# Code: Concept & Problems

### Goals

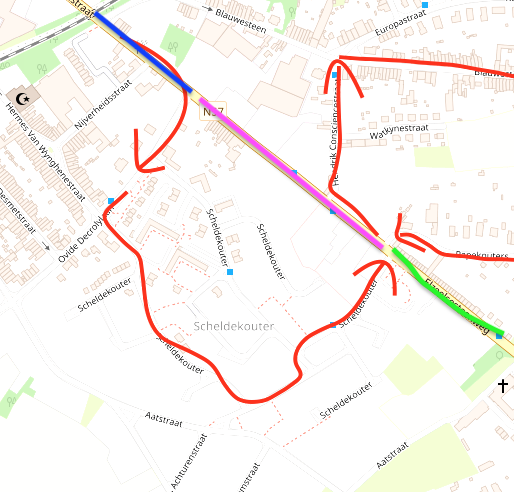
* Acquire the right data:
  + Data where a street is divided in parts each leading to another road. A garbage truck ends up multiple times in the same street sometimes.
* Visualisation & user Interface:
  + All parts of roads should be individual parts which can be routed to and be selected
  + A road should be checkable if the app failed to register the trucks progress
  + Make it possible to transfer a road to another driver (or optimally a part)
* Features and functionalities:
  + This map should optimally be offline, not depending on good internet connection
    - i.e. download an area or map using a custom api
  + Progress tracking for the driver: know how much of the street has been finished. This will require some lookup system to identify nodes out of coordinates
  + All basic functionalities should be able to be performed by an admin panel

Upgrades:

* Route Optimisation: AI learning system that can optimise the route out of statistics that are already being collected.
* Progress tracking: Currently progress tracking is not optimised in the sense that it can be either “off” or “on”. But in an ideal scenario it would only track the target street that it needs to reach.
* Database:

### Problem using existing software

There a re a lot of libraries and frameworks out there but none of them offer exactly what I needed.

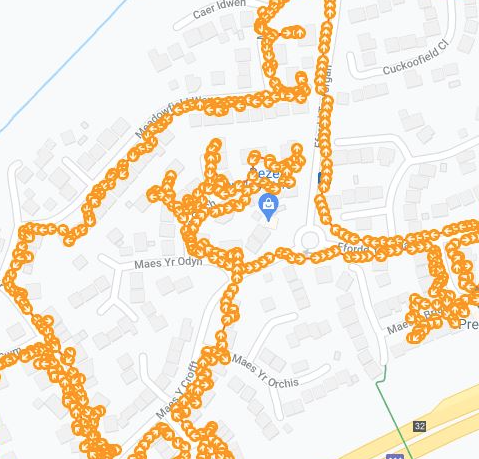
After doing a lot of research and experiments with Leaflet I started to think I could not use their standard Routing Machine (<https://tomickigrzegorz.github.io/leaflet-examples/#63.leaflet-routing-machine>), for the following reason:

The routing machine goes from street A to B. But a garbage truck need to do the whole street.

You could say add waypoints to each end and beginning of the street.

But in a street with a lot of side streets, the garbage truck goes from A to B to C and back to B, multiple times.

A possible solution is to add multiple waypoints in between every street, and make a route between those in the correct order.

Trackit247 (GPS Leaflet Tracking Software for Leaflet Distribution, n.d.) used this method but the result is rather messy and does not give you a lot of options as we can see on the following image.

But I contacted them, and by putting myself forward as a company owner and they gave me a free demo account which gave an interesting insight into their product. Their main product was tracking trucks and also could not assign routes to vehicles so could not be used for this project.

One of my requirements was to be able to transfer a road or part of a road to another user.

Combined with that not a single free mapping system allows infinite amount of waypoints, this all made me think there was only one way and that was to program it all myself.

# Basic Implementation

In art galleries I sometimes found the sketches of paintings far more interesting than the works themselves, as they give a deeper and more straightforward insight into the process of their creation.

So using this mentality, I will mainly discuss the basic implementation, which works as a sketch for the app, with all its basic features. This will give a clear insight in the workflow, progress and know-how of the project.

The goal here is to program the basic functionalities in a lightweight version, to grasp the core idea which we can later just scale upwards.

1. Data

### Research

In order to set up the map and defining a route we need the right data.

I specifically need data which can define a route in a certain metric so we can optimism the route using algorithms, for example an array of streets.

This array I found in the yearly Renewi magazine as it contained a lost of all street they takes care of in the city of Ronse. But now I needed their coordinates or some kind of data I could put on a map and calculate routes with.

After a long quest for open source, free and developer friendly data sources I discovered Open Streep Map or OSM (GmbH, 2022). It is the largest open source geographical data set available, with a lot of other interesting datasets like a map of GSP locations around the world.

The hard part was using their metric, as they defined all roads and routes by id’s; like the Veldstraat in Gent, has ‘way’ with id “762619506” (<https://www.openstreetmap.org/way/762619506>).

But a street exists out of multiple waypoints or ‘nodes’, and in this case: the veldstraat is their parent node. And some nodes had the same ID but a different location and the parent node defines the location.

It so happens that a street in Ronse has the same ID and name as a node in the Netherlands and one in France.

I used (Overpass Turbo, n.d.) to understand the data as it allows you to query all OSM data.

It took a while to find the right query as their documentation is quite succinct (Overpass Turbo/Examples - OpenStreetMap Wiki, n.d.).

Then I iterated the original array of streets of Ronse and requested with a Python Overpass Api all nodes of each street.

Having finally a bit of a result I found a guide called Smart Mobility Algorithms (Getting Started — AI Search Algorithms for Smart Mobility, n.d.).

This gave a comprehensive guide to all usable packages, algorithms and implementations.

Combined with the previous effort I could start programming a basic concept.

### Code & Testing

Then I created a couple of lightweight scripts (and multiple variations) for testing purposes:

#### Streets

Code: 2-basic-features/1-data/1.2-get-all-streets.py

This first python script had the purpose to queries data by using the “overpass” (Python Overpass, n.d.) package and outputs this into JSON.

#### Folium

Code: 2-basic-features/1-data/2.3-create-foli.py

This other python script converts the gathered JSON data into an instantly usable Leaflet HTML page using the “folium” (Folium, 2022) package. I used Leaflet as it is an open source library that can also be used in React to display OSM maps.

But nothing goes perfect and the map was empty. I finally found my streets back in shiny colours in the middle of the Arabic Sea, off the Horn of Africa. Apparently Overpass used [longitude, altitude] while in leaflet the coordinates where the other way around. So if you find yourself in the middle of that ocean wondering why you're home made GPS directed you that way, just switch the coordinates.

Now we could open the html page and have all out json perfectly visualised on a highly customisable map.

1. Basic Functionalities & Features

Here we will go over the more technical evolution and code and some of the problems that came along. I will cover a couple of technical highlights of each step, and keep in mind that this still only describing the end results of the process.

With this Leaflet template we already visualised all data, but there was still most of the work to be done. I added some functions and made a Marker walk the streets in a somewhat random order (<https://github.com/Jonatanfroeling-user/Bap-garbagetruck/blob/main/Basic-implimentation/v4-visualising/1.2-foli-allStreets-975.html>).

Visually there is still a lot to be done, but can work as a sandbox to do rapid experiments.

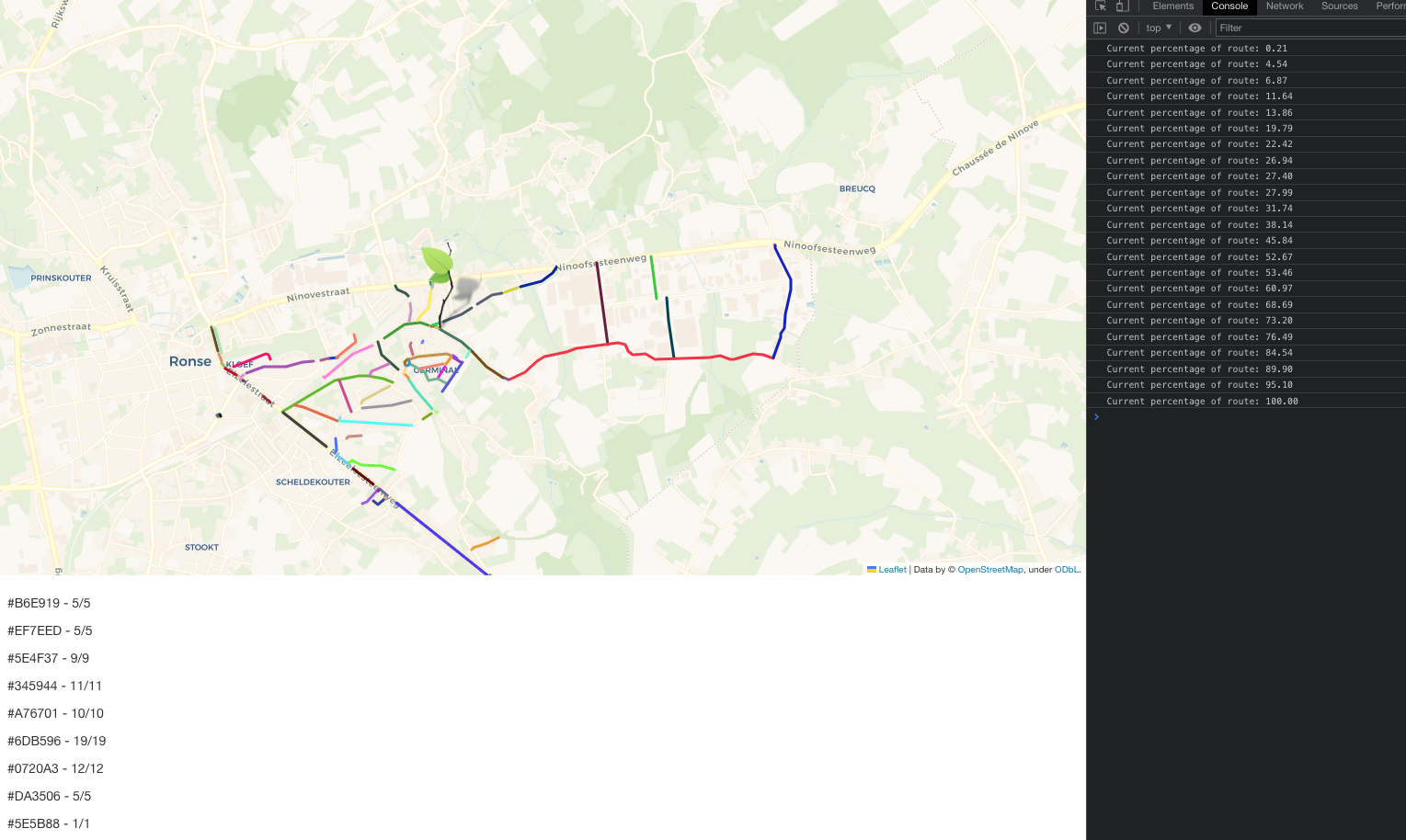
### Progress Tracking

#### Version 1: basic

Code: 2-basic-features/1-progress-tracking/1.1-progress.html.

One of the first problems I wanted to tackle was progress tracking.

So I made a new script that was able to track how much of each street was done, as my concept was that the app will use the trucks location to mark if a street is finished or not.

(street names are replaces with colours for testing purposes)

There where still a lot of problems and things that needled to be fixed (described in more detail in the second version).

#### Version 2: smooth

If you want to calculate how much of the road is finished and the road only consists of 2 nodes, the percentage will go from 0 to 100. So I started to think, and came upon the idea of adding multiple points in between, that would make up a street.

Challenge: in between

Code: 2-basic-features/1-progress-tracking/smoothness.html.

To make this smoother I used functions out of older projects of mine (<https://github.com/jonatan-verstraete/js-canvas/blob/main/setup.js>) where I experimented with animations with the Js canvas Api. These functions plot points in between every node every 8 steps, resulting in the following:

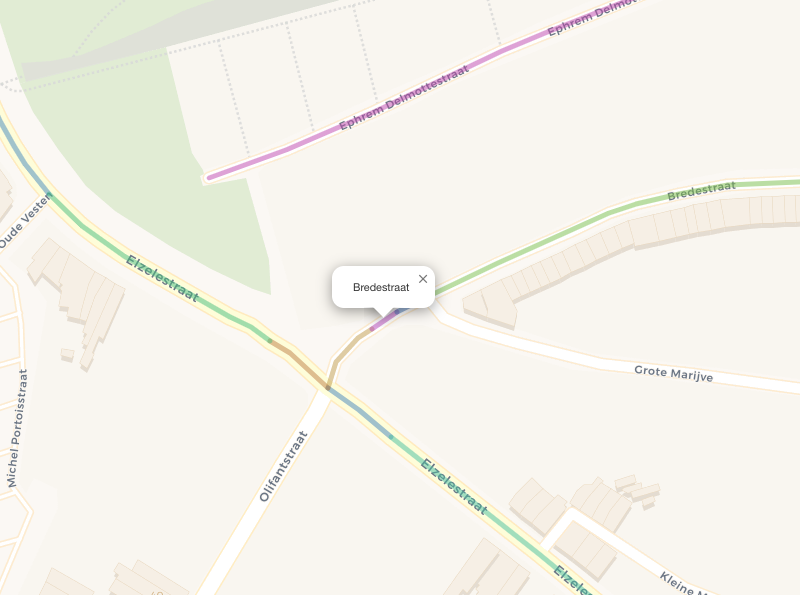
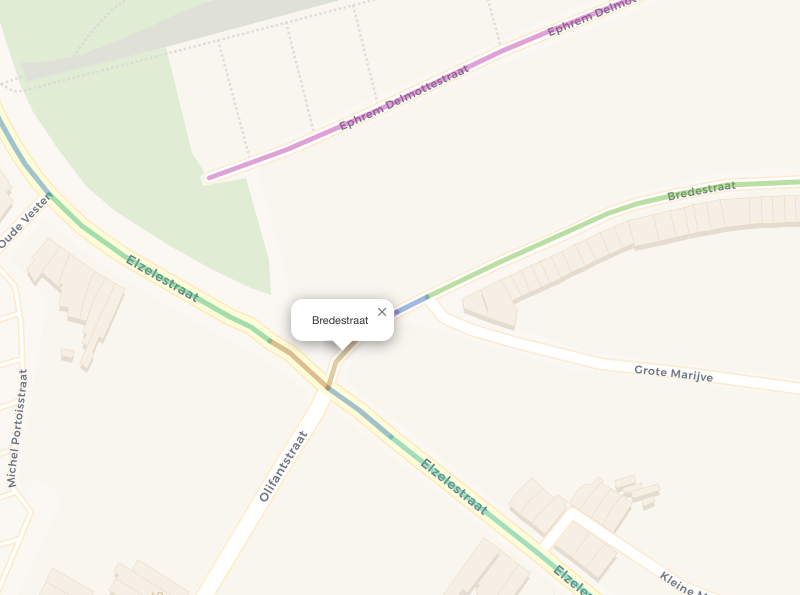
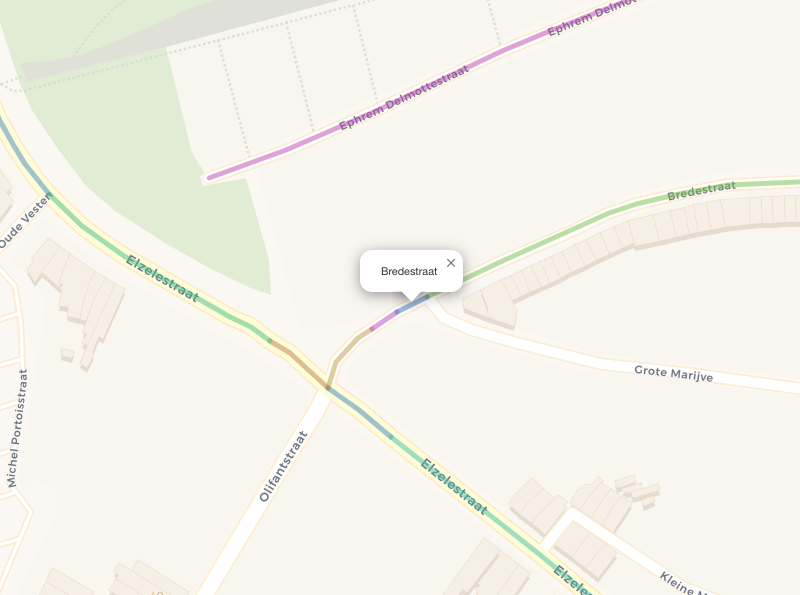
Now the percentage will be a more smooth transition and is just: checked-nodes / total-nodes.

This resulted in the following:

2-basic-features/1-progress-tracking/1.2-progress.html

Challenge: in between

Code: 1-data/1.3-update-data-conversion.py

Another problem was that each street was made out of multiple parts which bared the same name. Adding all waypoints together would create a huge mess of overlapping lines, so I created another script to filter the json data yet again.

Now each street is clearly made out of parts with an ID and each part exists out of a couple of nodes.

### Grid

#### Concept

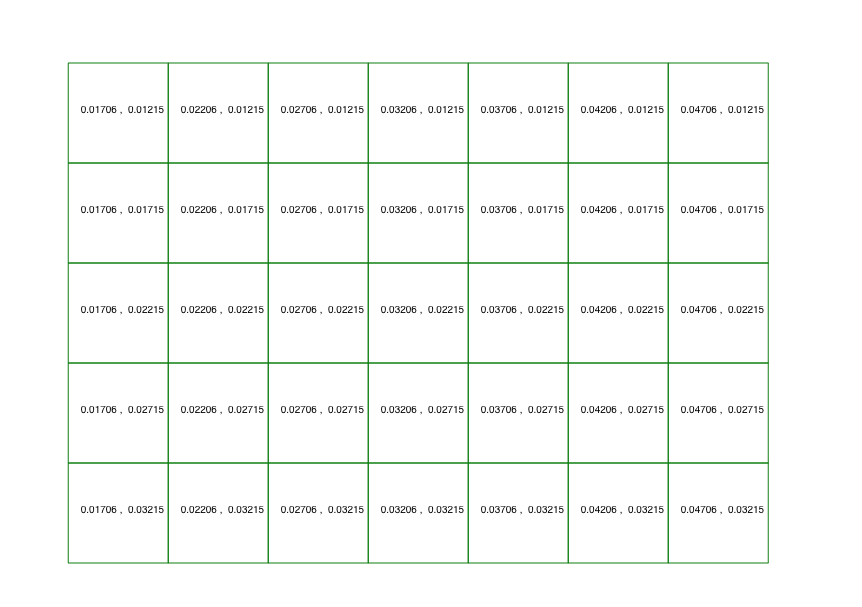
Code: 2-basic-features/2-grid/2.0-grid.html

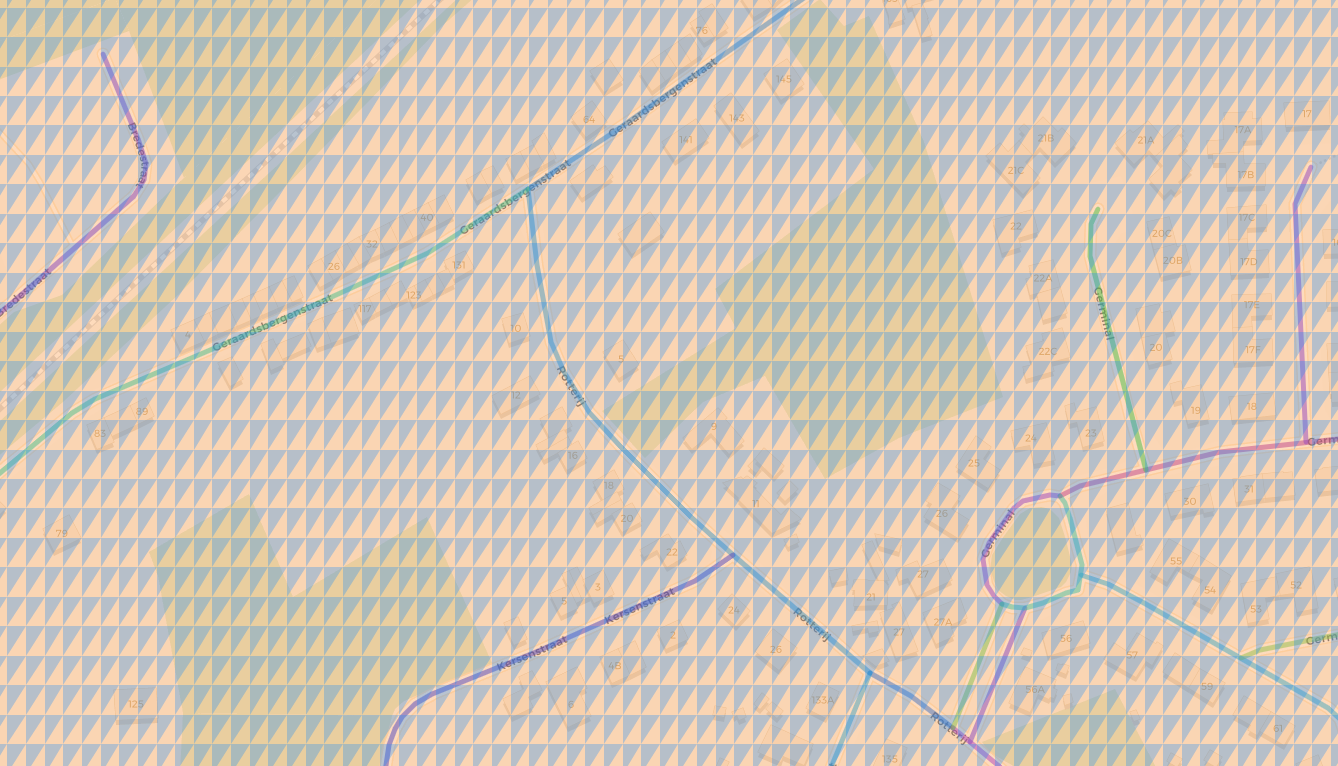
Another problem arrises when using real time location data. Eg. If the truck is in between two streets, we can’t just check the nodes of one street, because we don’t really know which street the truck is at.

So there will be need for some kind system that checks an area around the truck. But here comes a performance issue, because we would need to check all the coordinates in a certain area every n-steps. Depending on the precision, a coordinate system is to 1.11 meters (Decimal Degrees - GIS Wiki | the GIS Encyclopedia, n.d.) distance which would result in a lot of iterations.

Mapping systems used in Terravision (Terravision, n.d.) or more modern system like Google maps or terrain generators, all use systems to divide everything in a grid. I used this idea to create a grid like in the images of test phases below.

Now it was time to assign each street node to a node on the grid.

Then we can search in a certain radius around the truck’s position, iterating the grid by the detail of that grid.



#### Version 1: tiles

Code: 2-basic-features/2-grid/2.1-grid.html

After a first (messy) implementation (https://github.com/Jonatanfroeling-user/Bap-garbagetruck/blob/main/Basic-implimentation/v4-visualising/saves/1.5-foli-allStreets-975.html) the concept failed.

Performance became the main issue. It took 7826.071 ms to set up the map. Iterating all points in an area also took very long as calculating the distance to a point uses computationally heavy functions like “Math.hypot”, which calculated the Euclidean norm (the square root of the sum of squares of its arguments).

My strategy was to have complete control over every coordinate that if need be, coordinates could be added to the route which where not necessarily in the original area.

But it could not handle positions out of the route it’s area as that would require creating another grid which ended up with dire performance.

I thought about saving all the gathered data in another JSON file, but this might rather complicate things in the long run so I stopped coding and continued to research.

#### Version 2: uniform

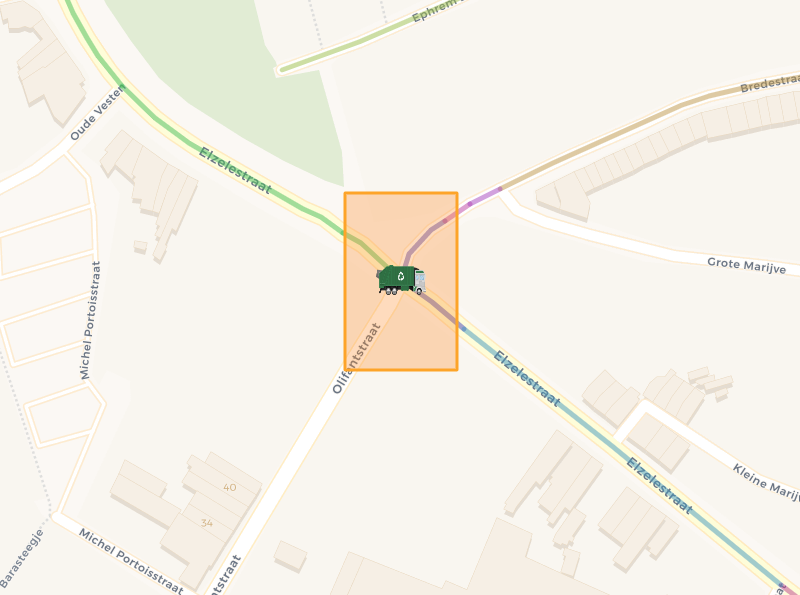
Code: 2-basic-features/2-grid/2.3-grid.html

I decided to use concept of a uniform grid as my current research pointed out that this method is rather common. The concept is rather easy: you divide the terrain or map into equal sized blocks. Each of the chunks (or commonly buckets), will contain a list or array of objects which reside in them.

But dividing everything up in a grid would result in a similar outcome as the past attempt, because checking in which tile the truck is currently at, would require calculating distances again, in order to verify that you are not in another tile (bad performance).

Another problem with using tiles is a scenario where the drivers gets a request to finish a street in another village, then we would also need to recalculate. So using a fixed grid is only good if you know the exact boundaries.

In order to prevent all this, I reformed the concept so that it only scans a tile surrounding the truck.



This resulted was a huge performance boost.

Console.time performance test:

112.260 ms - for setup; create the grid, assign all street nodes… etc.

6.363 ms - to scan the a tile in the grid.

One bad aspect was a lack of control over the grid: the coordinates of the grid are somehow a bit weird numbers, to generate the square on the above image I used the following, very messy function in order to translate the grid coordinates to real-world coordinates:

const getX =(*a*)=> +String(a).split('').splice(3,50).join('')/100000 + 50

I was content to a certain extend, a pity for the somewhat weird number and rectangular scanning area, but a relatively good performance so I couldn’t complain.

#### Version 3

Code: 2-basic-features/2-grid/2.4-grid.html

After spending a lot of time on these grids and functionalities, I got a small “Eureka” moment.

I though that it might be more easy to simply round the coordinates off. Coordinates in this system use 6 digits after the comma, which results in an accuracy of 1 meter.

Positions rounded to 5 digits would result in an accuracy of approximately 10 meter.

Positions that are within this 10 meter of each other would now have the same coordinates, so you could put them in a shared dictionary or map, with the rounded coordinate as key.

This would resulted in a similar effect as the tiles but it uses the grid system of the coordinates themselves. Not only that, but with the benefit that there is no need to create a grid or to scan the area surrounding the truck.

Therefore this would not only resolve all previous issues, but also boost performance, add control and increase scalability.

In case of adding a route outside of the original route, this system does not require to recreating anything; simply add the rounded new coordinates to the list.

Performance test of an average of the whole route (console.time, 20 iterations):

1.5 - 7826 ms

1.6 (previous, ‘classes-v2.js’) - 3633 ms

1.6 (current) - 32 ms

Coming from nearly 8000 ms, the current version is about 250 times faster and can probably still be improved on.

Isn’t is funny how long it sometimes takes to realise something so simple.

### Transferring parts

Code: 2-basic-features/2-grid/2.3-grid.html

The goal here was to be able to transfer a road, or part of a road to another user.

This is not very hard to do, but if done well requires a complete review of the current ecosystem.

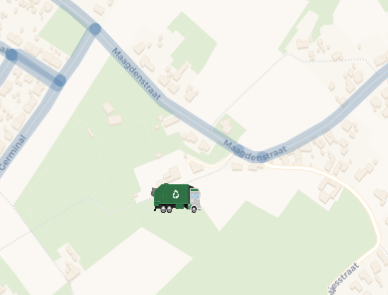
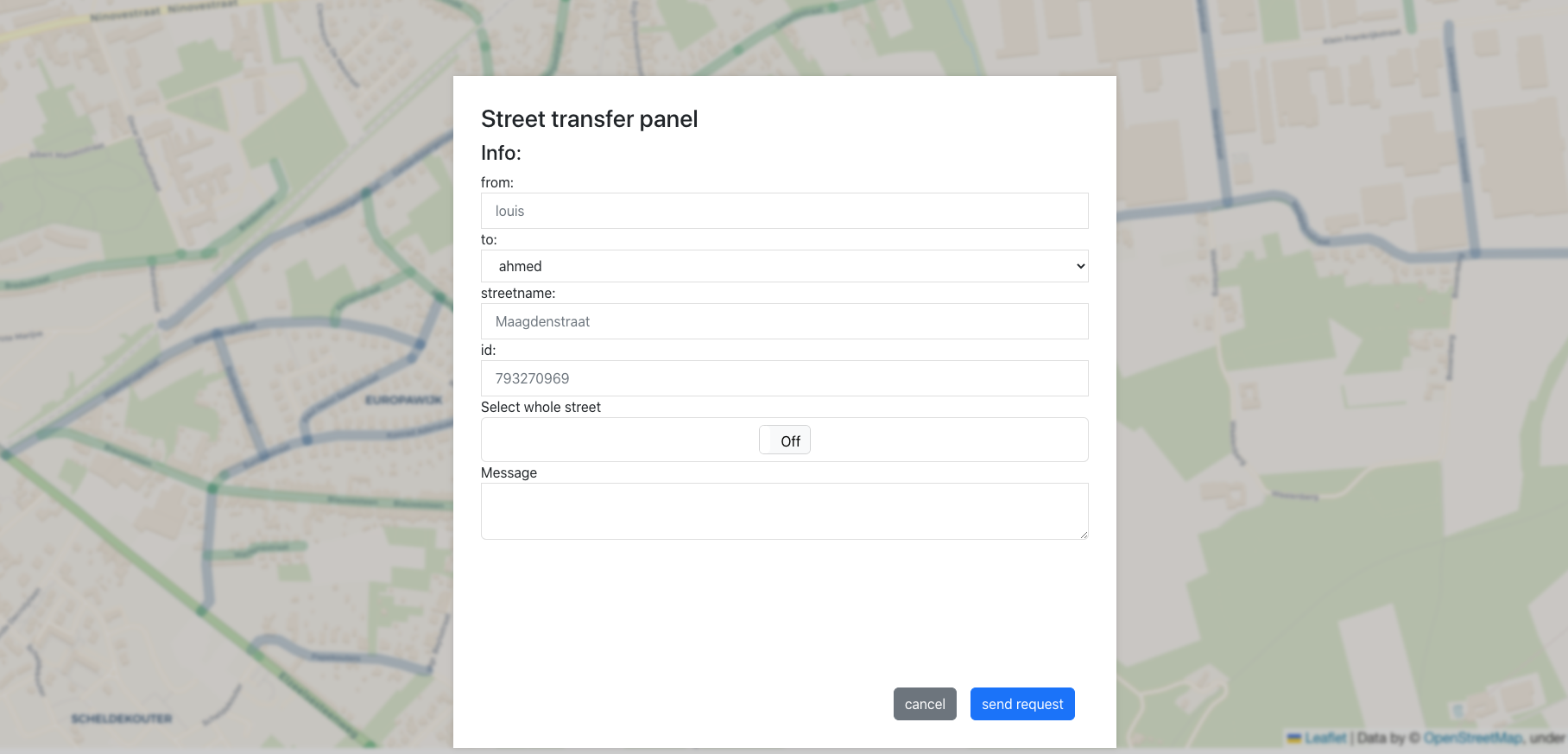
Basically everything needed to be updated to be as efficient as possible.

In order to transfer roads in a more realistic scenario, I started to work on something that would handle these requests from diver-A to driver-B.

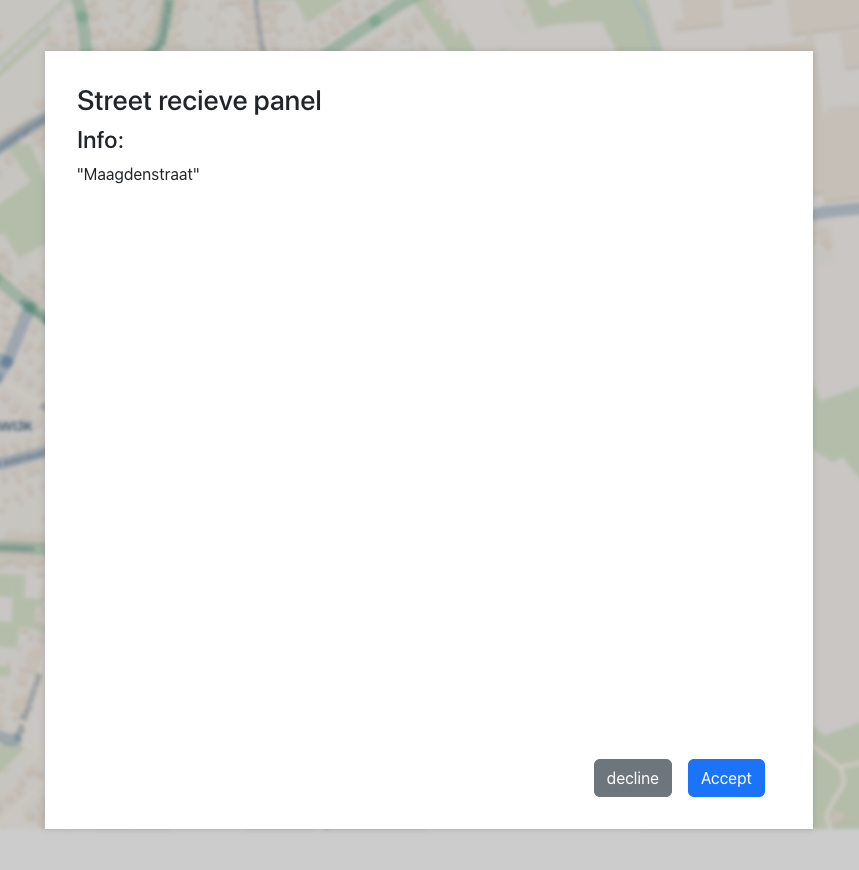
To simulate this as a lightweight example, without using the likes of Node Js, I create a sort of fake server object (in future versions this could be an Api, which enables the user to download routes and handles the traffic between users).

One of the features it has, it that a driver does not have a database of the roads, and that all data is passed on from this Server Object to the drivers.

This way the driver’s device does not need to make any heavy calculation and/-or configurations, and only needs to download or receive the right data.

The basic idea is to user-A get a popup form by clicking on a road (or part of), which gives him the option to select a user and an option to select the whole street.

This form submits the id of the street to the Server object. As this server has all raw data of all streets, so by receiving this id from userA it can retrieve the raw street, which it sends to user-B.

Then userB get a popup form which results in sending a response (accepted or declined) via the Server back to user-A. 

Challenge: response

When a road is transferred is basically is deleted at user-A, and recreated from scratch at user-B.

One challenge was that userA needs to wait for a response from userA, but I don’t find it optimal that user-A is waiting for user-B’s response.

So I created an object holding all pending actions. For userA the action has the original ID of the street as a key and a function to delete the street from its memory as value.

This is also more efficient for user-A, as the action is just stored and is only retrieved (and deleted) on the response of user-B.

### Offline Map

#### Concept

The reason I want to make the map offline, is not only for performance reasons, but also for better security and privacy and overall less costs of internet traffic.

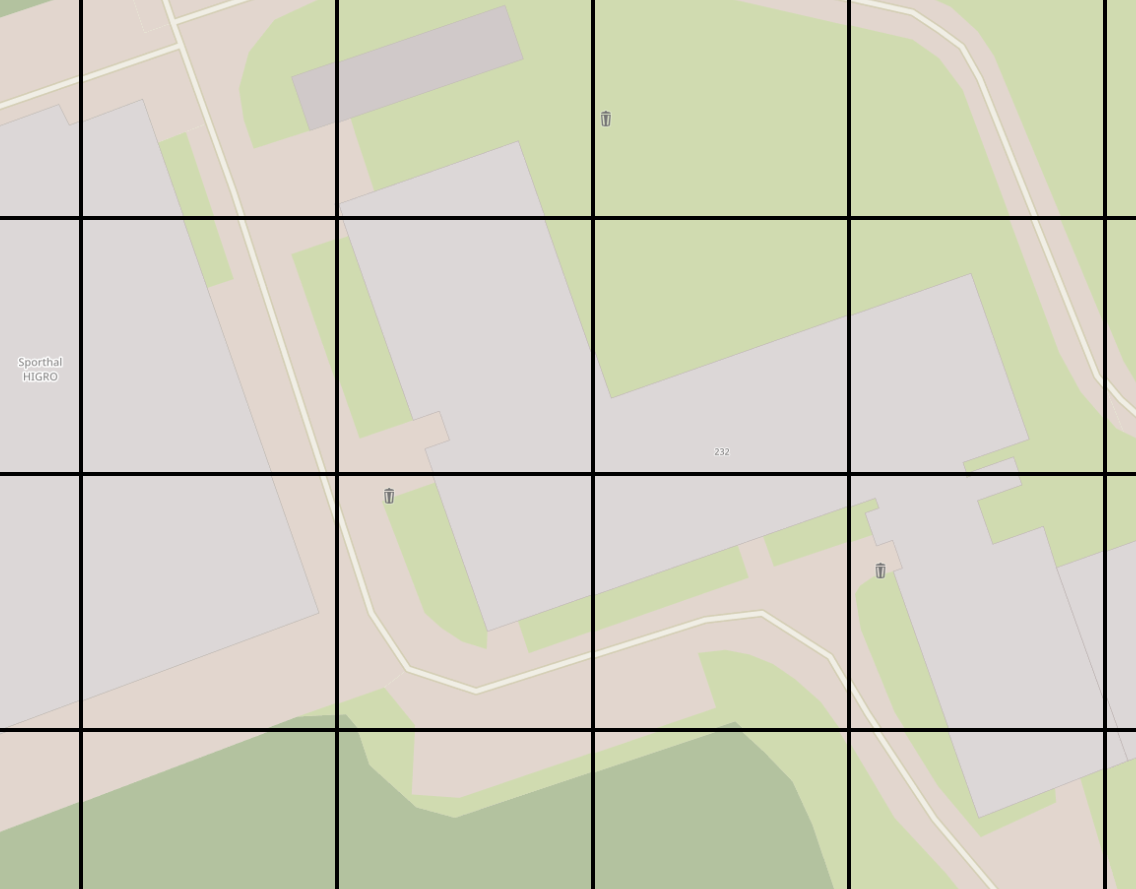
But the reason that instigated the idea to go offline is that in more remote areas or stormy weather their connection might be less reliable. So if you depend on a digital map, you need all time access.

I want a truck to be as autonomous as possible, so my goal is that that a map can be downloaded from the Api/server. So like Maps Me, my goal is to make a driver able to download a route.

A map is made out of individual images, each one arranges in such a way to give the illusion that it is one larger image. This illusion is broken, by the visible loading if those images.

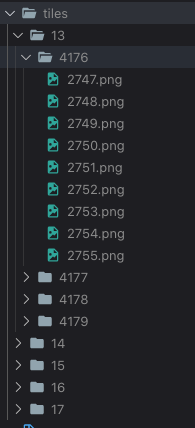
Even Google Maps has this, when you zooming, it gives a white grid off where the images should be, as they are not loaded yet and only starts rendering them once you stop zooming. But once the images are cached in memory, the zooming goes without hick-ups. The goals is to make this hiccup as small as possible.

If we take the two images below from the Open Street Map (<https://www.openstreetmap.org/#map=20/51.08733/3.66873&layers=H>). The first is zoomed on 20 (maximum) and the second on 19. Visually not a lot has changed, but actually all the images of ‘zoom 20’ have been replaced by images from ‘zoom-19’.

But if we add a simple css border around the images, we can clearly see that the zoom determines the images to load and that the size and amount of images stays relatively the same. The only difference is the amount of detail in the images (for performance reasons), as street names and icons are hidden with a when zoomed out.

#### Downloading the Offline Map

Code: 2-basic-features/1-data/3.0-download-area-offline.py

With the lightweight “shutil” and “urllib3” Python library’s you can download any image, so these map tiles as well. Source can be found here (<https://github.com/Jonatanfroeling-user/Bap-garbagetruck/blob/main/Basic-implimentation/v5-offline/1.0-download-map.py>).

This script allows me to download any area, to any zoom. The zoom factor is very important here, as each higher zoom level results in an exponential amount of images.

A seen in the following image, this folder contains about 5000 images.

It is the map of the route (a parts of Ronse) and doesn’t take long to download and is in total about 52mb.

Larger images are about 25kb (zoom 13) and smaller images can go down to 1kb (zoom 17) in size.

As I did more research I found that bulk downloading is not allowed (<https://operations.osmfoundation.org/policies/tiles/>).

And downloading more than 250 tiles or more than 13 zoom is also not allowed. So I had to figure out another way.

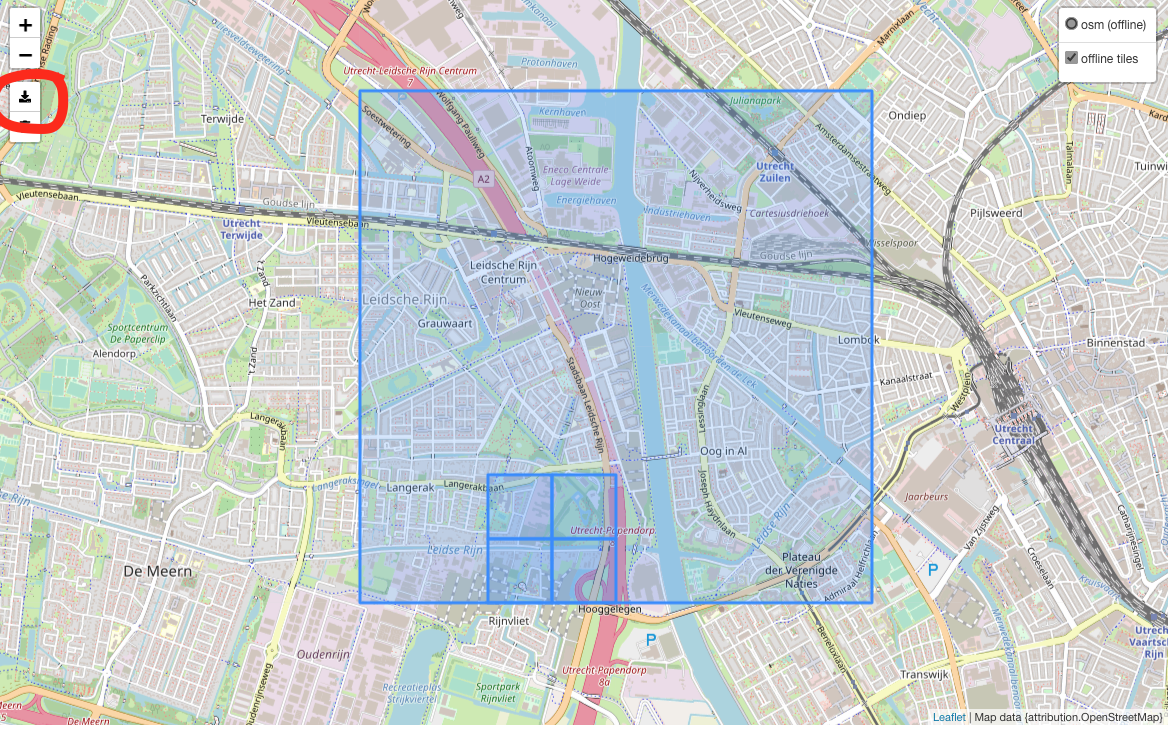
This article (Gibbard, n.d.) helped me a lot. The reason just downloading these maps is not allowed is because OSM is a none-profit organisation and they a have limited bandwidth on their self-paid servers.

His solution is to download a larger area and then download the necessary images from you a self made, image server.

But this is a rather time expensive solution. You also would need a very heavy computer and this seems like a project on it’s own.

After doing more research I found Osm Lab which allows to download tiles by country (OSM QA Tiles Country Extracts, n.d.), and Belgium was only 500mb. But the main problem was that it only supports 12 zoom, which does not include detailed street names, so is not usable for this project.

Then I found the ‘idb’ NPM package, which allows you to store these tiles/images on the fly.

As we can see in the following example (<https://allartk.github.io/leaflet.offline/>):

Is becomes very easy to download tiles, and even gives the user the option to use offline or online tiles.

I could reprogram this so that the downloading is automated and the user can download his current route if he wants to.

#### Completely offline

Code: 2-basic-features/4-offline/4.1-offline.html

### Real Time Location

The basic functionalities are all done, and it’s time to look at1 one of the key aspects of this project: real time location sharing of the truck.

One of the first ideas I had about this projects before I even started it, was to use a Raspberry Pi as a dummy truck driver which you can move on a normal sized physical map to simulate a truck driver.

The goal is to use a Raspberry Pi (4 B) with a sense-hat for this (as I do not have a gps tracking module and WIFI location tracking is not very accurate).

The idea is that, when the sense-hat moves, it adds the change in movement (detected from it’s accelerometer) from the previous movement.

This change in movement we then add to the current location which should simulate the truck’s movement in an accurate way.

We can also change the effect of the change so minimal movement become, a meter in the front .. . f..fs.f .sdf .sd.f s.df. CHANGE ME

#### Research - WebSockets

It’s back to the drawing board.

In most school projects we learned to use http and how to set up a REST api. But in the course Web Of Things I learned the basics of MQTT (paho-mqtt) and indirectly or accidentally also learned about the WebSocket technology, i.e. [socket.io](http://socket.io).

Some described [socket.io](http://socket.io) simply as a layer on top of the standard WebSocket or “ws” library (rsp, 2016), and that it actually slows down the process as it is made to be compatible with all browsers and versions, as it adds another calculating factor to the performance.

But it made sense to me that WebSockets have generally better performance than http, considering real-time data as they keep a direct connect or a bidirectional connection.

My suspicions where confirmed by a performance study (WebSocket vs HTTP Calls - Performance Study, 2019), which stated that WebSockets are expected to be about 5-7 times faster than http.

WebSockets are considered the next generation method of asynchronous communication from client to serve (Walsh, 2014), so for this project I wanted to use this method not only for performance reasons, but also as an interesting learning path for future development.

#### Concept

The basic project is fairly simply:

1. create a python script emits or sends the movement of the sense hat
2. Display that movement on a map
3. Optionally: create a server that server that also stores these coordinates in a database

Challenge: sense hat - communication

One of the problems is that the sense-hat cannot directly communicate to the front-end, as Javascript does not have direct access to the sense-hat. So in the end, after long discussions with ChatGPT, I ran a [Flask-Socket.io](http://Flask-Socket.io) server on the Raspberry PI which emitted data to the frontend on a MacBook which displayed this data in a front-end.

Challenge: sense hat - movement

Finally getting the whole thing working I found that the marker on the map (which moves by the received movement from the PI), made unpredictable and rather random movements.

But after rewriting the code and trying different angles, still the same random like movements appeared.

Most movement was given by tilting the PI and not by moving it in 2D directions.

Ultimately I let ChatGPT have a look at a my code, and it told me that it should work the way I intended it to work, so no luck there. But it did make the remark that maybe the problem lied with the sense-hat itself.

So I started to investigate and debug.

The first method was to only add the y-axis movements. By tilting the PI on its side the y coordinates on the map changed. But by only moving the x-axis the y-axis was again the only one moving

1. Use a properly indexed schema: The schema should be designed with indexes that support the required data access patterns, such as querying locations within a certain time range or within a specific geographic area.

2. Consider sharding: If the data volume is expected to be very large, sharding can distribute the data across multiple nodes for improved performance and scalability.

3. Batch updates: If the frequency of updates is high, consider batching the updates to reduce the number of database writes.

4. Use TTL (Time To Live) Indexes: If only the latest location data is needed, consider using TTL indexes to automatically delete old data.

5. Consider caching: If frequently accessed data needs to be served quickly, consider caching the data in memory using tools such as Redis or Memcached.

6. Monitor performance: Keep an eye on database performance and optimize the database configuration as needed to maintain good performance.

# Defining routes

<https://developer.here.com/documentation/routing-waypoints/dev_guide/topics/request-constructing.html>