CMP6207 Modern Data Stores

Report Title

Coursework Assignment Report

Zan Zver
18133498

Word count: 3685

BIRMINGHAM CITY
University

Faculty of Computing, Engineering and the Built Environment
Birmingham City University

Contents

List of Tables

List of Figures

Listings

1 Types of NoSQL databases

NoSQL stands for not structured SQL. Since there is no structure, a lot of companies have created NoSQL management system in different ways. All of them are trying to achieve the same goal of efficiency and uptime but with different approaches (Hackolade n.d.). Here are some of more well known and used approaches:

- NoSQL Document
  - basic unit of data is "Document", we would call that Row in SQL,
  - document is ordered set of key-value pairs,
  - are schema free,
  - represented simple structure is: database, collection (or table in SQL), document.
- NoSQL Columnar
  - data is stored with column oriented model,
  - uses keyspace model which is similar to schema in relational databases,
  - represented simple structure is: keyspace, column family, rows, columns.
- NoSQL Property Graph
  - organises data in graph represented structure,
  - is made of nodes (represents an entity) and edges (represents connection/relationships between two nodes).
- NoSQL Key-Value
  - every item is stored as key-value,
  - data is stored as a collection of key-value pairs,
  - key must be unique.
- NoSQL Multi-model
  - joins the previous NoSQL types (Document, Graph, Key-Value) together,
  - instead of using 3 different NoSQL types in a project, 1 multi-model can be used to provide the same functionality.

NoSQL type	DB provider	Well known for

Document	MongoDB, Couchbase, AmazonDocument DB, etc...	Used in big data companies (example: BCU, Tesco)
Columnar	Cassandra DataStax, HBase, BigTable, etc...	Used for big/enormous data storage (example: Facebook, Netflix)
Property Graph	Neo4j, JanusGraph, TigerGraph, etc...	Fast relationship queries
Key-Value	DynamoDB, Redis, Aerospike, etc...	Fast and simply scalable
Multi-model	CosmosDB, Marklogic, OrientDB, etc...	Combination of NoSQL types

Table 1: NoSQL types compared

2 Relational VS NoSQL databases

2.1 About Relational databases

Relational databases use SQL as their query language. Relational databases (RDBS) have a clear structure. There is DB, then table and data (Microsoft n.d.[a]). Database is collection of tables and table is collection of data. Schema is enforced, meaning we have strict data types and requirements where data needs to go.

Every table has fields (columns) and records (rows). Every record has the same number of fields unlike NoSQL.

To improve SQL performance, normalisation is performed. With this, one table is broken apart to multiple tables with primary keys and foreign keys (Köhler and Link 2018)
Types of relations:

- One to one,
- many to many,
- one to many,
- many to one.

2.2 About NOSQL

Mongo has the structure of: database, collection, documents. Database can have multiple collections and collection can have multiple documents (MongoDB n.d.[a]). As name suggests, NoSQL means no schema. That means collection can have documents with different sizes inside.
NoSQL is flexible → you can add new data to one document and that one document would be different than others(MongoDB n.d.[b]).
NoSQL does not have relations. They can be implemented (on software level) but they are not enforced with managment system. Due to this, if the data changes, you need to do multiple updates.

2.3 Cap theorem

Stands for:
C - Consistency - data is the same across the database
A - Availability - as the data is written, it is replicated to other nodes
P - Partition tolerance - if node goes offline, how can it be replaced
(Khazaei et al. 2016)

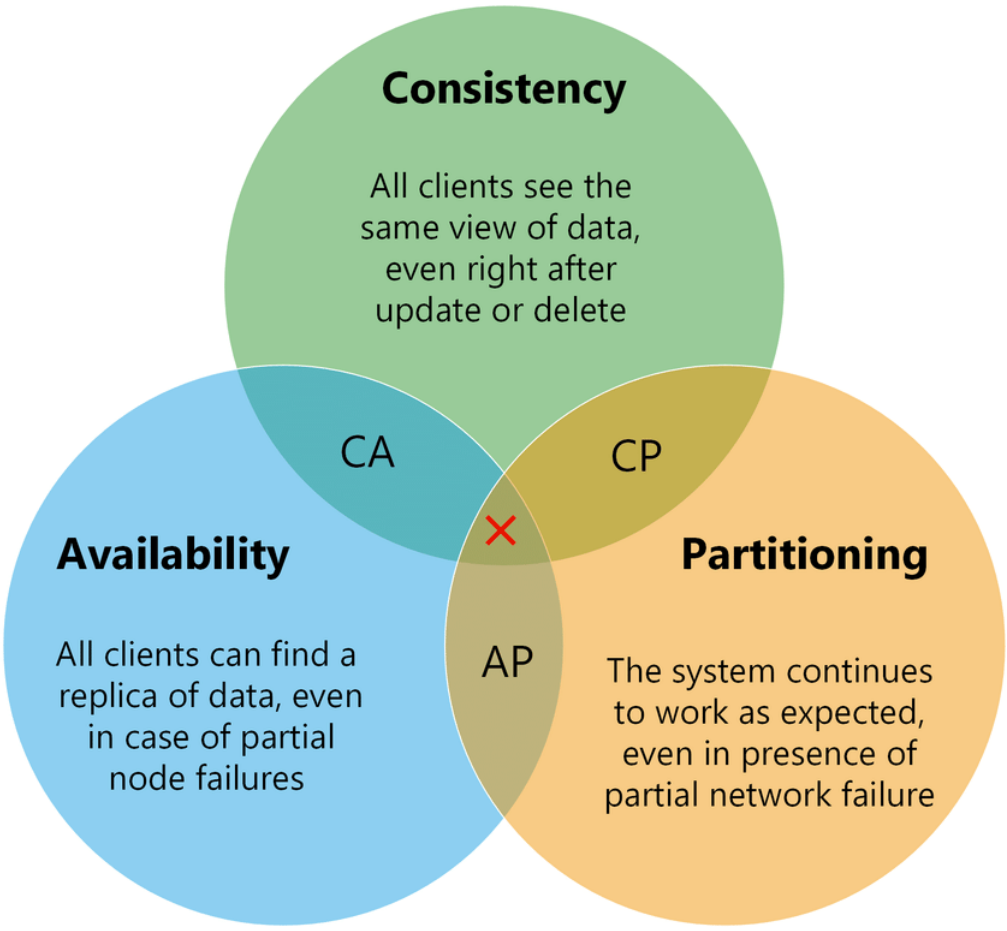

Figure 1: CAP theorem representation

MongoDB goes under CP, that meaning it is consistent and it has partition tolerance. This is achieved by MongoDB having primary node who receives all the writes and secondary nodes replicate the data after it has been written to primary (consistency). If primary node goes offline, secondary nodes can take over (partition tolerance). MongoDB does not have availability (or HA - high availability) due to it having one master node. So if master node goes offline it takes some time for voting process to be accomplished.

MySQL (relational database) would be CA. Data is written only to one node at the time, this is resulting in consistency. All of the data is also replicated to another node which is resulting in availability. But if master node goes down there is no partitioning in place.

Cassandra on the other hand, is classified as AP. All of the nodes are independent, that is preventing it to have consistency. But due to them not having a master node, read/write can be done from any node and this is resulting in high availability. Nodes are also copying data across to other nodes, this is resulting data (partition) tolerance. (IBM 2019)

2.4 Pros for SQL

- uses schemas → more fixed structure
- relations → no duplicate data, only update once
- data is distributed across multiple tables
- scaling is vertical

2.5 Pros for NoSQL

- no schema → data structure can change
- relations are not common → not querying relations
- data is nested in collections together
- can scale horizontal and/or vertical
- great performance for mass requests

2.6 Scaling

Scaling can be done in two ways (Cattell 2011)
Horizontal:

- add more servers (nodes) and merge data into one DB
- data is split across across the servers

Vertical:

- add more power (upgrade hardware) to existing server
- there is limit of to where you can scale

2.7 Which one is better: SQL or NoSQL

At the end, it depends on the scenario that we are in. Based on the CAP theorem, lets consider 3 cases:

- Consistency and availability (CA)
  - This would be for financial institutions, where data needs to be in sync as well as the same.
- Consistency and partition tolerance (CP)

  - This would be for e-commerce websites where data needs to be consistent and can operate even if node fails.

- Availability and partition tolerance (AP)
  - This would be for social media services (like Facebook) where data can be out of sync for some time.

Since there is not a clear winner, we can use what is the best for companies use case. Lets use Facebook as an example. They have social media which could be AP, marketplace could be CP and transactions on the site itself which are CA. With this, a company like Facebook can use SQL or NoSQL with correct CAP use case when needed.
To have a better representation on how data would be stored in data center, have a look on appendix A1-data centers.

3 IoT NoSQL database

3.1 Design

3.1.1 Stage 1

At first stage, we have a look at what our goal is. To explain it from top perspective, we are looking to have a connection to API which is going to access database.

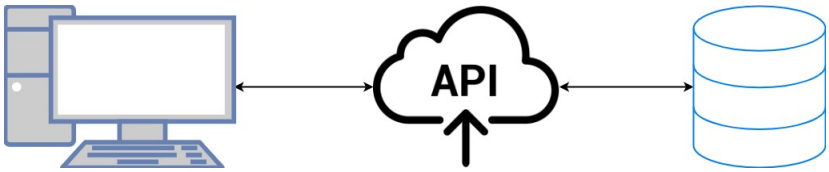

Figure 2: Diagram of stage 1

3.1.2 Stage 2

In the second stage, we define our tools. For database, it is going to be MongoDB. API used for connections is Node with Express (and Mongoose) API. Then we serve the data to the users as NodeJS website or to IoT devices.

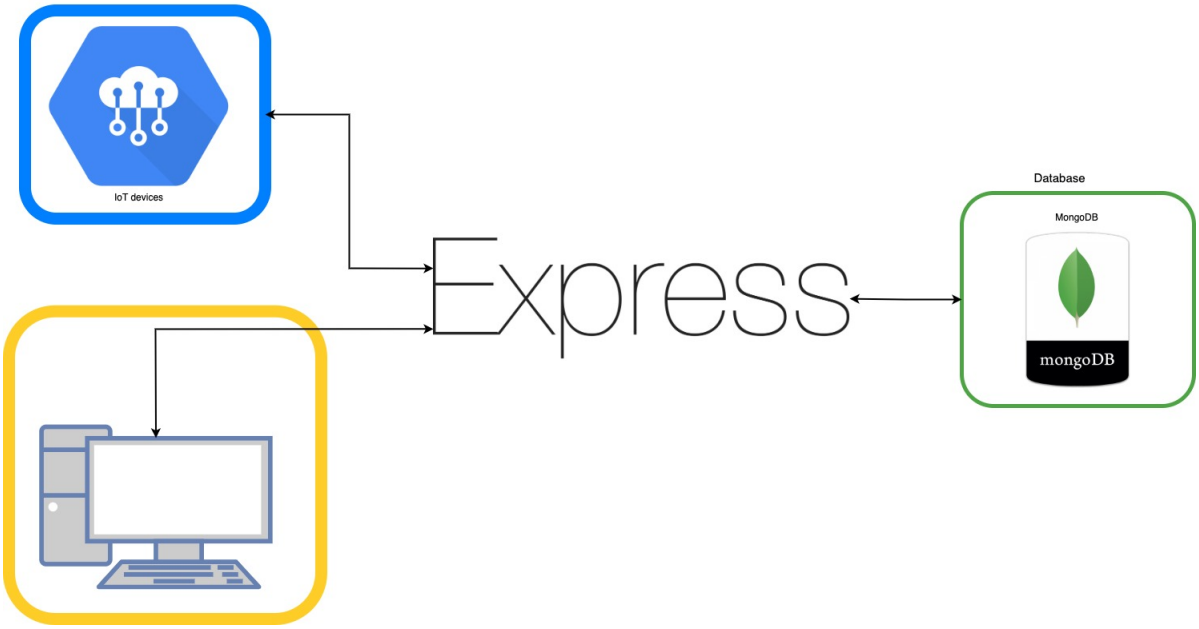

Figure 3: Diagram of stage 2

3.1.3 Stage 3

Third step defines structure for database and IoT. As seen in the diagram, Docker containers are going to be used for running the database. To improve database up time, we are going to shard it. This gives us 5 docker containers:

- 1 primary node - all of the writes come to this node
- 2 secondary nodes - reads are done from secondary node to balance load off the main node.
- 1 arbiter - in case that primary node goes down, this node is a tiebreaker of election between two of the nodes. This node cannot become primary node.
- 1 hidden node - this is backup node. It is hidden, meaning that it cannot participate in election process or in tie breaking. The only duty it has is to copy data from main node to itself. In case there is data loss, this node should have backups.

We have also expended on IoT side. Onece connection comes from API, it goes to user or IoT. Our API cannot communicate directly with IoT devices, but it needs some sort of "man in the middle". For that, we would expect API to communicate with the protocols (Apple home kit, Google home, Zigbee, Zwave) and they would comunicate with devices.

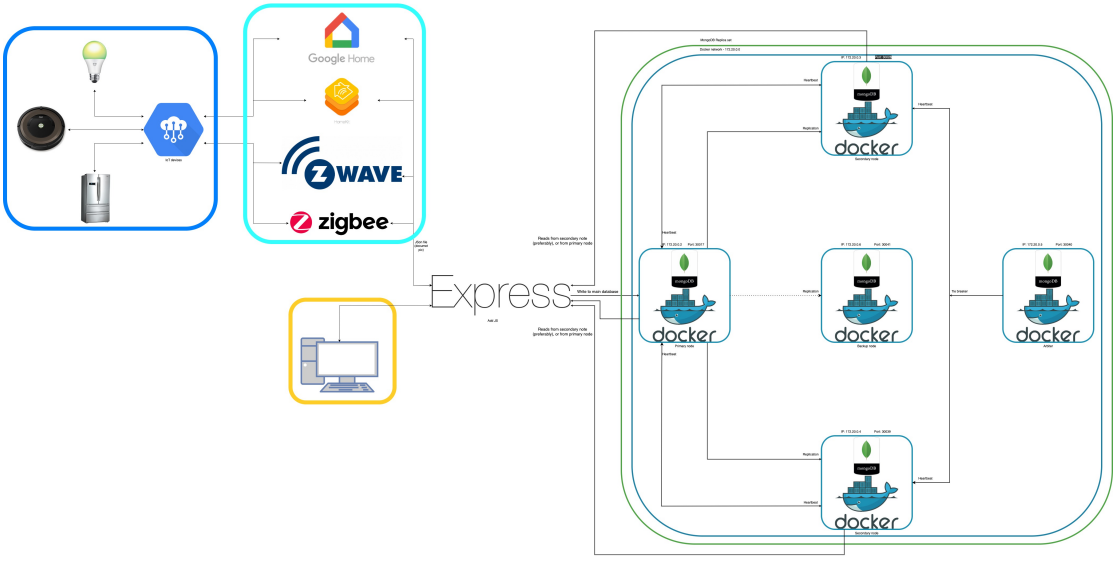

Figure 4: Diagram of stage 3

3.1.4 Stage 4

In stage four, we upgrade our sharded database structure to clustered database structure. With that in mind, shard from step 3 was duplicated to create shard

1-3. With this, two shards can go "offline" and we would still have our data. As seen, we have added 3 new docker containers between API and our cluster. Those are routers. Router is "directing" the traffic between API and shards in our cluster. We have 3 routers in case one of them goes down.

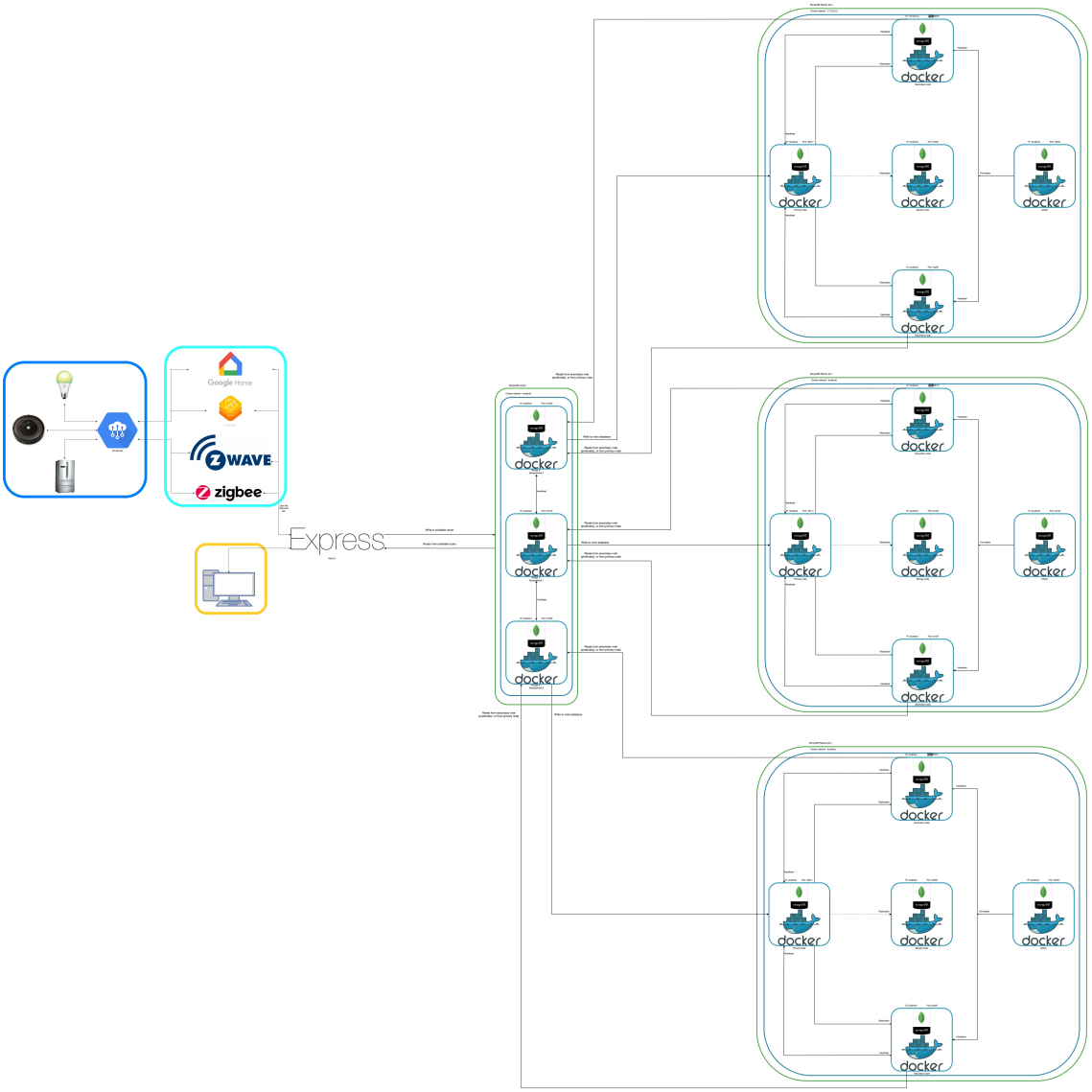

Figure 5: Diagram of stage 4

3.1.5 Stage 5

The final stage is just a concept of security. If we wanted to have all of the connections secured behind the private network (for security reasons), we would put a VPN in front of API. With that, users with VPN credentials can connect to API. Since devices cannot connect directly to VPN (except if network would route

directly to VPN), we would configure API to allow reads or writes from the protocols. With that, IoT devices can still communicate with our network but "hackers" cannot get to our API. If we wanted to add extra security, protocols would be set to read only outside of VPN.

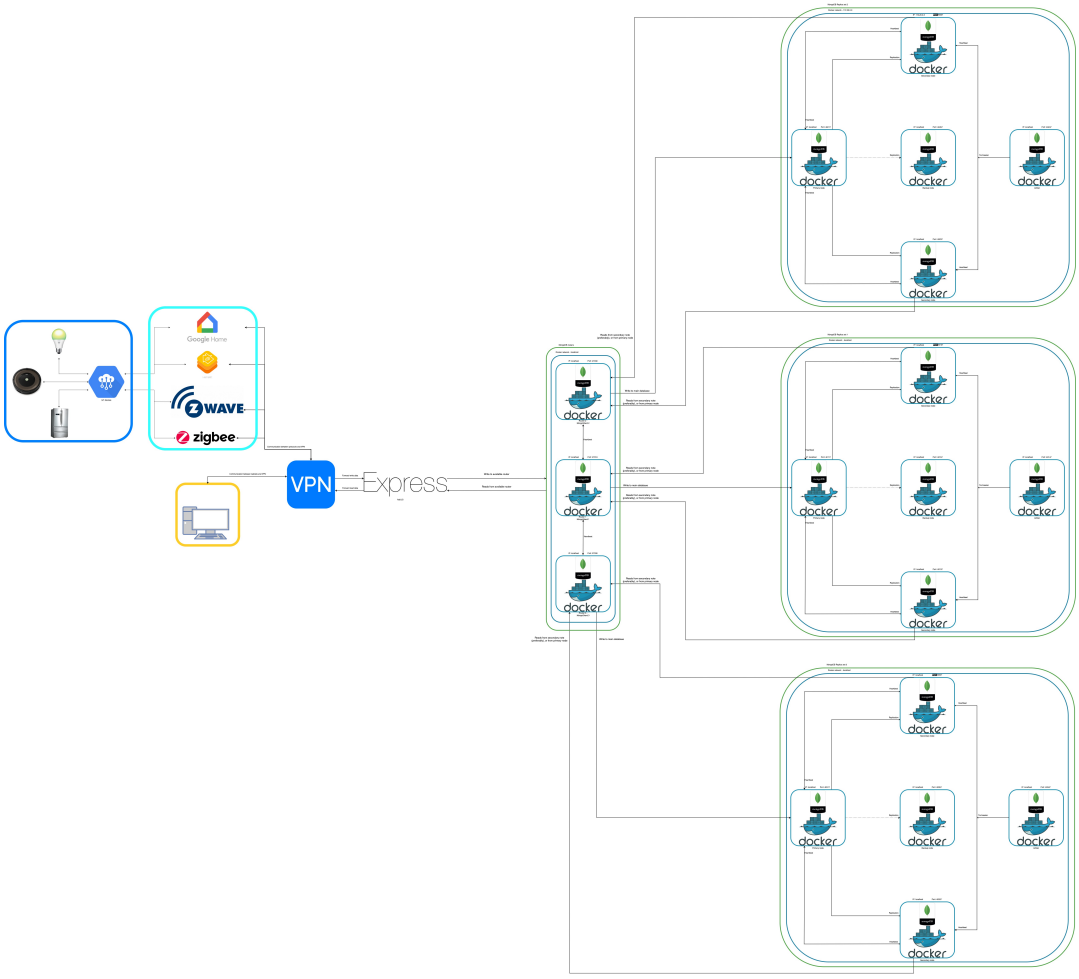

Figure 6: Diagram of stage 5

As seen from the diagrams above, Docker (Rad, Bhatti, and Ahmadi 2017) is going to be used for hosting the MongoDB (MongoDB n.d.[c]). Reasons for that are: efficiently, simplicity and scalability.

Reasoning	Description

Efficiently	• Docker containers are more light weight in comparison to VMs. This gives them an advantage in terms of OS storage and speed.
Simplicity	• Starting up new docker container is simple. For our use case, we are using Docker Compose. With this, we can have predefined YAML file that has specifications for desired Docker container.
Scalability	• Scaling with MongoDB and Docker is simple. We can scale vertically by adding more docker containers to our network or upgrading existing host server.

Table 2: Reasons to use docker

For our IoT database, we are using 4 collections:

Collection name	Collection description
users	• has user data inside • read writes from users and admins
iot_customer_devices	• saves customers devices • read writes from users and admins

iot_device_history	• historical list of device status changes • read writes from users and admins • this can be a capped collection, saving changes for a month
iot_device_info	• devices that are supported in the application • users are set to read only, while admins can do read and write

Table 3: Collections from DB and their descriptions

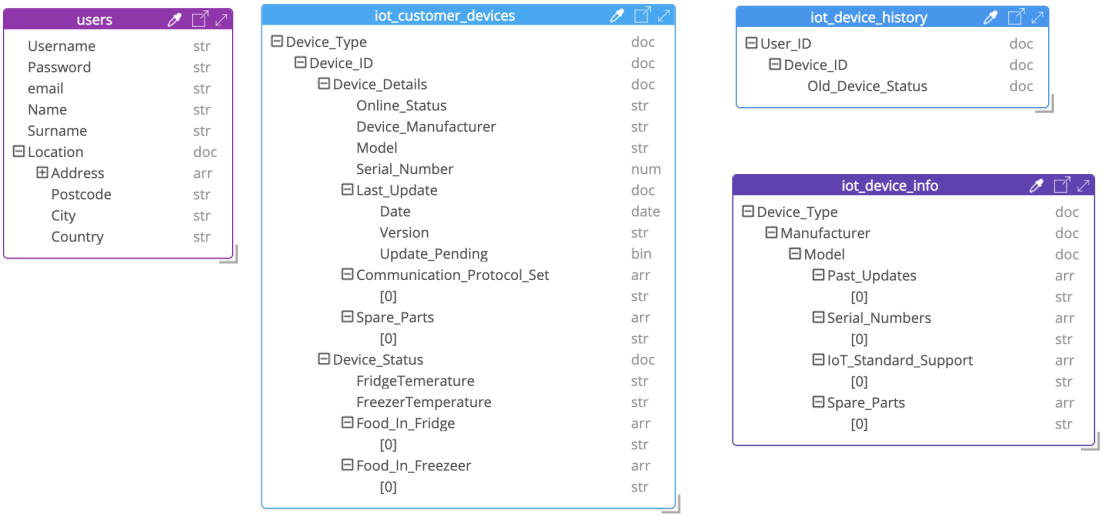

Figure 7: Collections diagram

users collection document description:

Document name	Document description

Username	• Type: string • Must be unique username that would be at least 6 characters long.
Password	• Type: string • User defined password that is at least 8 characters long and it has symbols in. Cannot be the same as username or email.
Email	• Type: string • Must be unique and must meet certain expectations (example: have @ in string).
Name	• Type: String • Name of the user
Surname	• Type: String • Surname of the user

Location	• Type: document • Is build with 4 other sub items – Address ∗ Type: Array (length of 2) ∗ Address line 1 is mandatory while address line 2 is optional – Postcode ∗ Type: String ∗ Postcode where user is located – City ∗ Type: String ∗ City where user is located – Country ∗ Type: String ∗ Country where user is located

Table 4: Documents from users collection described

iot_customer_devices collection document description

Document name	Document description
Device_Type	• Type: document • Key is device name (example: smart_light) and value is document which has devices inside
Device_ID	• Type: document • Key is ID of device and value is device_details and device_status

Device_Details	• Type: document • This document structure is the same across all the devices regardless of their type. Idea of this is to get device meta data which would be the same across the board. – Online_Status ∗ Type: boolian ∗ If the device is online or offline. – Device_Manufacturer ∗ Type: String ∗ Who produced the device – Model ∗ Type: String ∗ Model name – Serial_Number ∗ Type: number or String ∗ Serial_number of the device. – Last_Update ∗ Type: document ∗ When was device last updated. – Communication_Protocol_Set ∗ Type: array ∗ Based on available communication protocols, what protocols are being used. – Spare_Parts ∗ Type: array ∗ If there are any spare parts available for purchase, they are listed here.

Device_Status	• Type: document • This document depends on device type. In the image above we have an example of what it would look like for smart fridge. But the idea is that device_status gathers data that is from hardware level (example: battery level or last charge, etc...).

Table 5: Documents from iot_customer_devices collection described

iot_device_history collection document description

Document name	Document description
User_ID	• Type: document • Key is User_ID and value is document which has devices inside.
Device_ID	• Type: document • Key is Device_ID and value is old device status. When user updates device status, new version gets stored in the iot_customer_devices collection and old version gets written here.

Table 6: Documents from iot_device_history collection described

iot_device_info collection document description

Document name	Document description

Device_Type	• Type: document • Key represents device name (example: Smart_Vacuum). By theat, if we add new kind of device, we add new key (example: we don't have Smart_TV so we create new key with that name). Value is manufacturers that produce that device.
Manufacturer	• Type: document • List of manufacturer names. Key is manufacturer name and value is model of device.
Model	• Type: document • Key represents model name and value represents information of the model – Past_Updates ∗ Type: array ∗ List of past updates. – Serial_Numbers ∗ Type: array ∗ List of serial numbers that are registered on our platform under this model. – IoT_Standard_Support ∗ Type: array ∗ List of IoT standards that device supports. – Spare_Parts ∗ Type: array ∗ List of spare parts that can be bought.

Table 7: Documents from iot_device_info collection described

3.2 Implementation

3.2.1 Step 1 - Database design

Database design was done in design section above. From there, we are implementing concepts bellow.

3.2.2 Step 2 - Create Docker containers

3.2.2.1 Docker compose for nodes/replica sets

```
version: '2'
services:
```

Listing 1: Docker Compose Nodes Version

```
mongo_ReplicaSet1_Node1:
    container_name: mongo_ReplicaSet1_Node1
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet1 --
   dbpath /data/db --port 27017
    ports:
      - 40117:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet1_Node1:/data/db
  mongo_ReplicaSet1_Node2:
    container_name: mongo_ReplicaSet1_Node2
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet1 --
   dbpath /data/db --port 27017
    ports:
      - 40127:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet1_Node2:/data/db
  mongo_ReplicaSet1_Node3:
    container_name: mongo_ReplicaSet1_Node3
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet1 --
   dbpath /data/db --port 27017
```

```
    ports:
      - 40137:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet1_Node3:/data/db
  mongo_ReplicaSet1_Arbiter:
    container_name: mongo_ReplicaSet1_Arbiter
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet1 --
    dbpath /data/db --port 27017
    ports:
      - 40138:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet1_Arbiter:/data/db
  mongo_ReplicaSet1_Backup:
    container_name: mongo_ReplicaSet1_Backup
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet1 --
    dbpath /data/db --port 27017
    ports:
      - 40139:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet1_Backup:/data/db
```

Listing 2: Docker Compose Nodes for replica set 1

```
mongo_ReplicaSet2_Node1:
    container_name: mongo_ReplicaSet2_Node1
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet2 --
    dbpath /data/db --port 27017
    ports:
      - 40217:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet2_Node1:/data/db
  mongo_ReplicaSet2_Node2:
    container_name: mongo_ReplicaSet2_Node2
    image: mongo
```

```
    command: mongod --shardsvr --replSet mongo_ReplicaSet2 --dbpath /data/db --port 27017
    ports:
      - 40227:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet2_Node2:/data/db
  mongo_ReplicaSet2_Node3:
    container_name: mongo_ReplicaSet2_Node3
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet2 --dbpath /data/db --port 27017
    ports:
      - 40237:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet2_Node3:/data/db
  mongo_ReplicaSet2_Arbiter:
    container_name: mongo_ReplicaSet2_Arbiter
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet2 --dbpath /data/db --port 27017
    ports:
      - 40238:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet2_Arbiter:/data/db
  mongo_ReplicaSet2_Backup:
    container_name: mongo_ReplicaSet2_Backup
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet2 --dbpath /data/db --port 27017
    ports:
      - 40239:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet2_Backup:/data/db
```

Listing 3: Docker Compose Nodes for replica set 2

```
mongo_ReplicaSet3_Node1:
```

```
    container_name: mongo_ReplicaSet3_Node1
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet3 --dbpath /data/db --port 27017
    ports:
      - 40317:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet3_Node1:/data/db
  mongo_ReplicaSet3_Node2:
    container_name: mongo_ReplicaSet3_Node2
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet3 --dbpath /data/db --port 27017
    ports:
      - 40327:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet3_Node2:/data/db
  mongo_ReplicaSet3_Node3:
    container_name: mongo_ReplicaSet3_Node3
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet3 --dbpath /data/db --port 27017
    ports:
      - 40337:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet3_Node3:/data/db
  mongo_ReplicaSet3_Arbiter:
    container_name: mongo_ReplicaSet3_Arbiter
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet3 --dbpath /data/db --port 27017
    ports:
      - 40338:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet3_Arbiter:/data/db
  mongo_ReplicaSet3_Backup:
```

```
    container_name: mongo_ReplicaSet3_Backup
    image: mongo
    command: mongod --shardsvr --replSet mongo_ReplicaSet3 --
   dbpath /data/db --port 27017
    ports:
      - 40339:27017
    expose:
      - "27017"
    environment:
      TERM: xterm
    volumes:
      - mongo_ReplicaSet3_Backup:/data/db
```

Listing 4: Docker Compose Nodes for replica set 3

```
volumes:
  mongo_ReplicaSet1_Node1: {}
  mongo_ReplicaSet1_Node2: {}
  mongo_ReplicaSet1_Node3: {}
  mongo_ReplicaSet1_Arbiter: {}
  mongo_ReplicaSet1_Backup: {}
  mongo_ReplicaSet2_Node1: {}
  mongo_ReplicaSet2_Node2: {}
  mongo_ReplicaSet2_Node3: {}
  mongo_ReplicaSet2_Arbiter: {}
  mongo_ReplicaSet2_Backup: {}
  mongo_ReplicaSet3_Node1: {}
  mongo_ReplicaSet3_Node2: {}
  mongo_ReplicaSet3_Node3: {}
  mongo_ReplicaSet3_Arbiter: {}
  mongo_ReplicaSet3_Backup: {}
```

Listing 5: Docker Compose storage for nodes

3.2.2.2 Docker compose for config servers

```
version: '2'
services:
```

Listing 6: Docker Compose Config Version

```
mongo_Config1:
      container_name: mongo_Config1
      image: mongo
      command: mongod --configsvr --replSet mongo_ReplicaSet1_Conf
    --dbpath /data/db --port 27017
      environment:
        TERM: xterm
      expose:
        - "27017"
      volumes:
```

```
10          - mongo_Config1:/data/db
```

Listing 7: Docker Compose Config 1

```
mongo_Config2:
      container_name: mongo_Config2
      image: mongo
      command: mongod --configsvr --replSet mongo_ReplicaSet1_Conf
    --dbpath /data/db --port 27017
      environment:
        TERM: xterm
      expose:
        - "27017"
      volumes:
        - mongo_Config2:/data/db
```

Listing 8: Docker Compose Config 2

```
  mongo_Config3:
      container_name: mongo_Config3
      image: mongo
      command: mongod --configsvr --replSet mongo_ReplicaSet1_Conf
    --dbpath /data/db --port 27017
      environment:
        TERM: xterm
      expose:
        - "27017"
      volumes:
        - mongo_Config3:/data/db
```

Listing 9: Docker Compose Config 3

```
volumes:
  mongo_Config1: {}
  mongo_Config2: {}
  mongo_Config3: {}
```

Listing 10: Docker Compose storage for Config

3.2.2.3 Docker compose for routers

```
version: '2'
services:
```

Listing 11: Docker Compose Router Version

```
mongo_Shard1:
    container_name: mongo_Shard1
    image: mongo
    command: mongos --configdb mongo_ReplicaSet1_Conf/
   mongo_Config1:27017,mongo_Config2:27017,mongo_Config3:27017 --
   port 27017 --bind_ip 0.0.0.0
```

```
    ports:
      - 27019:27017
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
```

Listing 12: Docker Compose router 1

```
  mongo_Shard2:
    container_name: mongo_Shard2
    image: mongo
    command: mongos --configdb mongo_ReplicaSet1_Conf/
    mongo_Config1:27017,mongo_Config2:27017,mongo_Config3:27017 --
    port 27017 --bind_ip 0.0.0.0
    ports:
      - 27020:27017
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
```

Listing 13: Docker Compose router 2

```
  mongo_Shard3:
    container_name: mongo_Shard3
    image: mongo
    command: mongos --configdb mongo_ReplicaSet1_Conf/
    mongo_Config1:27017,mongo_Config2:27017,mongo_Config3:27017 --
    port 27017 --bind_ip 0.0.0.0
    ports:
      - 27021:27017
    expose:
      - "27017"
    volumes:
      - /etc/localtime:/etc/localtime:ro
```

Listing 14: Docker Compose router 3

3.2.2.4 Execute docker compose

Run docker compose files
setup replica → set nodes - 5 nodes per 3 replica sets → 15 docker nodes are created

```
sudo docker-compose -f docker-compose.yaml up -d
```

Listing 15: Create 15 Mongo nodes

setup config → 3 config (docker) servers are running

```
sudo docker-compose -f mongod.yaml up -d
```

Listing 16: Create config servers

setup router → 3 mongos (router) dockers are running

```
sudo docker-compose -f mongos.yaml up -d
```

Listing 17: Create router servers

Configure config servers replica set

Replica set 1

```
docker exec -it mongo_Config1 bash -c "echo 'rs.initiate(
    {_id: \"mongo_ReplicaSet1_Conf\",configsvr: true, members: [
        { _id : 0, host : \"mongo_Config1\" },
        { _id : 1, host : \"mongo_Config2\" },
        { _id : 2, host : \"mongo_Config3\" }
        ]
    })
' | mongosh"
```

Listing 18: Set config1

Replica set 2

```
docker exec -it mongo_Config2 bash -c "echo 'rs.initiate(
    {_id: \"mongo_ReplicaSet1_Conf\",configsvr: true, members: [
        { _id : 0, host : \"mongo_Config1\" },
        { _id : 1, host : \"mongo_Config2\" },
        { _id : 2, host : \"mongo_Config3\" }
        ]
    }
)' | mongosh"
```

Listing 19: Set config2

Replica set 3

```
docker exec -it mongo_Config3 bash -c "echo 'rs.initiate(
    {_id: \"mongo_ReplicaSet1_Conf\",configsvr: true, members: [
        { _id : 0, host : \"mongo_Config1\" },
        { _id : 1, host : \"mongo_Config2\" },
        { _id : 2, host : \"mongo_Config3\" }
        ]
    }
)' | mongosh"
```

Listing 20: Set config3

Check config server replica set status

```
docker exec -it mongo_Config1 bash -c "echo 'rs.status()' |
   mongosh"
```

Listing 21: Check replica set status

Build shard replica set

Replica set 1

```
docker exec -it mongo_ReplicaSet1_Node1 bash -c "echo 'rs.initiate(
    {_id : \"mongo_ReplicaSet1\", members: [
        { _id : 0, host : \"mongo_ReplicaSet1_Node1\" },
        { _id : 1, host : \"mongo_ReplicaSet1_Node2\" },
        { _id : 2, host : \"mongo_ReplicaSet1_Node3\" },
        { _id : 3, host : \"mongo_ReplicaSet1_Arbiter\", arbiterOnly: true},
        { _id : 4, host : \"mongo_ReplicaSet1_Backup\", hidden: true}
        ]
    }
)' | mongosh"
```

Listing 22: Build replica set 1

Replica set 2

```
docker exec -it mongo_ReplicaSet2_Node1 bash -c "echo 'rs.initiate(
    {_id : \"mongo_ReplicaSet2\", members: [
        { _id : 0, host : \"mongo_ReplicaSet2_Node1\" },
        { _id : 1, host : \"mongo_ReplicaSet2_Node2\" },
        { _id : 2, host : \"mongo_ReplicaSet2_Node3\" },
        { _id : 3, host : \"mongo_ReplicaSet2_Arbiter\", arbiterOnly: true},
        { _id : 4, host : \"mongo_ReplicaSet2_Backup\", hidden: true}
        ]
    }
)' | mongosh"
```

Listing 23: Build replica set 2

Replica set 3

```
docker exec -it mongo_ReplicaSet3_Node1 bash -c "echo 'rs.initiate(
    {_id : \"mongo_ReplicaSet3\", members: [
        { _id : 0, host : \"mongo_ReplicaSet3_Node1\" },
        { _id : 1, host : \"mongo_ReplicaSet3_Node2\" },
        { _id : 2, host : \"mongo_ReplicaSet3_Node3\" },
        { _id : 3, host : \"mongo_ReplicaSet3_Arbiter\", arbiterOnly: true},
        { _id : 4, host : \"mongo_ReplicaSet3_Backup\", hidden: true}
        ]
    }
)' | mongosh"
```

Listing 24: Build replica set 3

Check status primary-secondary

Status for replica set 1

```
docker exec -it mongo_ReplicaSet1_Node1 bash -c "echo 'rs.status()' | mongosh"
```

Listing 25: Check status of replica set 1

Status for replica set 2

```
docker exec -it mongo_ReplicaSet2_Node1 bash -c "echo 'rs.status()' | mongosh"
```

Listing 26: Check status of replica set 2

Status for replica set 3

```
docker exec -it mongo_ReplicaSet3_Node1 bash -c "echo 'rs.status()' | mongosh"
```

Listing 27: Check status of replica set 3

Introduce replica set to the routers

Add mongo_Shard1

```
docker exec -it mongo_Shard1 bash -c "echo 'sh.addShard(\"mongo_ReplicaSet1/mongo_ReplicaSet1_Node1\")' | mongosh"
```

Listing 28: Introduce replica 1

Add mongo_Shard2

```
docker exec -it mongo_Shard2 bash -c "echo 'sh.addShard(\"mongo_ReplicaSet2/mongo_ReplicaSet2_Node1\")' | mongosh"
```

Listing 29: Introduce replica 2

Add mongo_Shard3

```
docker exec -it mongo_Shard3 bash -c "echo 'sh.addShard(\"mongo_ReplicaSet3/mongo_ReplicaSet3_Node1\")' | mongosh"
```

Listing 30: Introduce replica 3

Get shard status

```
docker exec -it mongo_Shard1 bash -c "echo 'sh.status()' | mongosh"
```

Listing 31: Get shard status

Create test db

Create test db in shard mongo_ReplicaSet_1 for testing.

```
docker exec -it mongo_ReplicaSet1_Node1 bash -c "echo 'use testDb' | mongosh"
```

Listing 32: Create test db

Enable sharding for new db

Test sharding of new database.

```
docker exec -it mongo_Shard1 bash -c "echo 'sh.enableSharding(\"testDb\")' | mongosh"
```

Listing 33: Enable sharding for new db

Create test collection

Create test collection for further testing.

```
docker exec -it mongo_ReplicaSet1_Node1 bash -c "echo 'db.createCollection(\"testDb.testCollection\")' | mongosh"
```

Listing 34: Create test collection RS1

```
docker exec -it mongo_ReplicaSet2_Node1 bash -c "echo 'db.createCollection(\"testDb.testCollection\")' | mongosh"
```

Listing 35: Create test collection RS2

```
docker exec -it mongo_ReplicaSet3_Node1 bash -c "echo 'db.createCollection(\"testDb.testCollection\")' | mongosh"
```

Listing 36: Create test collection RS3

Shard based on the key

Test sharding collection based on the key.

```
docker exec -it mongo_Shard1 bash -c "echo 'sh.shardCollection(\"testDb.testCollection\", {\"shardingField\" : 1})' | mongosh"
```

Listing 37: Shard based on the key

3.2.2.5 Showcase of docker containers

At the end we have 21 docker containers that are database related. All of the Docker containers are on local network (localhost) in our case. In the table bellow, we can see the function they have.

Container name	Container description	Port
mongo_ReplicaSet1_Node1	Storage node 1 for replica set 1	40117
mongo_ReplicaSet1_Node2	Storage node 2 for replica set 1	40127
mongo_ReplicaSet1_Node3	Storage node 3 for replica set 1	40137
mongo_ReplicaSet1_Arbiter	Tie breaker for replica set 1	40147
mongo_ReplicaSet1_Backup	Backup node for replica set 1	40157

mongo_ReplicaSet2_Node1	Storage node 1 for replica set 2	40217
mongo_ReplicaSet2_Node2	Container 1 for replica set 2	40227
mongo_ReplicaSet2_Node3	Container 1 for replica set 2	40237
mongo_ReplicaSet2_Arbiter	Tie breaker for replica set 2	40247
mongo_ReplicaSet2_Backup	Backup node for replica set 2	40257
mongo_ReplicaSet3_Node1	Storage node 1 for replica set 3	40317
mongo_ReplicaSet3_Node2	Storage node 2 for replica set 3	40327
mongo_ReplicaSet3_Node3	Storage node 3 for replica set 3	40337
mongo_ReplicaSet3_Arbiter	Tie breaker for replica set 3	40347
mongo_ReplicaSet3_Backup	Backup node for replica set 3	40357
mongo_Config1	Configuration server 1	27017
mongo_Config2	Configuration server 2	27017
mongo_Config3	Configuration server 3	27017
mongo_Shard1	Main router	27019
mongo_Shard2	Backup router 1	27020
mongo_Shard3	Backup router 2	27021

Table 8: Containers used described

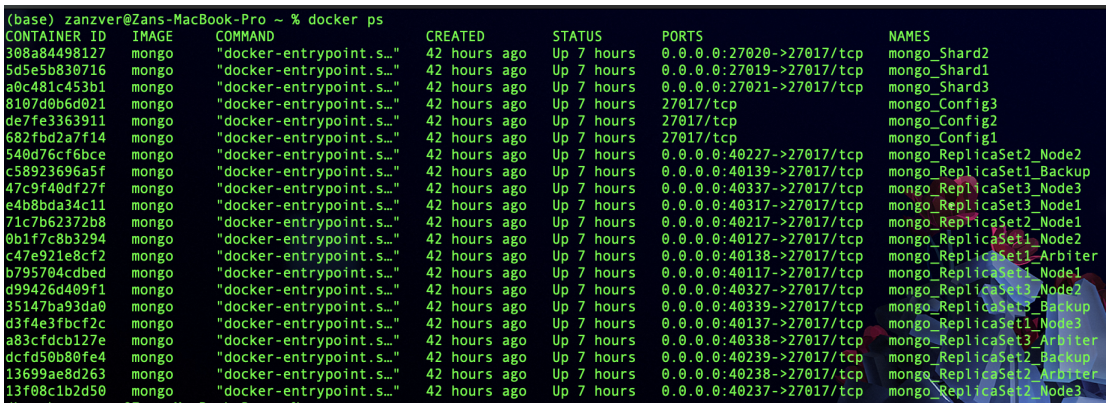

Figure 8: Containers displayed in terminal

3.2.3 Step 3 - Fill the database

To fill the database, Python scrip was used for inserting random data.
First thing is to specify how many users we want.

```
createUser(10)
```

Listing 38: Call createUser function

Based on the input number (10 in our case) users are going to be created.

```
def createUser(num):
    for i in range(num):
        firstname = names.get_first_name()
        lastname = names.get_last_name()
        items = ["", str(random.randrange(1,999))]
        random_item = random.choice(items)
        random_email = random.choice(emailProviders)
        random_address = real_random_address()
        #print(random_item)
        mydict = {
                "Username": firstname + lastname,
                "Password": randomword(10),
                "Email": firstname + "." + lastname + random_item
   + random_email,
                "Name": firstname,
                "Surname": lastname,
                "Location": {
                    "Address": [
                        random_address["address1"],
                        random_address["address2"]
                    ],
                    "Postcode": random_address["postalCode"],
                    "City": random_address["state"],
                    "Country": "United States"
                }
        }
        x = colUsers.insert_one(mydict)
        createItAll(x.inserted_id)
        createHistory(x.inserted_id)
```

Listing 39: Showcase of createUser function

When we create the user, we also create devices for the user.

```
def createItAll(UserID):
    myDict = createDevices()
    colIoT_Customer_Device.insert_one(
        { "UserID" : str(UserID),
          "smart_light": myDict["smart_light"],
          "smart_fridge": myDict["smart_fridge"],
          "smart_vacuum": myDict["smart_vacuum"]
        })

    dict1 = createDeviceInfo()
    try:
        dict2 = colIoT_Device_Info.find_one()
        for deviceTypes in dict1.keys():
            for manufacturer in dict1[deviceTypes].keys():
                for modelName in list(dict1[deviceTypes][
   manufacturer])[0].keys():
                    for serialNumber in list(dict2[deviceTypes][
   manufacturer])[0][modelName]["Serial_Numbers"]:
                        list(dict1[deviceTypes][manufacturer])[0][
   modelName]["Serial_Numbers"].append(serialNumber)
```

```
18
19          oldquery = colIoT_Device_Info.find_one()
20          newquery = { "$set": dict1 }
21
22          colIoT_Device_Info.update_one(oldquery, newquery)
23      except:
24          colIoT_Device_Info.insert_one(dict1)
```

Listing 40: Showcase of createItAll function

As noted in the code, history for device is also generated.

```
def createHistory(userID):
    lightHistoryGenerator = random.randrange(1,30)
    fridgeHistoryGenerator = random.randrange(1,30)
    vacuumHistoryGenerator = random.randrange(1,30)
    for i in range(lightHistoryGenerator):
        colIoT_Device_History.insert_one({str(userID): {"1":
    createTheSmartLight()["Device_Status"]}})

    for i in range(fridgeHistoryGenerator):
        colIoT_Device_History.insert_one({str(userID): {"2":
    createTheFridge()["Device_Status"]}})

    for i in range(vacuumHistoryGenerator):
        colIoT_Device_History.insert_one({str(userID): {"3":
    createTheVauum()["Device_Status"]}})
```

Listing 41: Showcase of createHistory

3.2.4 Step 4 - API

Connect the database with API

```
const mongoose = require("mongoose");

mongoose.connect('mongodb://localhost:27019,localhost:27020,
    localhost:27021/initialDB', {useNewUrlParser: true}, (err) => {
    if(!err){
        console.log("MongoDB Conncetion Succeeded!")
    }
    else{
        console.log("Error in DB connection: " + err)
    }
});

require("./employee.model");
require("./user.model.js");
require("./IoT_Customer_Device.model.js");
require("./IoT_Device_Info.model.js");
```

Listing 42: Showcase of Mongoose connection to database

3.2.4.1 Step 5 - Indexes

To improve database performance, we have added indexes (MongoDB n.d.[d]). With this, database management system has a faster way to identify location of document in our collection. Listed bellow we can see collections and their indexes.

Collections name	Index description
users	• Index type: regular • This collection can benefit on stock index _id. Any other index would not be beneficial at the moment since we are not using this collection a lot. Most of the queries are done by _id.
iot_customer_devices	• Index type: regular • In this collection we have a custom index with the name of UserID. As name suggests, this field has the value from collection users with users ID. Comparison bellow shows that index did not save us much time but on the document examined section, we have navigated straight to our document. If this would be larger database, time would be much bigger concern. • Performance without index – executionTimeMillis : 2 – totalKeysExamined : 0 – totalDocsExamined : 1200 • Performance with index – executionTimeMillis : 3 – totalKeysExamined : 1 – totalDocsExamined : 1

iot_device_info	• Index type: regular • This collection is small (at the moment) and it does not benefit much from indexing. But in case we wanted to index it, we would do it by device type and its manufacturer. • Performance without index – executionTimeMillis : 0 – totalKeysExamined : 0 – totalDocsExamined : 1 • Performance with index – executionTimeMillis : 0 – totalKeysExamined : 0 – totalDocsExamined : 1

iot_device_history	• Index type: regular • History is build as UserID (key): DeviceID (value) and in the DeviceID we have history. The idea was to create a temporary index on UserID. So, when user logs in, index is created in database and when they log out, index is removed. Upon creating index, we have noticed that indexing seems to not be performing as well. In the future, we would break the UserID (key): DeviceID (value) to UserID: value of key and to DeviceID: value of device. • Performance without index – executionTimeMillis : 37 – totalKeysExamined : 0 – totalDocsExamined : 53838 • Performance with index – executionTimeMillis : 55 – totalKeysExamined : 53838 – totalDocsExamined : 53838

Table 9: Indexes on collections

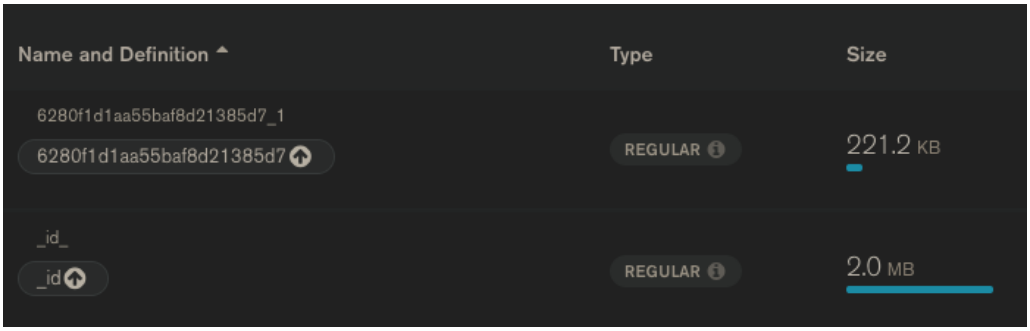

Figure 9: iot_device_history collection index

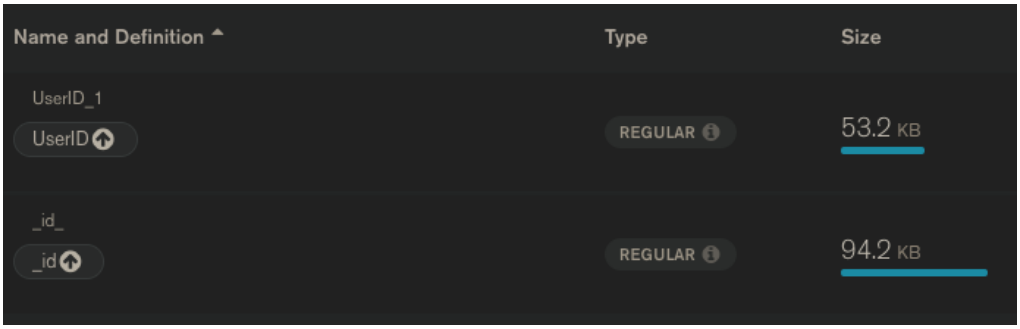

Figure 10: iot_customer_devices collection index

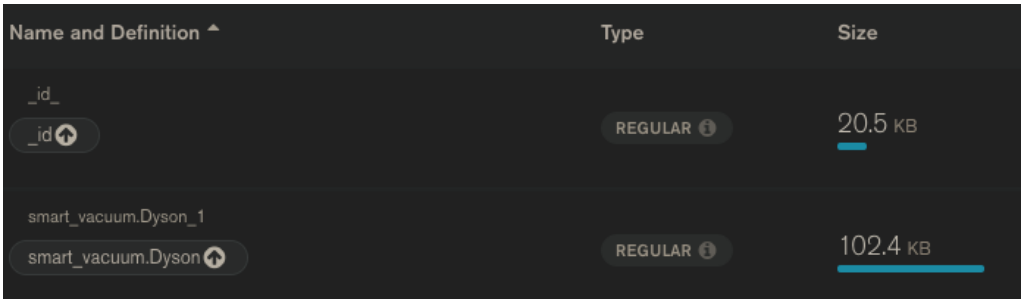

Figure 11: iot_device_info collection index

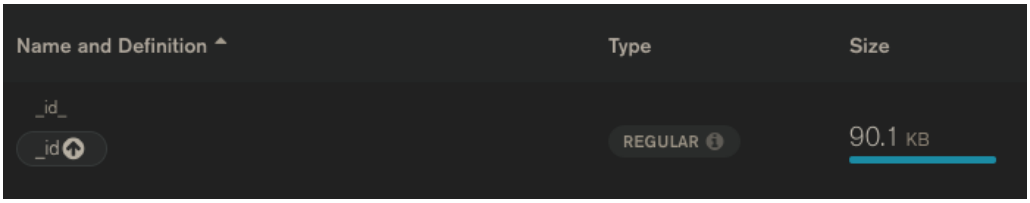

Figure 12: users collection index

4 API

4.1 Documentation

Design structure includes 5 main components:

- home page,
- forgotten password,
- user registration,
- user page,
- admin page.

Homepage would be something simple that users and admins have in common. When admin wants to login, login section would check if user signing in is actual user or admin. With that information, the correct page would be shown.

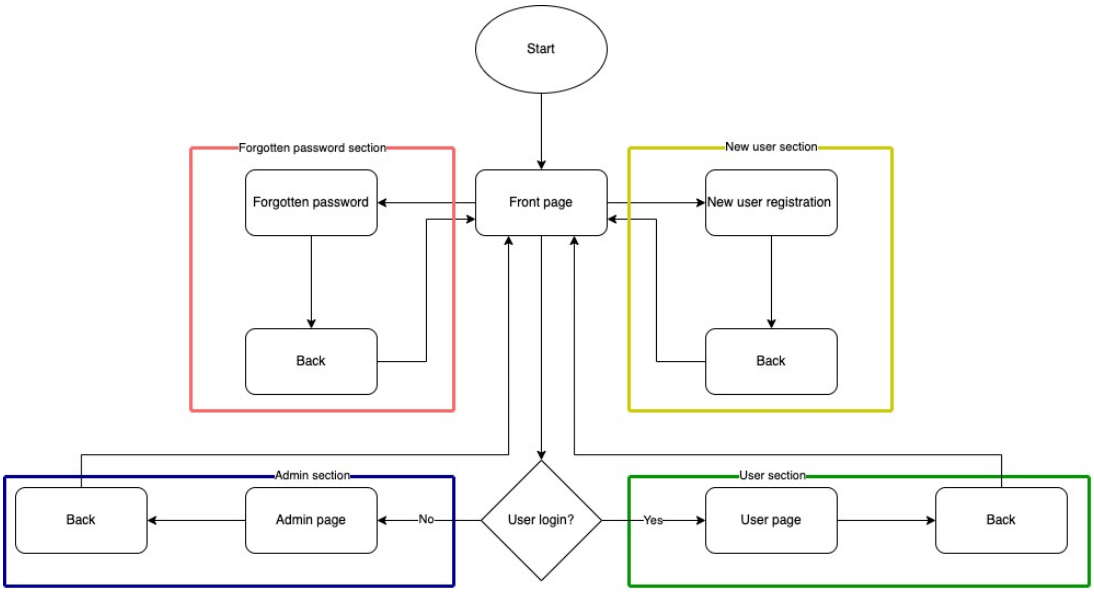

Figure 13: Diagram representing simple structure

Forgotten password is for users and admins to reset their passwords. Upon opening the page, user can reset the password (with inserting correct information) and be redirected to homepage. If user declines to to so, they can navigate back.

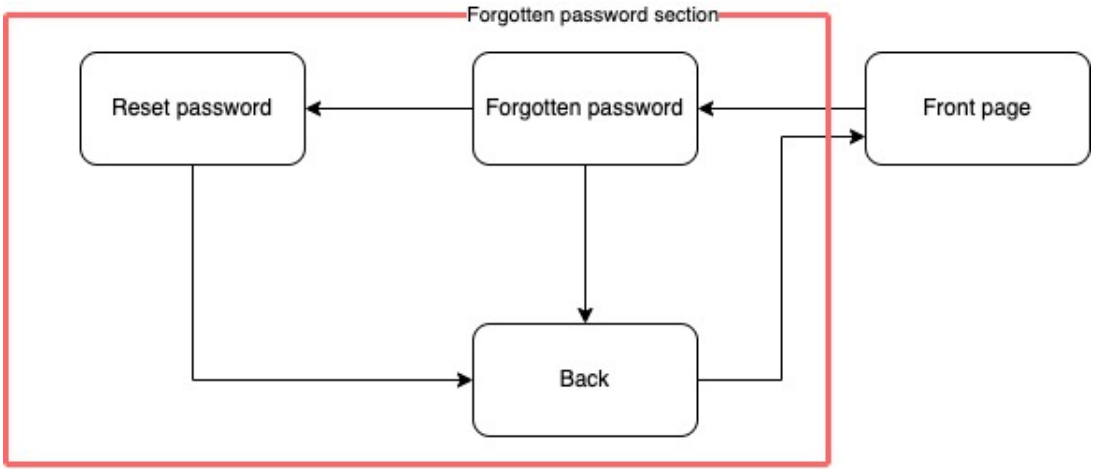

Figure 14: Diagram representing forgotten password

New user creation is for users to sign up for the service. On the site, user needs to fill the form in order to sign up. Once done, they are redirected to home page. If user declines to to so, they can navigate back.

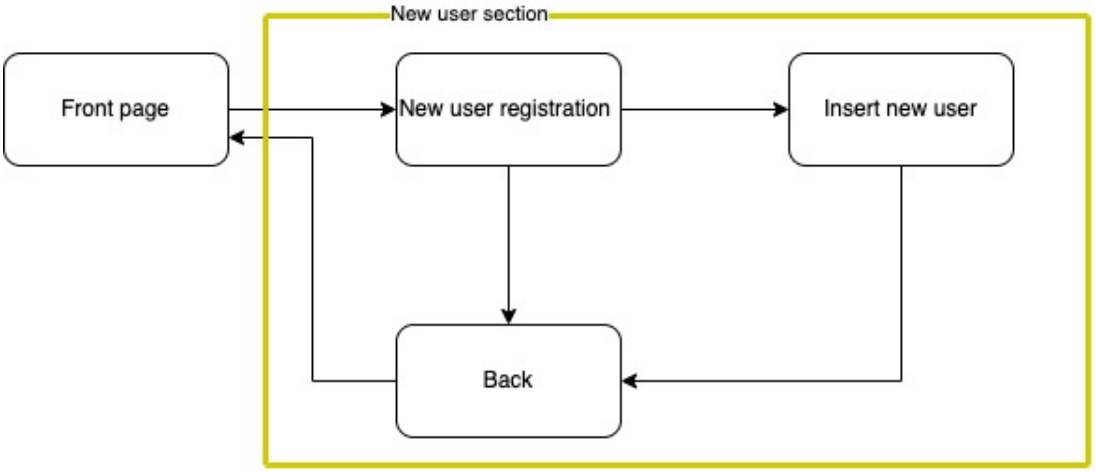

Figure 15: Diagram representing new user registration

On user login, user can see how many devices he/she has per section. From there they can select section to see all devices in there. Once all devices from sections are shown, they can perform CRUD operations by adding new device, listing existing devices, adding new devices and removing existing devices.

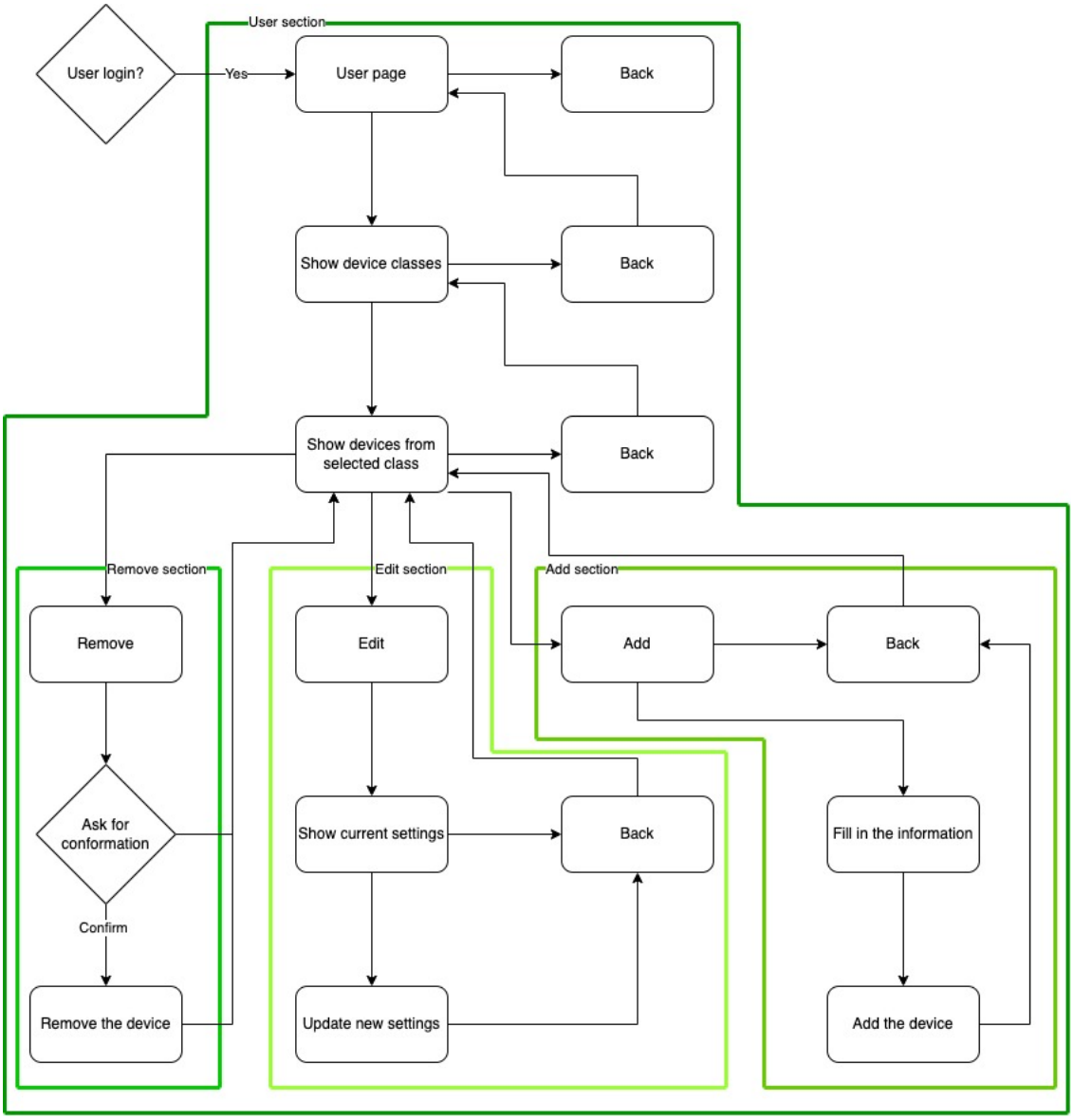

Figure 16: Diagram representing user view

If user logging in is admin, he/she will be redirected to this section. In here, admin can select "show users" or "show devices" section. In case admin wants to edit user settings (promote user to admin), option is there under "show users". Another thing they can see is analysis of users (for example, usual logins). If they select "show devices", where admin can edit existing device types. Admin can also add new supported devices in "add device" section. Under device analytic, admin can see how devices are performing.

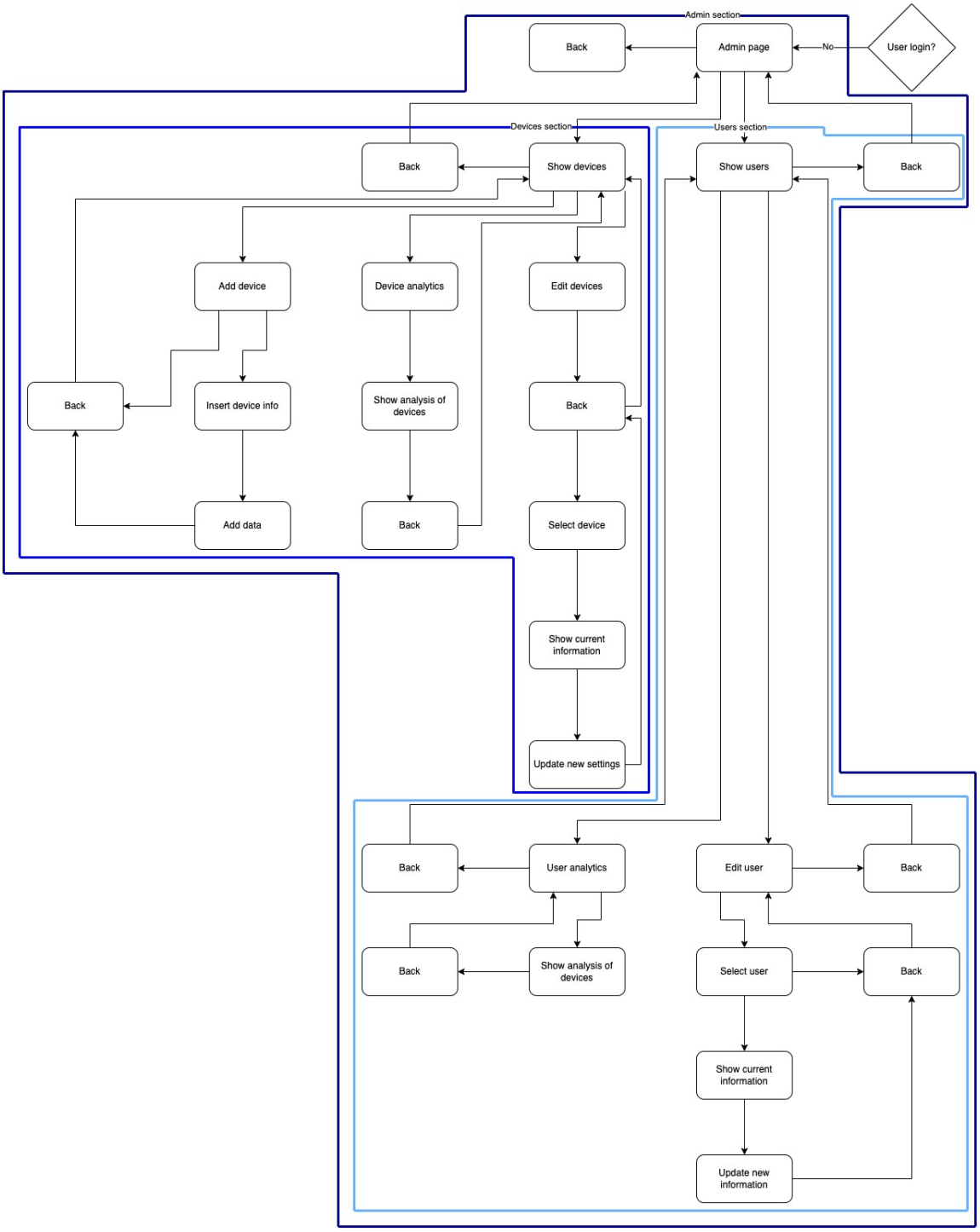

Figure 17: Diagram representing admin view

This is the same figure as on top but with all the sections expended.

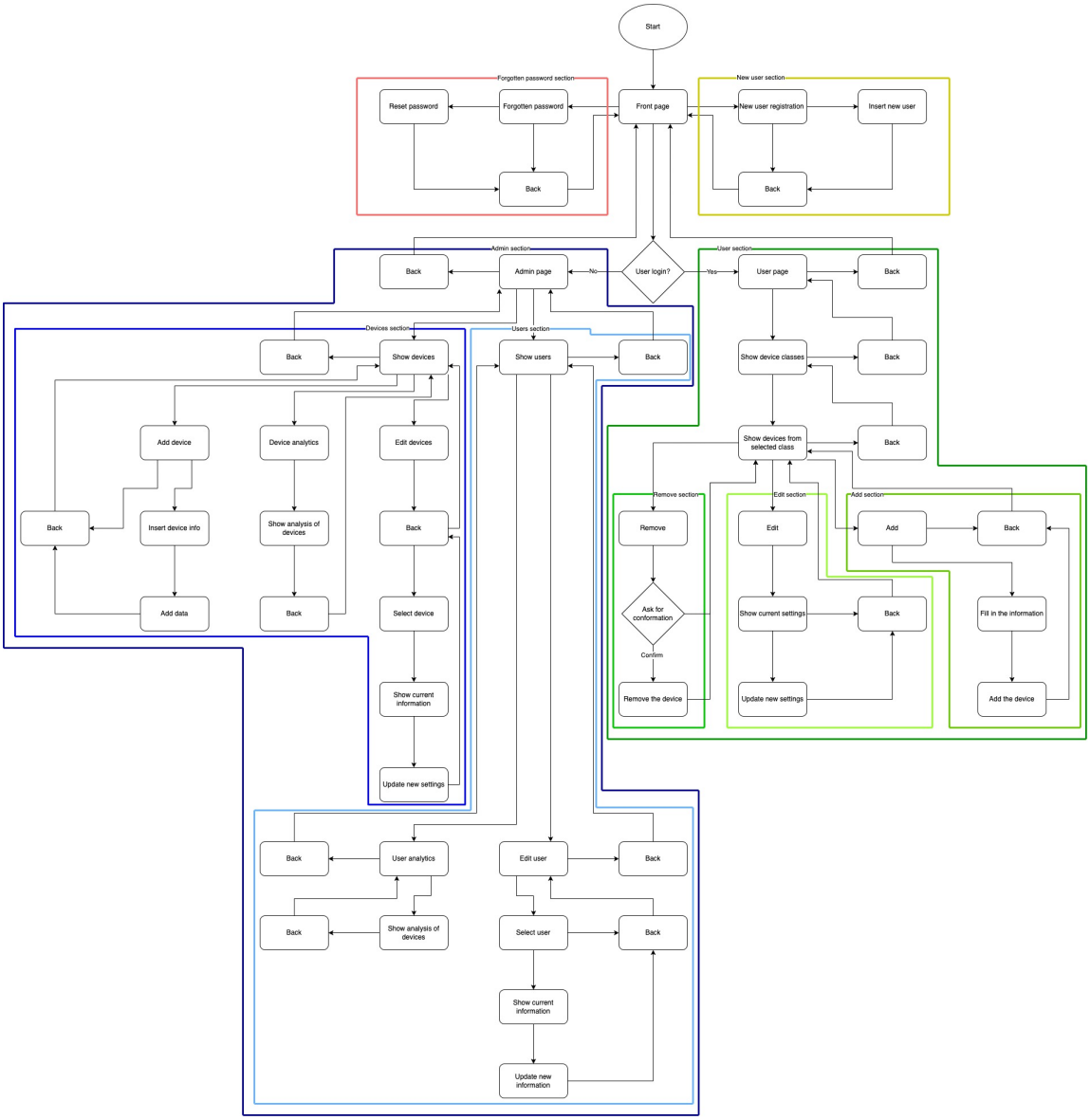

Figure 18: Detailed diagram of whole web application

4.2 Implementation

Main technologies used for creating web application are: NodeJS, Express, HandlebarsJS and Mongoose. Here is the role of each tool:

- NodeJS
  - Backbone of the project - backend
- Express
  - Routing across web pages

- HandlebarsJS
  - Rendering HTML with data inside
- Mongoose
  - Getting the data from MongoDB cluster.

Homepage, consisting of main actions:

- login,
- forgotten password,
- registration.

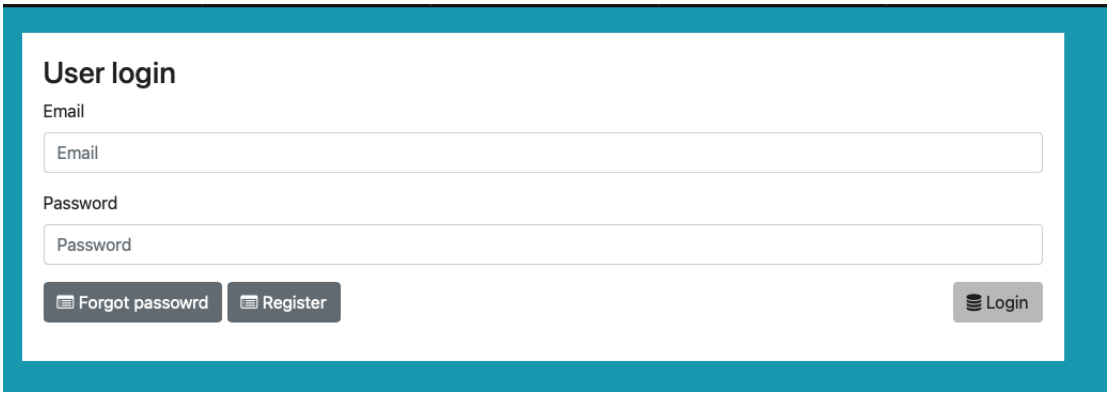

Figure 19: Home page

If user has forgotten the password, they would need to fill in the section with full name (name and surname) and email. To send emails, we can integrate sendgrid API. With that, automatic emails will be send for password reset. Unfortunately this is not implemented yet (email sending).

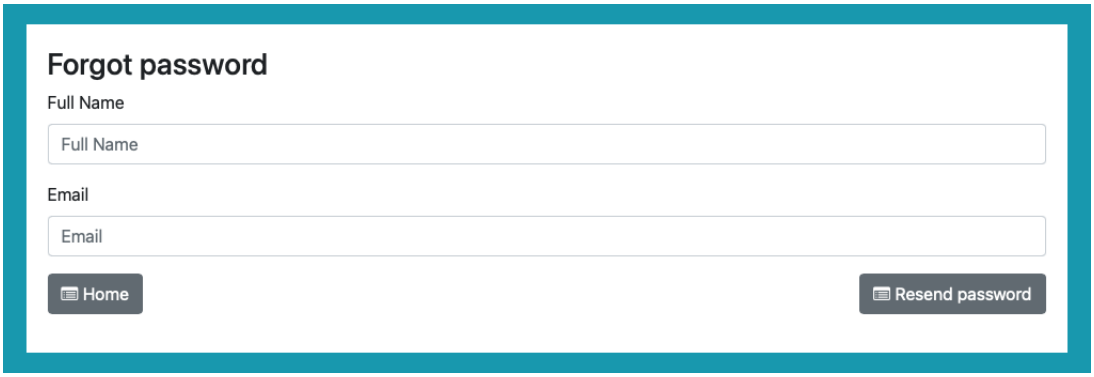

Figure 20: Forgotten password

User needs to fill in the form to register. Once done, he/she is saved to MongoDB and they are redirected to homepage.

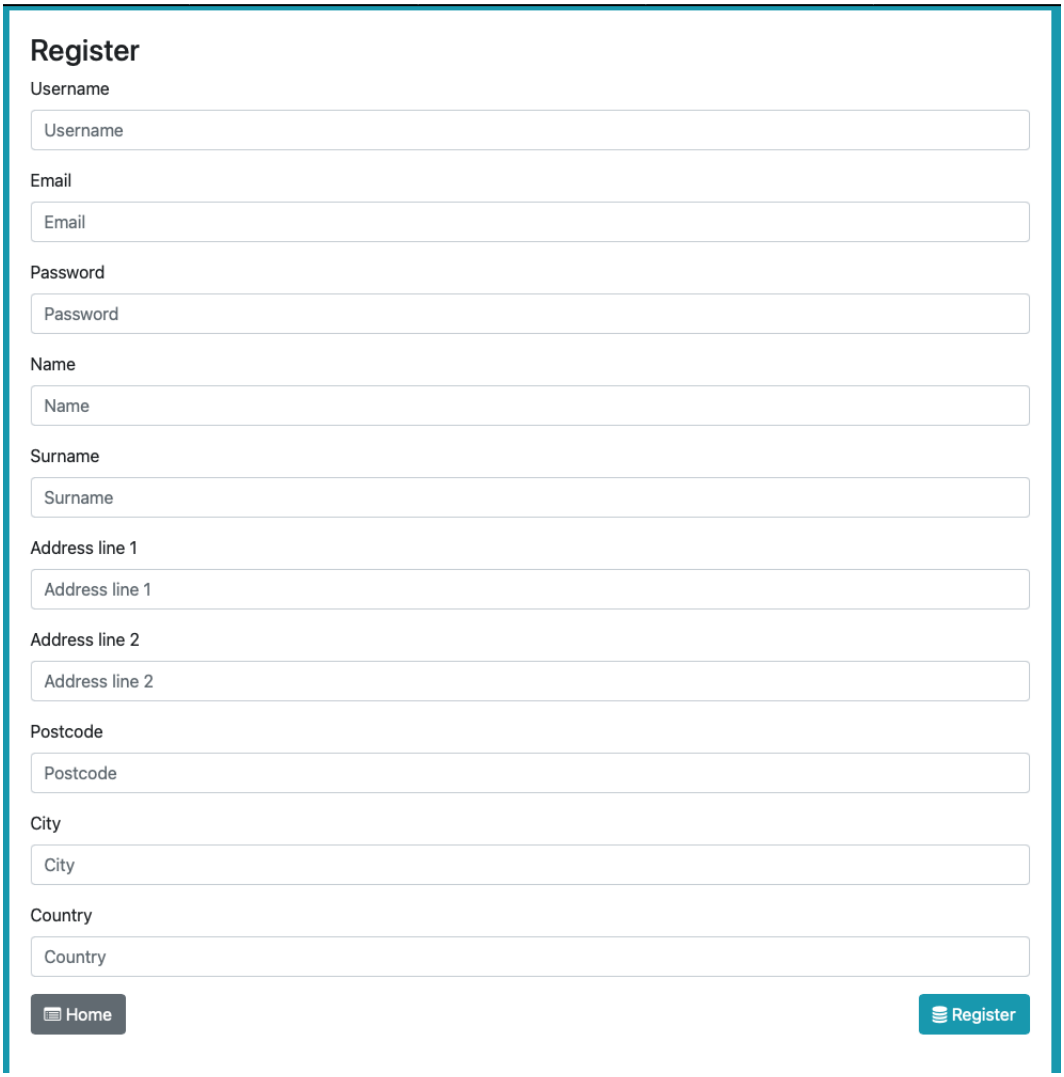

Figure 21: Register section

Once user is logged in, classes of devices are shown. This is list of supported classes (under device type) and number of devices user has registered. If new device type comes to support, user would have number of devices as 0 and could add new device by selecting pencil (see all).

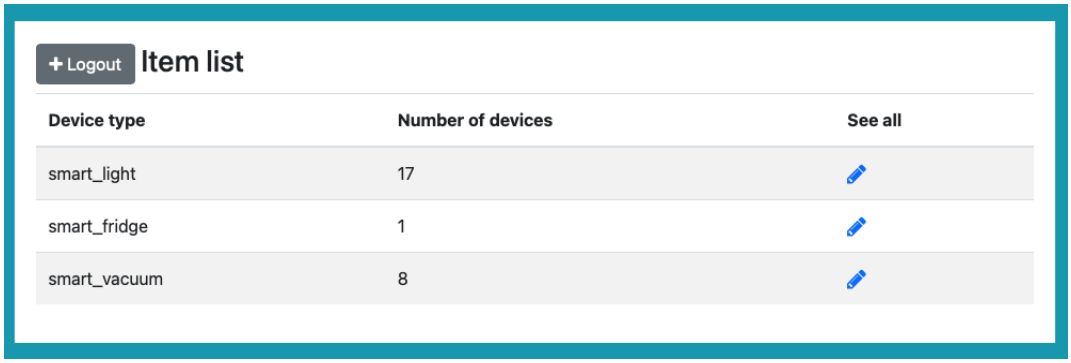

Figure 22: Devices classified

Upon selecting device type, all of the devices from that sections are displayed. User can also see some of the settings on top. There are also options to add, delete and update the device.

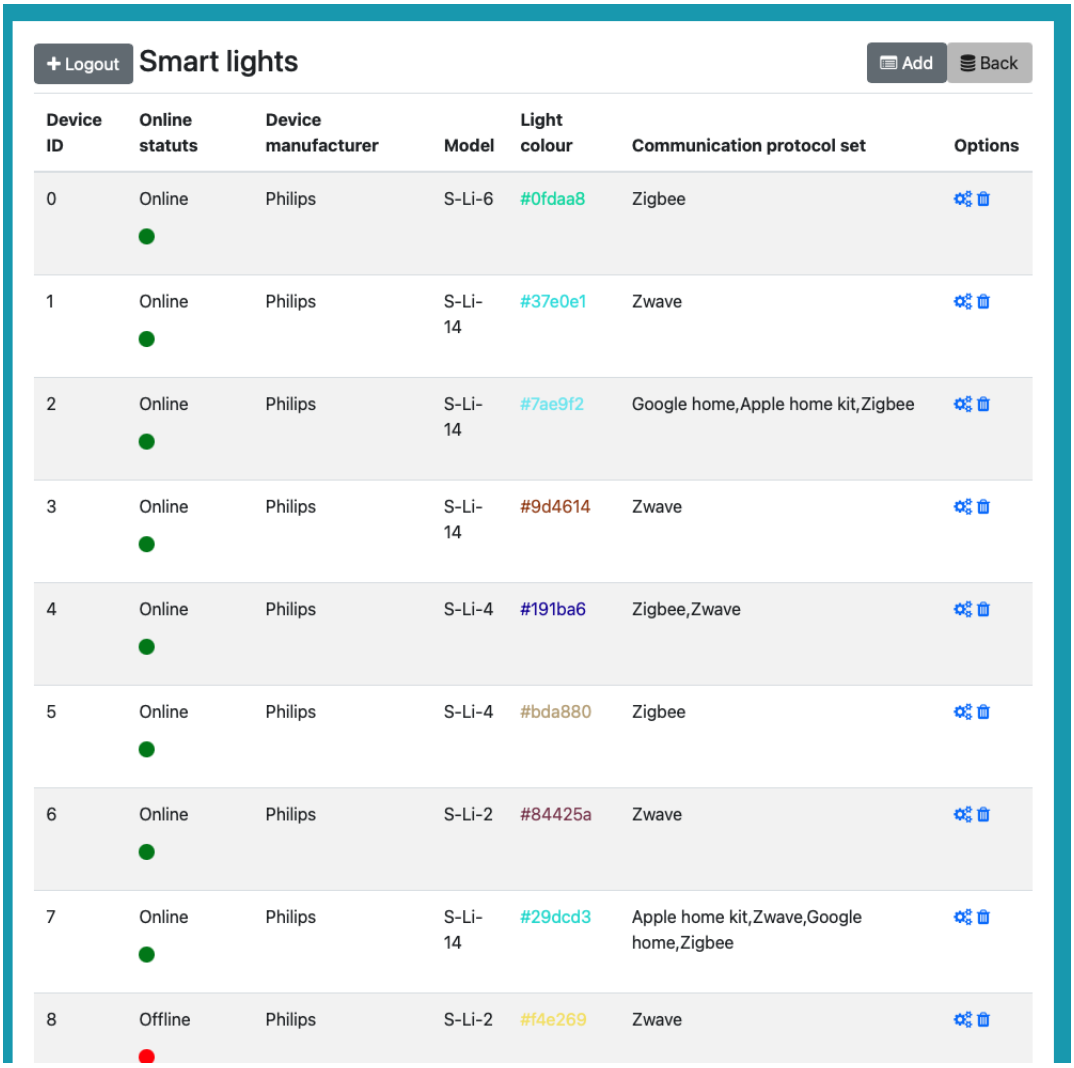

Figure 23: List of devices

Adding new device is simple. User needs to select information that is supported and correct for him/her. For example, on drop down menu, user selects manufacturer and then model from that manufacturer.

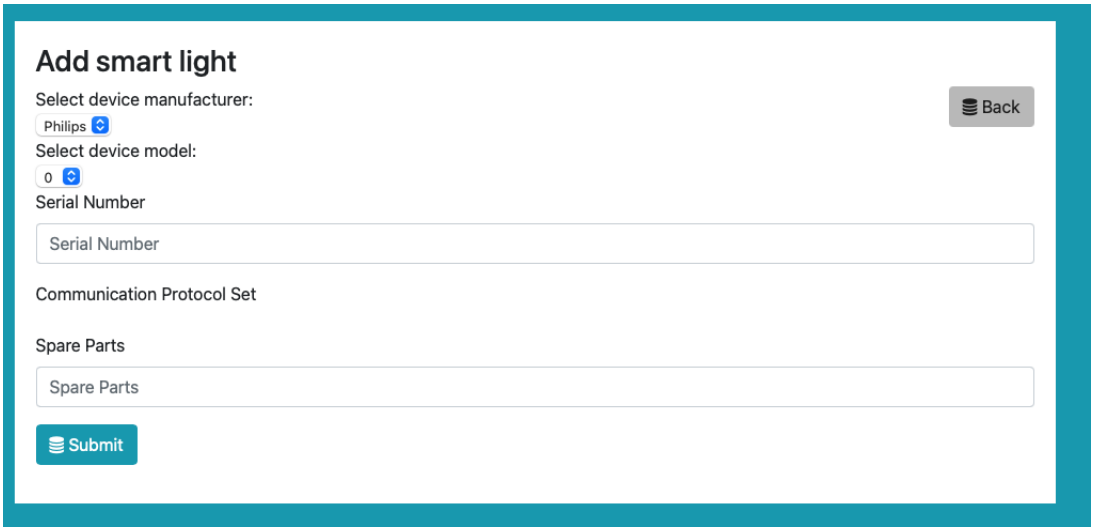

Figure 24: Add device

If user wants to remove the device, they can select bin icon and they will see a popup to confirm if they want to remove device or not.

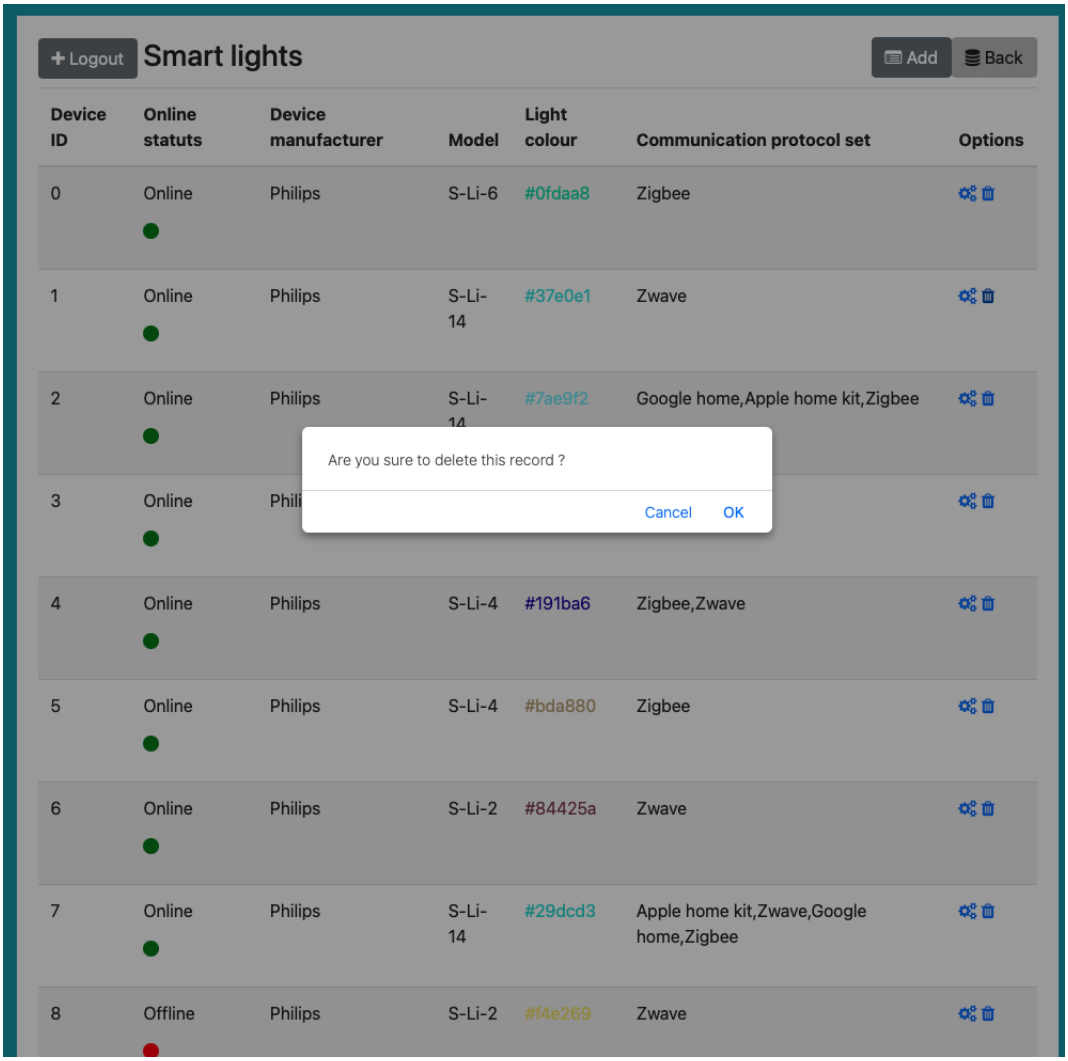

Figure 25: Remove device

Update section reads device information and displays it on web page. User can then edit section they desire.

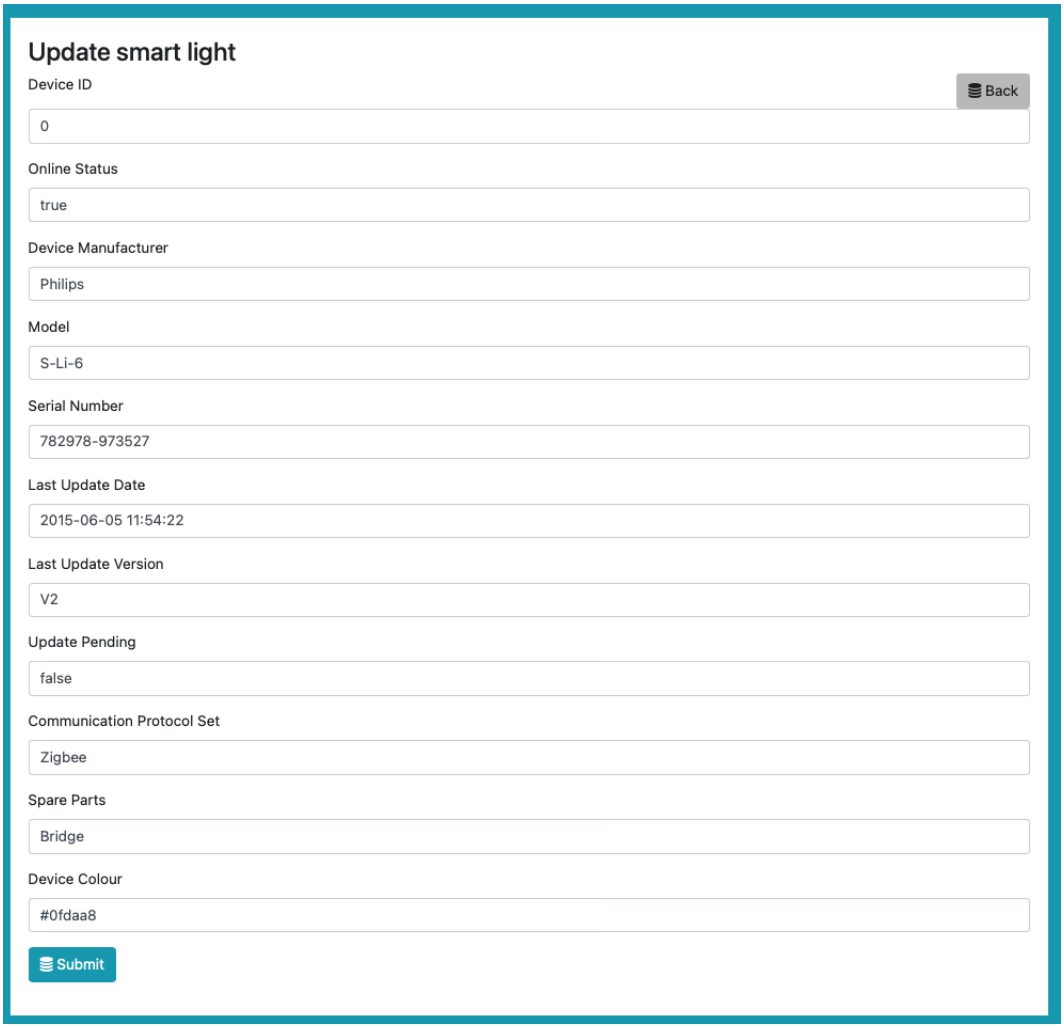

Figure 26: Edit device

5 Summary and conclusion

At the end, we have successfully produced a demo unit that could be implemented and scaled. On top of that, it is modular.

The database structure can scale as needed. For the company that is starting up, it wold make sense from financial standpoint to just use 1 shard instead of 3 that we have implemented. That is the great thing about NoSQL, it is simple to scale vertically or horizontally! This is specially possible with Docker. We can deploy, start/stop containers as we need.

User interface is modular as well. For our use, we have implemented NodeJS with combination of express, handlebars and mongoose. But since JavaScript is popular, this can be replaced with anything else! At the end of the day, we could use Flask (Python) with Bootstrap.

At the moment, there are still improvements that can be made. The improvements are more from technical side and range from "nice to have" to "necessary" ones. List of improvements can be found under appendix A2-improvements.

6 Appendixes

6.1 Data centers

In the industry, the big companies have their own data centers or they are renting them from other companies like Amazon and their AWS (Villamizar et al. 2016).

6.1.1 Data centers across the globe

Once you put data request across any of the websites, data is usually "piped" from your router, to ISP you have which connects you to the data center that requests is allocated for. This means, data needs to travel geographically which can take longer if it is across the globe (Benson et al. 2010). To increase speed, companies are deploying data centers across the globe. If our application would be used worldwide, we would need to split our database. This can be done with "sharding". Lets assume the application originated in UK and that resulted in us having data in one DB already - UK shard. After some data analysis, we see that some of our customers are originating from US. To increase the speed for them, we build another shard - US shard. Later on we discover that Asian markets are joining in as well. This can lead to creation of third shard - Singapore shard. Now we can observe the shard activity. If one shard becomes more active than the others, we can scale just one shard, or opposite, we can remove shard if it becomes financially unnecessary (Alizadeh et al. 2010).

6.1.2 Data rules

As mentioned in the subsection above, one of the data centers is in the UK. Since UK is no longer a part of EU, GDPR (Truong et al. 2019) is not effective on UK, but there are some other data protections placed by UK government that we would need to follow. In case our data would be in EU, we would need to respect GDPR policies in place. Mentioned above is also Asian market. Country with the biggest population there is China but, we decided against putting shard there. The reason is "the great firewall of China". Chinese data policies are strict in sense of their government having access to all of the data. Due to privacy and anonymity, it is a better decision to have data center close to China but not in the country. This is the reason why third shard is in Singapore and not China (Clayton, Murdoch, and Watson 2006).

6.2 Improvements

Here is the list of improvements that would be useful in the application. Node that improvements are not listed in any particular order. If application would go into production, some of them could not be ignored (example: HTTPS).

- NodeJS to docker

  - at the moment, web server is running in terminal. It would be better to have docker config for it as we do have for database. With this, it is much simpler to "move" the project

- TLS/SLL connection
  - connection to the database is not secure. To improve that, we would implement TLS/SSL.
- MongoDB LDAP user connection
  - At the moment, only specified users can connect to our cluster with MongoDB. If this would be used in real world, we wold need to monitor connections from inside of the company. With LDAP, we can simply assign roles to users as permissions are needed. Root user shall not be used.
- HTTPS with reverse proxy
  - If this application would be on public domain, HTTPS would be crucial for this since it encrypts the connection. Also, if this would be self hosted with the company, we would need to set up some network rules and reverse proxy is one of them. One of the industry best options would be to use CloudFlare in front to defend application against DDOS attacks or any other bots (Cloudflare n.d.).
- 2FA
  - 2 factor authentication is a must now days for security. This is something our application should have in production.
- encrypt data
  - Now days, security breaches are all over the world. To combat this, we would encrypt our data so hackers cannot see what we have stored.
- DevOPS
  - In a team, having DevOPS pipeline would improve the delivery of the code. For example, fronted and backend could work on one task independently and when it comes to code merge, it would be possible due to DevOPS and its day-to-day testing (Microsoft n.d.[b]).
- Improved frontend
  - At the moment, NodeJS application is just for demonstration purposes. It is simple but it needs improvement and some optimisation.
- Logging

  - Servers do not send any logs back in case that anything would fail. We could use 3rd party applications like Graylog and/or Grphana to produce and visualise logs from database.

- Analytics
  - In this version, no user analytics is being produced. Database is capable of producing it, but it was not implemented. Example of analytics that we can have: number of users, most common device, etc...

7 References

References

Hackolade (n.d.). *Hackolade*. Last accessed 15 May 2022. URL: https://hackolade.com.

Microsoft (n.d.[a]). *Database design*. Last accessed 15 May 2022. URL: https://docs.microsoft.com/en-us/sql/relational-databases/databases/databases?view=sql-server-ver15.

Köhler, H. and S. Link (2018). "SQL schema design: foundations, normal forms, and normalization". In: *Information Systems* 76, pp. 88–113.

MongoDB (n.d.[a]). *About MongoDB*. Last accessed 15 May 2022. URL: https://www.mongodb.com/what-is-mongodb.

— (n.d.[b]). *MongoDB - What is NoSQL?* Last accessed 15 May 2022. URL: https://www.mongodb.com/nosql-explained.

Khazaei, H., M. Fokaefs, S. Zareian, N. Beigi-Mohammadi, B. Ramprasad, M. Shtern, P. Gaikwad, and M. Litoiu (2016). "How do I choose the right NoSQL solution? A comprehensive theoretical and experimental survey". In: *Big Data & Information Analytics* 1.2&3, p. 185.

IBM (2019). *CAP Theorem*. Last accessed 15 May 2022. URL: https://www.ibm.com/cloud/learn/cap-theorem.

Cattell, R. (2011). "Scalable SQL and NoSQL data stores". In: *Acm Sigmod Record* 39.4, pp. 12–27.

Rad, B. B., H. J. Bhatti, and M. Ahmadi (2017). "An introduction to docker and analysis of its performance". In: *International Journal of Computer Science and Network Security (IJCSNS)* 17.3, p. 228.

MongoDB (n.d.[c]). *Docker and MongoDB*. Last accessed 15 May 2022. URL: https://www.mongodb.com/compatibility/docker.

— (n.d.[d]). *Indexes*. Last accessed 15 May 2022. URL: https://www.mongodb.com/docs/manual/indexes/.

Villamizar, M., O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, et al. (2016). "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, pp. 179–182.

Benson, T., A. Anand, A. Akella, and M. Zhang (2010). "Understanding data center traffic characteristics". In: *ACM SIGCOMM Computer Communication Review* 40.1, pp. 92–99.

Alizadeh, M., A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan (2010). "Data center tcp (dctcp)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74.

Truong, N. B., K. Sun, G. M. Lee, and Y. Guo (2019). "Gdpr-compliant personal data management: A blockchain-based solution". In: *IEEE Transactions on Information Forensics and Security* 15, pp. 1746–1761.

Clayton, R., S. J. Murdoch, and R. N. Watson (2006). "Ignoring the great firewall of china". In: *International Workshop on Privacy Enhancing Technologies*. Springer, pp. 20–35.

Cloudflare (n.d.). *About Cloudflare*. Last accessed 15 May 2022. URL: https://www.cloudflare.com/en-gb/.

Microsoft (n.d.[b]). *What is DevOps?* Last accessed 15 May 2022. URL: https://azure.microsoft.com/en-gb/overview/what-is-devops/.