

*Forfatter:*  
14.06-79 Sebastian O. Jensen GJX653

KEA

NODEJS

---

**Keywords:** Node.js, MongoDB, Express, AngularJS, Docker,  
RESTfull, Web service

---

DECEMBER 13, 2015

# Contents

0.1	Abbreviations . . . . .	2
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Abstract . . . . .	3
1.2	Overview of this report . . . . .	3
1.3	Background . . . . .	3
<b>2</b>	<b>Theoretical background</b>	<b>4</b>
2.1	REST . . . . .	4
2.2	Database system . . . . .	4
2.2.1	Relational database database . . . . .	4
2.2.2	NoSQL database . . . . .	4
2.3	Node.js . . . . .	5
2.4	Docker . . . . .	6
<b>3</b>	<b>Design of the backend</b>	<b>7</b>
3.1	Architecture . . . . .	7
3.2	Requirements of the backend . . . . .	8
3.2.1	MongoDB . . . . .	9
3.2.2	Nodejs . . . . .	9
<b>4</b>	<b>Implementation</b>	<b>9</b>
4.1	Folder structure . . . . .	9
4.2	Nodejs and Express . . . . .	10
4.2.1	Folder structure . . . . .	10
4.2.2	NPM, package.json and modules . . . . .	10
4.2.3	aap.js . . . . .	11
4.2.4	Model . . . . .	12
4.2.5	Routes and authentication . . . . .	13
4.2.5.1	routes/index . . . . .	14
4.2.5.2	routes/users and authentication . . . . .	14
4.2.6	Views . . . . .	16
4.2.7	bin/www and start.sh . . . . .	17
4.3	Docker Mongo and runsystem.sh . . . . .	19
4.3.1	Dockerfile . . . . .	20
4.3.2	hosting . . . . .	21
<b>5</b>	<b>Conclusion</b>	<b>22</b>
<b>6</b>	<b>References</b>	<b>23</b>

## 0.1 Abbreviations

IoT	Internet of Things
NoSQL	not only SQL
RDBMS	Relational Database Management Systems
REST	Representational State Transfer
SQL	Structured Query Language

# 1 Introduction

## 1.1 Abstract

Forecasts say that in 2020 there will be 25 billion devices connected to the Internet. In 2014 there was around 3.7 billion devices[1]. This put a big demand on common technologies such as relational databases, sessions based communication, etc. This project is a part of a larger project that investigates how to deal with these challenges. The project does this by designing a system that meets these challenges. The system consists of multiple devices which connect in a local network and send information to a backend system over the internet. This is a realistic example of how to deal with the future of IoT. The application consist of the following items

1. Battery powered devices that communicate via the ZigBee protocol.
2. ZigBee/Internet gateway.
3. Backend developed using Node.js and the database MongoDB
4. Frontend developed in AngularJS.

As this project is about the backend system (3) this report does only describe the other parts when this is needed to describe the functionality of the backend.

## 1.2 Overview of this report

In section 1.3 is the background for the system described. Section 2 will give a theoretical background on the technologies which will be used to implement the backend. Section 3 will give a technical description on the implementation. Section 4 contains the conclusion.

## 1.3 Background

Internet of Things is a term that describes networks of physical devices that are connected to the Internet. This can be anything like sensors, wearables, fridges, heating systems, light balls and what ever could be imagined to connect to the Internet. As mentioned there is a high growth in the number of these devices. This project is about solving a common problem in leisure harbors. In leisure harbors there are a limited number of moorings. Therefore it will be an advantage if a sailer can see if there are free moorings/berths in a harbor before arriving.

## 2 Theoretical background

### 2.1 REST

REST stand for Representational State Transfer. Its a method for designing networked applications. One of the most important constraints in REST is that it is stateless. Stateless means that every request should contain all information to process the event. In this way session state is stored on the client which avoid the need for sessions. When using sessions the server will need to hold information about all the devices sessions while communicating. This makes it more difficult to scale the system as request can not so easily be load ballanced between multiple servers. Another thing is that it put a higher demand on the server if it needs store session state for each connected object. Whith rest load balancing is easy it does not matter to which server the request is sent as the request holds all information to process the event.

### 2.2 Database system

#### 2.2.1 Relational database database

Relational Database Management Systems (RDBMS) store data in tables. Access or modification to data is done using Structured Query Language (SQL) . It was developed in the 1970s and has been the de facto standard for many years.

#### 2.2.2 NoSQL database

Doing the 2000s an alternative to the RDBMS start to become popular. The NoSQL Databases. NoSQL stands for “not only SQL”. There are different ways the NoSQL can store data. But what they have in common is that they use an object orientated approach. Some of the NoSQL databases are graph databases that are good at handling graph data. This could be data in a social network about who is connected to who. Other stores data in documents or wide-columns [5]. A NoSQL database is often very easy to scale as the developer can just start up another instance of the DB and the DB will then by it self distribute data among the DB instances. With SQL DBs it is also possible to divide data on more servers. But due to the way data is stored this require more work to do. A NoSQL DB is often many times faster than a SQL DB depending on the job it has to do. Three important things to consider when choosing a datastore for a distributed system are the following.

- Consistency.
- Availability.
- Partition tolerance.

Imagine that a database system is replicated out on to servers, server A and B. If data is updated on server A and the same data immediately being retrieved from B before the data has been replicated then you will get a wrong result back. While A and B are not fully synchronised the system is in an inconsistent state. Some datastores are highly consistent which means that it is not possible to receive “wrong” or not updated data. When a system is highly consistent it means that all nodes see and share the same data. Availability is about how available the system is. A highly available system guarantees that all requests will receive a response within short time. A partition tolerant system guarantees that the system will continue to work even when parts of the system are not accessible due to network link errors.

A theorem called the CAP theorem says that you can only choose two out of the three properties. This makes sense if you think about it. Imagine a system should continue to function when the link between server A and B breaks, at the same time be consistent and be able to answer all requests immediately.

Each of the three properties can have different importance depending on the system. Eg. money transfers need high consistency as this is very transaction critical.

## 2.3 Node.js

Node.js is a javascript runtime environment for developing server side applications. It uses Google's V8 engine. Traditionally javascript was only being used as a scripting language on the client side. But Node.js makes javascript available on the server side as well. What makes node.js special is its architecture. It's single threaded which means that it can only do one thing at a time and has one call stack. Imagine what happens when a resource requiring task is being executed on the stack. Then the program can't do anything else. This is called blocking. To avoid blocking the resource requiring task can be handed over to node.js underlying C++ APIs where the task will be processed outside node.js call stack. When handing over the task a callback function also needs to be provided. When the task is done the callback function is being put on the callback queue. When there are no tasks on the call stack, tasks from the callback queue will be processed. This mechanism is called the event loop. And it's important to never block the event loop. By blocking the event loop is happening if a resource requiring

job is handled synchronously in the node application. Instead the job should be handled asynchronously with the callback feature. This architecture makes node.js extremely fast and able to handle many concurrent connections compared to other server-side platforms. If REST is used in combination with load balance between more servers the system is very powerful and easily scalable. Therefore it is perfect for applications where there is a high load.

## 2.4 Docker

Docker can in some way be compared to virtual machines but at the same time it is very much different. A virtual machine simulates a computer which gives the possibility to run another operating system on top of the host system. Eg. you can run a Windows operating system in a virtual machine running on your Linux system. This gives a high overhead as the computer now needs to run two full operating systems and the simulation done by the virtual machine program also takes resources. In contrast Docker takes another architectural approach. Docker is containers which share the kernel and other parts of the operating system with the host system. In this way they are a lot more lightweight than virtual machines. Docker does only run on Unix and you can only build Unix containers. When starting a Docker container you need to have an image that defines the container. Eg. you can start an Ubuntu 14.04 container from an Ubuntu 14.04 image. There are two ways to obtain an image. One way is to receive it from the Docker hub or you can build your own images using a Dockerfile. A Dockerfile can be as simple as the single line "FROM ubuntu:14.04" which will build an Ubuntu image. For additional settings and installation of programs into the image additional lines are needed. Each line in the Dockerfile is actually a layer. Layers are also shared between different images if they fit for each other. But it will be out of the scope of this report to go into all the details about how Docker handles layers. A Docker container is only supposed to run one application. If a server needs to run more applications then you are supposed to start one container for each application. One of the main advantages of using Docker is that the containers are easy to move around so time is saved on installing servers and dealing with compatibility problems because of different environments. When starting more Docker containers on one system they can easily be connected by linking them to each other. It is also possible to mount host folders inside the containers.

### 3 Design of the backend

This section covers the design of the backend system. To give a better understanding of the role and responsibilities of the backend an brief description of the hole architecture is described in section — below. Then —

#### 3.1 Architecture

An architecture drawing of the hole system can be viewed in fig. 1 below.

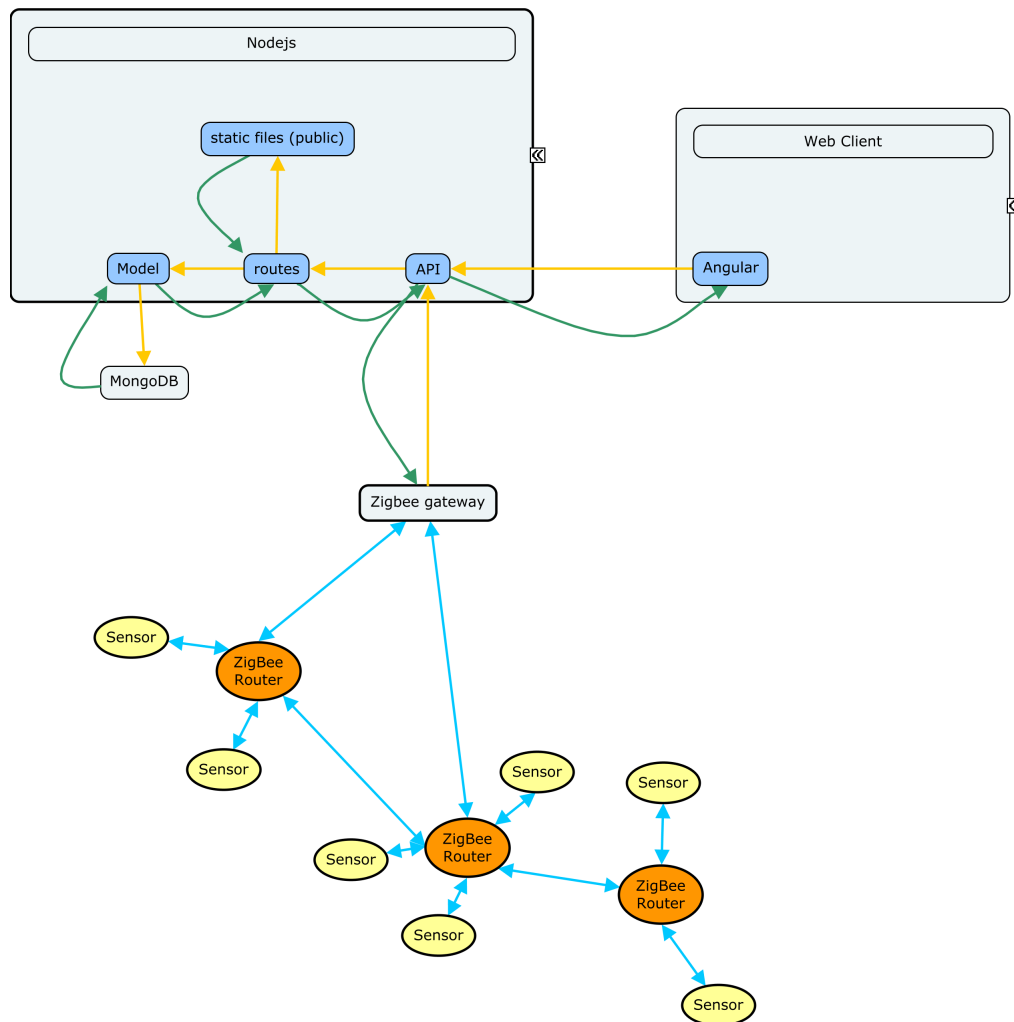


Figure 1: Architecture

Here follow a short description of each component



- Each sensor corresponds to one berth in the harbour. They communicate via the ZigBee routers to the ZigBee gateway. At regular intervals they check if there is a boat at the berth or not. When there are any changes they send a message about the change to the ZigBee gateway.
- The ZigBee gateway translates the messages from the sensors to a JSON message which holds information about the state (occupied / not occupied) of the sensor and the unique id of the sensor. It sends the JSON message as a PUT message to the backend. As the message contains both the id and the new state of the sensor it does not break the RESTful constraints.
- The Node.js application is the heart of the system. The application is responsible for exposing webservices and handling the datastore which in this system is a Mongo database. The application exposes RESTful webservices that are being consumed by the ZigBee gateway and the frontend.
- The Mongo database stores all the data. The data is user information and the information about the state of each berth.
- The webclient runs the frontend application which is developed in AngularJS. Angular is responsible for rendering and presenting the information received from the backend.

### 3.2 Requirements of the backend

To show how the backend work some requirements have been defined and implemented. In a full production ready system additional functionalities are required. These additional requirements will be discussed in the sections below and in the conclusion. But for now we will do with the following requirements.

- Expose a RESTful API which can receive information about the state of each berth in the port and store the information.
- Expose RESTful APIs which make data accessible to the frontend.
- Be able to handle authentication.
- Let admins update users and see all users.

### 3.2.1 MongoDB

As described in the theoretical section a NoSQL database is often many times faster than a relational database[7]. Mongo is an object related database which fits our data as each sensor can be seen as an object. Mongo can be used as a highly consistent and partition tolerant database. It prioritizes these properties on the cost of availability. When mongo is distributed on more servers the consistency is guaranteed if you use locking. You can lock on different levels (document, collection or database level). Locking on collection or database level can be useful when dealing with transactions [6] Therefore Mongo is a good choice to use as a database.

### 3.2.2 Nodejs

Node.js comes with the NPM package manager which can be used to easily install libraries and extensions. For this application the Express web framework is used. Express is unopinionated about how you are building stuff. This means that you Express does not set any rules about how you build your application. This makes it very flexible and a good choice for building web APIs. Express features middleware which give the possibilities to divide different tasks to different middlewares. Eg. all requests to restricted services can be passed by a middleware which checks for authentication. Each middleware can be passed the request and response objects, which the middleware can access and modify. When the middleware is done it passes on the request and response object to the next middleware until a response is sent to the requesting client. Things will be more clear when describing the source code in section —————

## 4 Implementation

### 4.1 Folder structure

The project is structured in folders. A list of folders and files in the root folder is shown in fig 2. The Dockerfile and runSystem.sh files are related to setting up and running the system. The folder mongoVol is a folder where the Mongo database stores its data. README.md is a file used by GIT<sup>1</sup> to describe the project. port.iml is a project file for an IDE which is not relevant. The folder named port contains the Node.js and Express application. The content of this folder will be described in the following section

---

<sup>1</sup>GIT is a version control system. It is outside the scope of this report to explain details about git

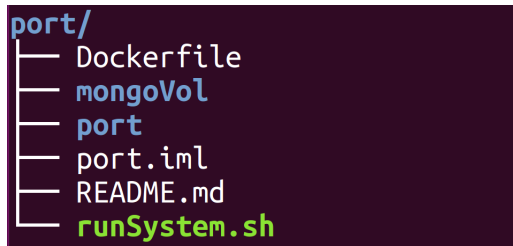


Figure 2: Rootfolder of the project. Blue is folders, white is files, green is executable files.

## 4.2 Nodejs and Express

The project have been build over the MVC pattern by deviding the data model, controler, and the views.

### 4.2.1 Folder structure

In fig 3 the folder structure of the Node, Express and AngularJS is shown. The sub folder public contains the frontend AngularJS application. The other parts is related to the backend application which will be described in details in the next sections.

### 4.2.2 NPM, package.json and modules

As mentioned erlier Node comes with the package manager NPM. NPM packages can be installed globally on the system or local to the project. Packages which an application is dependent on can be specified in the file package.json. When you install a new package to use in you project you can apply the save atribute to the npm command. This will put the dependecy in to the package.json file. You can also edit the file manualy. If you enter the command NPM install all packages in the file will be installed into the folder node\_modules. The content of package.json is showed here.

```
1 {  
2   "name": "port",  
3   "version": "1.0.0",  
4   "private": true,  
5   "scripts": {  
6     "start": "node ./bin/www"
```

```
7   },
8   "dependencies": {
9     "body-parser": "~1.13.2",
10    "cookie-parser": "~1.3.5",
11    "debug": "~2.2.0",
12    "ejs": "~2.3.3",
13    "express": "~4.13.1",
14    "jsonwebtoken": "^5.4.1",
15    "mongo": "^0.1.0",
16    "mongodb": "^2.0.48",
17    "morgan": "~1.6.1",
18    "serve-favicon": "~2.3.0"
19  }
20 }
```

This application are using 10 modules. Here follow a brief description of each module.

- body-parser: Used for parsing the content from and to the body of a html messages. In this project it is used for parsing Json data
- cookie-parser: Used forhandling cookies
- debug: Used by express to write debug messages
- ejs: For handling embedded Javascript templates on the frontend. (Not used by this project. but it comes with express. Could actually be removed)
- express: The web framework
- jsonwebtoken: Used for authentication using tokens
- mongo: a wrapper for the mongoDB driver. (Is not used in this project)
- mongodb: The native driver for Mongo
- morgan: Used for logging
- serve-favicon: Used for seting the favicon

#### 4.2.3 aap.js

In the file app.js the initialication is defined. A big part of this file is generated by express it self. Here an object named app is created which is the nerve of the application. Here folow a description of what has been changed for this project. In the top of the file, needed modules are imported with the requier statement. 2 lines has been added

```

1 var config = require('./config')
2 var db = require('./model/db');

```

s

The first file config is a file that has been made for holding special configuration parameters. For now the only configuration parameter that is handled here is a secret used to create and validate tokens. But more stuff should be moved here. Eg. database url and login would be good to place in the config file.

The line

```

1 app.set('superSecret', config.secret);

```

Sets a variable in the app object to hold the value of the secret.

The connection to the database is established

```

1 db.connect('mongodb://db/zigbee', function(err) {
2   if (err) {
3     console.log('Unable to connect to Mongo.')
4     process.exit(1)
5   }
6   else {
7     console.log('Connected to DB')
8   }
9 })

```

As node js is asynchron it is important to mention

The line

```

1 console.log("running in " + app.get('env') + " mode");

```

Prints the environment mode. This will be explained later.

The line

```

1 app.use(express.static(__dirname + 'public'));

```

set the public folder to the path for static content.

#### 4.2.4 Model

The model handles access to the mongo database. The model makes an abstraction to the application about how access to an underlying datastore is done. In this way the application does not need to concentrate on how data is accessed or which data store is used. In the model folder is one file named db.js. It contains the following functions

- connect: this we saw used in the file app.js for connecting to the database.

- get: To tell the state of the database connection
- close: which is used for closing the connection

In top of the file the mongo driver is imported with the require statement. The code can be viewed in the provide source.

The model folder also contains to other files *apiModel.js* og *userModel.js*

These files defines functions which update and get data from the database. The functions are exposed to the controller which is defined in the routes folder. An example of one of the functions is shown below

```
1 exports.updateBerth = function(json, callback) {
2   var collection = db.get().collection('berths')
3   collection.replaceOne({_id: json._id}, json, {upsert: true
4     }, function(err, response) {
5     callback(err, response)
6   })
7 }
```

This function updateBerth updates the status of a berth. The upsert:true option means that a record should be created if it does not exist. The rest of the functions can be seen in the source files.

The userModel expose the following functions.

- all: Return all the users in the system
- updateUser: update or create a user
- getUser: return a user from the database

See the source files for the full code.

#### 4.2.5 Routes and authentication

The routes control how the application respond to requests. This is done using middleware and pattern matching on the URL and request method. In the *app.js* file the following to lines

```
1 app.use('/', routes);
2 app.use('/users', users);
```

defines that all request with the path starting with users should be handled by the routes defined in routes/users.js and all other requests should be handled by routes/index.js

**4.2.5.1 routes/index** The routes in routes/index.js defines tree routes. Which is the following

- getAllBerths: GET method for returning all the berth and their status as json
- updateBerth: PUT method for updating or creating a berth
- root path: This will send the index file of the angular application.

Below is the code for the updateBerth path

```
1 router.put('/updateBerth', function(req, res) {  
2   json = req.body  
3   apiModel.updateBerth(json, function(err, response) {  
4     res.json(response);  
5   })  
6 });
```

As we see this middleware will only match request of the type PUT and the path /updateBerth. first the body of the request is encoded into the variable json. Next the function updateBerth in the apiModel is called with the json object. When the apiModel has done its magic the response from updating the database is send as the response. This is an other example of using callback. And this is one of the very important places to use the asynchron approach, because database access is one thing that can slow down the system if we were to wait for every database task to finalize before doing anything else.

**4.2.5.2 routes/users and authentication** The routes in the routes/users.js files is a little more complicated. The routes are the following.

- /authenticate: POST method which Handles authentication
- /\*: ALL methods. This catch every request and is used to check if a user is authenticated
- /getAllUsers: GET method for returning all the users
- /updateUser: POST method for adding a user to the system

The first method handles the authentication. The code is provided here

```

1 router.post('/authenticate', function(req, res, next) {
2   received = req.body;
3
4   userModel.getUser(received, function(err, data) {
5     //if there was no data in the db
6     if(data === undefined){
7       res.json({success: false, message: "Authentication
          failed"});
8     } else{
9       if (data.password !== received.password){
10        res.json({success: false, message: "Authentication
          failed"});
11      } else {
12        var token = jwt.sign({userType: 'admin'}, req.app.get
          ('superSecret'));
13        res.json({success: true, message: "you are
          authenticated", token: token})
14      }
15    }
16  })
17 });

```

The function expect to receive a username and a password in the body of the post message. The the coorsponding user is retrieved from the database vie the user model. Then there is no coorsponding user or the password of the user does not mach a response message with text "authentication failed" is send back to the client. If the password mach then a jason web token is created and send back. The token consist of tree parts. A header, payload and signature. In the header we can define the usertype, the payload can contain information about expiration and other detail, the signature is used for checking that the token is valid. The signature is hashed using the secret key which was defined in the config file and set in the app variable seper-Secret. The header and payload are not encrypted so the client can read the conten by decoding it with the base 64 algorithm. So its important not to put sentitive stuff here in case a hacker sniff the token. When a user is outhenticated it should send the web token with all the request. In this app this is being done by setting the token in the header.

When a user access any other path handeled by the user router, the midleware matching the path /\* and regardles of the method will catch everything. This midleware is shown below

```

1 router.all('/*', function(req, res, next) {
2   token = req.headers.token;
3   console.log(token);

```



```

4   jwt.verify(token, req.app.get('superSecret'), function(err,
    decoded) {
5     console.log(decoded) // bar
6     if (decoded){
7       req.authenticated = decoded.userType;
8       next();
9     } else {
10      req.authenticated = null;
11      next();
12    }
13  });
14 });

```

This middleware check if a user is authenticated. It first get the token from the header. Then the token is verified. If the token is valid then a variable named authenticated in the request object is set to the usertype of the user. In this app there are only the usertype admin. If the token is not valid authenticated is set to null. One thing to notice with this middleware is that the callback function get past an object called next. This is used to parse the request on to the next handler. So instead of setting the response the middleware parse on the request which will then be caught by one of the handlers below if any there are any match. Each of the handlers that require authentication ofcourse need to be defined below the middleware that checks for authentication. And the handler can then check the authenticated variable and respond depening on the usertype or if no type is set. Here is an example of the getallusers handler

```

1   if(req.authenticated == 'admin'){
2     userModel.all(function(err, data) {
3       console.log(data);
4       res.json(data);
5     })
6   } else {
7     res.json({message: 'you do not have access to this page'
8       });
9   }
10  });

```

It checks if the user is admin and if so all the users is send back else the messages "you do not have access to this page" is send back.

#### 4.2.6 Views

The view folder contains views which is not used in this project as the frontend is handeled by angularJS. Express defaults to use views in a format called JADE . But for testing it was changed to EJS. But as this is not used in the project. Nothing more will be mentioned about this.

#### 4.2.7 bin/www and start.sh

Running a node application is as simple as running the following command

```
1 node <application>
```

But Express provide an executable file bin/www. In the top of this file the line

```
1 #!/usr/bin/env node
```

defines that when running the file it should be run with the node application. Instead of using node this project uses nodemon instead. The difference between node and nodemon is that nodemon listens for file changes and automatically restarts when any changes have been made to the source files. This makes development faster as you do not need to manually restart node to see changes. Therefore the line has been changed to

```
1 #!/usr/bin/env nodemon
```

This project uses another wrapper *start.sh* the code in this file is shown below.

```
1 #!/bin/bash
2 cd /www
3 npm install
4 #NODE_ENV=production /www/bin/www
5 /www/bin/www
```

When this script is run it first changes directory into the folder /www on the system. This is where the application should be placed. Then all dependencies are installed. The next line which is commented out can be used to run the application in production instead of development mode. Running in production mode makes some changes to how the application is running. Eg. error messages are not sent to the clients and the performance is also better. Last line runs the application.



Figure 3: Folder structure of Nodejs, Express and AngularJS. Blue is folders, white is files, green is executable files.

### 4.3 Docker Mongo and runsystem.sh

The application is run using the docker echo system. This is all handled by the runSystem.sh script. Downloading the project and running the script on a system that has docker installed will handle all the setup and running the application. The code in the file is showed below.

```
1 #!/bin/bash
2 # uncomment all lines below to make a fresh build of the
   system
3
4 docker stop $(sudo docker ps -a -q)
5 docker rm $(sudo docker ps -a -q)
6 rm -rf port/node_modules
7 docker build -t seeeb/port .
8 docker pull mongo
9 docker run --name thisMongo -v ~/port/mongoVol:/data/db -it -
   d mongo:latest
10 docker run -d --name nodejs --link thisMongo:db -p 80:80 -it
   -v ~/port/port:/www seeeb/port
```

- Line 1 define that its a script which should be run by bash
- Line 4 use the docker ps command to get all running containers and fetch it to the docker stop command. In this way we are shure no other containers are running.
- Line 5 Remove all the containers. (remeber a container is an instance of a docker image)
- Line 6 Removes all the modules from the node application to forse a clean install of all the modules.
- Line 7 builds a docker images named seeeb/port using the Dockerfile. This docker images run the node application and will be explained in the next section.
- Line 8 pull the latest official mongo image from the docker hub.
- Line 9 Starts a container of the mongo images, name it thisMongo and maps the host folder /port/mongoVol in the container folder /data/db. The -d option demonize the container so it is runing in the background.
- Line 10 Starts a container using the images seeeb/port and name it nodejs.

- “-d” tell docker to demonize the container. If you want to see the output from node, this should be removed
- “-link thisMongo:db” tell that this container should link to the container named thisMongo with the alias db. Note that inside the node application when defining the address to the mongo database we only need to specify “db” as the address.
- “-p 80:80” map the external port 80 of the host system to external port 80 of the the node container
- “it” tell that we want to interact with the container and get a shell. in this way the output from the node application is printed to the screen of the host system.
- “-v ”/port/port:/www” mounts the host folder /port/port in the container folder www.

Its important that the project is placed in the home folder of the host system because the shell script expect to find the files here.

#### 4.3.1 Dockerfile

The Dockerfile defines the node images. See the code below

```
1 FROM ubuntu:14.04
2
3 RUN apt-get update && apt-get install -y git python build-
  essential curl nano wget libkrb5-dev
4 RUN wget https://nodejs.org/dist/v4.2.1/node-v4.2.1-linux-x64
  .tar.gz
5 RUN sudo tar -C /usr/local --strip-components 1 -xzf node*
6
7 RUN npm install express -g
8 RUN npm install express-generator -g
9 RUN npm install -g nodemon
10
11 CMD /www/start.sh
12 EXPOSE 80:80
```

The file is pretty much self explanatory. First we specify to start from an ubuntu 14.04, then all the the dependent packages are installed. Then express and nodemon are installed globally. The line *CMD /www/start.sh* tell that this script should be run when the system has started. *EXPOSE 80 : 80* Make port 80 available as a port that can be accessed on the container.

### 4.3.2 hosting

The project is hosted on an Amacon EC2 ubuntu server. And a dns record pointing the subdomain port.lapela.dk to it has been seed.

The function can be tested by sending json requests using Chrome Postman to the server or by using the angular frontend application by accessing the the address in a browser. The Angular application provide four functions which is the following

- Berth: Whitch shows all the berth and their status
- Users: Which shows all the users if you are outhenticated
- Add user: Which let you add a user if you are loged in
- Login: Which authenticated a user

A user account with the username "admin" and password "nodeJSkursus" can be used to test the system. As this server is only running to show the system is working it is ok to do all kinds of test like adding users, updating berths status, etc. Also note that under add user and login some fields is defined in the bottom that is bind to response message and the stuff you enter in the form. This is to show what goes on under the hood. This would ofcourse not be there in a production version of the system.

For testing the updateBerth API a simulator has been developed using Java. It first choose some random berth id's and then start sending random status update messages simulating status change. This will result in some new berth are being added to the system and that those new berth will be updated randomly. The code of the simulator is handed in together whit this report.

## 5 Conclusion

The aim of the project is to show how a realistic Node.js application can be build also taking into account performance and scalability. It has been showed that Node.js together with MongoDB is a high performance system that easely scale. Looking at the Life in Vista Prints test [7] comparing Mongo with a SQL database it is clear that the mongo database is faster than a realtional database when working with object data. There some things that would need improvement for an system put into production. This is that passwords should never be stored in the database as clear tekst. The password strength should also be checked when creating a user and it should not be allowed to have a week password. Right now you can actually create a user with no username or password. It is also important that the access to the side is using ssl. If not, everybody can grap tokens or passwords flying over the network. Another thing that should also be protected with JWT is the updateBerth API which at the moment can be mish used by anyone

All in all this project conclude and show how Node.js, Mongo and Express can be used to build a backend system with high performance and scalerbility

## 6 References

- [1] Gartner, *An American information technology research and advisory firm*, <http://www.gartner.com/newsroom/id/2905717>
- [2] ZigBee Pro Specifications, *ZigBee Alliance*, Full specification can be downloaded from bottom of this page <http://www.zigbee.org/zigbee-for-developers/network-specifications/zigbeepro/>  
ZigBee Document 053474r20 September 7, 2012 10:19 pm Sponsored by: ZigBee Alliance Accepted by ZigBee Alliance
- [3] ZigBee Alliance, *ZigBee Pro, Technical Summary*, <http://www.zigbee.org/zigbee-for-developers/network-specifications/zigbeepro/>
- [4] 802.11ac A Survival Guide, *Matthew S. Gast*
- [5] MongoDB, *What is NoSQL*, <https://www.mongodb.com/nosql-explained>
- [6] MongoDB manual. *Concyrency*, <https://docs.mongodb.org/manual/faq/concurrency/>
- [7] MongoDB vs. SQL Server's XML Data Type *Lifeinvistaprint.com*, <http://lifeinvistaprint.com/techblog/mongodb-vs-sql-servers-xml-data-type/>
- [8] AlfaZeta *flipdots specification*, <http://www.flipdots.com/electromagnetic-status-indicators.html#.Vi8uApcy1hE>
- [9] Texas Instruments *CC2530 Specifications*, <http://www.ti.com/lit/ds/symmlink/cc2530.pdf>
- [10] Texas Instruments *Measuring Power Consumption of CC2530 With Z-Stack*, <http://www.ti.com/lit/an/swra292/swra292.pdf>
- [11] Ultrasonic sensor HC-RF04 *ELEC Freaks*, <http://www.electroschematics.com/wp-content/uploads/2013/07/HCSR04-datasheet-version-1.pdf>
- [12] Specification of LM 17500 battery *SAFT batteries*, [http://www.saftbatteries.com/force\\_download/LM17500\\_datasheet\\_0515.pdf](http://www.saftbatteries.com/force_download/LM17500_datasheet_0515.pdf)
- [13] Specification of Raspberry model 2 *Raspberry Pi*, <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>



- [14] ZNP interface specification *Texas Instruments*,  
[http://e2e.ti.com/cfs-file/\\_\\_key/  
communityserver-discussions-components-files/158/3286.  
CC2530ZNP-Interface-Specification.pdf](http://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/158/3286.CC2530ZNP-Interface-Specification.pdf)

# APPENDICES