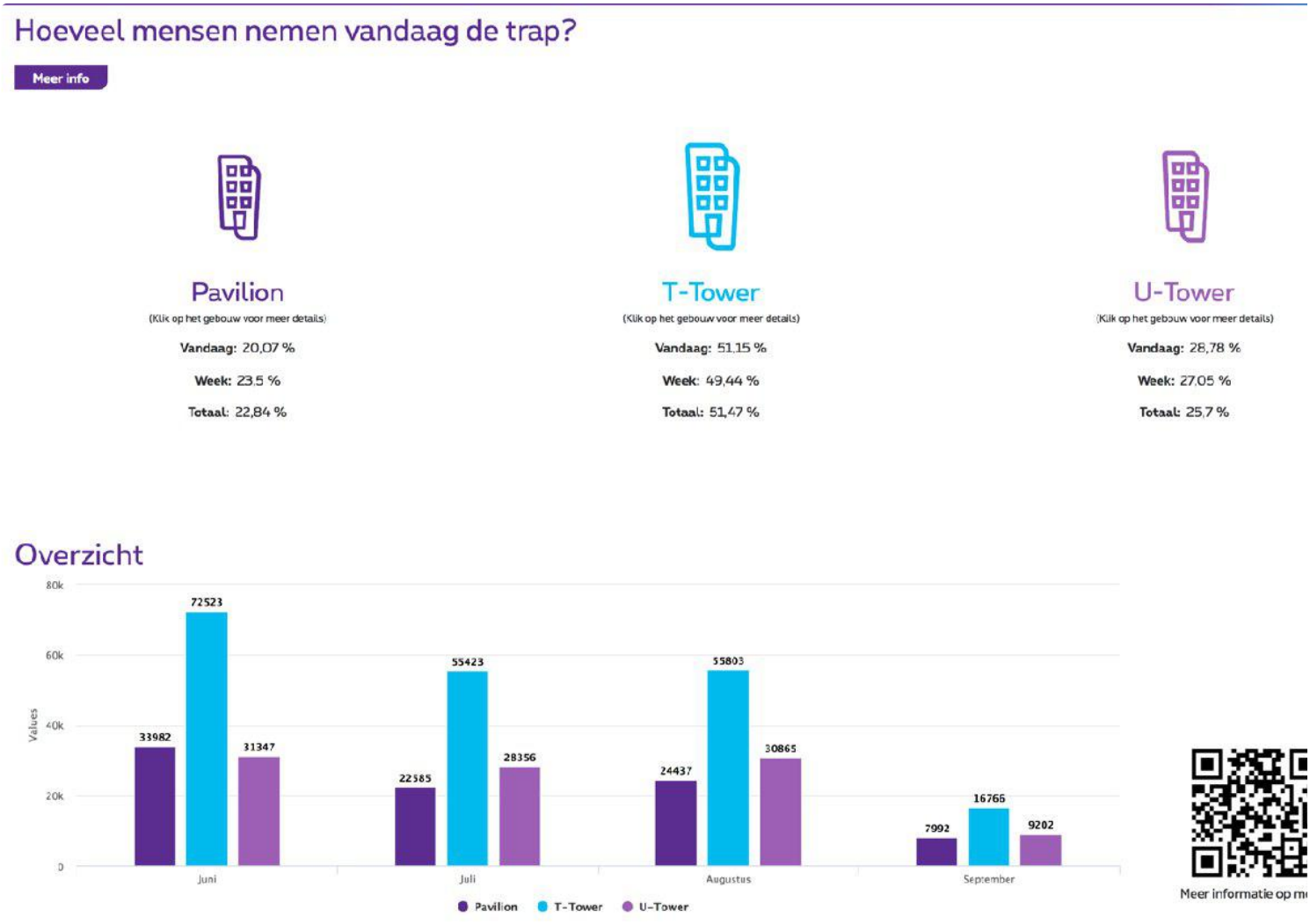
> *Healthier at the office with the 'Internet of Things'.*

WHAT IS STAIRWAY TO HEALTH

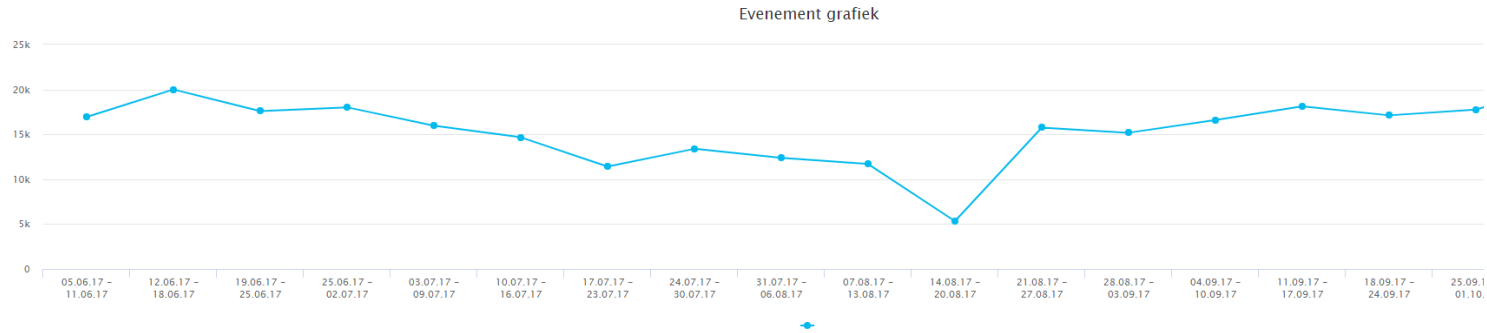
In an effort to improve worker health in a fun and engaging way, Proximus wanted to encourage their employees to choose for th instead of using the elevator. This is when they came with the idea of a little game between the three towers. On different dashbc across proximus and online employees could see which tower had the most employees that are taking the stairs.



They can also see a more detailed look of how many people taking the stairs in which tower of each week, day and even each hc

T-Tower

?



Data

Exp

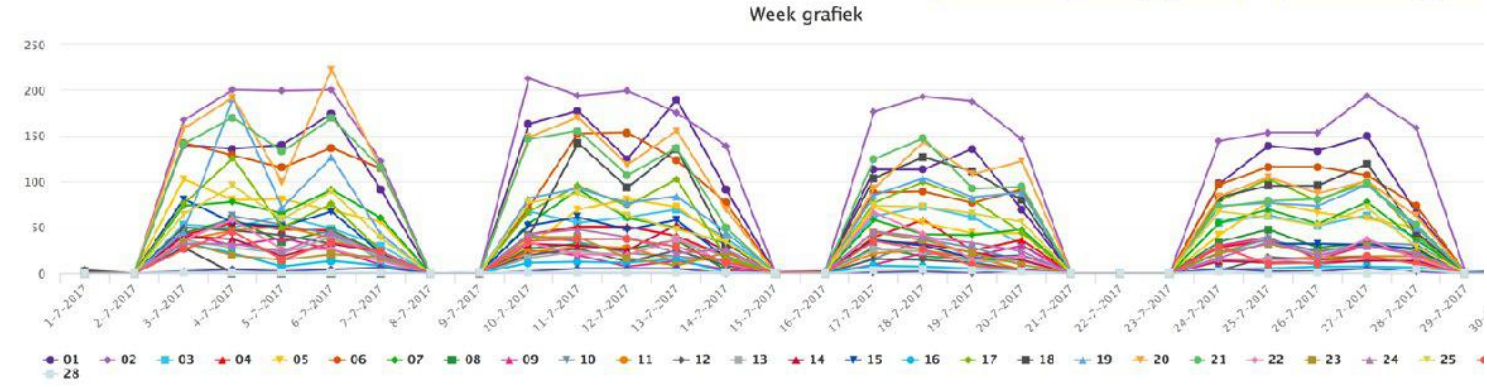
Label	Start	Einde	Teller
05.06.17 - 11.06.17	5-6-2017	11-6-2017	16.926
12.06.17 - 18.06.17	12-6-2017	18-6-2017	19.998
19.06.17 - 25.06.17	19-6-2017	25-6-2017	17.589
25.06.17 - 02.07.17	26-6-2017	2-7-2017	18.011
03.07.17 - 09.07.17	3-7-2017	9-7-2017	15.960
10.07.17 - 16.07.17	10-7-2017	16-7-2017	14.671

U-Tower

?

Zie/verberg alle verdiepingen

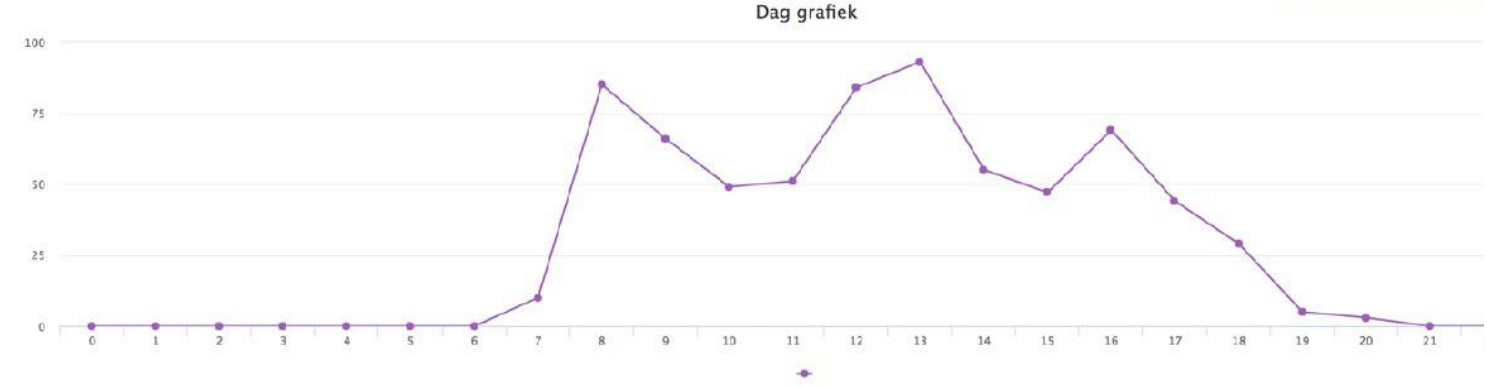
Vorige grafiek



U-Tower

?

Vorige grafiek

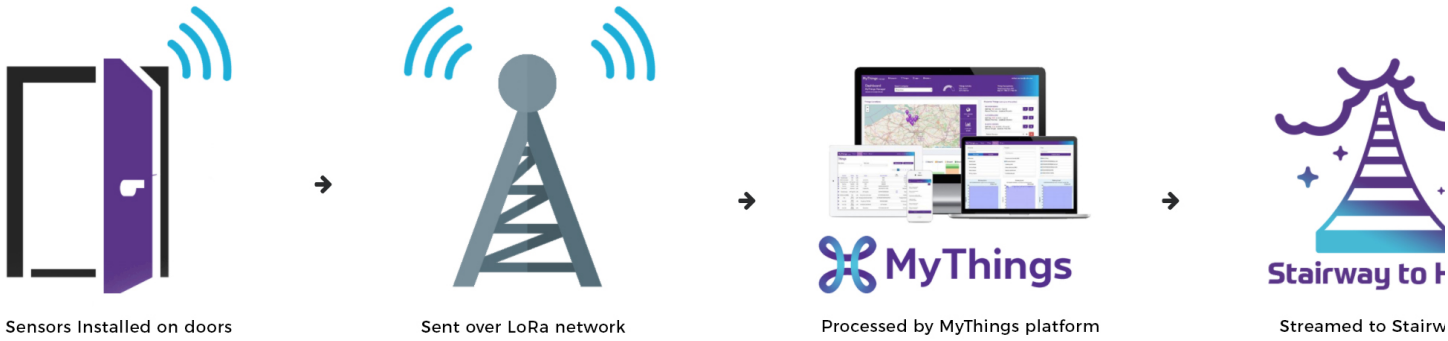


WHAT DOES IT DO?

The stairway to health project is a simple, yet great example to show what the internet of things is can do:

- LoRa sensors detect door openings, these are installed on the doors of the staircases
- These sensors communicate via the Proximus LoRa network to report their status
- Sensor data is sent to the Proximus MyThings platform which processes the data
- The data gets sent to the StairwayToHealth application
- The StairwayToHealth application interprets and visualises the data

In summary: We install sensors on the doors (things) to measure usage and we analyse the data to persuade people to move mc result is a good example of how IoT can influence our daily lives. Proximus was able to provide us with all the building blocks ava offer a complete end-to-end solution!



MYTHINGS AND STAIRWAY TO HEALTH

MyThings is the Proximus IoT platform for onboarding, managing, configuring and monitoring IoT sensors. By registering (onboar sensors to the platform, we can let MyThings take care of decoding the messages and set up a stream to our application so that e a log comes in from the sensor, we get the decoded data posted to our designated endpoint.

THE REQUIREMENTS

The usage of the stairways is measured and the results should be visualised on large screens in the towers. These screens shoulc code so that employees can easily visit the application on their mobile devices. When visiting the website, they should be able tc the results to get a more detailed view of the data. The frontend application should be available in dutch and french and the das should switch between these languages every minute when viewing it on the large screens. Admins should be able to manage lc (towers) and chart timespans. It should have an info page with some information about the project and it's purpose.

So technically this translates to build an application that:

- has an endpoint to receive logs from the MyThings Application,
- stores the data to it's own database,
- show the data in charts that have multiple layers to see more/less details
- shows the ratio of the results per tower
- the frontend dashboard data has to reload automatically (since it is shown on some big screens @ Proximus)
- add multi-language (automatically switch languages when viewing on tower's large screens)
- is performant (able to handle many logs coming in and calculate the data to be displayed in the graphs)
- CRUD's for managing timespans and locations.
- use the timespans / locations when displaying data

oh, and did I mention we were given 4 weeks to complete this mission…

THE INGREDIENTS

So given all the requirements listed above and the fact we didn't have a lot of time to waste, we chose to use a **MEAN(TS)** stack. stands for MongoDB Express Angular and NodeJS. It's possible to use the mean stack with plain JavaScript, we chose to impleme TypeScript since we wanted some strong typings on the backend application and we were going to use Angular 4 on the front-er comes with TypeScript as well.

NodeJs: write event driven applications with asynchronous I/O powered by the ultra fast Google V8 Engine. Mostly known for rur local dev environment and automating build tasks for front-end developers. NodeJS is probably one of the best and easiest opti there for real-time applications (with socket.io), which is exactly what we needed for our application.

MongoDB: Great to work with when dealing with JavaScript Objects. Good driver support with mongoose for NodeJs. Documen structure, which makes it really flexible when it comes to modelling and it's extremely scalable. We also took advantage of the ve performant aggregation functionality for dealing with large amounts of data.

ExpressJS: A node frameworkt that comes with some great functionality for setting up your node server and makes it easy to cre middleware, handeling requests/responses, serving files from the filesystem, configuring static files, easy connections to the data much more.

Angular(4): A TypeScript-based open-source front-end web application platform led by the Angular Team at Google and by a co of individuals and corporations to address all of the parts of the developer's workflow while building complex web applications.

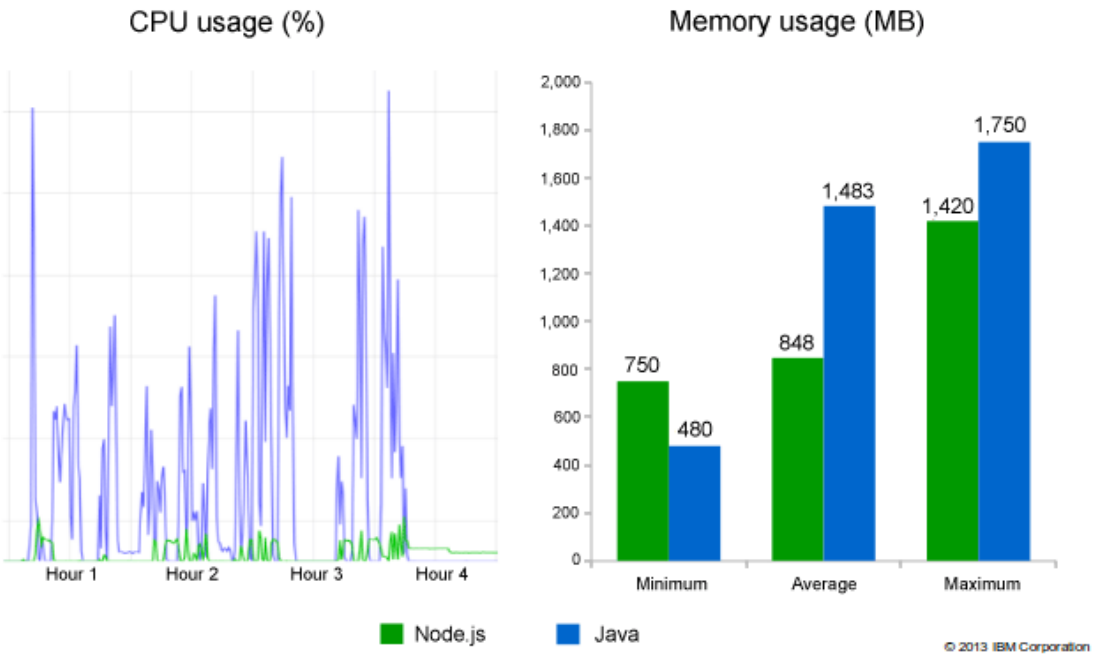
Socket.IO Socket.IO enables real-time bidirectional event-based communication. It works on every platform, browser or device, equally on reliability and speed. To trigger events on our front-end application we used this great library to be able to detect whe

data has been received and refresh the dashboard.

Highcharts: Interactive JavaScript library for creating dynamic charts. Highcharts is based on native browser technologies and t reinvent the wheel. Thousands of developers have contributed their work for us to use in our own projects. Also backwards comp IE.

JAVASCRIPT ACROSS THE STACK

Not only does it make development a lot faster and easier by having a large community with lots of reusable code for your applic (npm), It also lowers the barriers between front-end and backend developers by using the same programming language over the stack, so more efficiency and faster, leaner development which in turn means lower development costs. Also worth noting is tha JavaScript currently is THE most popular programming language, so more developers will be able to easily understand and cont the application if needed. And probably the most important criteria: when it comes to cloud hosting, RAM is probably the main ir factor when it comes to pricing. NodeJs uses less RAM than comparable Java applications.



Source and more about these tests

Now that I've listed some of the pro's of full-stack JS, I should also mention that it might not be the best solution for computatior backend applications. For projects like machine learning or heavy mathematical calculations the single CPU core and having on thread that processes one request at a time might be easily blocked by a single compute-intensive task. Yet, there are numerous overcome this limitation. By simply creating child processes or breaking complex tasks into smaller independent microservices.

Let me just note that the comparison with Java above here is not because we are claiming that one is better than the other, it's ju demonstrate that they both have their use cases and can be equally worth considering when choosing a technology for your app

Some great use cases for JavaScript across the stack are:

- real-time chat,
- Internet of Things,
- real-time finance (stocks),
- monitoring applications,
- event-driven applications
- server-side proxies
- many more…

Blocking vs. Non-Blocking

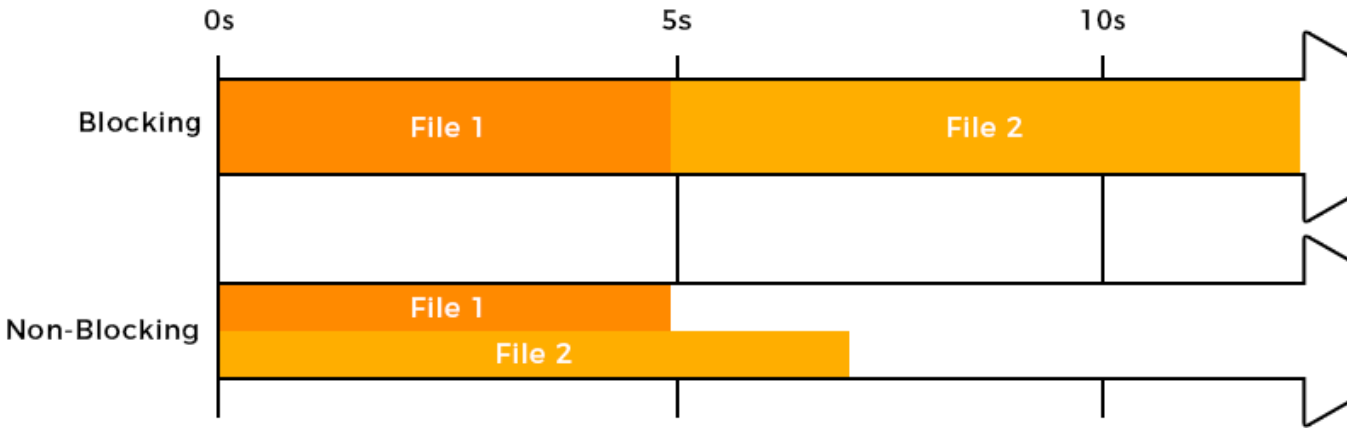
In NodeJs you can take advantage of JavaScript promises. One of the benefits of this is that we can write non-blocking code. To demonstrate how this works, I'll give you an example in psuedo code for reading a file from the filesystem.

Blocking:

```
read file from filesystem, set equal to "contents"
print content
do something else
```

Non-Blocking

```
read file from filesystem
    Whenever we're complete print contents (callback)
do something else
```



SETTING UP OUR DEV ENVIRONMENT / BUILD

The front-end part of this was really easy. We used angular-cli to generate a new project. In the future this also gave us the advan generating new components, services, pipes and much more. For the backend we decided to go with gulp. We added some task transpile our server site TypeScript files to javascript so that node can execute it. For local serving we created a sequence task tha combines running 'ng build' from the angular-cli and a gulp task to use 'nodemon' for running our server and restarting on chan working on the frontend, doing an 'ng build' was a bit too slow, therefore we added a –standalone flag, to the serve task so that just build the backend application and do the frontend serve with 'ng serve' which is a lot more performant than having to do a ' on every change. Since we are using TypeScript throughout the application, it only felt right to use the TypeScript version of gulp takes a little effort to get used to, but once you get the hang of it it makes writing gulp tasks a lot more fun and less error prone. U provided decorators our gulp tasks look something like the following:

```
@Task()
    public environment(cb) {
        return gulp.src('./dist/app/server/config/mongo.connection.js')
                    .pipe($.if((yargs.env === 'prod'), $.replace('mongodb://localhost:27017/stairway',
                    .pipe(gulp.dest('./dist/app/server/config'));
    }
```

and create sequence tasks with:

```
@SequenceTask()
    public mocha() {
        return ['buildApp', 'runMochaTests'];
    }
```

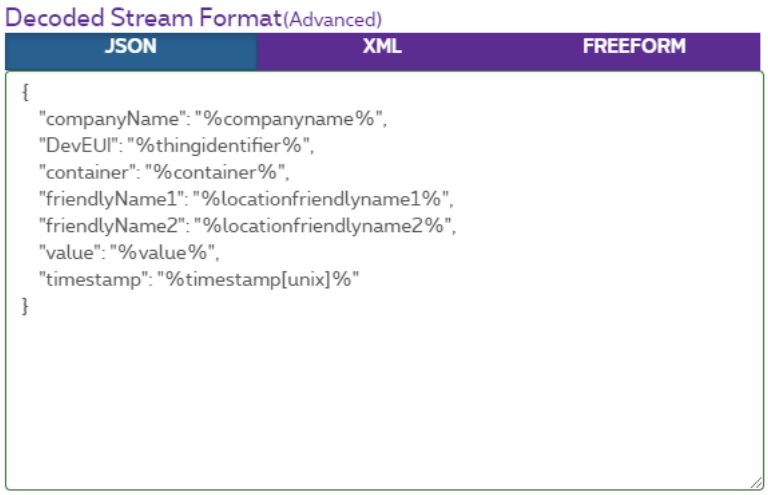
Now that we have a gulpfile.ts file, we need to ensure that the gulpfile gets transpiled as well, we did this by adding an npm scrip we can use 'tsc' to transpile the file and make sure we are using the latest changes every time we use gulp. (to get the tsc comma typescript globally with npm)

BUILDING STAIRWAY TO HEALTH

After setting up our dev environment, database and getting a simple application up and running it's time to start implementing c features.

Receiving data from MyThings

So first things first, on MyThings we took a look at how we were going to structure the data that was going to be streamed to the Health application.



In the MyThings application every sensor can have a friendlyName1 and 2, we used these to specify which tower and which floor represent. The sensors send a lot more data than just the magnetic pulse counts, therefore we needed the container field, to be filter on 'counter' logs only (however, we store the other messages as well, maybe for future use). The 'value' field is the amount c sensor was triggered, in other words, the actual counts. And ofcourse a timestamp since we will show the data in time based gra

The timestamp represents the time that the sensor has sent it's message to the MyThings application, we also wanted to keep tr when our (Stairway applicaition) has received the log, so before saving we added one extra field to store this in our database.

After we defined our model / schema of our logs, it was simply adding an endpoint to our express router and our first feature was Well not exactly, we needed to trigger an event to refresh the data on our dashboard, but we'll get back to this later.

The Dashboard

Since we created an angular(4) application, we took advantage of the great features of angular-cli which makes it really easy to g project up and running and generate new components, services and much more. We started working on our dashboard that shc building icons with the total counts per day, week and total, added a nice graph below to show an overview of the competition c event.After adding the Proximus styles and importing the highcharts library from npm, the most important part of the applicatio take shape.

In the mean time we started to get an idea on how to model our data to display it in the charts, since we got it running with some data in the frontend. Thus we were able to start implementing our dashboard api endpoints.

Mongo Aggregates

So for displaying the daily, weekly and total counts below the buildings, we had to get this data from the database, keeping in mi would have to iterate over millions of sensor logs (at the time of writing this blogpost 1.4 million over 4 months). We had to make was performant. This is where the mongo aggregates come in handy. In stead of (say) looping over the results and adding them mongo take care of this with the '$sum' operator which in code looks like the following:

```
this.sensorLogModel.aggregate([
                {$match: {container: 'counter', value: {$ne: 0}}},
                // group them by fn1 (tower) and add up all 'value' fields
                {$group: {_id: '$friendlyName1', total: {$sum: '$value'}}}
            ]);
```

Remember, we store all the logs, but we only need counter logs. So for a little more performance, we leave out the ones with value 0 them in the weekends), that's what the $match is for The result: an array with objects that have a '_id' field with 'friendlyName1' as a 'total' field with the sum of all (counter) values per tower. We repeat this for daily and weekly, but add a start and end date (whi simply create with TypeScript). $match then looks something like this:

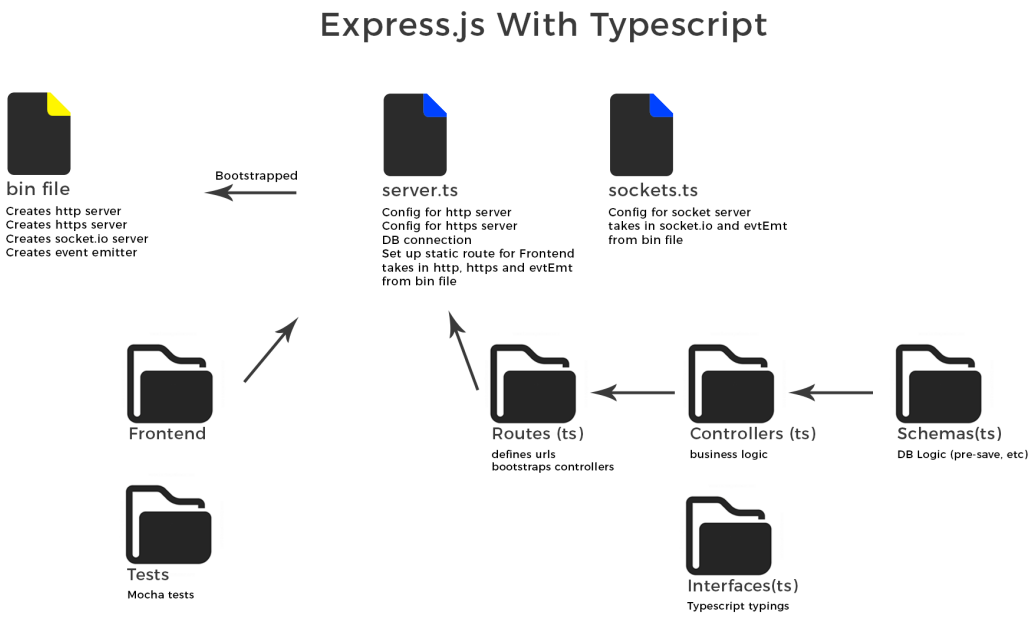
```
{$match: {container: 'counter', value: {$ne: 0}, timestamp: {$gt: start, $lt: end}}}
```

Later on we added some more calls to get the data by timespan and location for the more detailed chart data, but you get the id simply edit the timestamps or friendlyName1 (also by friendlyName2 on the hourly chart, wich displays the hourly data per floor

Socket.IO

Now that we have data that can be retrieved and displayed on the frontend, time to implement some way to let our frontend app know when we receive some new data, so that it in turn can do a request for that new data.

For this one to be clear we're going to skip ahead in time and show a high level scheme of how the application is made up.



The bin (js) file is where we create our http, https and socket servers. To communicate between them, we use the node event em server.ts file (let's call it the app) gets bootstrapped on to these servers and when creating the app, we pass the created event em This enables us to listen and broadcast events back and forward. The event emitter emits events between the backend services a socket.io server emits events to our front-end application.

So in our case, to let the frontend know when the sensor-log endpoint has received a message, we emit a 'log-received' event on event emitter. In the socket IO server we are listening on this event and we broadcast a 'data' event to every connected frontend application. The frontend applications are listening for this 'data' event and refresh their data by calling the dashboard endpoint However, since we have about 60 sensors sending data, this event was triggering quite a lot and with the chart rendering animat our frontend application we had to wrap the 'log-received' in a timeout so that we would only refresh it once every 30 seconds (if received).

I've picked a few lines of code from our bin file to demonstrate how we pass the eventEmitter when bootstrapping our applictior http and https services from node.

```
const server = require('../dist/app/server');
const http = require('http');
const https = require('https');
const events = require('events');
const eventEmitter = new events.EventEmitter();

const httpServer = http.createServer(server.Server.bootstrap(eventEmitter).app);
const httpsServer = https.createServer(options, server.Server.bootstrap(eventEmitter).app);
```

After that, we bootstrap the created https server on to the socket.io application. It too gets the same EventEmitter instance passe constructor.

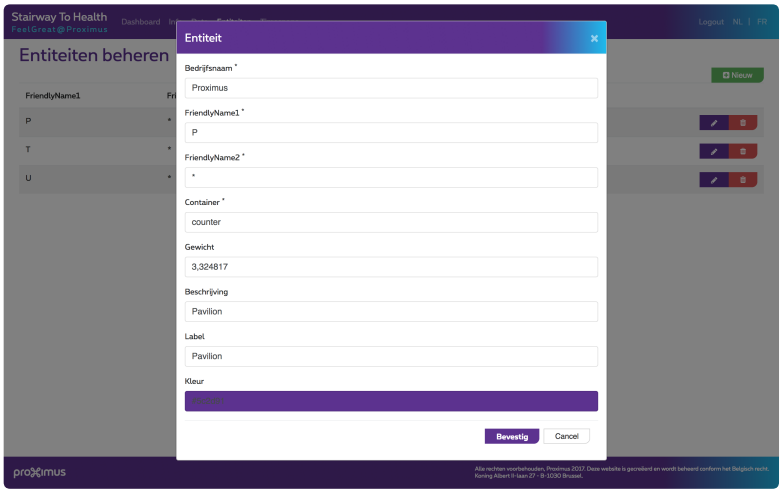
```
const io = require('socket.io')(httpsServer);
const sockets = require('../dist/app/sockets');
const ioApp = sockets.Sockets.bootstrap(io, eventEmitter).io;
```

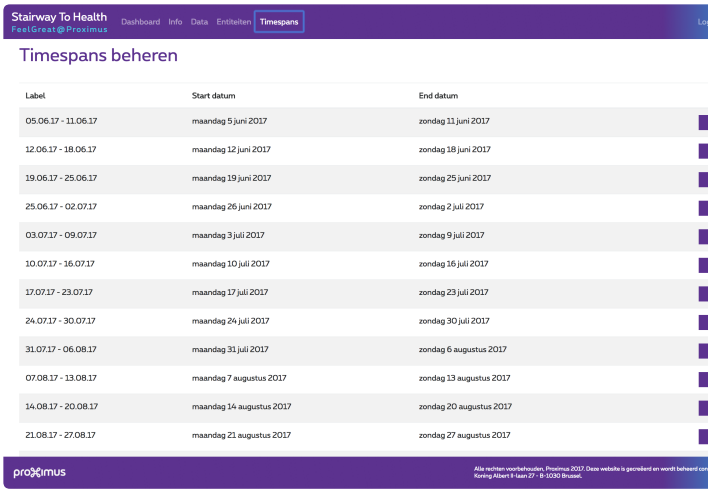
In our sockets file, the method that gets executed will listen on the 'logsReceived' from our passed EventEmitter, and emits a 'dat on our 'io' instance.

```
public sockets(eventEmitter, io){
    eventEmitter.on('logsReceived', (logs) => {
        io.of('/socket/').emit('data', logs);
    });
}
```

CONFIGURATION CRUD

Since we did not want our configuration to be hard coded, we added some configuration screens to be able to change the times entities (towers).





By the way, 'gewicht' in the first image stands for weight. To make sure the ratio's are fair, we made sure that every tower has a 'w multiply it's log values by. These weights are calculated by the amount of employees/tower, with the largest tower having a weig

Lets take a look at how we set up our backend structure for creating crud endpoints. In our '/routes' directory we keep all files tha the urls and methods of every endpoint, and tell it which controller and method to use:
timespan.route.ts

```
router.get('/timespan/', (req: Request, res: Response, next: NextFunction) => {
    this.timespanController.getTimespanList(req, res, next);
});
router.post(('/timespan/', this.authenticate, (req: Request, res: Response, next: NextFunction) => {
    this.timespanController.createTimespan(req, res, next);
});
```

next under our '/controllers' directory we have our controllers where all our functionality/logic is
timespan.controller.ts

```
public getTimespanList(req: Request, res: Response, next: NextFunction) {
    return this.timespanModel.find({}, [], {sort: {start: 1}})
```

```
        .then((result) => {
            res.json(result).status(200);
        }, (err)=>{
            res.statusCode = 500;
            res.statusMessage = err;
            res.send();
        });
    }
```

AUTHENTICATION

To prevent everyone from changing these configurations of course we had to add some authentication functionality. As you can router code above, we created some authentication middleware so that on every route that we want the user to be authenticate simply add 'this.authenticate()' to the route. This checks a JWT token in the headers. We check the token to be valid. If it's not val an 'unauthorised' response, and if it is valid, we decode it and add it as a user object on the request. This way we can access it in controller and do some logic depending on it's role, etc. 'this.authenticate' is a method we added to the 'core.route.ts' every rout this superclass so that we can put common code and middleware in this file.

JWT stands for JSON Web Token and is a JSON-based open standard for creating access tokens that assert some number of clai example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could th that token to prove that it is logged in as admin.

DEPLOY

Finally we deployed it to the Proximus Datacenter and watched the Proximus employees take on the challenge.

Full stack JavaScript developer, specializing himself in backend development with Express. With a solid background of frontend/hybrid mobile development. Part of C core 'Internet of Things' team where he has collaborated on some challenging and IoT applications.

Kevin is a Principial Java consultant at Ordina, passionate about all Java and JavaS related technologies. In his roll as Competence Leader Internet of Things he uses hi knowledge of building custom software to build innovative solutions using new tec Currently focussing on the internet of things and sensor networks using LoRa. Loves with gadgets.

Blarenberglaan 3B,
2800 Mechelen, Belgium

+32 15 29 58 58

jworks@ordina.be

Facebook
LinkedIn
Google+
YouTube
GitHub
RSS Feed