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

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 bassed 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 difficoult to scale the system as request can not so easely be load ballanced between multiple servers. Another thing is that it put a higer 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.

- Consitency.
- Availerbility.
- 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 imediatly being retrieved from B before the data has bein replicated then you will get a wrong result back. While A and B are not fully syncronised the system is in an inconsistant state. Some datastores are higly consitent which means that it is not posible to receive "wrong" or not updated data. When a system is higly consistant it means that all notes se and share the same data. Availerbility is about how availerble the system is. A highly availerble system garantie that all request will receive a response whitin short time. A partition tolerant system garantie that the system will continiu to work even when parts of the system is not acceserble do to network link erros.

A theorim called the CAP theorim says that you can only choose to out of the tre properties. This makes sense if you think about it. Imagine a system should continiue to function when the link between server A and B breaks, at the same time be consistent and be able to answer all request imediantly.

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

2.3 Node.js

Node.js is a javascript runtime environment for developing server side aplication. It use googles V8 engine. Traditionaly javascript was only being used as a scripting language on the client side. But Node.js makes javascript availerble on the serverside as well. What makes node.js speciel is its architecture. Its single threded which means that it can only do one thing at a time and has one call stack. Imagine what happens when a resource requireing task is being executed on the stack. Then the program can't do anything else. This is called bloking. To avoid bloking the resource requireing task can be handed over to node.js underlying c++ APIs where the task will be proccesed outside node.js call stack. When handing over the task a callback function also need 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 mecanisem is called the eventloop. And its important to never block the eventloop. By blooking the event loop is happening if a resource requireing

job is handeled syncroious in the node application. Instead the job should be handeled asyncrone whit the callback feture. This architecture makes node.js extreamly fast and able to handle many concurent connections compared to other serverside platforms. If REST is used in combination with load balance between more servers the system if very powerfull and easely scalerble. 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 simulate a computer which gives the posibility to run another oporating system on top of the host system. Eg. you can run a windows oporating system in a virtual machine running on your linux system. This gives a high overhead as the computer now needs to run to full oporating systems and the simulation done by the virtual machine program also take resources. In contrast Docker take an other arcitectual aproach. Docker is containers which share the kernel and other parts of the oporation 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 conainers. When starting a docker container you need to have an images that defines the container. Eg. you can start an ubuntu 14.04 container from an ubuntu 14.04 images. There are to ways to obtain an images. One way is to receive it from the Docker hub or you can build you own images using a Dockerfile. A docker file can be as simple as the single line "FROM ubuntu:14.04" which will build an ubuntu image. For aditional settings and instalation of programs into the images aditional lines is needed. Each line in the Dockerfile is actualy a layer. Layers is 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 suposed to run one application. If a server needs to run more applications then you are suposed 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 safed on instaling servers and dealing with compability problems because of different enviroments. When starting more docker containers on one system they can easely be connected by linking them to each other. It is also posible 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 acrhitecture drawing of the hole system can be viewed in fig. 1 below.

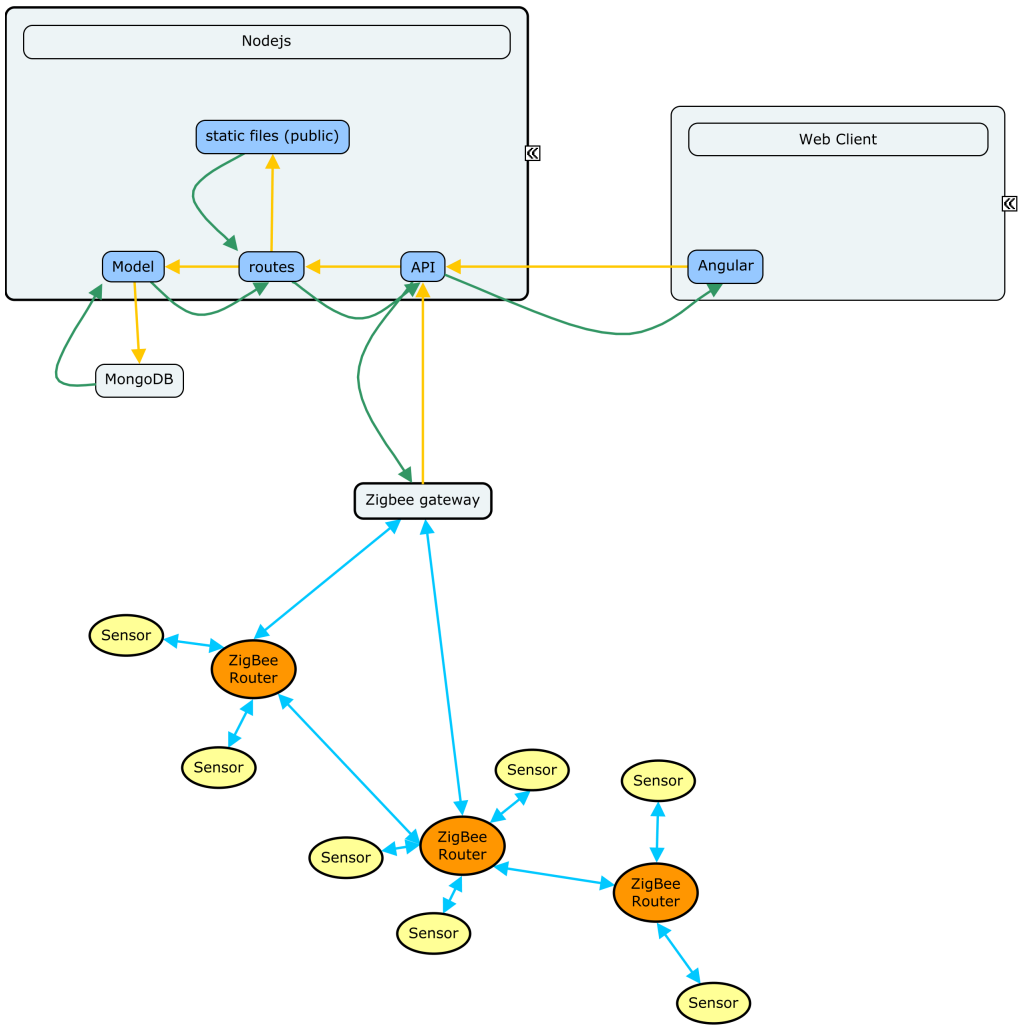

Figure 1: Architecture

Here follow a short description of each component

- Each sensor coorspond to one berth in the harbour. They comunicate via the ZigBee routers to the ZigBee gateway. At regular interval 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 translate the messages from the sensors to a JSON message which holds information about the state (ocupied / not ocupied) of the sensor and the unic id of the sensor. It sends the json message as a PUT messages to the backend. As the message contains both the id and the new state of the sensor it does not break the RESTfull constraints.
- The Node.js application is the heart of the system. The application is responsible exposing webservices and handle the datastore which in this system is a Mongo database. The application expose RESTfull webservices that are being consumed by the ZigBee gateway and the frontend.
- The Mongo database store all the data. The date 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 render and present the information received from the backend.

3.2 Requirements of the backend

To show how the backend work some requirements has been defined and implemented. In a full production ready system aditional functianlities is required. These aditional requirements will be discused in the sections below and in the conclution. But for now we will do with the following recurements.

- Expose a RESTfull api which can receive information about the state of each berth in the port and store the information.
- Expose RESTfull api's which make data accesible to the frontend.
- Be able to handle authentication.
- Let admins update users and see all users.

3.2.1 MongoDB

As descibed 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 higly consitant and partition tolerant database. It priorities these proberties on the cost of availerbility. When mongo is distributed on more servers the consitancy is garantied if you use looking. You can look on different levels (document, collection or database level). Locking on collection or database level can be usefull when dealing with transactions [6] Therefore Mongo is a good choise to use as a database.

3.2.2 Nodejs

Node.sj comes with the NPM package manager which can be used to easely 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 you application. This makes it very flexible and a good choice for building web APIs. Express feature midleware which give the posibilities to devide different task to different middle ware. Eg. all request to restricted services can be passed by a midleware which check for authentication. Each middleware can be passed the request and response objects, which the midleware can access and modify. When the midleware is done it passes on the request and response object to the next middleware until a response is send 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 store its data. README.ml is a file used by GIT[1] to describe the project. port.iml is a projectfile for and IDE which is not relevant. The folder named port contains the Node.js and Express application. The content of this folder wil be described in the following section

[1]GIT is a version control system. It is outside the scope of this report to explain details abput git

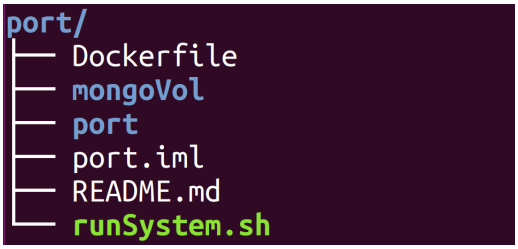

Figure 2: Rootfolder of the project. Blue is folders, white is files, green is executerble files.

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 globaly 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 speciel configuration parameeters. For now the only configuration parameeter that is handeled here is a secret used to create and validate tokens. But more stuff shuld 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 estabilised

```
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 asyncron 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 consentrate on how data is accessed or which data store is used. In the model folder is one file named db.js. It contains the folowing tree 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 bee 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 the to the controler 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
      }, function(err, response) {
4     callback(err, response)
5   })
6 }
```

This function updateBerth updates the status of a berth. The upsert:true option means that a record shoud be created if it does not exist. The rest of the functions can be seen in the source files.
The userModel expose the foloving 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 maching 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 reguest with the path starting with users should be handeled by the routes deffined in routes/users.js and all other requests should be handeled 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 theis 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 midleware will only mach 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 wery important places to use the asyncron aproach, 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 litle 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

```
router.post('/authenticate', function(req, res, next) {
  received = req.body;

  userModel.getUser(received, function(err, data) {
    //if there was no data in the db
    if(data === undefined){
      res.json({success: false, message: "Authentication
         failed"});
    } else{
      if (data.password != received.password){
        res.json({success: false, message: "Authentication
           failed"});
      } else {
        var token = jwt.sign({userType: 'admin'}, req.app.get
           ('superSecret'));
        res.json({success: true, message: "you are
           authenticated", token: token})
      }
    }
  })
});
```

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

```
router.all('/*', function(req, res, next) {
  token = req.headers.token;
  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 seting the response the middleware parse on the request which will then be catched by one of the handlers below if any there are any match. Each of the handlers that requiere 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   });
```

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 runnng the following command

```
1 node <application>
```

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

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

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

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

This project use 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 in to a the folder /www on the system. This is were 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 make some changes to how the application is running. Eg. error messages is not send to the clients and the performance is also better. Last line run the application.

4.3 Docker Mongo and runsystem.sh

The

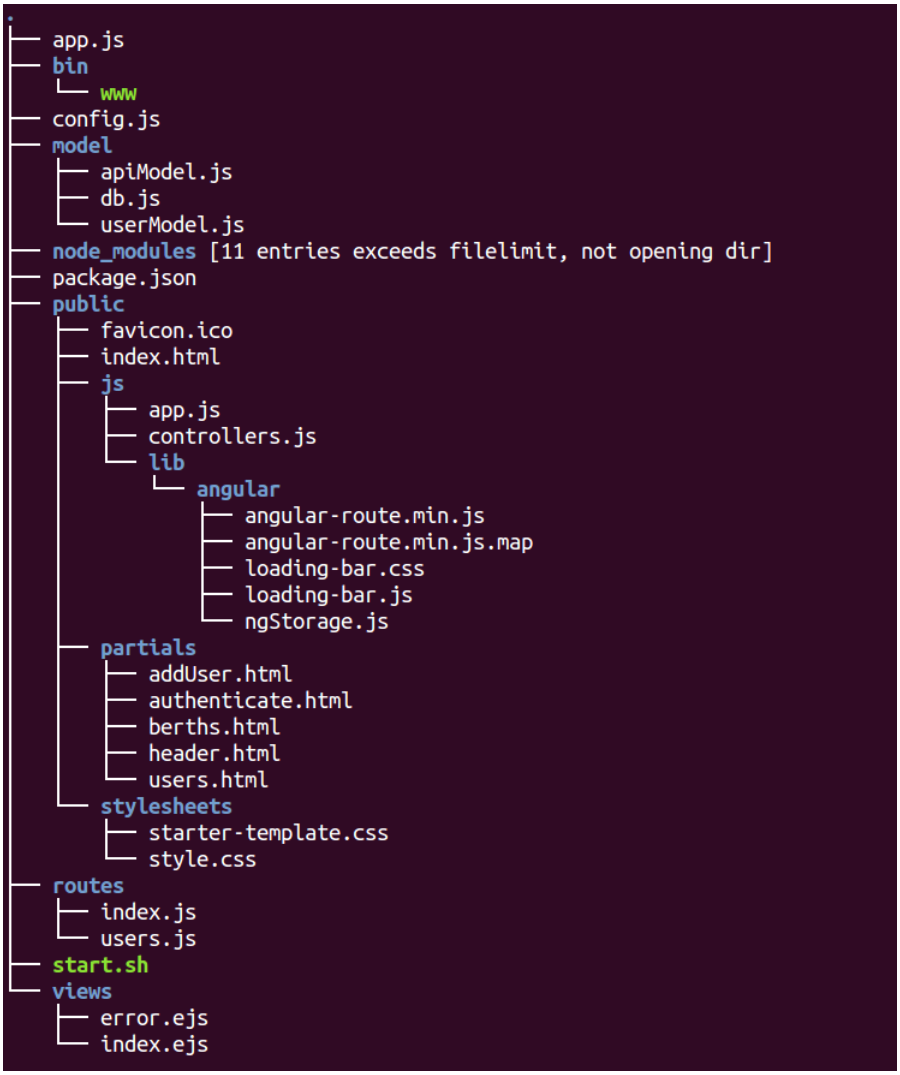

Figure 3: Folder structure of Nodejs, Express and AngularJS. Blue is folders, white is files, green is executerble files.

5 Conclusion

encrypt password services instead of logic in the controlers. ssl sertificate

The aim of the project is to purpose how a realistic IoT system can be implemented taking all parts into consideration. Focus has been on how the devices can be designed so they can run on battery for many years and how they communicate. It has been show that by using the CC2530 SoC from Texas Instruments together with flipdots it is fully possible to power the devices with a relatively small battery. It has been shown that the ZigBee protocol can be used to provide realiable transfer in a mesh networking topology so a network can be spanned over a distance many times longer than each device range. Also it has been shown that a zigbee network can contain up to 2^{64} bit addresses which is more than enough addresses for any imaginable size of network. This conclude that bulding the devices is posible with the purposed technology.

Even so the code for the gateway runing on the Raspberry Pi is not finalized, it is shown connecting a CC2530 to the Raspberry Pi can work as a gateway to the internet allowing communication with the devices over the internet.

An important part of the application servers performance is the database. Locking at the Life in Vista Prints test comparing Mongo with a SQL database it is clear that the mongo database is faster than a realtional database when working with object data.

It is show that useing MongoDB together with Nodejs can handle a very high load and that it easely scale when there is need for handling a higher load.

All in all this project conclude and show how one IoT system can be designed.

The sensors and ZigBee gateway are simulated by a java program which continusly hit the backend API wiht jason data.

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/symlink/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

[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

Appendices