_name,

ts\_name: item.ts\_name,

ts\_unitsymbol: item.ts\_unitsymbol,

station\_no: item.station\_no,

warning\_value: context.global.warningValues[item.station\_no] // Use warning value from the context

};

});

// Create a new message object with the array of objects

msg.payload = enrichedData;

} else {

// If the response is not as expected, set payload to null or handle accordingly

msg.payload = null;

}

return msg;

This Node-RED function processes the API response received from the HTTP request. The code checks if this response contains an array with at least one element. If this condition is met, a new array, 'enrichedData,' is created by selecting specific properties from each item in the original array and combining them into new objects. Within this process, a 'warning\_value' is added to each object, retrieved from the global context ('context.global.warningValues'). The resulting array of objects is then assigned to 'msg.payload' for further processing in the Node-RED flow.

The second function node contains the following code:

// Assuming msg.payload contains an array of objects

var apiResponse = msg.payload;

// Create an object to store critical stations

var criticalStations = {};

for (var i = 0; i < apiResponse.length; i++) {

// Get the current object

var currentObject = apiResponse[i];

// Your conditional check

if (currentObject.ts\_value > currentObject.warning\_value) {

// Store the critical station information in the object

criticalStations[currentObject.station\_no] = currentObject;

}

}

// Check if there are critical stations to send an email

if (Object.keys(criticalStations).length > 0) {

// Send a single email with information about all critical stations

sendEmail("strategicthinktank1@gmail.com", criticalStations);

}

return null; // Return null to stop further processing, assuming you've handled the conditions.

// Declaration of the sendEmail function

function sendEmail(recipient, criticalStations) {

// Create the email message

var emailMessage = {

from: 'strategicthinktank1@gmail.com',

to: recipient,

subject: 'Alarm! Critical Stations',

// Use HTML for the email body

html: '<h1 style="color: #2F2483;">Stations with Critical Values</h1>'

};

// Grafana base link

var grafanaBaseLink = 'https://grafana-group3.smartville-poc.mycsn.be/d/LD8McX5Iz/mathieutest?orgId=1&var-station\_name=';

// Add information about each critical station to the email text

for (var stationNo in criticalStations) {

var stationInfo = criticalStations[stationNo];

// Convert timestamp to a Date object

var timestamp = new Date(stationInfo.timestamp);

// Format date and time

var formattedTimestamp = timestamp.toLocaleString(); // Adjust the format as needed

// Add HTML content for each critical station with reduced spacing

emailMessage.html += '<h2 style="color: #E5007D;">Station ' + stationNo + '</h2>' +

'<strong style="color: #009EE3;">Tijd:</strong> ' + formattedTimestamp + '<br>' +

'<strong style="color: #009EE3;">Waarde:</strong> ' + stationInfo.ts\_value + 'm'+'<br>' +

'<strong style="color: #009EE3;">Station Naam:</strong> ' + stationInfo.station\_name + '<br>' +

'<strong style="color: #009EE3;">parametertype\_name:</strong> ' + stationInfo.parametertype\_name + '<br>' +

'<strong style="color: #009EE3;">ts\_name:</strong> ' + stationInfo.ts\_name + '<br>' +

// Add Grafana link as clickable with the text "Check Graph"

'<strong style="color: #009EE3;"><a href="' + grafanaBaseLink + encodeURIComponent(stationInfo.station\_name) + '">Check Graph</a></strong><br>' +

// Add Google Maps link as clickable

'<strong style="color: #009EE3;"><a href="https://www.google.com/maps?q=' +

stationInfo.station\_latitude + ',' + stationInfo.station\_longitude + '">Google Maps</a></strong><br><br>';

}

// Set the message object as the payload

msg.payload = emailMessage;

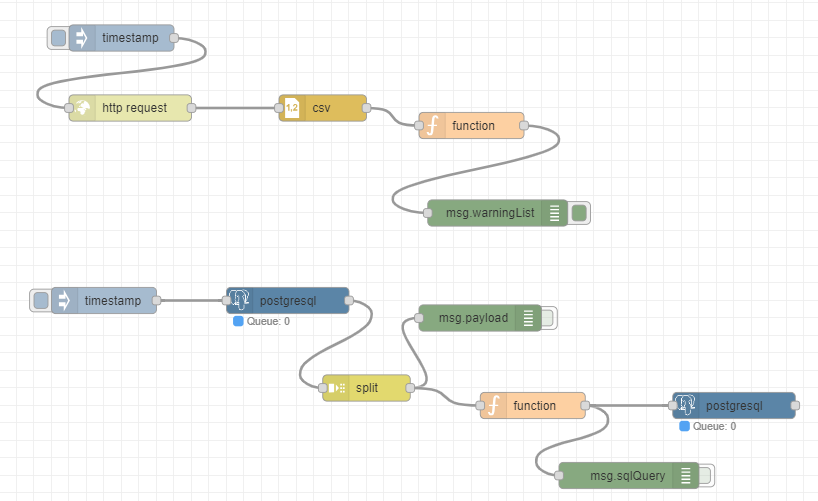
// Continue the flow to the email sending node

node.send(msg);

}

This Node-RED function processes an array of objects in msg.payload. It identifies critical stations where ts\_value exceeds warning\_value, stores their information, and sends an email if critical stations are found. The email includes details like timestamp, value, and clickable links to Grafana graphs and Google Maps locations for each critical station. The function then stops further processing by returning null.

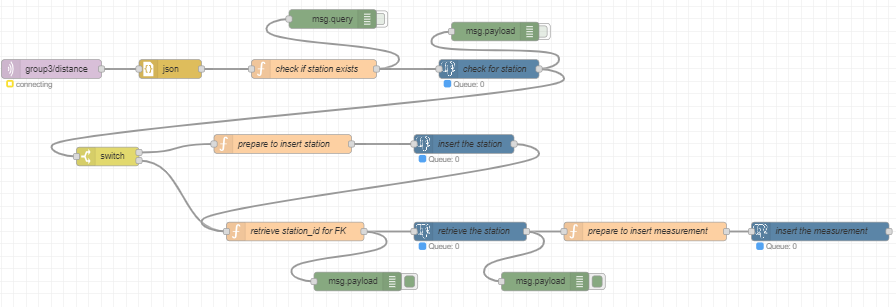
## Fill warning levels Flow



This flow was to insert the values for the warning levels that we generated based on our predictive model that takes the values that we knew and can predict an accurate warning level based on this.

The flow is not really used unless you need to mass insert new warning values because you retrained the model.

## Subscribe to MQTT flow



This flow is used to insert the measurements made by the diy sensor into the database. It works based on an MQTT publish and subscribe system.

# How to add new endpoints in Node-Red

When you want a new endpoint to access / publish / edit for example different information you will need to set up a new endpoint in Node-RED.

To do this, you head to the flow containing all the endpoints. Here you need at first a HTTP IN node to setup the actual link to the endpoint.

After this you will need to create a function to extract the information you need from what the HTTP request sends through.

Now there are 2 main things to keep in mind:

1. Simple HTTP requests
2. Headers

What do we mean by that. Basically, the Node-RED server is set up in a way that “complicated” HTTP requests, can’t pass through as their preflight request gets a 401 response. A way to work around this is, by sending only simple HTTP requests (GET or POST). This means that if you need a PUT request, you will send a POST request, but handle in Node-RED that it is a PUT request.

Secondly, the content-type, there are only a few content-types that are allowed in simple requests. We decided to use plain/text, as this is the easiest to convert things back and forth. So, when you send the request, it will be sent in plain-text and Node-RED handles the conversion back to the right format like json for example.

Now for the headers, you will need an access-control-allow-origin header. These need to be manually added with a function node in Node-RED. To do this you only need to connect the last node to the node that has the function to add the headers.

msg.headers = {

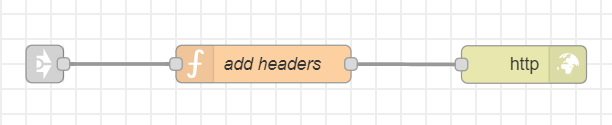
'Access-Control-Allow-Origin': ‘link to the origin’,

'Access-Control-Allow-Methods': 'GET,HEAD,PUT,PATCH,POST,DELETE',

'Access-Control-Allow-Headers': 'Authorization',

'Access-Control-Allow-Credentials': 'true',

};



# WaterHub DIY Sensor

**Necessary tools:**

1. ESP32 DevKit V1
2. Ultrasonic HS-SR04 Sensor
3. 4x Female-Female jumper cables
4. USB to micro-USB cable
5. Arduino IDE

**How it works:**

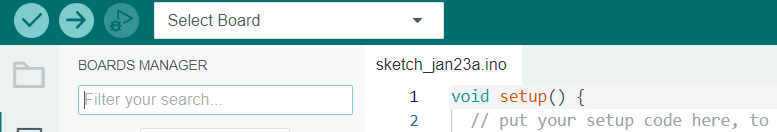
This document is to help you set up your own sensor and start recording your own water level data using an ESP32 microcontroller and an ultrasonic sensor to measure water levels and send the data to a remote server using the MQTT protocol. You need to connect the ESP32 to your Wi-Fi network and the ultrasonic sensor, following the provided instructions. The ESP32 periodically triggers the sensor (once every hour), calculates the distance based on the response, and sends this data, along with additional information which you provided such as your station name and location, to an MQTT broker. The information is then accessed, processed, analyzed and used in creating dashboards and visualizations. You will then be able to view your data on the web app by selecting your sensor from the options.

**Assembling the sensor:**

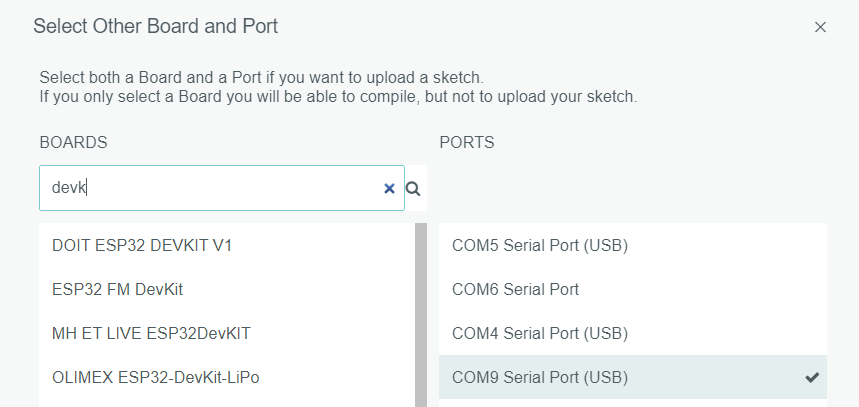
1. Connect the GND pin of your ESP32 to the GND pin of your sensor
2. Connect the ECHO pin of your ESP32 to the D22 pin of your sensor
3. Connect the TRIG pin of your ESP32 to the D23 pin of your sensor
4. Connect the VIN pin of your ESP32 to the VCC pin of your sensor

**Preparing your code for your sensor:**

1. Download and install the Arduino IDE : <https://www.arduino.cc/en/software>
2. Navigate to **LINK HERE**
3. Fill in the form with the correct information to receive the code you will need to use on your sensor.
4. Once you have your code, open the Arduino IDE
5. Connect your ESP32 Devkit board to your computer with the USB cable
6. Choose your board from the top left dropdown menu



1. Select the DOIT ESP32 Devkit V1 option and the appropriate port



1. Paste the code you received into the sketch area and hit the upload button on the top left next to where you selected your board.
2. Now on the top right you can click on the “Serial Monitor” button to view the output of your sensor.

### Code

#include <Arduino.h>

#include <WiFi.h>

#include <PubSubClient.h>

#include <ArduinoJson.h> // Include ArduinoJSON library

// WiFi credentials

const char\* ssid = "";

const char\* password = "";

// MQTT broker details

const char\* mqttServer = "mqtt.mycsn.be";

const int mqttPort = 1883;

const char\* mqttUsername = "group3";

const char\* mqttPassword = "";

const char\* mqttTopic = "group3/distance";

// Sensor details

const int trigPin = 23; // GPIO pin for the trigger

const int echoPin = 22; // GPIO pin for the echo

// Variables to store sensor data

long duration;

int distance;

// Additional properties

const char\* station\_name = "Geel/ThomasMore";

const char\* station\_no = "TM\_G\_03";

const float latitude = 51.160950;

const float longitude = 4.961660;

// WiFi and MQTT client instances

WiFiClient espClient;

PubSubClient client(espClient);

// Function prototype

void callback(char\* topic, byte\* payload, unsigned int length);

// Variables to manage time intervals

unsigned long previousMillis = 0;

const long interval = 3600000; // 1 hour

void setup() {

// Start the serial communication

Serial.begin(9600);

// Connect to Wi-Fi

WiFi.begin(ssid, password);

while (WiFi.status() != WL\_CONNECTED) {

delay(1000);

Serial.println("Connecting to WiFi...");

}

delay(1000); // Add a delay after successful connection

Serial.println("Connected to WiFi");

// Set the trigger pin as an output

pinMode(trigPin, OUTPUT);

// Set the echo pin as an input

pinMode(echoPin, INPUT);

// Connect to MQTT broker

client.setServer(mqttServer, mqttPort);

client.setCallback(callback);

while (!client.connected()) {

Serial.println("Connecting to MQTT...");

if (client.connect("ESP32Client", mqttUsername, mqttPassword)) {

Serial.println("Connected to MQTT");

} else {

Serial.print("Failed with state ");

Serial.print(client.state());

delay(2000); // Add a delay before retrying

}

}

// Subscribe to the MQTT topic

client.subscribe(mqttTopic);

}

void loop() {

// Get the current time

unsigned long currentMillis = millis();

// Check if 1 hour has elapsed since the last transmission

if (currentMillis - previousMillis >= interval) {

// Save the current time

previousMillis = currentMillis;

// Trigger the ultrasonic sensor

digitalWrite(trigPin, LOW);

delayMicroseconds(2);

digitalWrite(trigPin, HIGH);

delayMicroseconds(10);

digitalWrite(trigPin, LOW);

// Measure the duration of the pulse on the echo pin

duration = pulseIn(echoPin, HIGH);

// Calculate the distance in centimeters

distance = duration \* 0.034 / 2;

// Create a JSON object

StaticJsonDocument<200> jsonDocument; // Adjust the size according to your data

jsonDocument["distance"] = distance;

jsonDocument["name"] = "group3\_selfmade";

jsonDocument["station\_name"] = station\_name;

jsonDocument["station\_no"] = station\_no;

jsonDocument["latitude"] = latitude;

jsonDocument["longitude"] = longitude;

// Convert the JSON object to a string

char payload[1000]; // Adjust the size according to your data

serializeJson(jsonDocument, payload);

// Publish the JSON payload to the MQTT topic

client.publish(mqttTopic, payload);

// Print the distance to the Serial Monitor

Serial.print("Distance: ");

Serial.print(distance);

Serial.println(" cm");

}

// Handle MQTT messages

client.loop();

}

// Callback function

void callback(char\* topic, byte\* payload, unsigned int length) {

// Handle incoming MQTT messages (if needed)

// This function is called when a message is received on the subscribed topic

}

# The postgresql structure

Afbeelding met tekst, Lettertype, nummer, schermopname

Automatisch gegenereerde beschrijving

Important to note are the alternate keys in the ERD.

Station has station\_no as an alternate key. The station\_no comes from the waterinfo.be API, and we presume that it is some kind of identifier. Because the option of adding new stations is available, we opted to use our own id field. The station\_no has to be unique for incoming stations, so that there are no duplicate entries.

The alternate key for Measurement is the combination of timestamp, timeseries\_name and station\_id. This ensures that no duplicate measurements get added to the database.

# Web application

We also made a web application for showing the dashboard of stations, adding a station and filling in a measurement you see. You can also change information about a station, by changing the warning value, adding an image, or supplying additional information.

## Homepage

When you open our web application, you will be at the home page where you be able to filter by a station, a number or a province. If you apply a filter, the table will change according to the filters you have added. You also see a map with all the sensors on it. You can zoom in on a province and then you see that you can go to the sensor itself. The raindrops you see are the sensors. You can also zoom in on your location to see if there are any sensors nearby.

Afbeelding met tekst, schermopname, kaart

Automatisch gegenereerde beschrijving

## Station dashboard

In the table you can also click on a name of a station and then you go to a page with all the details of that station. It displays a line chart of recent data, the location of the sensor, and a table of the measurements. Along with it, you can find the warning value, an uploaded image and additional station info. If you are logged in, you can change the warning value, add a measurement and change the image.

Afbeelding met tekst, kaart, schermopname, software

Automatisch gegenereerde beschrijving

## Authentication

We’ve implemented authentication through Auth0. If you want to add a station or you want to insert a measurement, you have to log in. If you are not logged in, you can’t get to these pages. You won’t be able to update the warning value or the image of a station either.

Afbeelding met tekst, schermopname, software, multimedia

Automatisch gegenereerde beschrijving

## Comparing stations

To compare stations, you first need to make a selection. You can search for stations by filtering and selecting them in the dropdown or by selecting them on the map. You can choose as many stations as desired. Chosen stations will be highlighted. Pressing the compare stations button will bring you to the comparison page.

Afbeelding met tekst, schermopname, kaart

Automatisch gegenereerde beschrijving

The main focus of the comparison page is the line chart where recent measurements of all selected stations will be displayed. You can select the measurements of the past 6, 12 or 24 hours. It’s possible to download this graph as an SVG, PNG, or to save the data in CSV format through the burger icon.

Individual line charts are displayed below, where you can navigate to the overview page of the respective stations.

Afbeelding met tekst, schermopname, lijn, nummer

Automatisch gegenereerde beschrijving

## Adding a station

If you are logged in, you can go to the add station tab and then you can add a station. You can fill in the name of the sensor, on the map you can click on the location where the sensor is installed, the province, the device type, the data provider, the sensor number and the sensor warning level. When you have filled in everything, you can click on the submit. If you click on the submit button, the sensor will be added immediately.

Afbeelding met tekst, kaart, schermopname, diagram

Automatisch gegenereerde beschrijving

## Adding a measurement

If you want to add a measurement for a specific sensor, you can go to the insert measurement tab. On the page you see a form where you can choose the province where the sensor is in, the name of the sensor, the time series, the unit symbol, the value you have measured and the timestamp. If you have filled in everything, you can click on the submit button. If you click on the submit button, the value will be added immediately.

Afbeelding met tekst, schermopname, nummer, Lettertype

Automatisch gegenereerde beschrijving

# The Grafana dashboard

We also made a Grafana dashboard with all the data we have. If you go to the grafana dashboard, you can go to ‘cipal dashboard’ and then you can click the homepage to start.

## Homepage

The homepage displays 2 tables with content of everything in the database.

Afbeelding met schermopname, tekst, software, Multimediasoftware

Automatisch gegenereerde beschrijving

If you go to the dropdown on the right, you see all the provinces which have sensors.

Afbeelding met tekst, schermopname, Lettertype

Automatisch gegenereerde beschrijving

## Province

If you click on a province, you see all the values of a sensor over time, the place where the sensor is and a table with all the values that sensor has.

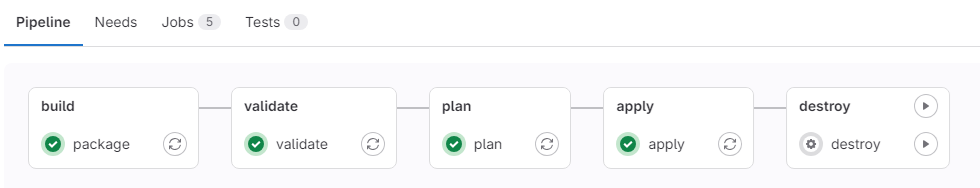
Afbeelding met tekst, schermopname, kaart, Multimediasoftware

Automatisch gegenereerde beschrijving

If you want to change the sensor you can just click on the dropdown with the name of the sensor. This dropdown can you find above the title of the dashboard.

Afbeelding met tekst, schermopname, software, Multimediasoftware

Automatisch gegenereerde beschrijving

7. CI/CD pipeline   
  


To automate the deployment of the infrastructure, a pipeline was used in this case this one in GitLab. A pipeline allows processes to be automated such as in this case building the Docker container and setting up the infrastructure in AWS.

When changes in the main repository are detected, the GitLab pipeline is automatically started and goes through all defined phases, with each phase representing a specific task, such as building the Docker image, planning, and setting up the infrastructure. The pipeline is defined in the repository in gitlab in the gitlab-ci.yml file.

## Components of the Pipeline:

* The Build Stage: This is the first stage of the pipeline and is triggered on every push/merge to main. In this stage, the Docker engine is installed and configured. A Docker image is built based on the Docker file present in the repository. In the process, a configuration file ('config.json') is also generated and copied into the source code. The resulting Docker image is than pushed to the GitLab Container Registry.
* The Validate Stage: The stage follows the previous one. Terraform is initialized using GitLab as the backend for storing the Terraform state. This file stored in GitLab and keeps track of the state of the infrastructure and configuration. It also makes sure that all dependencies are installed. After this, the code is checked for any errors.
* The Plan Stage: In this stage, the plan is generated based on the current state of the infrastructure and Terraform configuration. Terraform calculates what actions are needed to achieve the desired infrastructure configuration and displays the proposed changes.
* The Apply Stage: This is the final step that is performed automatically. The Apply Stage applies the changes to the infrastructure.
* The Destroy Stage: This stage can only be performed manually. If infrastructure is no longer needed, this stage can be activated to destroy the entire infrastructure created by Terraform.

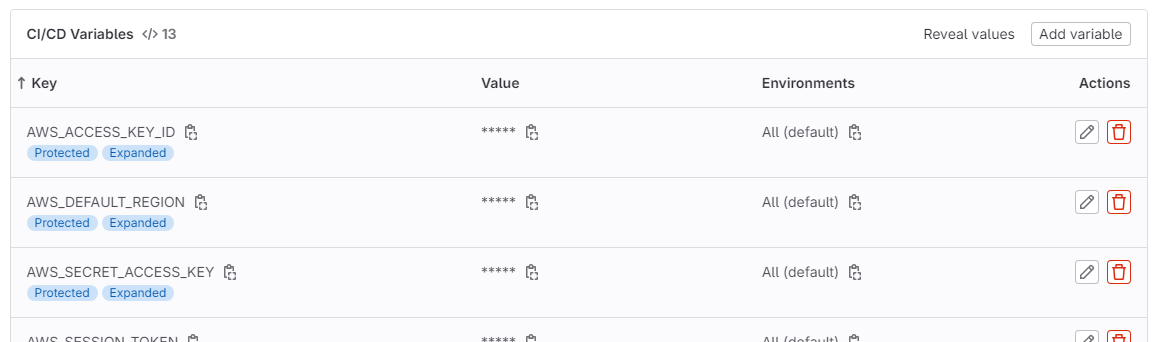
## The gitlab-ci.yml file

Stages:

These are the different phases of the pipeline.

### Variables:

These are variables that are passed to the pipeline and are securely stored in GitLab variables. These are used to perform the necessary configurations needed in the pipeline. Such as Terraform, Docker image, GitLab, AWS. These variables mainly contain credentials needed to build, deploy and manage the infrastructure.



* AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY, AWS\_SESSION\_TOKEN: These variables are the credentials used to configure AWS services.
* AWS\_DEFAULT\_REGION: This is the region where the infrastructure is implemented.
* GITLAB\_DEPLOY\_USER, GITLAB\_DEPLOY\_TOKEN: These variables are provided by Terraform and stored in AWS Secret Manager so that AWS ECS can populate the Docker image va, GitLab Container Registry.
* GITLAB\_USERNAME, GITLAB\_ACCESS\_TOKEN: These variables are used by terraform for initializing the backend and saving terraform state file.
* CLOUDFLARE\_API\_TOKEN, CLOUDFLARE\_ZONE\_ID: These variables cause the CNAME record to be updated.
* NODE\_RED\_BASE\_URL: This is the web address of the Node-RED service.

### Image:

The pipeline also needs to execute actions for this is used in gitlab runners, This is a Docker image that acts as the operational environment for executing commands.

### Code:

stages:

- build

- validate

- plan

- apply

- destroy

variables:

TF\_VAR\_aws\_access\_key: $AWS\_ACCESS\_KEY\_ID

TF\_VAR\_aws\_secret\_key: $AWS\_SECRET\_ACCESS\_KEY

TF\_VAR\_aws\_region: $AWS\_DEFAULT\_REGION

TF\_VAR\_aws\_token: $AWS\_SESSION\_TOKEN

TF\_VAR\_gitlab\_deploy\_token\_username: $GITLAB\_DEPLOY\_USER

TF\_VAR\_gitlab\_deploy\_token\_token: $GITLAB\_DEPLOY\_TOKEN

NODE\_RED\_BASE\_URL: $NODE\_RED\_BASE\_URL

NODE\_RED\_USERNAME: $NODE\_RED\_USERNAME

NODE\_RED\_PASSWORD: $NODE\_RED\_PASSWORD

USERNAME: $GITLAB\_USERNAME

ACCESS\_TOKEN: $GITLAB\_ACCESS\_TOKEN

PROJECT\_ID: $CI\_PROJECT\_ID

TERRAFORM\_DIR: ${CI\_PROJECT\_DIR}/infrastrucutre

TERRAFORM\_STATE\_NAME: ${CI\_PROJECT\_NAME}.tfstate

GITLAB\_TF\_ADDRESS: <https://gitlab.com/api/v4/projects/$PROJECT_ID/terraform/state/$TERRAFORM_STATE_NAME>

TF\_VAR\_cloudflare\_api\_token: $CLOUDFLARE\_API\_TOKEN

TF\_VAR\_cloudflare\_zone\_id: $CLOUDFLARE\_ZONE\_ID

TF\_VAR\_pipeline\_tag: $CI\_PIPELINE\_IID

package:

stage: build

before\_script:

- apk add --update docker

image: docker:20.10.12

services:

- docker:20.10.12-dind

script:

- echo "{\"nodeRedBaseUrl\":\"$NODE\_RED\_BASE\_URL\",\"nodeRedUsername\":\"$NODE\_RED\_USERNAME\",\"nodeRedPassword\":\"$NODE\_RED\_PASSWORD\"}" > ./src/config.json

- docker build -t $CI\_REGISTRY\_IMAGE:$CI\_PIPELINE\_IID -t $CI\_REGISTRY\_IMAGE:latest .

- echo $CI\_REGISTRY\_PASSWORD | docker login -u $CI\_REGISTRY\_USER $CI\_REGISTRY --password-stdin

- docker push $CI\_REGISTRY\_IMAGE:$CI\_PIPELINE\_IID

- docker push $CI\_REGISTRY\_IMAGE:latest

only:

- main

image:

name: hashicorp/terraform:latest

entrypoint: [""]

.common\_before\_script: &common\_before\_script

before\_script:

- cd ${TERRAFORM\_DIR}

- terraform init -backend-config="address=$GITLAB\_TF\_ADDRESS" -backend-config="lock\_address=$GITLAB\_TF\_ADDRESS/lock" -backend-config="unlock\_address=$GITLAB\_TF\_ADDRESS/lock" -backend-config="username=$USERNAME" -backend-config="password=$ACCESS\_TOKEN" -backend-config="lock\_method=POST" -backend-config="unlock\_method=DELETE" -backend-config="retry\_wait\_min=5"

validate:

stage: validate

<<: \*common\_before\_script

script:

- terraform validate

only:

- main

plan:

stage: plan

<<: \*common\_before\_script

script:

- terraform plan

only:

- main

apply:

stage: apply

<<: \*common\_before\_script

script:

- terraform apply -auto-approve

only:

- main

destroy:

stage: destroy

<<: \*common\_before\_script

script:

- terraform destroy -auto-approve

when: manual

only:

- main

# 8. Dockerfile

The Dockerfile has the code to build a Docker image that executes a Web application so the web application can be served. The process is divided into two main steps:

### Build stage (builder):

In this stage, a temporary container is created in which the application is compiled the required software and libraries are installed.

### Production stage (nginx):

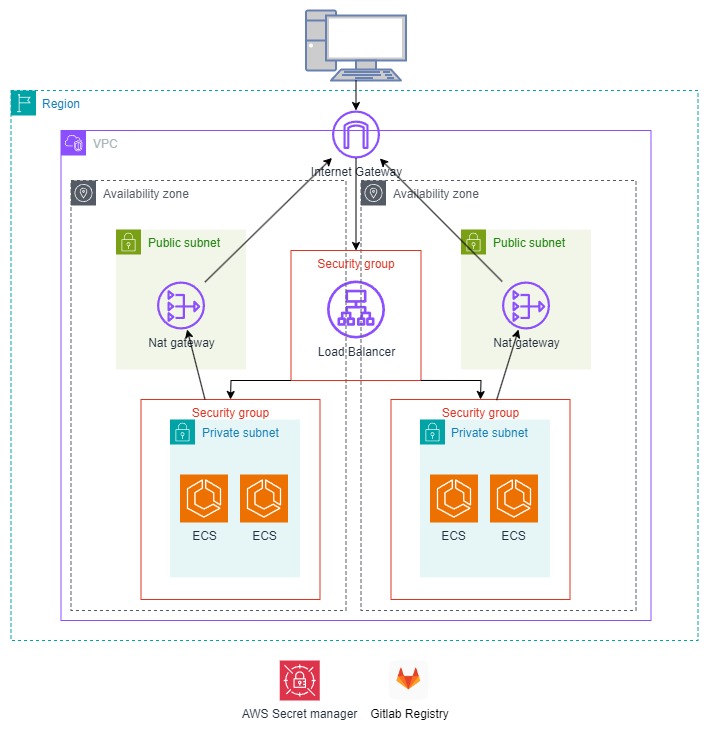
In this stage, an Nginx web server is implemented. The files from the previous contianer are copied into the nginx container. The nginx image will handle alle traffic on port 80 (HTTP) and serve the webpage.

## Nginx default.conf

In the Nginx default.conf, the line "try\_files $uri $uri/ /index.html" was added to support the routing of a React application. This rule will cause Nginx to forward requests to index.html and allow the application to handle page routing.

# 9. Infrastructure

## Schema

$

As shown in the diagram, there is a VPC with 4 subnets, of which 2 are private subnets and 2 are public subnets. The nginx container is inside the private subnet.

The image is pushed through the pipeline to Gitlab's Container Registry. This image is pulled by ECS, which runs Fargate. The credentials for the repository are stored in AWS Secret Manager. ECS can be accessed via the load balancer. The load balancer is accessible on the Internet and balances traffic between the different ECS instances.

# 10. Terraform

The infrastructure is automatically set-up by terraform. The code can be found in the repository in the folder infrastructure. Every file in terraform is needed to handle a certain task.

## aws\_acm.tf

This file defines a Cloudflare record for a CNAME record pointing to an AWS Load Balancer (LB). It takes care of managing DNS settings for the application.

## aws\_ecs.tf

This file defines the ECS cluster, task definition and service. It creates an ECS cluster with container insights enabled. It also creates an ECS task definition and service to execute containers. These containers are executed as Fargate tasks and have specific CPU and memory requirements. The container definition also includes configuration for logging and access to Secrets Manager for secret data.

## aws\_loadbalancer.tf

This file defines an AWS Load Balancer, a target group, listener and associated security groups. The load balancer distributes traffic among containers run by ECS services. It also includes configuration for health checks and sticky sessions, which ensures session persistence and does not switch between ecs.

## aws\_secretsmanager\_secret.tf

This file generates a Secrets Manager in aws and contains the credentials for being able to pull the docker image this is stored in gitlab repository. It also gives ecs access to this data.

## aws\_vpc.tf

This file defines an AWS Virtual Private Cloud (VPC) with associated subnets. It creates both public and private subnets, as well as NAT gateways for accessing the Internet from private subnets.

## terraform.tf

This file configures the Terraform provider for AWS and Cloudflare. It also contains the backend configuration for storing the Terraform state in gitlab.