

Trevor Gutierrez

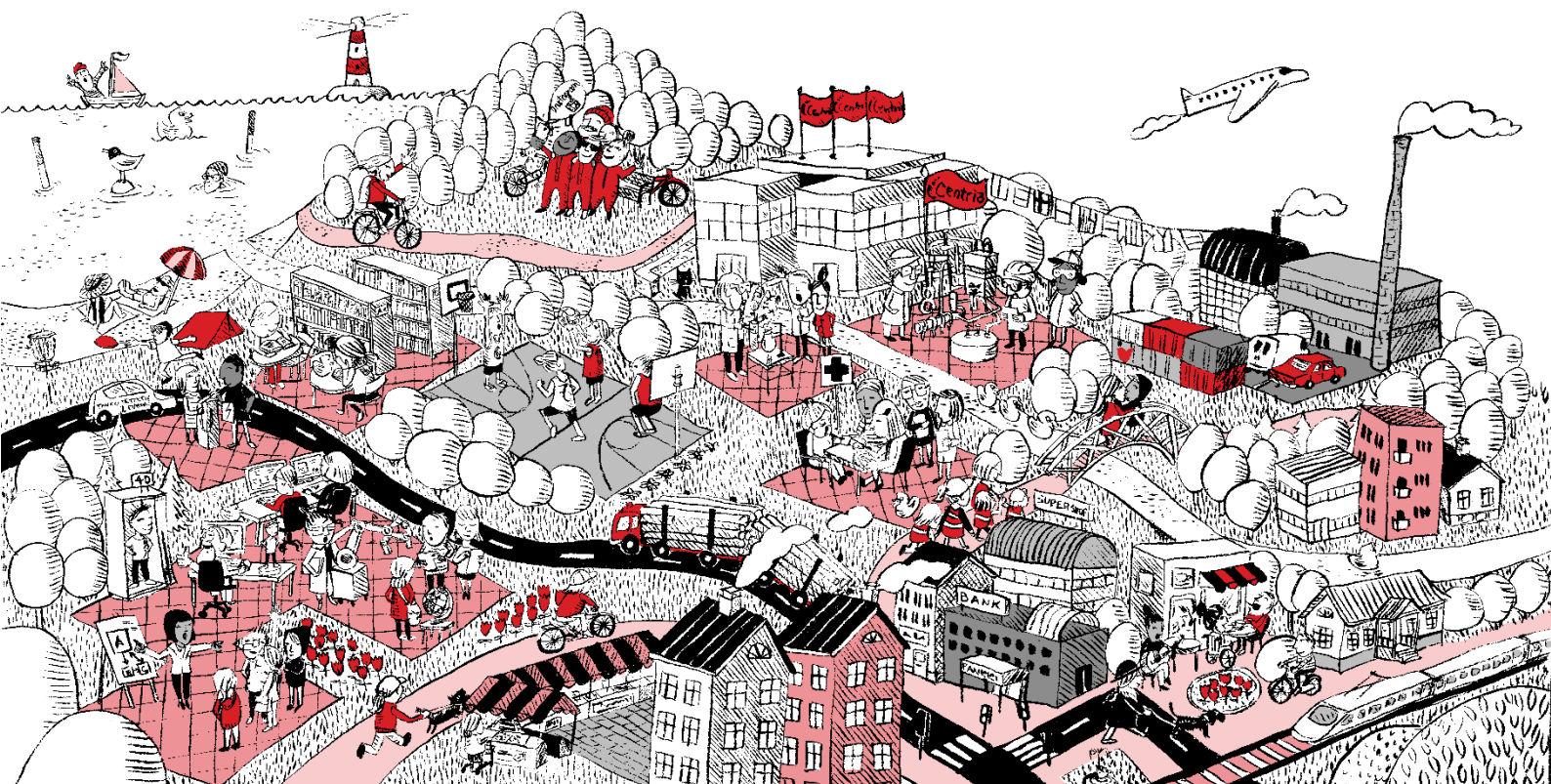
FULL-STACK WEB DEVELOPMENT & AZURE IoT HUB

Thesis

CENTRIA UNIVERSITY OF APPLIED SCIENCES

Information Technology

December 2021



ABSTRACT

Centria University of Applied Sciences	Date December 2021	Author Trevor Gutierrez
Degree programme Information Technology		
Name of thesis FULL-STACK WEB DEVELOPMENT & AZURE IoT HUB		
Supervisor Kauko Kolehmainen		Pages 43 + 25
Instructor Marko Mattila		
<p>The aim of the thesis was to provide a company-accessible website using JavaScript. It depicts the basic methods and ideas that are needed to create a web application. The thesis follows the creation of a website made to track device data from Azure IoT Hub.</p> <p>The technologies used in this thesis work are Microsoft/Azure SQL, Express, React/Redux, Node.js, and Azure IoT Hub. It allows for the company employees to access device data via a service which is configured to not be publicly accessible.</p>		
Key words Full-stack, IoT, REST API, web development		

CONCEPT DEFINITIONS

HTTP - Hyper Text Transfer Protocol

XML - eXtensible Markup Language

SPA - Single Page Application

I/O – Input/Output

CRUD - Create, Read, Update, Delete

REST - Representational State Transfer

CMS - Content Management System

JSON - JavaScript Object Notation

API - Application Interface

UI - User Interface

HTML - Hypertext Markup Language

CSS – Cascading Style Sheets

IoT – Internet of Things

ABSTRACT
CONCEPT DEFINITIONS
CONTENTS

1 INTRODUCTION.....	1
2 SULAON – THE COMPANY OF IOT DEVICES.....	2
3 FULL-STACK APPLICATION.....	3
3.1 Single Page Application	3
3.2 Technology Stack	3
3.2.1 Node.js.....	4
3.2.2 Express.....	5
3.2.3 React/Redux	6
3.2.4 Microsoft/Azure SQL and Azure IoT Hub.....	6
3.3 REST API and JSON.....	7
3.3.1 REST API.....	7
3.3.2 JSON	8
3.4 Content Management System	8
4 PLANNING	9
5 INSTALLATION.....	10
5.1 Node.js.....	10
5.1.1 Installing with GUI	10
5.1.2 Installing via terminal	11
5.2 Microsoft/Azure SQL and Azure IoT Hub.....	12
5.3 Useful additional installations and modules	13
5.3.1 Async.....	13
5.3.2 Sequelize	14
6 IMPLEMENTATION	15
6.1 Application Architecture	15
6.2 Database Architecture	17
6.3 Application’s Operation	18
6.3.1 Registration and Login.....	19
6.3.2 Role-Based Pages	22
6.3.3 Devices List	23
6.3.4 Device View	27
6.3.5 Update Device.....	29
6.3.6 Get Device Twin	32
6.3.7 Delete Device	37
6.3.8 Delete All Devices.....	38
7 CONCLUSION	39
REFERENCES.....	45

FIGURES

FIGURE 1. Diagram of Full-Stack concept.....	10
FIGURE 2. Node.js architecture	11
FIGURE 3. REST API design.....	13
FIGURE 4. Static site architecture.....	22
FIGURE 5. Dynamic site architecture	22
FIGURE 6. Database diagram	23

PICTURES

PICTURE 1. Node.js download page	17
PICTURE 2. Download of Node.js via terminal	17
PICTURE 3. IoT Hub connection steps in Visual Studio Code	18
PICTURE 4. Example of an async function	20
PICTURE 5. Application's login page	26
PICTURE 6. Admin dashboard	29
PICTURE 7. Devices List page after synchronization	30
PICTURE 8. Dynamic search bar	31
PICTURE 9. Device page	34
PICTURE 10. Device 'devStatus' update in application	35
PICTURE 11. Device description update	35
PICTURE 12. Twin query process console log with SQL query	37
PICTURE 13. Device twin queried in UI	38
PICTURE 14: Devices List after delete function.....	42
PICTURE 15. Devices List page after clicking "Delete All Devices" button	43

1 INTRODUCTION

This thesis regards full-stack web applications, the concept of single page application, the utilized technology stack, planning, building, and testing of the program. Websites are utilized for nearly every single purpose which involves the propagation of any information type involving any subject. A web browser is a software application that accesses the World Wide Web for the purpose retrieving content from a web server corresponding to a particular web address and presenting it to the end user's device. Once HTML was created and HTTP communication technology was written, the proper websites that we know today were born, one of the key traits being the web address format. CSS and JavaScript completely revolutionized how information is presented and the speed. Previously, server-side development required programming languages, such as Java, PHP, and C#, which is usually associated with Microsoft. Later, many programs were created which enabled JavaScript server-side development, such as Node.js. Because of JavaScript libraries like React and faster transmission of data, very creative and complex websites are the norm and often consist of many individual components which change without even a page reload required. As web technology and means of connectivity have increased, IoT (Internet of Things) concepts and devices became much more prevalent and more companies utilized them more often, as it is a type of smart concept in terms of technology. With the advent of fast data transfer and modern processing power and developmental concepts, many of these technologies were quickly written and designed for cross-communication and many possibilities have been revealed. (Lea, 2021.)

2 SULAON – THE COMPANY OF IOT DEVICES

Sulaon's main business focus involves electronics and embedded systems development. It specializes in high-performance micropower electronics, software control that includes power management features, high-efficiency SMPS and Class-D technology, and radio frequency and wireless local area network data. While Sulaon is usually focused on development involving physical electronics, it also does some Linux-related work. Most projects are for customers and based on contractual agreements, but the project for this thesis has been conceptualized by the company for the purpose of internal use to track its own device data from Azure IoT Hub. (Sulaon Oy, 2015.) Sulaon has about fifty employees. There is an office in Vantaa and there is a factory in Salo.

3 FULL-STACK APPLICATION

Web development has drastically changed over time, thanks to the improvement of information technology and processors. HTML could only be used to serve static information, but JavaScript has made it possible to serve dynamic elements to the browser. A full-stack application includes the server-side, the client-side, and the database. (Mozilla Inc, 2021.) A full-stack application connects to the server to retrieve data dynamically. It only exists in the browser. Upon initial access, the app is rendered by the server and its back-end services are supported. Deriving from modern web applications, Sulaon's internal project is designed to build an API (Application Programming Interfaces) and use JSON (JavaScript Object Notation) as the formatting language. (Mozilla Inc, 2021.)

3.1 Single Page Application

Unlike the early days of the Internet where it was required to change pages to do anything, most modern services use the concept of the single-page application (SPA). The main goal of SPA is providing fluid user experience with dynamically loaded content as information is requested or needed. It can be a simple application, such as a CREATE, READ, UPDATE, DELETE (CRUD) service, like a to-do list. SPA can also handle many requests and support a very complex application. The largest advantage of SPA is to be able to dynamically load and update different components of the page. This can be done without sending more requests than necessary or reloading a full page and the ease of use has led to widespread adoption. Many different frameworks and libraries exist to make SPA development much simpler, such as React, Vue, Ember, and Angular. (Bloomreach, [n.d].)

3.2 Technology Stack

The presented technology stack consists of Microsoft/Azure SQL, Express, React, and Node.js. Microsoft/Azure SQL is the database, Express is back-end web application framework, React is the JavaScript library for building the user interface, and Node.js is the back-end runtime environment. The reason why the technology stack is selected because Sulaon already had prior work done on Azure and an active account is already available, as a result, and the other tools are very simple to use and have widespread use and assist greatly in the speed of development. In addition, Azure's IoT Hub environment is utilized, as the application involves integration with that environment and its

respective devices that Sulaon owns. Full-stack web applications involve the development of back-end and front-end services and use of a database and can be developed with many different combinations of technologies, many which are known by key acronyms, such as MEAN (Mongo, Express, Angular, Node), MERN (Mongo, Express, React, Node), and PERN, which uses PostgreSQL instead of Mongo. (Doshi, 2019.)

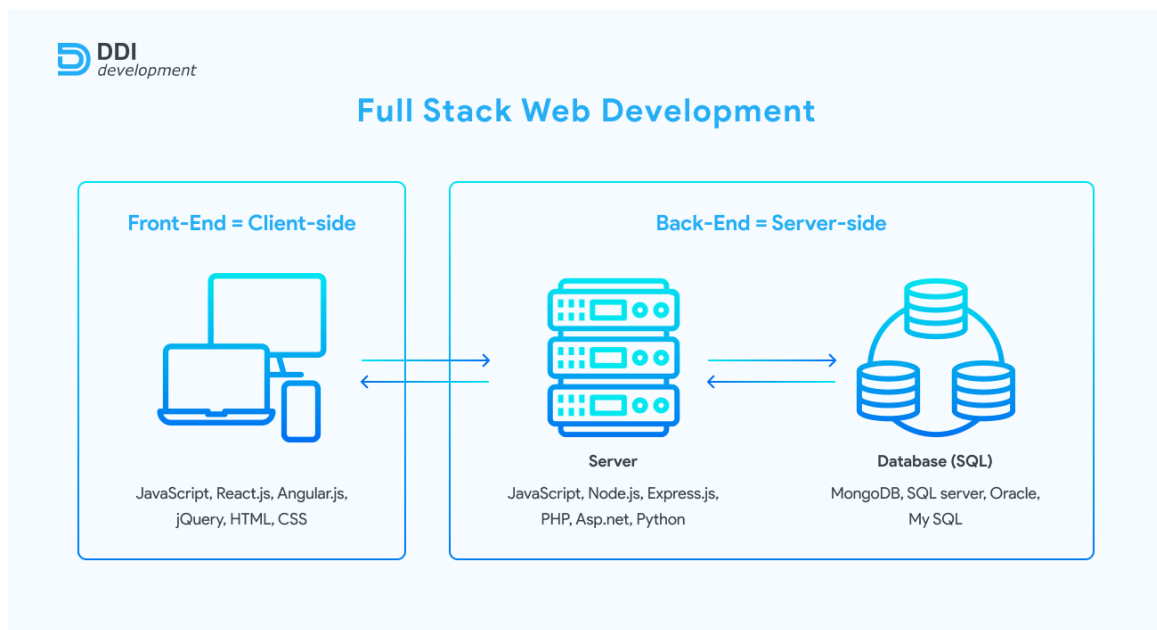


FIGURE 1. Diagram of Full-Stack concept (DDI Development, 2020)

3.2.1 Node.js

Node.js is a runtime environment which allows developers to create scalable web servers and build web applications and execute them outside of a web browser. However, it is not limited only to web development; Node.js has also been utilized in the development of mobile phone applications and other desktop applications. Node.js provides many features to assist in creating applications that run JavaScript on the server side, such as asynchronous I/O, single-threaded or multi-threaded, sockets, and HTTP connections. When building web applications using other server-side programming languages, such as Java and Python, a new thread is created for every connection with event looping. This approach is suitable if the server has enough hardware infrastructure for the traffic. The single threaded, non-blocking I/O is handled differently by a server. Instead of having one thread for every connection, the web server receives the requests and they are put in the Event Queue. The Event

Loop starts to process a request. If the request does not include any blocking I/O operation, the server processes it and responds to the client. If I/O blocking happens, an available thread will take up the request. Node servers can support tens of thousands of simultaneous connections, as evidenced by its use by many of the biggest corporations. (TutorialsPoint, [n.d].)

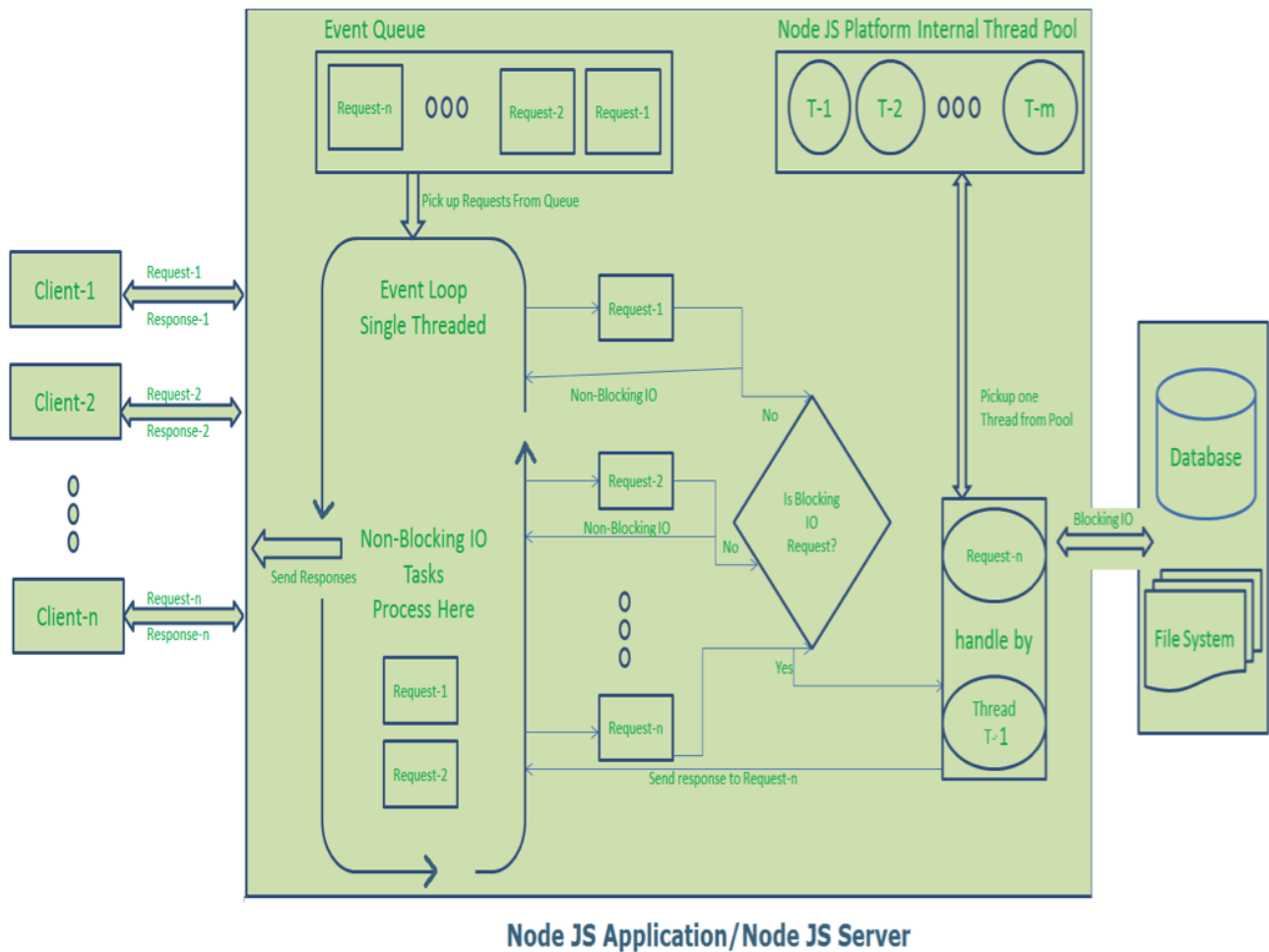


FIGURE 2. Node.js architecture (Jörg 2016)

3.2.2 Express

While Node.js is a great choice to make a web server, it is ultimately only a run time environment that happens to have a few modules already built in. The default Node API is not able to handle complex routing in applications, thus a web framework is required. Express is a very versatile web framework to use to build web applications. (OpenJS Foundation, 2017.) Furthermore, Express reduces difficulties and time in setting up a web server to handle incoming requests and returning a response. While

Express is somewhat limited, many middleware packages and libraries have been created to handle any web elements, including cookies, sessions, security headers, data routes, and much more. It is also “unopinionated” which means that there are very few restrictions for how to set up a web server. (Mozilla Foundation, 2021.)

3.2.3 React/Redux

There are many libraries and frameworks which work to support front-end web development and many which are devoted to JavaScript alone. React is a JavaScript library for building UIs. The main attraction is that components dynamically change as new data is introduced and it changes only components which are necessary. Another main feature of React is state management and because everything is written in JavaScript, data can easily be passed through the application. (Facebook, 2021.) Redux is another JavaScript library that goes well with React. It makes testing easier by allowing problems to be easier to track, it simplifies state control and persistence, and it provides a very organized view when paired with Redux DevTools. (Abramov, 2021.)

3.2.4 Microsoft/Azure SQL and Azure IoT Hub

Microsoft Azure is a cloud computing service which is very widely used. Microsoft also has Microsoft SQL Server, their main relational database management system, and Azure SQL is a version of SQL Server which is operated within Azure, so they use the same syntax. It is also scalable and performance ranks high. (Microsoft, 2021.) Data is able to be stored in the key-value format, which means that JSON is supported by Microsoft SQL. One major note about Microsoft/Azure SQL is that, because they are Microsoft’s property, they use Transact-SQL (T-SQL), an extension of the standard SQL language. In short, T-SQL adds procedural programming, local variables, string processing, data processing, and mathematics. (JavaTpoint, 2021.) Azure IoT (Internet of Things) Hub is a Microsoft application for clients to connect, manage, and monitor their devices’ communications in accordance with their IoT application. It allows per-device records, authentication, and management. (Microsoft, 2021.)

3.3 REST API and JSON

XML-RPC is the method that was previously used the most to transfer data via HTTP protocols. It is much more complicated and resource intensive than JSON, which has become a standard for REST API development. JSON is syntactically identical to the JavaScript object syntax, hence the acronym, but JSON itself is pure text. Any programming language can be used to read and generate JSON. As stated before, JSON utilizes a key-value pair. (W3Schools, [n.d].)

3.3.1 REST API

A REST (Representation State Transfer) API is an application with an architectural style based on standardizing communication between computer systems to make it much easier. It uses JSON as a formatting language and uses the HTTP protocol for communication. The simplest type of application to utilize the REST API style is an application which can create, read, update, and delete (CRUD) data. CRUD applications can be basic or more complex. A CRUD app has a stateless interface which can process basic POST (create), GET (read), PUT (update), and DELETE requests. (Domareski, 2021.)

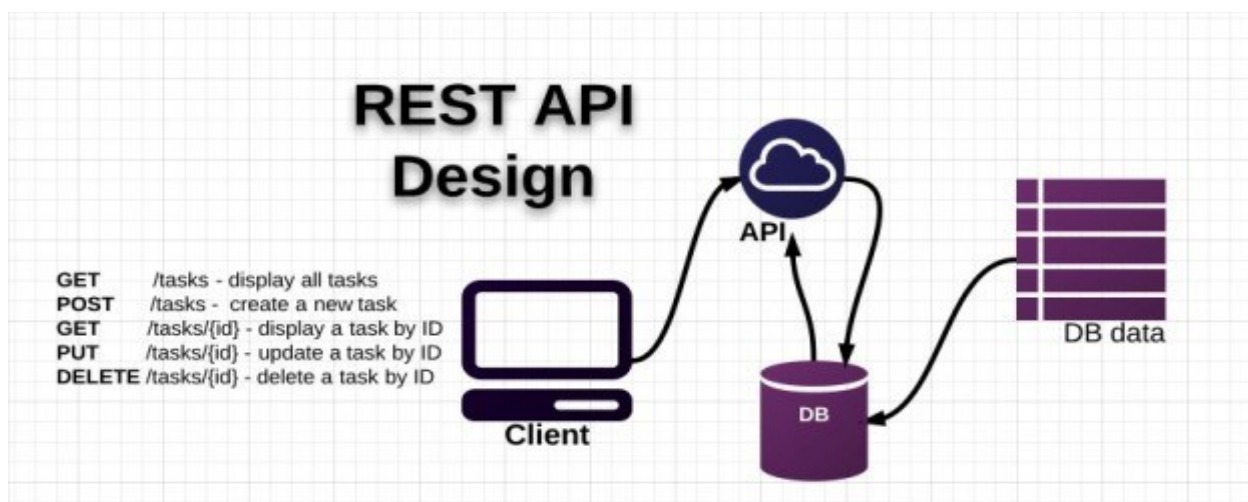


FIGURE 3. REST API design (Singhal, 2018)

3.3.2 JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is very similar to JavaScript object syntax, as it is derived from it. Although this is the case, JSON is a very simple format for any developer to learn, regardless of JavaScript experience. JSON can be used with many programming languages. XML was the previous standard, but it is much more complicated and XML was not completely JavaScript-friendly. JSON has been standardized because of its seamless transition to JavaScript and has become the standard. It is written in a key-value format. (Ecma-International, [n.d].)

3.4 Content Management System

A CMS (content management system) is an application which manages web content and allows multiple users to create new content, read existing content, update existing content, or delete existing content. Because it is usually browser-based, it is usually a collaborative effort. (Spilotro, 2018.)

4 PLANNING

The project was conceptualized by Marko Mattila, the development director of Sulaon Oy. The main goal was to create a web application for the purpose of managing the company's devices which are connected on their Azure IoT Hub. The application provides a consolidated environment in which all users can see all devices on their Hub. The application is only meant for the company, so steps were taken to limit the service to that audience. JSON was utilized throughout the project and a goal was to use Azure SQL because of the company's already-existing Azure account, where their IoT Hub is also hosted. It was a very difficult project nearly half of concepts were unknown upon entrance, but optimism was kept because it was the ultimate learning opportunity. There were no other developers in this project. The planning of a web application involves a few basic steps, though not all of these were able to be utilized in this process, as time was essential. Technology stack is important, but the planning of the front-end/state is also essential, as good planning can prevent the necessity to backtrack a decision because not everything was thought through enough. Knowing the userbase and planning the database schema is extremely important too. Planning API endpoints in association with the requirements of the database activity and planning the component hierarchy are the two steps that could be the most crucial to the planning phase, as it will ultimately affect the direct user experience. (Windmill, 2015.)

5 INSTALLATION

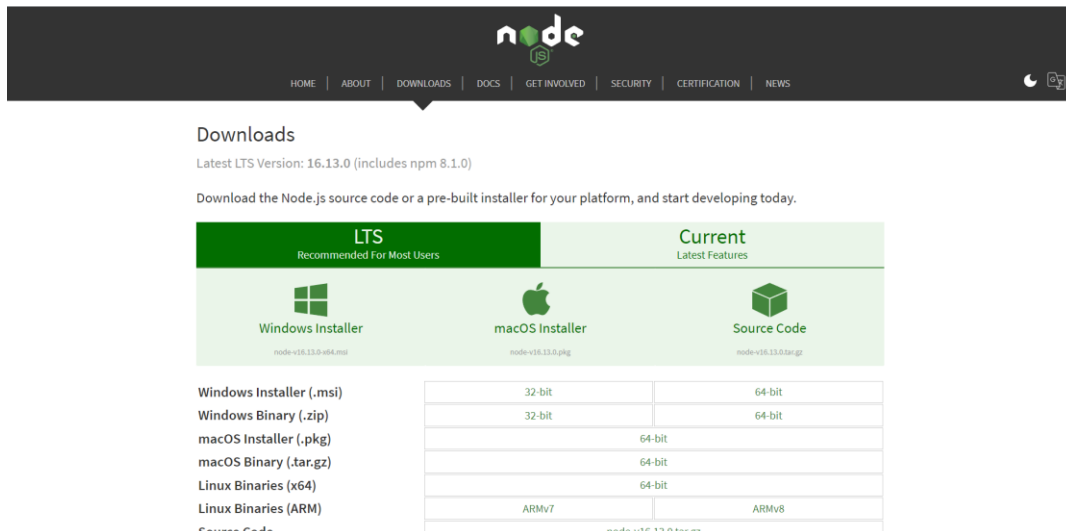
For the project to start, Node.js, Express, React, and Azure SQL needed to be installed. Then, familiarization with React and Azure IoT Hub was necessary. The OS in this case was Debian 10, a flavor of Linux. Installation is very similar for MacOS; Windows installation differs quite some from this process. (Tucakov, 2019.)

5.1 Node.js

While Node.js is very important, npm is a package manager for the JavaScript language which is used to install different modules. npm is community-supported and is made up of many volunteers and skilled developers around the world who share their code and download code from others for reuse in a project or if a particular task was developed efficiently, so that rewriting is not necessary. (npm Inc, [n.d].)

5.1.1 Installing with GUI

Node.js download can be done from the official website if the Node.js installation package is downloaded. It is mostly advisable to download the LTS (long-term support) version because it is the version which is deemed for stable use. There is another option available for download, which is the latest version; however, it is not recommended unless one seeks to develop Node.js itself because there could be potential bugs or new features which are not completely tested yet. Picture 1 displays the official download page for Node. (OpenJS Foundation, [n.d].)



PICTURE 1. Node.js download page (OpenJS Foundation, [n.d.])

5.1.2 Installing via terminal

While the package manager for Debian is called dpkg, use of the utility APT is necessary to download certain packages for Debian, including Node.js. Picture 2 displays the terminal during Node installation. After installation, it can be checked with the command “node -v”. (Rahul, 2017.)

```
shovon@linuxhint: ~
File Edit View Search Terminal Help
shovon@linuxhint:~$ sudo apt install nodejs
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libnode64 libuv1 nodejs-doc
Suggested packages:
  npm
The following NEW packages will be installed:
  libnode64 libuv1 nodejs nodejs-doc
0 upgraded, 4 newly installed, 0 to remove and 123 not upgraded.
Need to get 6,667 kB of archives.
After this operation, 30.3 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

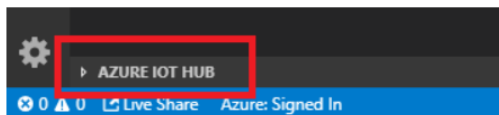
PICTURE 2. Download of Node.js via terminal

5.2 Microsoft/Azure SQL and Azure IoT Hub

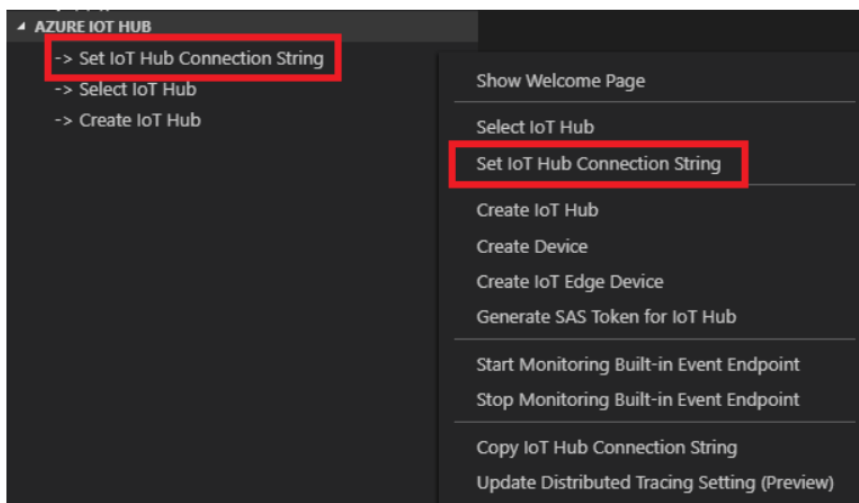
Both Azure products require a subscription to Microsoft Azure, which Sulaon Oy already has. While it is ultimately owned by Sulaon, connection still established to the IoT Hub via connection string and the ability to view all the devices and their properties was enabled. The program Visual Studio Code includes support for Azure IoT Hub. Azure SQL database connection was established via use of the program/IDE (integrated development environment) called DataGrip (APPENDIX 1), developed by JetBrains. (Microsoft, 2021.) Picture 3 is a screenshot of connection instructions.

Prerequisites

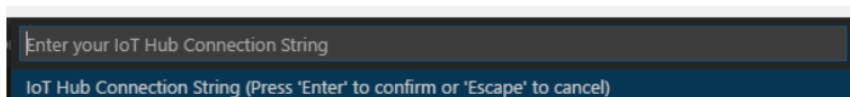
1. In Explorer of VS Code, click "Azure IoT Hub" in the bottom left corner.



2. Click "Set IoT Hub Connection String" in context menu.



3. An input box will pop up, then enter your IoT Hub Connection String (It is one-time configuration, and please make sure it is **IoT Hub Connection String** not **Device Connection String**. The format is `HostName=<my-hub>.azure-devices.net;SharedAccessKeyName=<my-policy>;SharedAccessKey=<my-policy-key>`).



PICTURE 3. IoT Hub connection steps in Visual Studio Code (Microsoft, 2021)

5.3 Useful additional installations and modules

While the project could technically be done without additional tools at this time, npm and other contributors have provided many packages which can contain libraries, frameworks, and other useful tools and bits of code which can greatly assist in any kind of task, depending on what specific task is at hand. Async utility module is used in the program and the Sequelize ORM (object-relational mapping) tool was utilized to assist in manipulating and writing code for the database within JavaScript code instead of using plain SQL language within the JavaScript code. In addition, React and Express are crucial to this project, but they are easy npm package installations done via the terminal in Debian 10. React is installed upon entering “npm install -g create-react-app” into the terminal and then the first React app can be created upon entering “create-react-app project-name-here” into the terminal. For Express, the process is similarly very easy, as it requires only the line “npm install express – save”, which also saves Express to the dependencies list for the potential project, (Nguyen, 2020.)

5.3.1 Async

Async is a utility module which provides straightforward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with Node.js and installable via the command “npm install –save async”, it can also be used directly in the browser. (npm, Inc., 2021.) An "async function" in the context of Async is an asynchronous function with a variable number of parameters. The final parameter is a callback, which must be called once the function is completed. An error should be the first callback, just in case. Otherwise, the desired return value should occur. (McMahon, [n.d.].) Picture 4 shows an example of an async function in code.

```
// for use with Node-style callbacks...
var async = require("async");

var obj = {dev: "/dev.json", test: "/test.json", prod: "/prod.json"};
var configs = {};

async.forEachOf(obj, (value, key, callback) => {
  fs.readFile(__dirname + value, "utf8", (err, data) => {
    if (err) return callback(err);
    try {
      configs[key] = JSON.parse(data);
    } catch (e) {
      return callback(e);
    }
    callback();
  });
}, err => {
  if (err) console.error(err.message);
  // configs is now a map of JSON data
  doSomethingWith(configs);
});
```

PICTURE 4. Example of an async function (npm, Inc., 2021)

5.3.2 Sequelize

Sequelize is an open-source Node.js module (npm package, as well) which is an ORM tool. It enables manipulation of relational databases in the JavaScript language and syntax and within JavaScript code and without using raw SQL queries or very limited. It causes code to appear cleaner and more organized and more consistent. As SQL can look messy when written in with another programming language, it can drastically improve readability. Sequelize will work with many relational database management systems, including MySQL, PostgreSQL, MariaDB, and Microsoft SQL Server. This serves to be useful because of its compatibility with Microsoft SQL Server. (Pius, 2021.)

6 IMPLEMENTATION

To start implementation and due to the newness of React and Sequelize, an extremely basic CRUD app was created as a template for the actual project to fit into. It allowed for basic data entry into the database, basic updates, basic viewing, and basic deletions, but it did not utilize any real-world items, like later with Azure IoT Hub. Building a basic CRUD application allowed more time to understand some of the basic concepts before more serious data was handled. Developing a full-stack application consists of server-side development and client-side development (back-end and front-end, respectively). Ideally, development should first be done on the server-side because the information is the most important part of the entire concept and the front-end will be useless if there's no data for it to handle nor if the server communication is not even written for that data handling to occur. One should develop the server-side and have a basic template done so that when development starts on the client-side, those developmental decisions can be made more efficiently. Also, part of implementation is API testing. This ensures that all data and errors are properly handled. This will be demonstrated for every step. A popular method of API testing is to use cURL, which allows for many different supported protocols. cURL can be a little bit more difficult to technically configure, as it takes place in the terminal. It is easy to make syntax errors if a user does not have deeper knowledge. (Newell, 2020.) The best alternative that is very popular is the application Postman. It provides a very clean and organized interface that allows for quick selection of the request type (POST, GET, PUT, DELETE, PATCH, etc.), a field for the request URL, and a convenient tab for inserting necessary headers. (Kotecha, 2018.)

6.1 Application Architecture

In basic applications like this, data is retrieved by a browser via Hyper Text Transport Protocol (HTTP). The back-end code is what determines how that communication happens and the other events can be configured to happen upon initiation in the browser, which is the client-side. The browser's request is then returned with an HTTP message to indicate what action has taken place. There are static websites and dynamic websites. Static websites return all hardcoded content when requested while dynamic websites are more complex and can return specific content only when that content is needed. (Mozilla, 2021.)

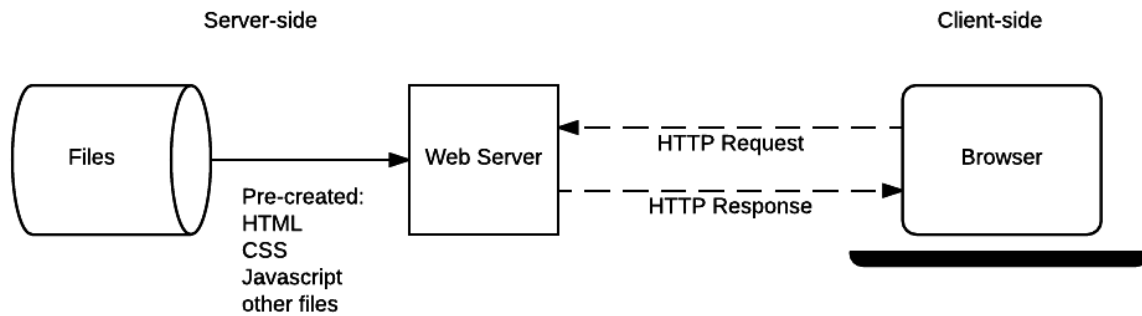


FIGURE 4. Static site architecture (Mozilla, 2021)

Figure 4 and Figure 5 show models of static and dynamic sites, respectively. The differences between the static site and the dynamic site become very clear in the intermediate communications of the files and server and database. As has been explained, dynamic has a lot more complexity. There are GET/POST requests involved, which are not involved in a static website. No data needs to be fetched in a static website beyond all data already available to the website.

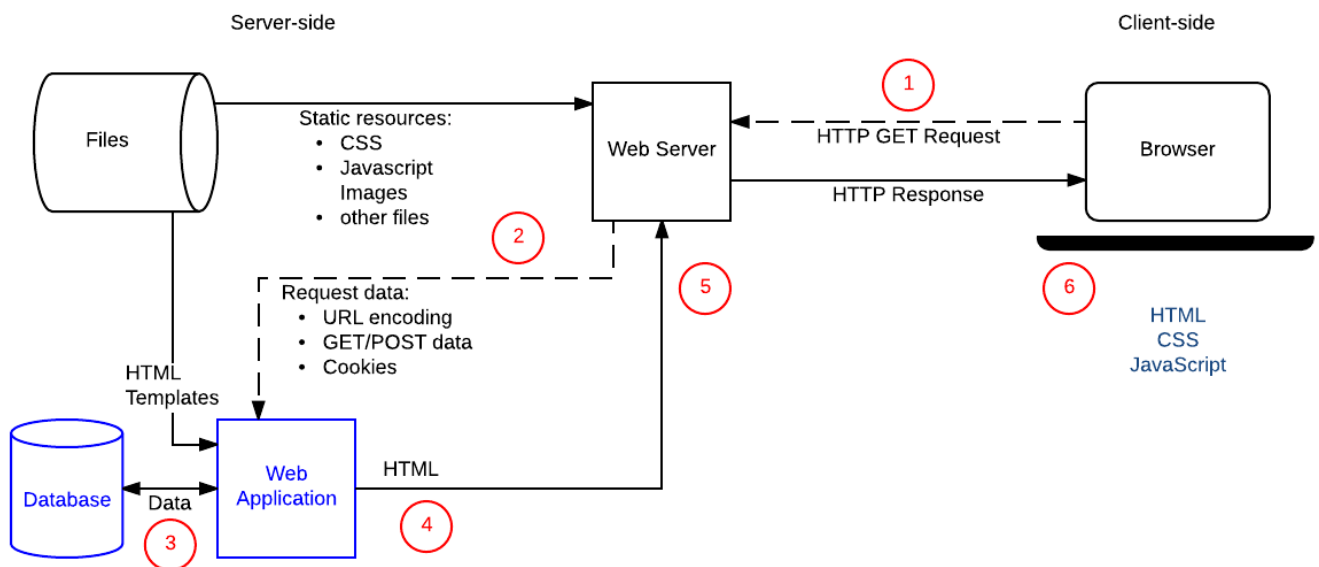


FIGURE 5. Dynamic site architecture (Mozilla, 2021)

6.2 Database Architecture

The database diagram is below, which displays schema and relations between certain tables. The ‘roles’ table holds the three roles that are available in the application: user, admin, and pm. Each role has an ‘id’ assigned to it and the ‘user_roles’ table is a consolidation of the ‘id’ from ‘users’ table and the ‘id’ from the ‘roles’ table and, together, they indicate as to which role that a specific user is assigned. The database diagram is shown in Figure 6. The ‘devices’ table consists of a few fields of important information about the devices; ‘deviceId’ is the ‘deviceId’ from Azure IoT Hub, ‘devStatus’ reads if the device is ‘enabled’ or ‘disabled’ and is set as a Boolean value, and ‘freeDescription’ allows for the company to make any comments on the device and it has been written so that any synchronization with the IoT Hub itself will not overwrite that field. (Gribkov, 2020.)

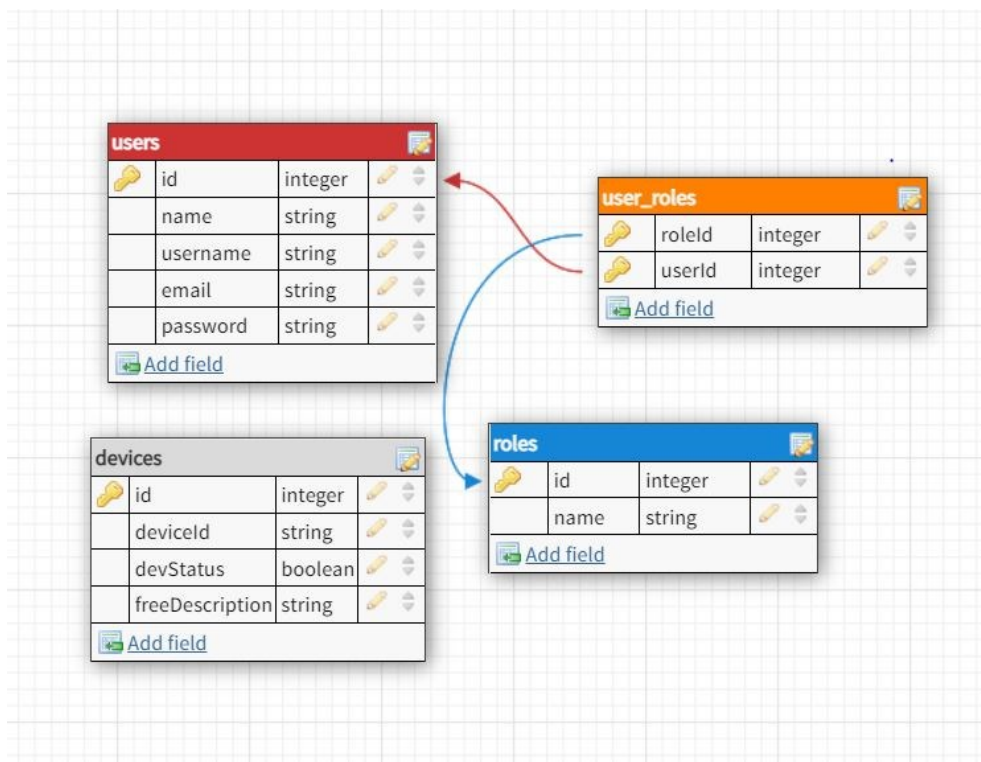


FIGURE 6. Database diagram

Each table also has timestamp fields ‘createdAt’ and ‘updatedAt’, which properly update. The ‘roles’ table has been placed below, as well, so that the ‘roleId’ of each role is known for later actions in this project. I created the tables with Sequelize. First, Sequelize variables must be declared to be able to use the libraries and, like any database configuration, database credentials are required. Then,

Sequelize needs the models, so those should be imported. Finally, before declaring the file as an export for later use, roleId and userId associations need to be coded in. For all those steps, these are snippets of code from that process:

CODE 1:

Declaration of Sequelize variables:

```
const Sequelize = require('sequelize');

const sequelize = new Sequelize(credentials and other info here)

.....

db.Sequelize = Sequelize;

db.sequelize = sequelize;
```

Model importation/declaration example:

```
db.user = require('../model/user.model.js')(sequelize, Sequelize);
```

Associations example:

```
db.role.belongsToMany(db.user, { through: 'user_roles', foreignKey: 'roleId', otherKey:
'userId'});
```

6.3 Application's Operation

The application will present the user with the login screen. The application has been configured so that it forces the user to stay on the login screen until an existing user has logged in and a JSON Web Token (JWT) has been returned to the user. Additionally, since the application is meant only for the use of the company and its employees, it was written so that user registration cannot be done via the application itself; it must be done either by direct manipulation of the database via a management system or through a program which tests the API itself.

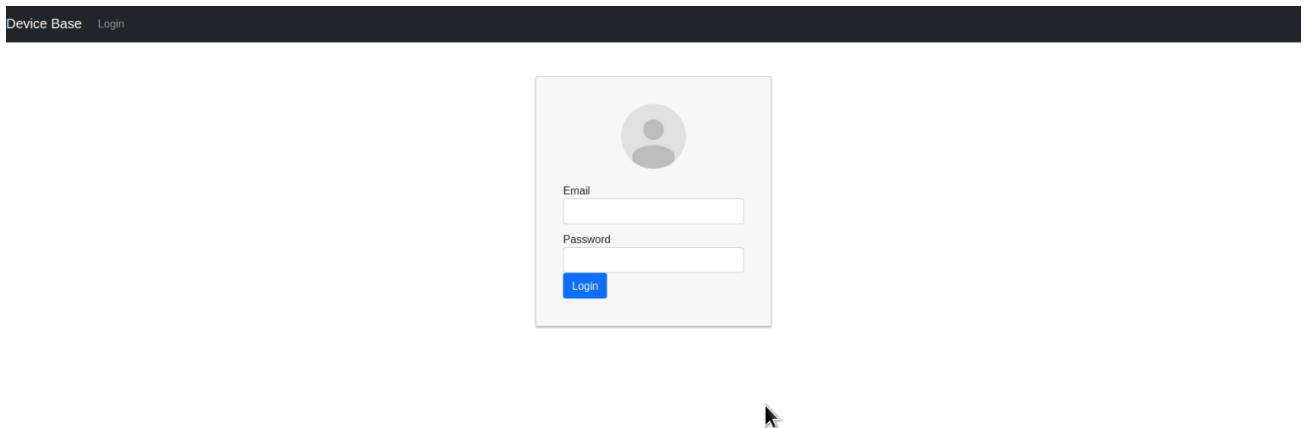
After login, the user will be presented with their dashboard, which confirms login and displays their information conveniently for them. I have made sample pages, which are labelled as “User Page”, “Admin Page”, and “Moderator Page”, whose purpose is to show that I have been able to properly configure varying levels of access to certain parts of the website based on which role is assigned to the current user who is logged in.

The device list is queried from the database upon page loading. The device list can be synchronized with the Azure IoT Hub. When a device is clicked and “Edit” is clicked, options for the specific device are shown. One can write anything in ‘freeDescription’ and update it, enable/disable, or delete the device from the database. Even if deletion occurs, the device will return to the database if Hub synchronization is done. On the devices list, when a device is clicked, attributes appear on the side and when scrolling, it stays static so that the attributes stay with the page scroll action for convenience.

6.3.1 Registration and Login

As stated above, registration is not a feature in the main application because the application is not meant for public use; it is for the company’s internal use. Thus, either the database must be manipulated directly or an API testing program (like Postman) can be used to so that it is done directly via the API (APPENDIX 2). There are three roles: admin, user, and moderator (pm). During registration, passwords are hashed and salted via bcrypt so that even a database manager is not able to view them. SHA encryption algorithms were outdated and bcrypt became a newer and more secure standard. (Arias, 2021.) Sequelize syntax is used to verify registration (‘findOne’ for email). For login, there is a very basic login system which returns a JSON Web Token (JWT) for verification and utilizes Sequelize syntax to handle such login. JWT is an open standard (RFC 7519) for transmitting information as a JSON object and can be trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. (Auth0, Inc., 2016). After all users are registered, the database will display all stored users (APPENDIX 3). There are three users because three users were registered. There is a table which holds the ‘role’ names in correspondence to integer values (APPENDIX 4). Finally, there is a table which holds the integer ‘role’ values for the ‘id’ integer values to correspond the different roles, ‘user_roles’ (APPENDIX 5). There are three to correspond with each type and are stored in integer format and correspond with the integers of users’

'id' integer and ultimately assign the actual role's function together. They will affect which pages and actions can be done for each type. The login test is successfully done in the API test (APPENDIX 6). The user information is properly returned in the JWT. The user will be permitted to do the actions which the 'role' label has been granted. The application forces the login screen if there is no user logged in. The purpose is to prevent any unauthorized users from accessing any other page. The login page is shown in Picture 5.



PICTURE 5. Application's login page

The application has proper error handling implemented for the login system (APPENDIX 7). There will be a notice and no access is granted if no correct and existing information has been entered. After all actions have been done, the actions can be seen in the console. In addition, the SQL queries which happen during the entire process are visible. Relevant registration code is here. Note the 'User.create' line and the fields which correspond with the 'users' database table. That is Sequelize in action:

CODE 2:

```
User.create({
  name: req.body.name,
  username: req.body.username,
  email: req.body.email,
  password: bcrypt.hashSync(req.body.password, 10)
}).then(user => {
  if (req.body.roles) {
    Role.findAll({
```

```

        where: {
            name: req.body.roles
        }
    }).then(roles => {
        user.setRoles(roles).then(() => {
            res.send({ message: "User and role
registered successfully." });
        });
    });
} else {
    user.setRoles([1]).then(() => {
        res.send({ message: "Registered as role 'USER'."
});
    });
}
})

```

Relevant login code. Sequelize is again used to find the user via email and the entered password is compared to the database password:

```
let passwordIsValid = bcrypt.compareSync(req.body.password, user.password);
```

JWT is then signed and returned to allow successful login, with all relevant and safe user information on the JWT. Roles are sent and stored as an array literal:

```

        let token = jwt.sign({ id: user.id } , config.secret, {
            expiresIn: 86400 // expires in 24 hours
        });

        let authorities = [];
        user.getRoles().then(roles => {
            for (let i = 0; i < roles.length; i++) {
                authorities.push(roles[i].name.toUpperCase());
            }

            res.status(200).send({auth: true, accessToken: token, id: user.id, name: user.name, username:
                user.username, email: user.email, roles: authorities })
        });
    });

```

This code forces the login page if there is no user stored in 'currentUser' (no user logged in):

```

{currentUser ? (
    <div className="navbar-nav ml-auto">
        <li className="nav-item">
            <a href="/signin" className="nav-link" onClick={log-
Out}>

```

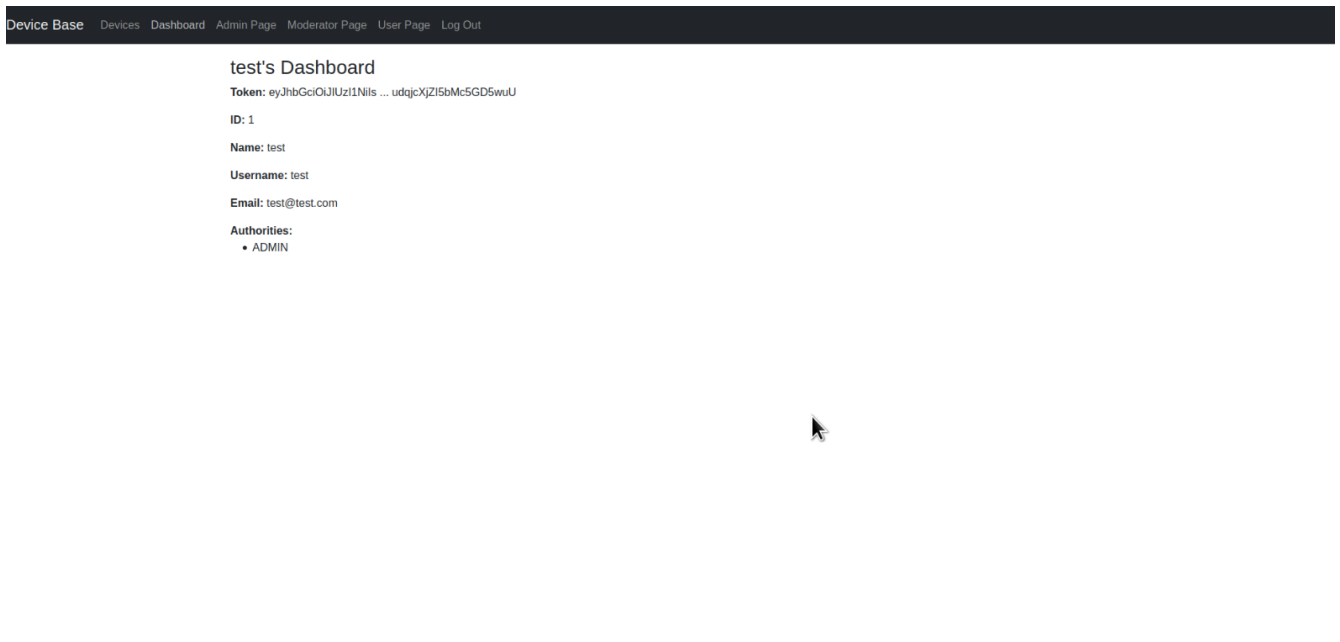
```

                Log Out
            </a>
        </li>
    </div>
) : (
    <div className="navbar-nav ml-auto">
        <Redirect strict from="/" to="/signin" />
        <Route path="/signin">
            </Route>
            <li className="nav-item">
                <Link to={"/signin"} className="nav-link">
                    Login
                </Link>
            </li>
        </div>
    )}

```

6.3.2 Role-Based Pages

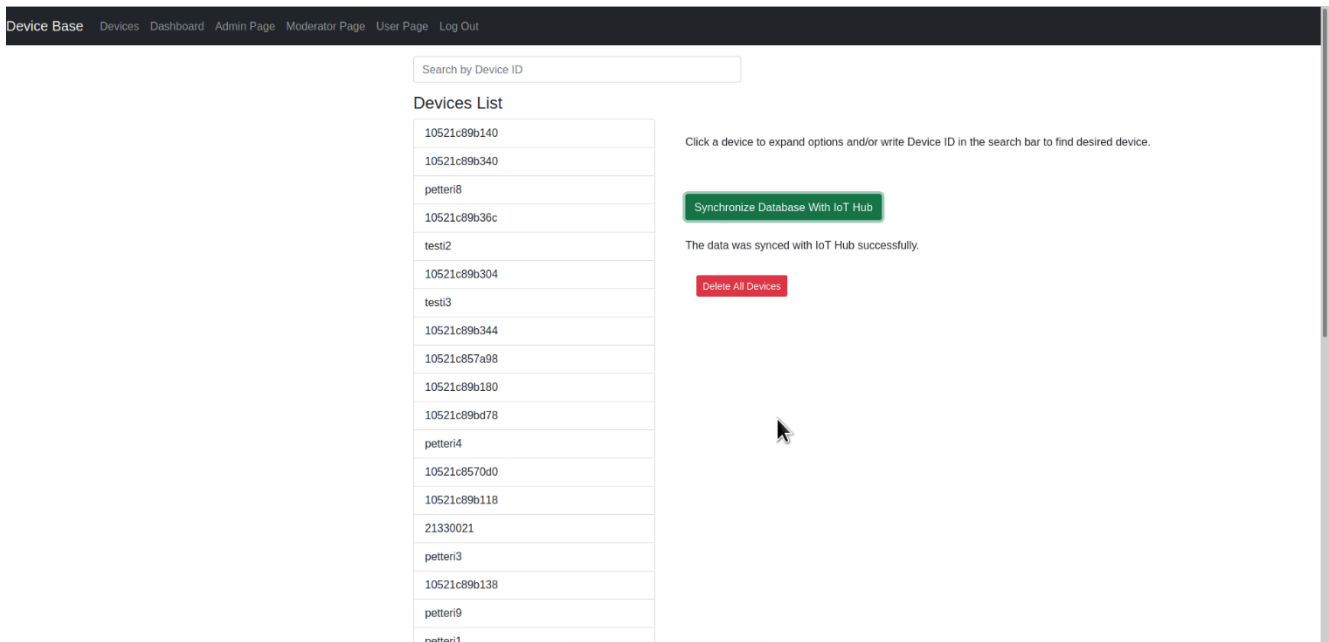
After the user has logged in, there is automatic redirection to the dashboard page, which displays all information for the user's convenience. This information is from the JWT. Admins can see everything on the website, but moderators and users can see only pages which are relevant to their role. It is similar to the previous section in how 'currentUser' storage determines if the website is open to the user or if the login page will be forced. This time, it checks what the role of 'currentUser' is from the JWT, in which the role is stored and changes the view for each role. Viewing of the pages is also restricted and will be blank if the address of an unauthorized page is entered. API tests are done on a GET request and the dashboard example can be seen in Picture 6. A screenshot of SQL queries in this process is also provided. The API tests for the pages are done in Postman and are done on the different roles. The 'role' field, as seen in the dashboard example, is different for each 'role' type. This corresponds to the labeled roles. The moderator has similar capabilities in most CRUD apps, aside from widescale data manipulation. Moderator and administrator can access 'user' page. The 'user' page test is done, as well. 'User' cannot access the 'admin' and 'pm' pages the tabs do not show for 'user'. (Degges, 2018.)



PICTURE 6. Admin dashboard

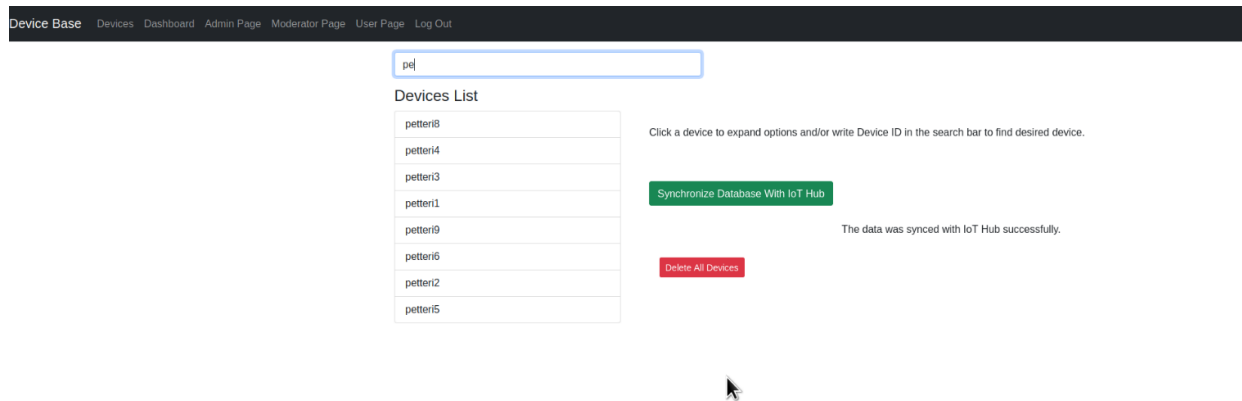
6.3.3 Devices List

Assuming that devices are already in the database, when the user clicks to go to the Devices List page, the page load will query the database to select all devices (`SELECT * FROM devices`) and all devices will show on the page, as can be seen in Picture 7. This has been done by placing the retrieval action inside of the 'useEffect' React hook. When the button 'Synchronize Database With IoT Hub' is clicked, a query is sent, which follows examples from the query language guide from the Azure IoT Hub documentation, which is very similar to regular SQL language. (Microsoft, 2021.) An API test for synchronization was done before client-side development (APPENDIX 8). The request was programmed as a PUT request and the resulting SQL table is also shown (APPENDIX 9).



PICTURE 7. Devices List page after synchronization

Along with the list is a search bar. It has been programmatically connected with the queried results of the database to allow for dynamic changes to occur as a user enters characters into the text field, as seen in Picture 8. Also, as a user scrolls the Devices List page, the section of the interface with the buttons and where device attributes are displayed will be static so that the view seems to go with the page as scrolling happens (APPENDIX 10). Clicking on a device will change the view from the text to device's attributes. Ultimately, whether the devices are queried from the database or synchronized with the Hub, both actions will be a type of "RETRIEVE_DEVICES" action in Redux DevTools (APPENDIX 11).



PICTURE 8. Dynamic search bar

This is the relevant piece of code which uses the Azure IoT Hub query language and retrieves all devices:

CODE 3:

```
let query = registry.createQuery('SELECT * FROM devices');
```

One important aspect of this code is that this queries all device twins. While the concept of ‘device twin’ will be explained in further detail in the later, designated section for ‘Get Device Twin’, the basic information regarding this step is that it is a JSON document which stores the device state information, including metadata, configurations, and conditions. (Microsoft, 2021.) One of those pieces of information is the ‘deviceId’ of each device.

After that piece of code, the ‘forEach’ function is used and Sequelize is used to insert the data into the database. Then, Sequelize’s ‘findAll’ function is used to list the devices from the database which have just been synchronized and stored into the database. On a regular database query when devices are already inserted, the basic ‘findAll’ function is used to make the database entries appear on the application’s Device List.

This is the relevant piece of code for the aesthetic aspect of the list and dynamic search bar:

CODE 4:

```
<ul className="list-group">
```

```

        {devices &&
        searchResults.map((device, index) => (
            <li
                className={
                    "list-group-item " + (index === currentIndex ? "active"
: "")
                }
                onClick={() => setActiveDevice(device, index)}
                key={index}
            >
                {device.deviceId}
            </li>
        ))}
    </ul>

```

These are the relevant codes for the search bar function. Notice ‘useEffect’ is also used here but is separate from the device retrieval function call. During development, placing both together caused an infinite loop of database queries, so it was crucial to fix that and it was done:

```

const [searchTitle, setSearchTitle] = useState("");
const [searchResults, setSearchResults] = useState([]);
.....
useEffect(() => {
    setSearchResults(devices.filter(device => device.deviceId.toString().toLowerCase().includes(searchTitle.toLowerCase())));
}, [devices, searchTitle]);

const onChangeSearchTitle = e => {
    const searchTitle = e.target.value;
    setSearchTitle(searchTitle);
};

```

This is the relevant piece of code which keeps the Device attributes panel static (namely, the ‘position’ property):

```

<div style={{ position: "fixed", right: "-20px", top: "160px" }} className="col-md-6 device">

```

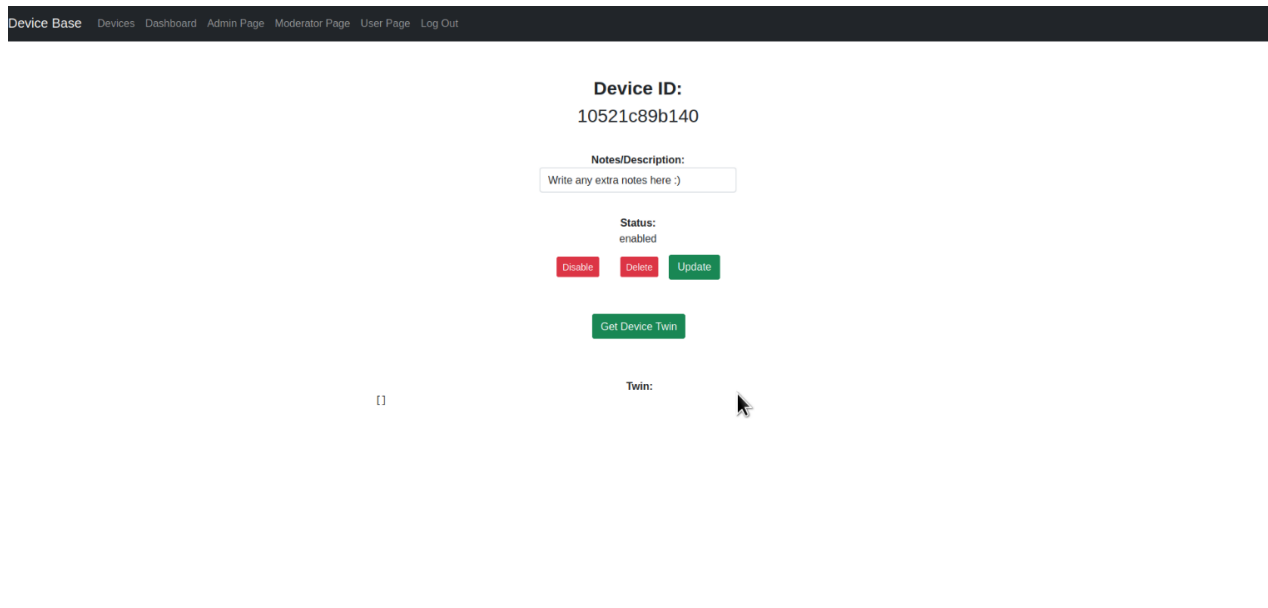
Finally, this is the SQL query that is responsible for the synchronization function and they can be seen in the console logs (APPENDIX 12). To explain it simply, if a device exists which already has a certain ‘deviceId’ that matches ‘deviceId’ from all device twins which were just queried (refer to the above code that used the Azure IoT Hub query language), it will update all other fields which are correlated with that device; if this device, otherwise, does not exist in the database, it is added along with all corresponding information. Also, remember that the ‘freeDescription’ column is never updated, as the use case specifies that this is meant to stay consistent on the UI for employees’ use, despite any synchronizations:

CODE 5:

```
IF EXISTS(SELECT deviceId FROM devices WHERE deviceId = (?) ) UPDATE devices SET devStatus =
(?) WHERE deviceId = (?) ELSE INSERT INTO devices (deviceId, devStatus, freeDescription) SE-
LECT (?, (?), (?))
```

6.3.4 Device View

After a user clicks the ‘Edit’ button on a selected device from Devices List, the user is presented with an interface which shows the stored data about that specific device. One way which I have limited the amount of database queries is that after the initial query upon first navigation to Devices List, all devices stay in the state, so no reloading or dropping of data happens while a user is signed in. Sequelize is used to query the device by the primary key (id) of the selected device, taken from the URL via the ‘req.params’ JavaScript line, from the user interface (UI). I ensured that the device information would be stored in the ‘currentDevice’ variable by creating a blank ‘initialDeviceState’ variable which mirrors the properties of the devices from the database and storing that same data in that variable as an array (APPENDIX 13). On this page, the user is presented with the following action options: ‘Disable’/‘Enable’, ‘Delete’, ‘Update’, and ‘Get Device Twin’. Basic CSS (Cascading Style Sheets) was used to alter the styling and placement of components. There are relevant screenshots of the API tests, done on a GET request (APPENDIX 14), and some application screenshots and code are attached to show how this was implemented. Picture 9 shows an example of one device’s page. This operation is a type of READ action, which is part of the basic CRUD model. (Degges, 2018.)



PICTURE 9. Device page

The relevant code follows this. This first snip of code is the blank ‘initialDeviceState’ variable which is used as a reference for the ‘currentDevice’ variable in which the data is later stored in:

CODE 6:

```
const initialDeviceState = {
  id: null,
  deviceId: "",
  freeDescription: "",
  devStatus: true,
};

const [currentDevice, setCurrentDevice] = useState(initialDeviceState);
```

This code snip is where the device data is stored into the ‘currentDevice’ variable for later possible uses, as there was no need to clear the state and do another ‘dispatch’ of a Redux action to retrieve the device data again when the entire database has already been loaded on the prior Devices List page. Thus, Redux is not used in this case:

CODE 7:

```

DeviceDataService.retrieveDevice(id)

    .then(response => {

        const currentDevice = {

            id: response.data.id,

            deviceId: response.data.deviceId,

            freeDescription: response.data.freeDescription,

            devStatus: response.data.devStatus,

        };

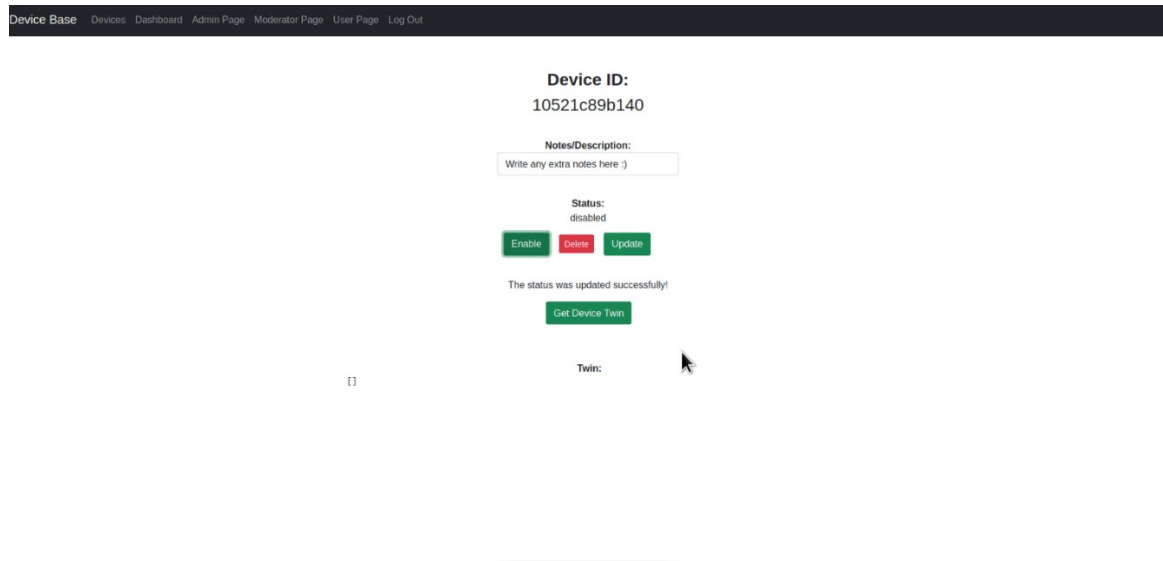
        setCurrentDevice(response.data);
    });

```

6.3.5 Update Device

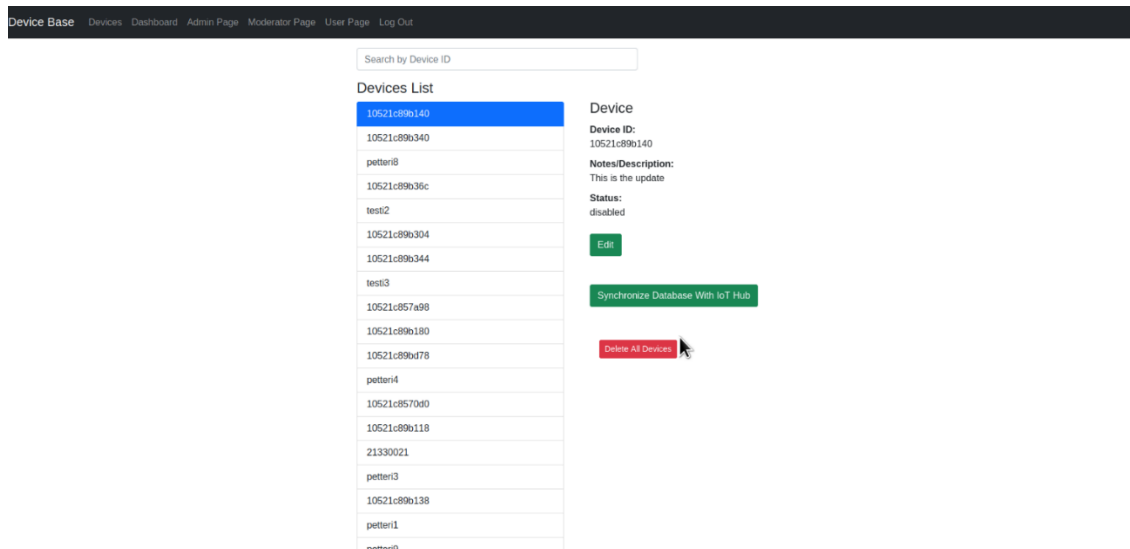
While there is technically only one function within the program that is truly labelled as an update function, there are two updates which can be done on the Device page: one is to click the ‘enable’/‘disable’ button, which has been written as a Boolean and operates as ‘true’/‘false’ in the actual database (APPENDIX 15), and the other is to write new text in the ‘Notes/Description’ field and click the ‘Update’ button, which updates that field in the database for the corresponding device. When synchronization is done, all are as ‘enabled’ (‘true’ in the database) by default. Changing the Boolean does not require the ‘Update’ button to be clicked; in fact, changing the Boolean value ‘enable’/‘disable’ (again, ‘true’/‘false’) is coded 100% on the front-end and the only back-end code which contains the Boolean value itself is to update the database. The back-end is not responsible for changing the value of it, though the value could be changed in API testing (any value other than ‘true’ or ‘false’ would cause an error, as the front-end is solely responsible for ‘enable’ and ‘disable’). The front-end code for the aesthetical factor of changing the Boolean value and the code from the Devices List page which shows the ‘Device’ attributes tab upon clicking a device from the actual list are very similar; it is coded within some HTML code and essentially functions as a Boolean in that it shifts the view one way or another depending upon the condition and there are only two available conditions. The code that is responsible for the functioning of the Boolean change is similar to the Device page code, which created the blank ‘initialDeviceState’ variable to store the data for storage in the ‘currentDevice’ variable, which now

comes into use. The ‘currentDevice’ variable and the ‘devStatus’ field are utilized. The process having been done in the UI is depicted in Picture 10.



PICTURE 10. Device ‘devStatus’ update in application

Similarly, updating the ‘Notes/Description’ field (‘freeDescription’ in ‘devices’ database table) uses the ‘currentDevice’ variable and the ‘id’ of ‘currentDevice’. What differentiates the two is that updating the description does not require creation of a blank variable for initial storage purposes. Also, clicking the ‘Update’ button to update the description pushes the user back to the Devices List page. Updating is another basic function of the CRUD model. (Degges, 2018.) A screenshot and code which are relevant to these processes are shown below. Picture 11 depicts the UI showing that the ‘Notes/Description’ field has been updated.



PICTURE 11. Device description update

The back-end code is very basic Sequelize syntax for updates and appears very similar to code from previous steps. This is the relevant code that shows the blank variable for changing the Boolean value and the storage and utilization of the 'currentDevice' variable:

CODE 8:

```
const data = {
  id: currentDevice.id,
  deviceId: currentDevice.deviceId,
  freeDescription: currentDevice.freeDescription,
  devStatus: status,
};

.....

dispatch(updateDevice(currentDevice.id, data))

.then(response => {
  console.log(response);
  setCurrentDevice({ ...currentDevice, devStatus: status });
  console.log(currentDevice);
  setMessage("The status was updated successfully!");
})
```

This is the relevant code for updating the ‘Notes/Description’ field. Note that it also utilizes the ‘currentDevice’ variable:

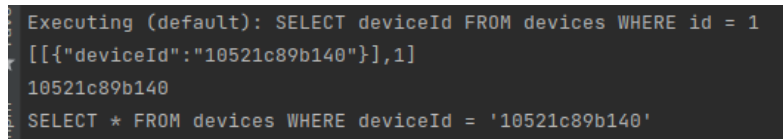
```
const updateContent = () => {
    dispatch(updateDevice(currentDevice.id, currentDevice))
        .then(response => {
            console.log(response);
            setMessage("The device was updated successfully!");
            props.history.push("/devices");
        })
        .catch(e => {
            console.log(e);
        });
};
```

6.3.6 Get Device Twin

An important part of working with devices on Azure IoT Hub is to understand what the ‘device twin’ is. It was explained previously that it’s a JSON document which stores the device state information, including metadata, configurations, and conditions. It can be used for simple checks, but can also be used, for example, to store important information about such a device, like location, last update time, and even the process of long-running workflows, such as updates. The device twin can also report if said device is connected to Wi-Fi or cellular data. Generally, the sections of a device twin are tags, desired properties, reported properties, and device identity properties. Properties within those sections can also be replaced and/or changed and be reflected in the latest device twin. (Microsoft, 2021.)

The server-side code for getting the device twin was complicated for me to resolve and took quite some time and many attempts mixed in while sometimes working on other tasks which I was able to solve quicker. First, the server-side code utilizes Sequelize and actually requires a hard-coded query this time and that query finds the ‘deviceId’ of the corresponding ‘id’ of the device page the user is

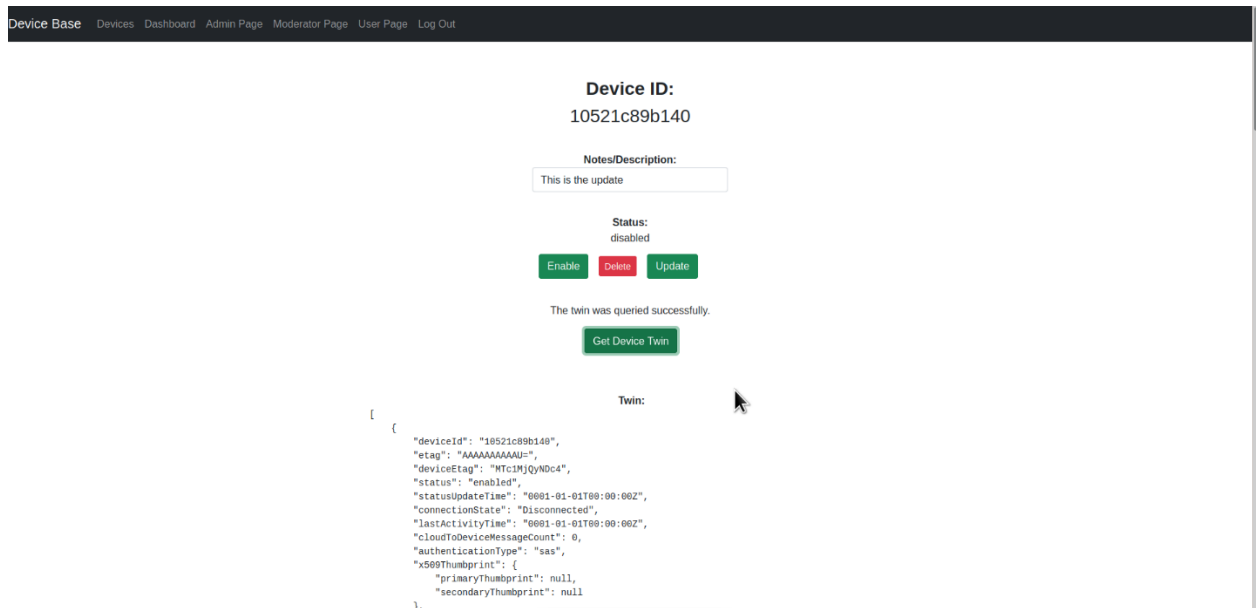
currently on by using the 'req.params' property to take the 'id' from the URL. Then, the result of that query presents 'deviceId' in the format "[{"deviceId": "10521c89b140"}],1"; the problem here is that my next step is to do an Azure IoT Hub query, but the query will not recognize 'deviceId' in that format. To solve this, I used the JavaScript 'slice' function to remove characters from both ends of "[{"deviceId": "10521c89b140"}],1" so that only "10521c89b140" is what remains, as seen in the logs in Picture 12. Then, I stored the remaining sliced result as its own variable and used that variable within the Azure IoT Hub query and the device twin was finally obtained.



```
Executing (default): SELECT deviceId FROM devices WHERE id = 1
[{"deviceId": "10521c89b140"}],1
10521c89b140
SELECT * FROM devices WHERE deviceId = '10521c89b140'
```

PICTURE 12. Twin query process console log with SQL query

On the client-side, it was very basic implementation to make it appear on the UI. I used the HTML '<pre>' tag, as it is used to render preformatted text on a browser page, and the device twin is certainly preformatted JSON, which made rendering very easy. The purpose of the only JavaScript code was to render the twin from the Redux store/state, in which the twin is stored (APPENDIX 16). When the user navigates away from the Device page, any twin information will be cleared from the state (APPENDIX 17). The relevant screenshots and code will show how this is done. A real device was utilized to ensure that the obtained data was the latest device twin data. Picture 13 shows the UI after the twin is queried. API tests were probably the most helpful here, as I was never sure exactly when the twin data would finally appear and those are included (APPENDIX 18), as well.



PICTURE 13. Device twin queried in UI

The action of querying the device twin is also done via Redux principles (dispatch, reducer, etc). For context, ‘reducer’ in Redux principles is what controls the state and is how the ‘twin’ state is cleared after navigation away from the Device page; the empty ‘initialState’ is returned on default. For more context, this is some code which is relevant to the ‘reducer’ concept:

CODE 9:

```
const initialState = [];

const twinReducer = (twin = initialState, action) => {

  const { type, payload } = action;

  switch (type) {

    case RETRIEVE_DEVICE_TWIN:

      return payload;

    default:

      return initialState;

  }
}
```

```
};
```

This code is relevant to the ‘actions’ concept in Redux:

```
export const queryDeviceTwin = (id, data) => async (dispatch) => {
  try {
    const res = await DeviceDataService.queryDeviceTwin(id, data);

    dispatch({
      type: RETRIEVE_DEVICE_TWIN,
      payload: res.data,
    });

    return Promise.resolve(res.data);
  } catch (err) {
    return Promise.reject(err);
  }
};
```

This code is an example of a line from the ‘device.service’ file that is responsible for routing the data through the API address:

```
const queryDeviceTwin = (id, data) => {
  return axios.post(API_URL + `device/${id}`, data, { headers: authHeader() });
};
```

As a reminder, this is what it looks like to use the Redux concept ‘dispatch’ to truly use the actions and reducers:

```
const queryTwin = () => {
  dispatch(queryDeviceTwin(currentDevice.id))
}
```



```

.then(response => {

    console.log(response);

    setCurrentTwin({ ...currentTwin, twin: response});

    setMessage("The twin was queried successfully.");

})

```

As it can be seen, when the action “queryDeviceTwin” is called in the main file (in this case, on the click of a button, which the previous function “queryTwin” is connected to), it is dispatched through the API and then is processed by the server to render the twin. Then, the reducer verifies the payload and, again as in this case, the final part of the reducer will return the twin state to empty so that any other possible twin to query can be stored and rendered later. The twin is rendered on the page with this code:

```
<pre>{JSON.stringify(store.getState().twin, null, 4)}</pre>
```

This code explains the concept of JavaScript’s ‘slice’ function that was described in the beginning of section 6.3.6. The variable ‘result’ is the original returned data that needed to be sliced in order to get ‘deviceId’ in the proper format. The eventual proper returned value has been assigned as the variable ‘properResult’:

```

let beginningSliceCharacterCount = 15;

let endingSliceCharacterCount = -6;

let properResult = JSON.stringify(result).slice(beginningSliceCharacterCount, endingSliceCharacterCount);

```

Finally, the following query is the Azure IoT Hub query which takes the newly created variable ‘properResult’ (the pure ‘deviceId’), and queries the twin (in Azure IoT Hub, “SELECT *” will query the entire device twin):

```
SELECT * FROM devices WHERE deviceId = '${properResult}'
```

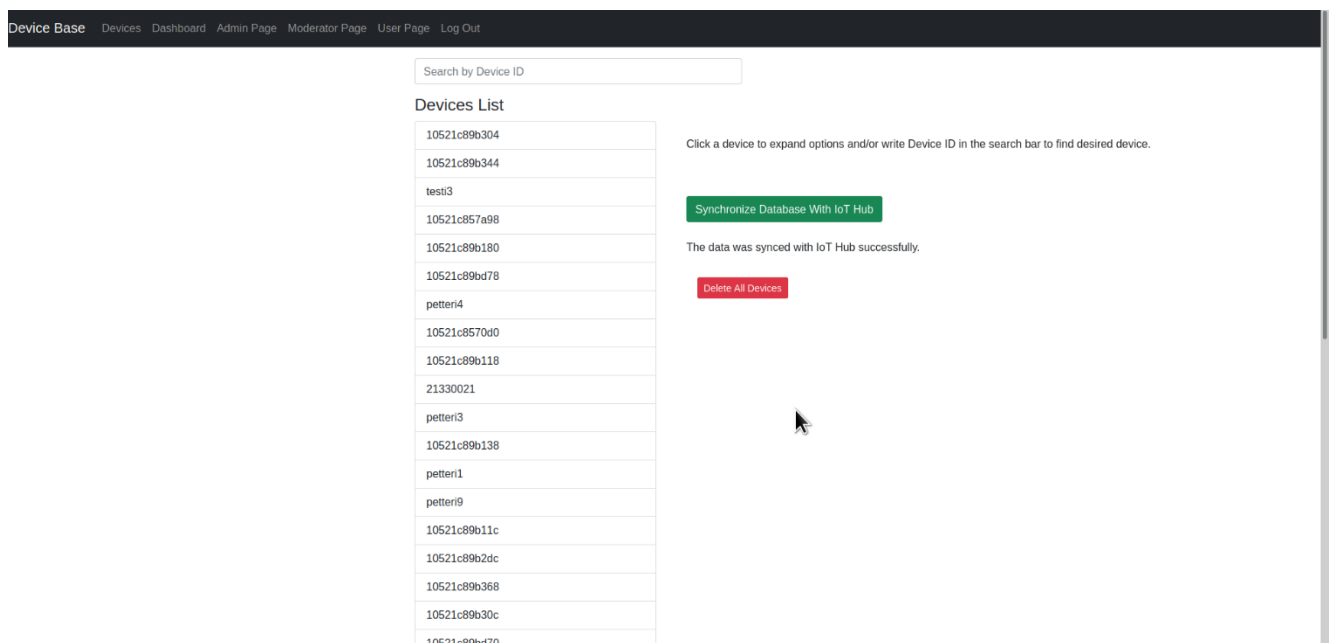
6.3.7 Delete Device

The delete function is very straightforward, after having gone through all other processes. The server-side code uses Sequelize syntax and the condition is based on the 'id' of the device which was just chosen for deletion. The application will then push the user back to the Devices List page and the reflected change will be apparent. The client-side is automatically updated after the button is clicked because the state is changed. Deletion is another basic function of the CRUD model. (Degges, 2018.)

API tests were conducted (APPENDIX 19) before the UI was developed. The relevant screenshot for the UI is depicted in Picture 14 and minimal required code for explaining the concept are below. The SQL table was also affected as according to the actions (APPENDIX 20) and the queries were logged (APPENDIX 21). The state change was reflected in Redux DevTools, confirming the action (APPENDIX 22). Not much code needs to be shown to understand how the delete function happens. After the Redux action is dispatched and the device is deleted from the server, using Sequelize syntax in the code, the application pushes the user back to the Devices List page with this relevant code:

CODE 10:

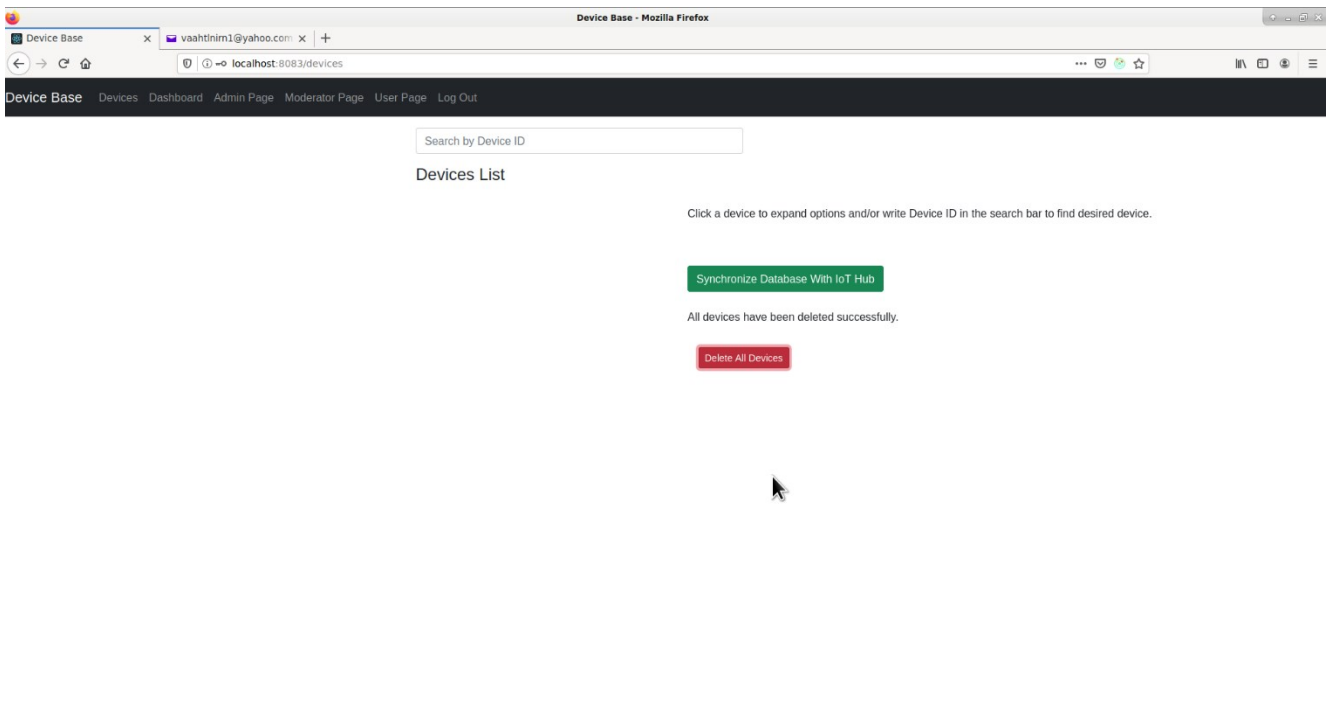
```
props.history.push("/devices");
```



PICTURE 14: Devices List after delete function

6.3.8 Delete All Devices

This works the same as the function to delete one device, except that this button is located back on the Devices List page. Besides that, it uses Sequelize with an empty condition, which signifies all conditions. In SQL, an empty condition is a reflection for “all”. The UI is automatically updated as the state changes. API tests were done before the client-side development started (APPENDIX 23). The screenshot of the UI displaying a successful message for deleting all devices is depicted in Picture 15. The SQL table was affected and all devices were gone (APPENDIX 24). In addition, the state change was reflected in the Redux DevTools window (APPENDIX 25). Devices can be obtained again via synchronization method which was demonstrated prior.



PICTURE 15. Devices List page after clicking “Delete All Devices” button

7 CONCLUSION

The thesis intended to convey the process of how a full-stack application is created in the JavaScript language. There were many challenges involved, including the inexperience in React, Redux, Azure SQL, or IoT Hub, it was mostly solo work with some assistance, and that the project idea was very abrupt after an issue with the previous project idea, but a good application was able to be built, despite any obstacles which complicated the project. The project was the best opportunity to learn some of the skills which could be necessary in a real-life workplace or hobby setting. It also allowed experience for real-world situations in which problem-solving skills would be necessary. The application was developed in about two months because one month was spent to learn some of the involved technologies. It followed the CMS model, including user authentication, web interface, editable content, and it represents a REST API. While the application is meant only for in-house use among employees for work purposes, adequate security and good role security are necessary and the application has a very basic system implemented to reflect that idea.

Ultimately, the project will experience further development, as many steps should be completed to bring the application to a functioning state for the company, but the groundwork has been made and is something that has been made very easy to improve upon. Because of the short development time and because development was done on an internship, the application unfortunately could not experience that practical use during the project's elapsed time, but it should be much easier for the company or another future developer or intern to really tweak everything and clean up certain parts of the application and add in the necessary features, especially because as this project was not originally the first intended project for the contract, this idea was more of an idea in the long-term for the company. The opportunity arose and, with the combination of completed work on the prior project idea, some code was able to be reused and an absolute fresh start was not necessary.

REFERENCES

- Abramov, D. 2021. Redux – A predictable state container for JavaScript apps. Available: <https://redux.js.org/>. Accessed 6.11.2021.
- Arias, D. 2021. Hashing in Action: Understanding bcrypt. Available: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/>. Accessed 6.11.2021.
- Auth0, Inc. 2016. JSON Web Token Introduction. Available: <https://jwt.io/introduction>. Accessed 6.11.2021.
- Bloomreach, [n.d.]. What is a Single Page Application. Available: <https://www.bloomreach.com/en/blog/2018/07/what-is-a-single-page-application.html>. Accessed 5.11.2021.
- DDI Development. 2020. Pros and Cons of JavaScript Full Stack Development. Available: <https://ddi-dev.com/blog/programming/advantages-and-disadvantages-of-javascript-full-stack-development/>. Accessed 5.11.2021.
- Degges, R. 2018. Tutorial: Build a Basic CRUD App with Node.js. Available: <https://developer.okta.com/blog/2018/06/28/tutorial-build-a-basic-crud-app-with-node>. 30.11.2021.
- Domareski, H. 2021. Designing a RESTful API. Available: <https://henriquesd.medium.com/designing-a-restful-api-736e53b10d3f>. Accessed 5.11.2021.
- Doshi, A. 2019. What is full stack development?. Available: <https://www.geeksforgeeks.org/what-is-full-stack-development>. Accessed 5.11.2021.
- Ecma-International, [n.d.]. Introducing JSON. Available: <http://www.json.org/>. Accessed 5.11.2021.
- Facebook. 2021. React. Available: <https://reactjs.org/>. Accessed 5.11.2021.
- Gribkov, E. 2020. SQL Database Design Basics with Examples. Available: <https://blog.devarart.com/sql-database-design-basics-with-example.html>. Accessed 29.11.2021.

JavaTpoint. 2021. T-SQL Tutorial. Available: <https://www.javatpoint.com/t-sql>. Accessed 5.11.2021.

Jörg, K. 2016. Programming Web Application with Node, Express and Pug. Available: <https://link.springer.com/book/10.1007/978-1-4842-2511-0>. Accessed 5.11.2021.

Kotecha, K. 2018. API Testing using Postman. Available: <https://medium.com/aubergine-solutions/api-testing-using-postman-323670c89f6d>. Accessed 7.11.2021.

Lea, T. 2021. A Brief History of Web Development. Available: <https://devdojo.com/tnylea/a-brief-history-of-web-development>. Accessed 29.11.2021.

McMahon, C. [n.d.]. Async Documentation. Available: <https://caolan.github.io/async/v3/global.html>. Accessed 6.11.2021.

Microsoft. 2021. Azure IoT Hub – Visual Studio Marketplace. Available: <https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-toolkit>. Accessed 6.11.2021.

Microsoft. 2021. Azure SQL. Available: <https://azure.microsoft.com/en-us/products/azure-sql/#product-overview>. Accessed 5.11.2021.

Microsoft. 2021. IoT Hub. Available: <https://azure.microsoft.com/en-us/services/iot-hub/#overview>. Accessed 5.11.2021.

Microsoft. 2021. Understand Azure IoT Hub device twins. Available: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins>. Accessed 6.11.2021.

Microsoft. 2021. Understand the Azure IoT Hub query language. Available: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-query-language>. Accessed 6.11.2021.

Mozilla Foundation. 2021. Express/Node introduction. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. Accessed 5.11.2021.

Mozilla Foundation. 2021. Introduction to the server side. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction. Accessed 8.10.2021.

Newell, G. 2020. Example Uses of the Linux Curl Command. Available: <https://www.lifewire.com/example-uses-of-the-linux-curl-command-4084144>. Accessed 7.11.2021.

Nguyen, S. 2020. What is npm? A Node Package Manager Tutorial for Beginners. Available: <https://www.freecodecamp.org/news/what-is-npm-a-node-package-manager-tutorial-for-beginners/>. Accessed 29.11.2021.

npm, Inc. [n.d.]. About npm. Available: <https://docs.npmjs.com/getting-started/what-is-npm>. Accessed 5.11.2021.

npm, Inc. [n.d.]. async. Available: <https://www.npmjs.com/package/async>. Accessed 6.11.2021.

OpenJS Foundation. 2017. Express. Available: <https://expressjs.com/>. Accessed 5.11.2021.

OpenJS Foundation. [n.d.]. About. Available: <https://nodejs.org/en/about/>. Accessed 5.11.2021.

OpenJS Foundation. [n.d.]. Download. Available: <https://nodejs.org/en/download/>. Accessed 29.11.2021.

Pius, O. 2021. Introduction to Sequelize ORM for Node.js. Available: <https://www.section.io/engineering-education/introduction-to-sequelize-orm-for-nodejs/>. Accessed 6.11.2021.

Rahul. 2017. How to Install Node.js on Debian 10/9/8. Available: <https://tecadmin.net/install-latest-nodejs-npm-on-debian/>. Accessed 29.11.2021.

Singhal, R. 2018. Restful API Design. Available: <https://medium.com/@rachna3singhal/restful-api-design-95b4a8630c26>. Accessed 5.11.2021.

Spilotro, C. 2018. What is a Content Management System (CMS). Available: <https://www.zesty.io/mindshare/marketing-technology/what-is-a-content-management-system-cms-the-complete-guide/>. Accessed: 5.11.2021.

Sulaon Oy, 2015. Available: <http://www.sulaon.fi/english/index.html>. Accessed 8.10.2021.

Tucakov, D. 2019. How to Install Node.js and NPM on Windows. Available: <https://phoenixnap.com/kb/install-node-js-npm-on-windows>. Accessed 29.11.2021.

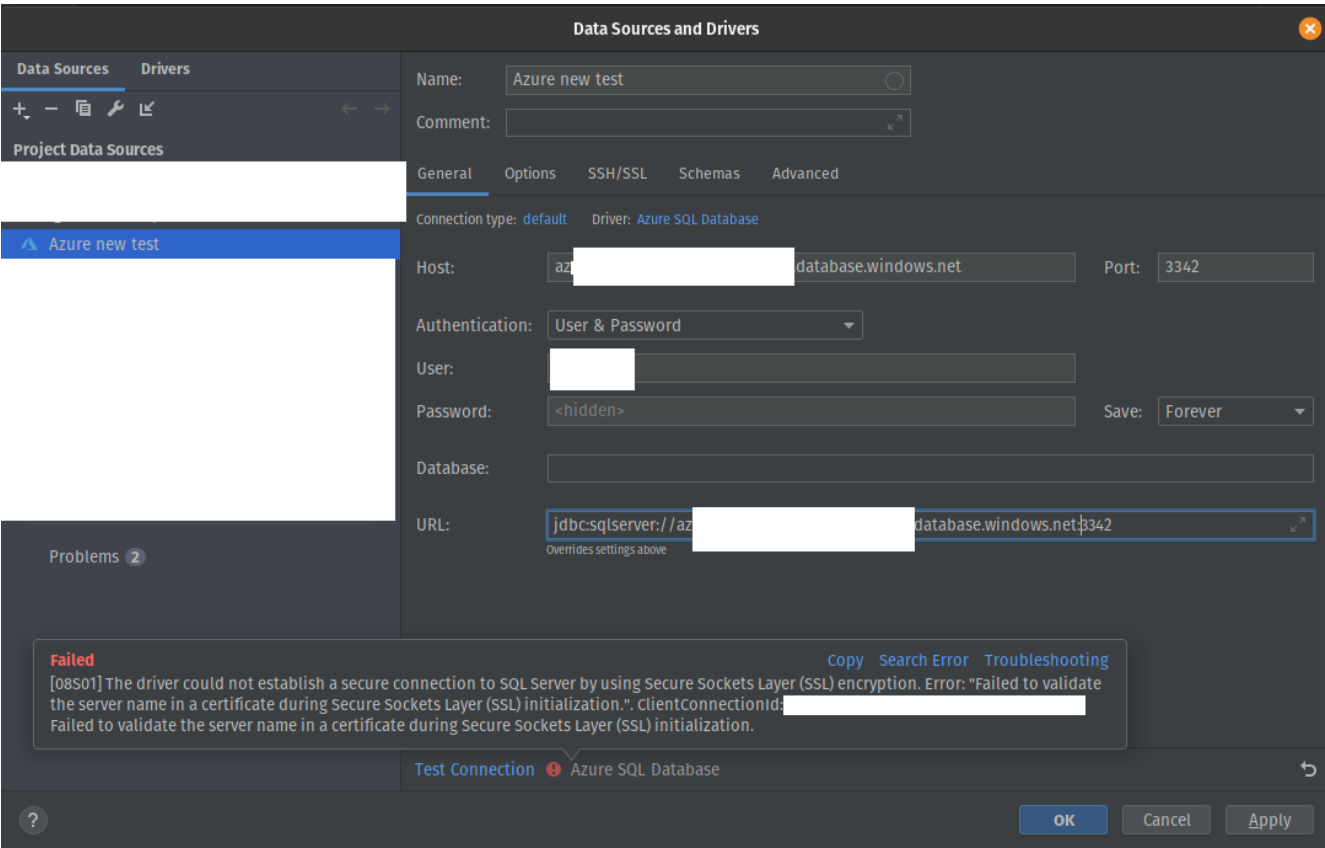
TutorialsPoint. [n.d.]. Node.js Tutorial. Available at: <https://www.tutorialspoint.com/nodejs/index.htm>. Accessed 5.11.2021.

W3Schools. [n.d.]. JavaScript JSON. Available: https://www.w3schools.com/js/js_json.asp. Accessed 5.11.2021.

Windmill, E. 2015. Step By Step: Planning a Web Application. Available: <https://medium.com/@ericwindmill/step-by-step-planning-a-web-application-ddaa010a8353>. Accessed 29.11.2021.

APPENDICES

APPENDIX 1. DataGrip connection window for Azure SQL



APPENDIX 2. Admin registration via API in Postman

The screenshot displays the Postman interface with a workspace named "My Workspace". A collection named "Anon" is visible on the left sidebar. The main area shows a POST request to the endpoint `http://localhost:3000/signup`. The request body is a JSON object with the following fields:

```
1 {
2   "name": "test",
3   "username": "test",
4   "email": "test@test.com",
5   "roles": "ADMIN",
6   "password": "12345"
7 }
```

The response is displayed in the "Body" tab, showing a JSON object with a success message:

```
1 {
2   "message": "User and role registered successfully."
3 }
```

The status bar at the bottom indicates a successful response with a status of 200 OK, a time of 1408 ms, and a size of 353 B. The "Save Response" button is visible.

APPENDIX 3. Database table ‘users’ after registration of each role

[illegible]

APPENDIX 4. Roles table

The screenshot displays the Azure Data Studio interface with the 'roles' table selected in the 'testing-database' database. The table structure and data are as follows:

id	name	created_at	updated_at
1	USER	2021-09-07 11:05:56.5920000 +00:00	2021-09-07 11:05:56.5920000 +00:00
2	ADMIN	2021-09-07 11:05:56.5940000 +00:00	2021-09-07 11:05:56.5940000 +00:00
3	FW	2021-09-07 11:05:56.5940000 +00:00	2021-09-07 11:05:56.5940000 +00:00

The bottom panel shows the following log entries:

```
[2021-09-07 14:26:19] Connected
[2021-09-07 14:26:19] use (testing-database)
[2021-09-07 14:26:19] (conn113761) Changed database context to 'testing-database'.
[2021-09-07 14:26:19] completed in 48 ms
[2021-09-07 14:26:19] SELECT TOP 50 t.*
FROM (testing-database).dbo.roles t
[2021-09-07 14:26:19] 5 rows retrieved starting from 1 in 99 ms (execution: 47 ms, fetching: 32 ms)
[2021-09-07 14:26:19] SELECT TOP 50 t.*
FROM (testing-database).dbo.roles t
[2021-09-07 14:26:19] 5 rows retrieved starting from 1 in 93 ms (execution: 81 ms, fetching: 12 ms)
[2021-09-07 14:26:19] SELECT TOP 50 t.*
FROM (testing-database).dbo.roles t
[2021-09-07 14:26:19] 5 rows retrieved starting from 1 in 79 ms (execution: 68 ms, fetching: 7 ms)
[2021-09-07 14:26:19] SELECT TOP 50 t.*
FROM (testing-database).dbo.roles t
[2021-09-07 14:26:19] 5 rows retrieved starting from 1 in 92 ms (execution: 78 ms, fetching: 14 ms)
```

APPENDIX 5. Database table 'user_roles'

The screenshot displays the Azure Data Studio interface with the 'testing-database' open. The 'user_roles' table is selected in the 'Database' pane on the left. The main pane shows the table's data, which is as follows:

id	createdat	updatedat	roleId	userId
1	2021-09-07 11:40:09.0510000 +08:00	2021-09-07 11:40:09.0510000 +08:00	1	3
2	2021-09-07 11:36:27.7210000 +08:00	2021-09-07 11:36:27.7210000 +08:00	2	1
3	2021-09-07 11:37:44.1510000 +08:00	2021-09-07 11:37:44.1510000 +08:00	3	2

The 'Services' pane at the bottom shows the execution of a query. The query text is:

```
SELECT TOP 500 t.*
FROM (testing-database).dbo.user_roles t
```

The output shows that 3 rows were retrieved, starting from 1 in 110 ms (execution: 80 ms, fetching: 30 ms).

The screenshot displays the Postman application interface. At the top, the 'File Edit View Help' menu bar is visible. Below it, the 'Home Workspaces Reports Explore' navigation bar is present. The main workspace area shows a REST client request setup for a POST method to the endpoint 'http://localhost:3000/signin'. The request body is set to JSON format and contains the following data:

```

1 {
2   "email": "test@test.com",
3   "password": "12345"
4 }

```

The response is displayed below the request, showing a status of 200 OK, a time of 893 ms, and a size of 550 B. The response body is formatted as JSON and contains the following data:


```

1 {
2   "auth": true,
3   "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IWRXVjZ9.eyJpZiI6Im5wcm90YV5lC1leHAiOiJlZ2MzExMDEyNj19.d1sa8NlkaWQpb767fYg19uoocEVr1Jv9AmC-1LccQw",
4   "id": 1,
5   "name": "test",
6   "username": "test",
7   "email": "test@test.com",
8   "roles": [
9     "ADMIN"
10 ]
11 }

```

The interface also includes a sidebar on the left with options like 'Collections', 'Environments', 'Mock Servers', 'Monitors', and 'History'. The bottom status bar shows 'Find and Replace', 'Console', 'Bootcamp', 'Runner', and 'Trash' icons.

APPENDIX 7. Login error handling implemented



Email

This field is required!

Password

This field is required!

Login

APPENDIX 8. IoT Hub-Device List synchronization test in Postman

The screenshot shows the Postman interface with a PUT request configured to `http://localhost:3000/devices`. The request is set to the 'Headers' tab, showing 10 headers. The 'Body' tab is selected, displaying a JSON array of three device objects. The status bar at the bottom indicates a successful response (200 OK) with a time of 2.31s and a size of 6.77 KB.

Request Headers:

Key	Value	Description
Content-Length	<calculated when request is sent>	
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.28.0	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ8.eyJpZCI6IjM5SWiaWF0IjoxNjM0MTM0MD00.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ8.eyJpZCI6IjM5SWiaWF0IjoxNjM0MTM0MD00	
Authorization	SharedAccessSignature sr=Subson-IoTHub.azure-devices.net&sig=jB888U	

Request Body (JSON):

```
1 {
2   {
3     "id": 1,
4     "deviceId": "10521c89b1a0",
5     "devStatus": true,
6     "freeDescription": "Write any extra notes here :)",
7     "createdAt": "2021-09-07T11:54:29.126Z",
8     "updatedAt": "2021-09-07T11:54:29.126Z"
9   },
10  {
11    "id": 2,
12    "deviceId": "10521c89b340",
13    "devStatus": true,
14    "freeDescription": "Write any extra notes here :)",
15    "createdAt": "2021-09-07T11:54:29.126Z",
16    "updatedAt": "2021-09-07T11:54:29.126Z"
17  },
18  {
19    "id": 3,
```

Response: Status: 200 OK, Time: 2.31 s, Size: 6.77 KB, Save Response

APPENDIX 9. Database table ‘devices’ after synchronization done via UI

[illegible]

APPENDIX 10. Devices List page after scroll

10521c89b180
10521c89bd78
petteri4
10521c8570d0
10521c89b118
21330021
petteri3
10521c89b138
petteri1
petteri9
10521c89b11c
10521c89b2dc
10521c89b368
10521c89b30c
10521c89bd70
21330011
petteri6
21330020
10521c89b33c
petteri2
21330018
10521c89b334

Device

Device ID:
10521c8570d0

Notes/Description:
Write any extra notes here :)

Status:
enabled

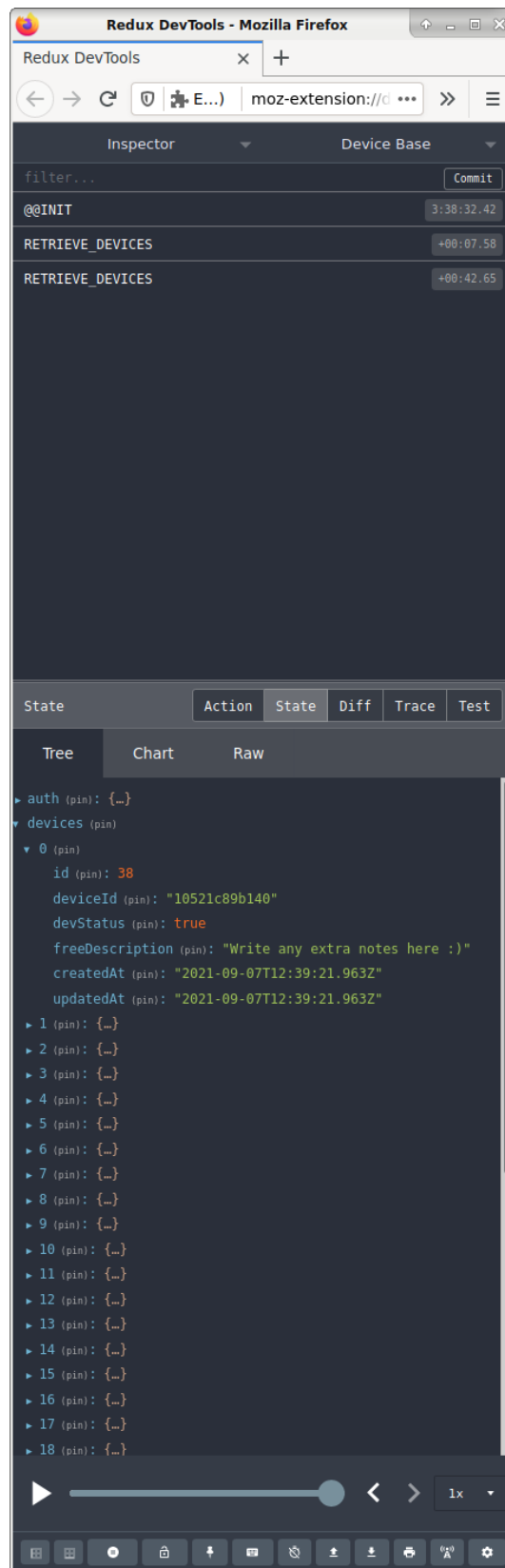
Edit

Synchronize Database With IoT Hub

Delete All Devices

APPENDIX 11. Redux DevTools state view after database query or Hub synchronization

Redux DevTools device synchronization example



APPENDIX 12. Iot Hub synchronization SQL queries

[illegible]

APPENDIX 13. Device page console log showing “initialDeviceState” utilization

The screenshot shows a web browser window with the URL `localhost:8083/device/1`. The page displays the status of a device as "enabled". Below the status, there are three buttons: "Disable" (red), "Delete" (red), and "Update" (green). A message states "The twin was queried successfully." with a "Get Device Twin" button below it. The "Twin:" section shows a JSON object with the following properties:

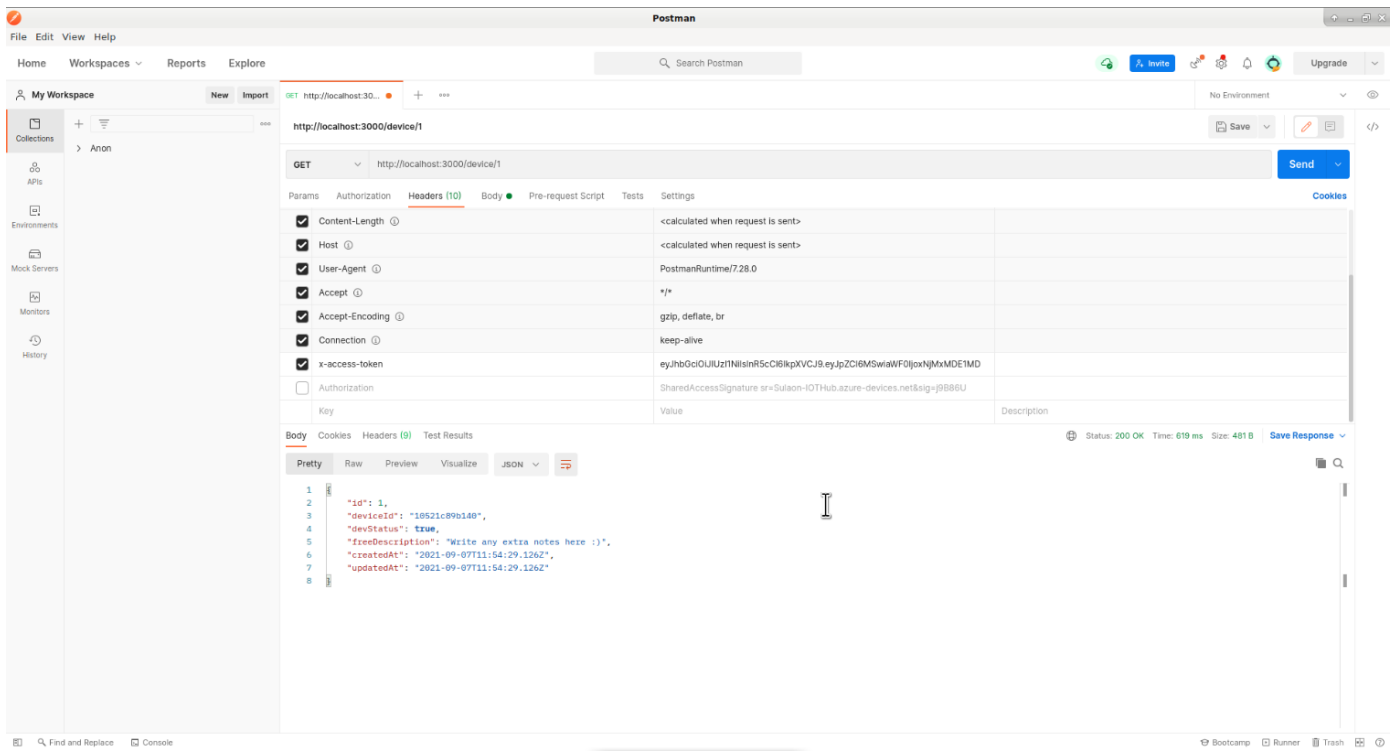
```
{
  "deviceId": "10521c89b140",
  "etag": "AAAAAAAAAA=",
  "deviceEtag": "MTc1MjQyNDc4",
  "status": "enabled",
  "statusUpdateTime": "0001-01-01T00:00:00Z",
  "connectionState": "Disconnected",
  "lastActivityTime": "0001-01-01T00:00:00Z",
  "cloudToDeviceMessageCount": 0,
  "authenticationType": "sas",
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  }
}
```

The bottom of the image shows the Chrome DevTools Console. The "Console" tab is active, displaying the following log:

```
[HMR] Waiting for update signal from WDS...
> Object { id: null, deviceId: "", freeDescription: "", devStatus: true }
1
> Object { id: 1, deviceId: "10521c89b140", freeDescription: "Test update :)", devStatus: true }
> Array [ (-) ]
```

The console also shows the source files for the log entries: `log.js:24`, `Device.js:23`, `Device.js:24`, `Device.js:34`, and `Device.js:87`.

APPENDIX 14. Device page test in Postman



APPENDIX 15. Device description update in ‘devices’ database table after update via UI

The screenshot displays the Azure Database for PostgreSQL console. The top section shows the 'devices' table with columns: id, devId, devStatus, devDescription, createdAt, and updatedAt. The table contains 37 rows, all with a status of 'true' and a description of 'Write any extra notes here'. The bottom section shows the 'Services' tab, which lists various services and their execution times. The services listed are: console (411 ms), devices (410 ms), user_roles (410 ms), roles (410 ms), and users (410 ms). The console output shows the execution of a query to retrieve 37 rows from the 'devices' table, starting from row 1 in 98 ms (execution: 70 ms, fetching: 28 ms).

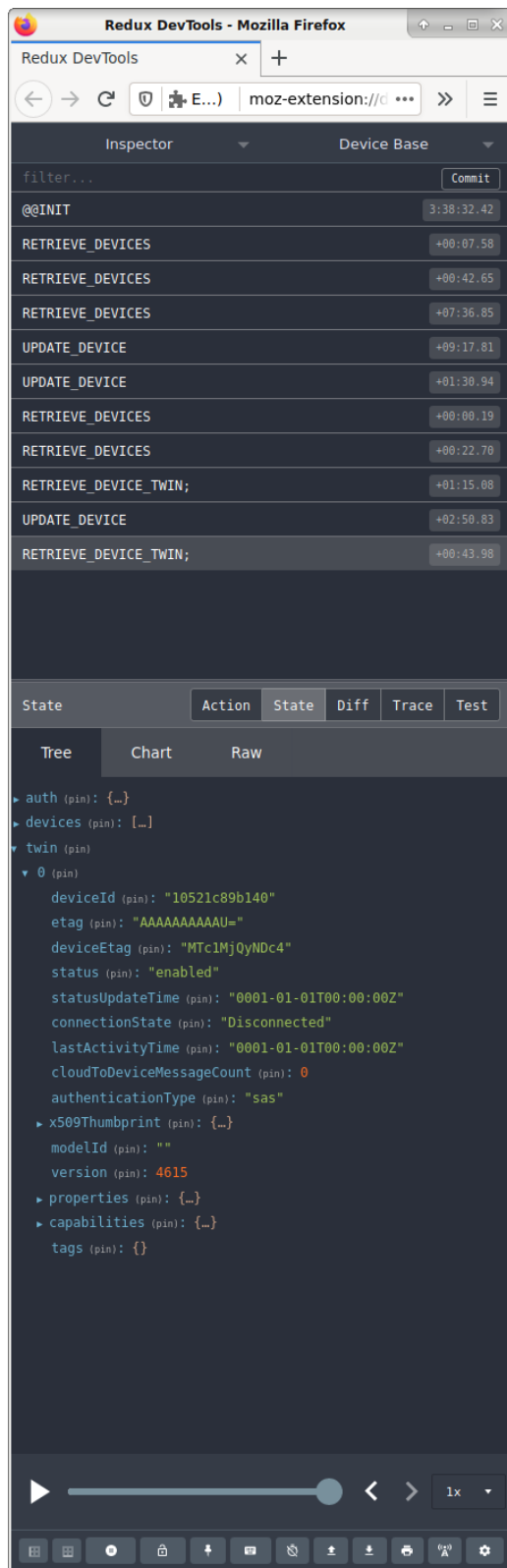
id	devId	devStatus	devDescription	createdAt	updatedAt
1	10521090100	true	Write any extra notes here	2021-09-07 12:19:21.9430000 +00:00	2021-09-07 12:19:21.9430000 +00:00
2	10521090350	true	Write any extra notes here	2021-09-07 12:19:21.9466667 +00:00	2021-09-07 12:19:21.9466667 +00:00
3	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9500000 +00:00	2021-09-07 12:19:21.9500000 +00:00
4	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9533333 +00:00	2021-09-07 12:19:21.9533333 +00:00
5	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9566667 +00:00	2021-09-07 12:19:21.9566667 +00:00
6	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9600000 +00:00	2021-09-07 12:19:21.9600000 +00:00
7	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9633333 +00:00	2021-09-07 12:19:21.9633333 +00:00
8	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9666667 +00:00	2021-09-07 12:19:21.9666667 +00:00
9	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9700000 +00:00	2021-09-07 12:19:21.9700000 +00:00
10	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9733333 +00:00	2021-09-07 12:19:21.9733333 +00:00
11	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9766667 +00:00	2021-09-07 12:19:21.9766667 +00:00
12	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9800000 +00:00	2021-09-07 12:19:21.9800000 +00:00
13	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9833333 +00:00	2021-09-07 12:19:21.9833333 +00:00
14	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9866667 +00:00	2021-09-07 12:19:21.9866667 +00:00
15	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9900000 +00:00	2021-09-07 12:19:21.9900000 +00:00
16	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9933333 +00:00	2021-09-07 12:19:21.9933333 +00:00
17	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9966667 +00:00	2021-09-07 12:19:21.9966667 +00:00
18	10521090360	true	Write any extra notes here	2021-09-07 12:19:21.9999999 +00:00	2021-09-07 12:19:21.9999999 +00:00
19	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0000000 +00:00	2021-09-07 12:19:22.0000000 +00:00
20	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0033333 +00:00	2021-09-07 12:19:22.0033333 +00:00
21	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0066667 +00:00	2021-09-07 12:19:22.0066667 +00:00
22	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0100000 +00:00	2021-09-07 12:19:22.0100000 +00:00
23	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0133333 +00:00	2021-09-07 12:19:22.0133333 +00:00
24	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0166667 +00:00	2021-09-07 12:19:22.0166667 +00:00
25	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0200000 +00:00	2021-09-07 12:19:22.0200000 +00:00
26	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0233333 +00:00	2021-09-07 12:19:22.0233333 +00:00
27	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0266667 +00:00	2021-09-07 12:19:22.0266667 +00:00
28	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0300000 +00:00	2021-09-07 12:19:22.0300000 +00:00
29	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0333333 +00:00	2021-09-07 12:19:22.0333333 +00:00
30	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0366667 +00:00	2021-09-07 12:19:22.0366667 +00:00
31	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0400000 +00:00	2021-09-07 12:19:22.0400000 +00:00
32	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0433333 +00:00	2021-09-07 12:19:22.0433333 +00:00
33	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0466667 +00:00	2021-09-07 12:19:22.0466667 +00:00
34	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0500000 +00:00	2021-09-07 12:19:22.0500000 +00:00
35	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0533333 +00:00	2021-09-07 12:19:22.0533333 +00:00
36	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0566667 +00:00	2021-09-07 12:19:22.0566667 +00:00
37	10521090360	true	Write any extra notes here	2021-09-07 12:19:22.0600000 +00:00	2021-09-07 12:19:22.0600000 +00:00

Services

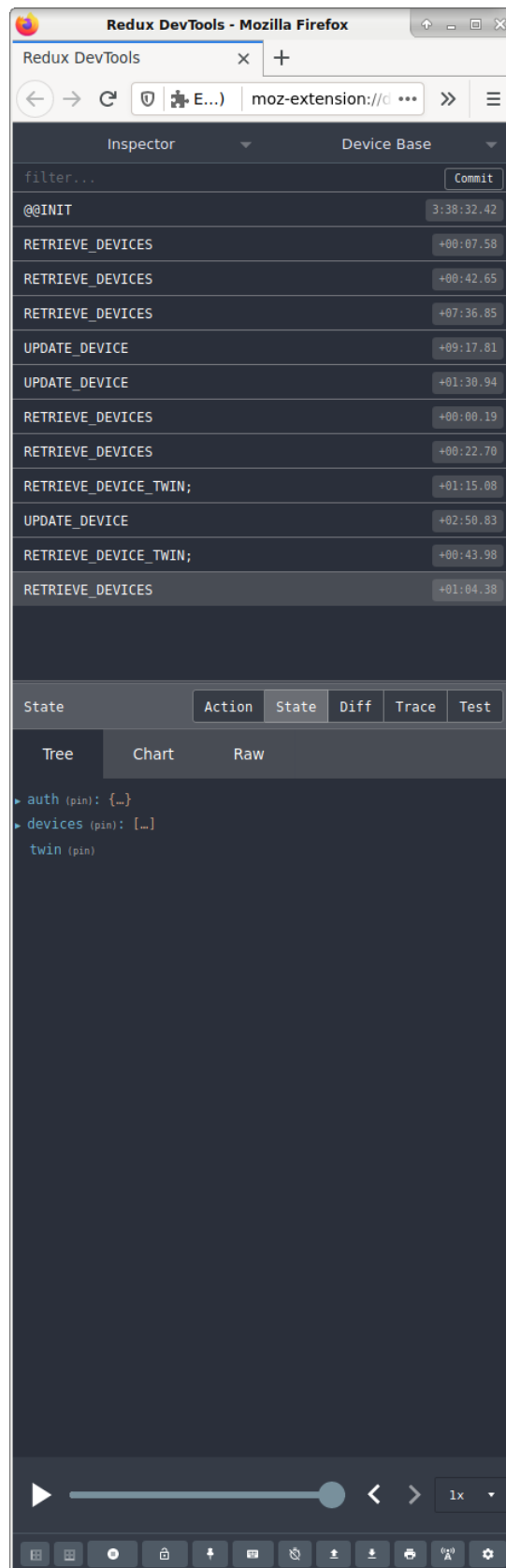
- console 411 ms
- devices 410 ms
- user_roles 410 ms
- roles 410 ms
- users 410 ms

37 rows retrieved starting from 1 in 98 ms (execution: 70 ms, fetching: 28 ms)

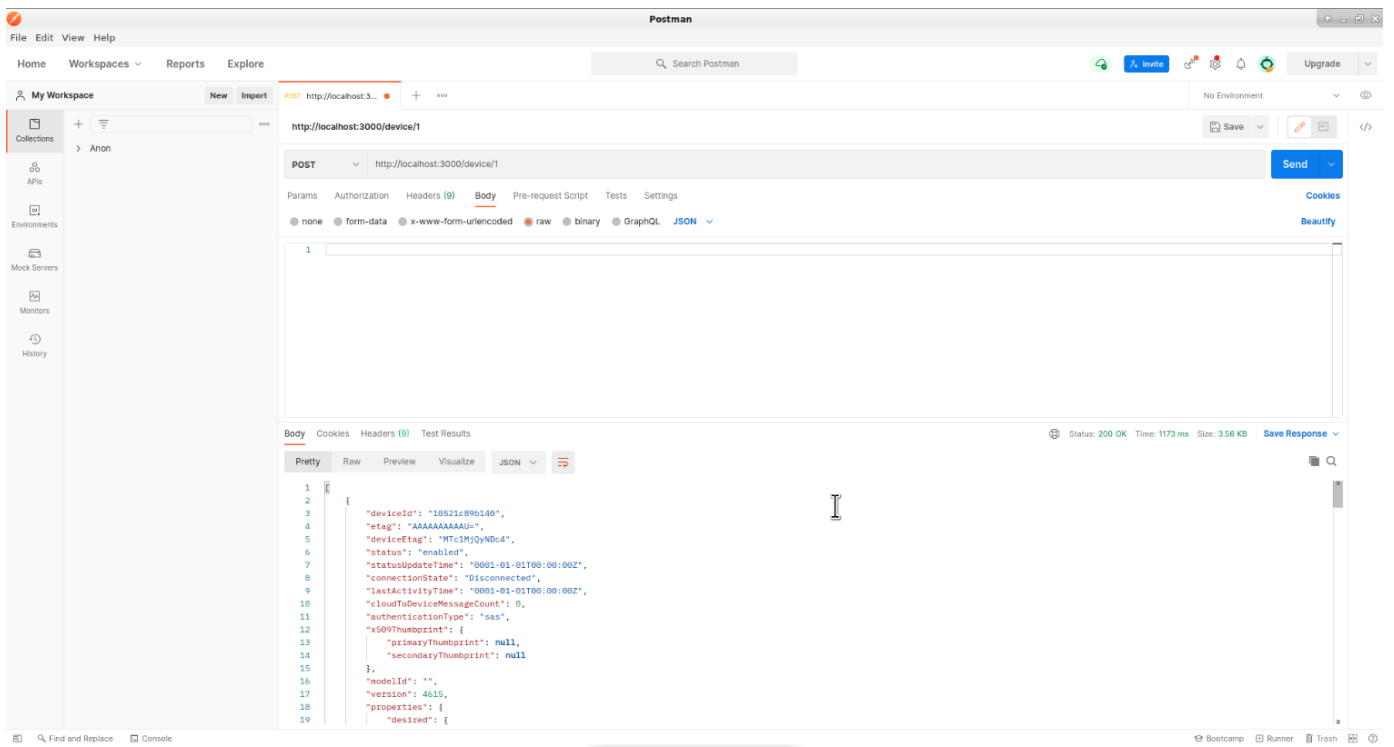
APPENDIX 16. Twin state after query in Redux DevTools



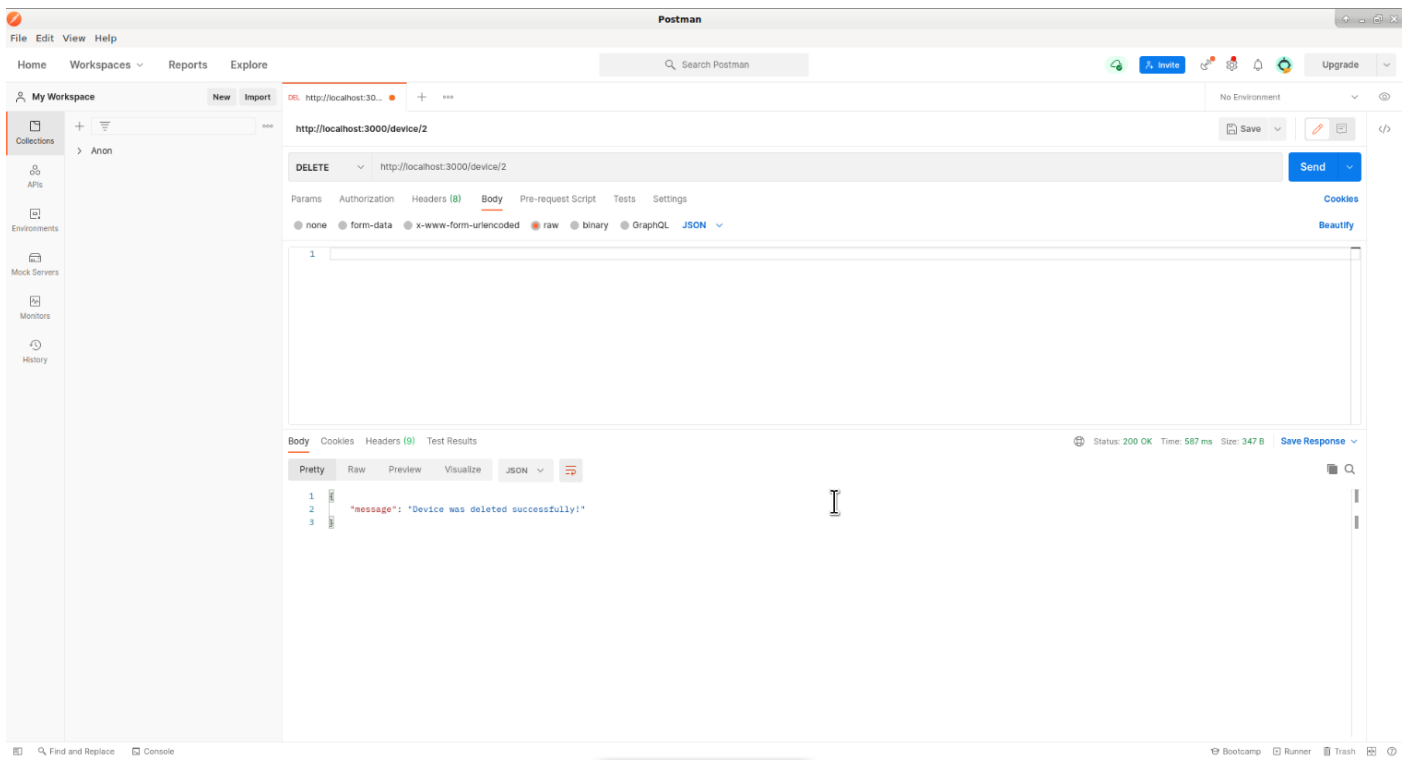
APPENDIX 17. Twin state after navigation away from Device page



APPENDIX 18. Device twin query test in Postman



APPENDIX 19. Device delete test in Postman



APPENDIX 20. Reflected deletion of five devices from 'devices' table

The screenshot displays the SQL Server Enterprise Manager interface. The left pane shows the 'testing-database' database with the 'devices' table selected. The right pane shows the 'devices' table structure with columns: id, deviceid, devstatus, devdescription, createtime, and updateat. The table contains 33 rows of data, all with 'devstatus' set to 'true' and 'devdescription' set to 'Write any extra notes here.'.

The bottom pane shows the 'Query History' window, which displays the execution of a query that deleted five devices from the 'devices' table. The query is as follows:

```
DELETE FROM [testing-database].[dbo].[devices] t
WHERE t.id IN (
    SELECT TOP 5 t.id
    FROM [testing-database].[dbo].[devices] t
    ORDER BY t.id
)
```

The query history shows the execution of this query on 2021-09-07 15:17:03.151, with 33 rows retrieved starting from 1 in 101 ms (execution: 78 ms, fetching: 23 ms).

APPENDIX 21: Delete function SQL queries of five devices

```

Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices];
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices] WHERE [devices].[id] = N'38';
1
Executing (default): SELECT [id], [name], [username], [email], [password], [createdAt], [updatedAt] FROM [users] AS [users] WHERE [users].[id] = 1;
1
Executing (default): SELECT [roles].[id], [roles].[name], [roles].[createdAt], [roles].[updatedAt], [user_roles].[createdAt] AS [user_roles.createdAt], [user_roles].[updatedAt] AS [user_roles.updatedAt], [user_roles].[roleId] AS [user_roles.roleId], [user_roles].[userId] AS [user_roles.userId] FROM [roles] AS [rol
+1 INNER JOIN [user_roles] AS [user_roles] ON [roles].[id] = [user_roles].[roleId] AND [user_roles].[userId] = 1;
ADMIN
ADMIN
Executing (default): DELETE FROM [devices] WHERE [id] = N'38'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices];
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices] WHERE [devices].[id] = N'38';
2
Executing (default): SELECT [id], [name], [username], [email], [password], [createdAt], [updatedAt] FROM [users] AS [users] WHERE [users].[id] = 1;
1
Executing (default): SELECT [roles].[id], [roles].[name], [roles].[createdAt], [roles].[updatedAt], [user_roles].[createdAt] AS [user_roles.createdAt], [user_roles].[updatedAt] AS [user_roles.updatedAt], [user_roles].[roleId] AS [user_roles.roleId], [user_roles].[userId] AS [user_roles.userId] FROM [roles] AS [rol
+1 INNER JOIN [user_roles] AS [user_roles] ON [roles].[id] = [user_roles].[roleId] AND [user_roles].[userId] = 1;
ADMIN
ADMIN
Executing (default): DELETE FROM [devices] WHERE [id] = N'38'; SELECT @@ROWCOUNT AS AFFECTEDROWS;

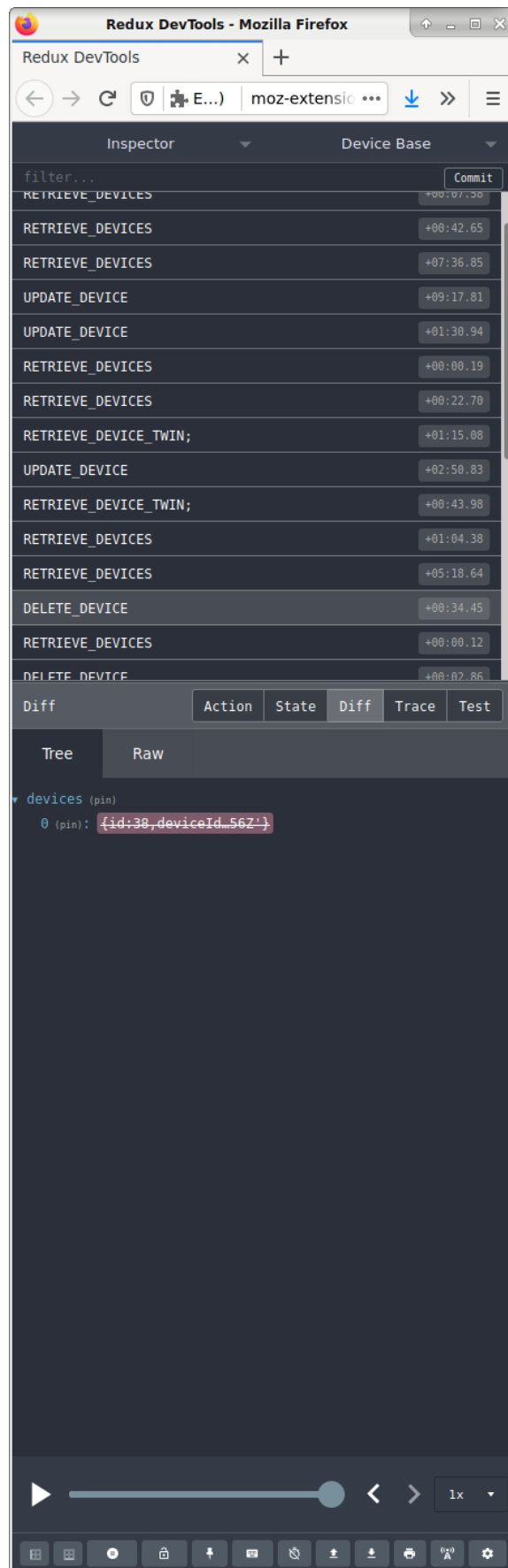
```

```

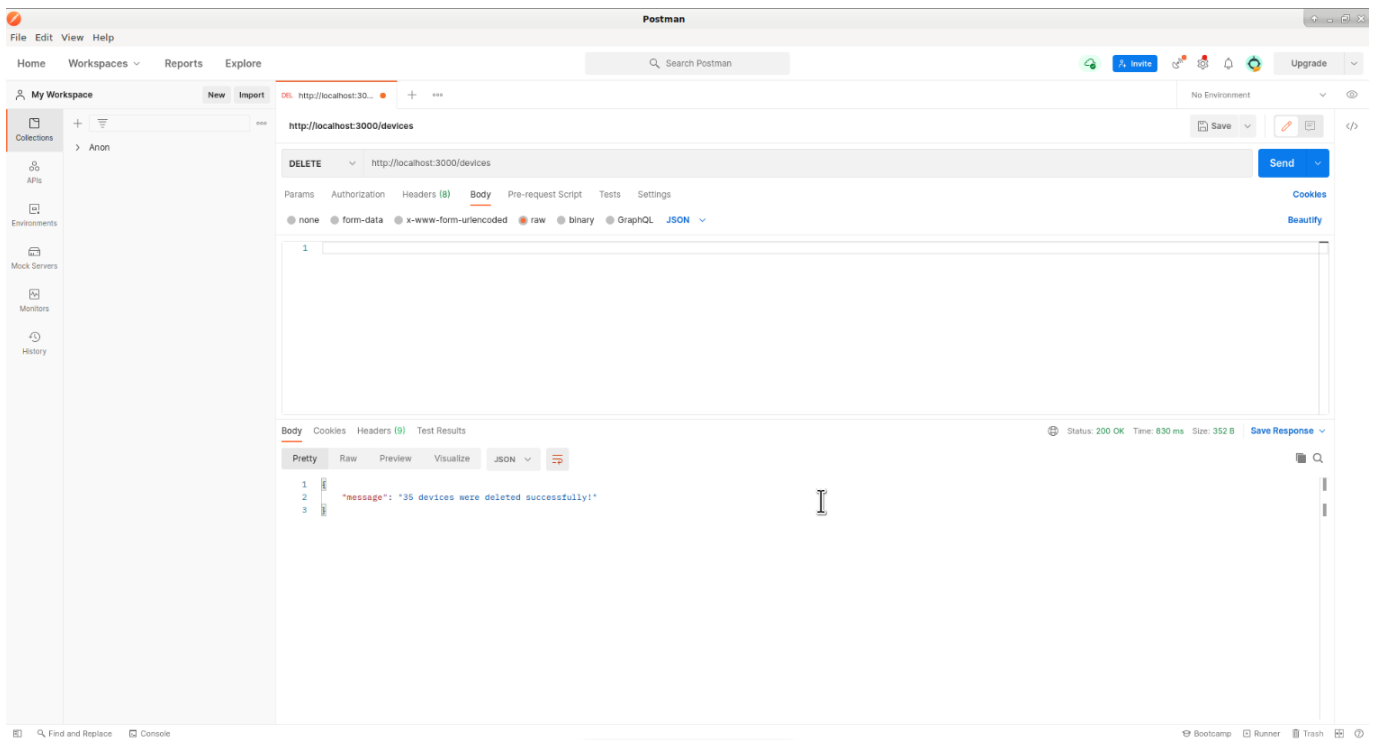
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices];
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices] WHERE [devices].[id] = N'40';
Executing (default): SELECT [id], [name], [username], [email], [password], [createdAt], [updatedAt] FROM [users] AS [users] WHERE [users].[id] = 1;
Executing (default): SELECT [rules].[id], [rules].[name], [rules].[createdAt], [rules].[updatedAt], [user_rules].[createdAt] AS [user_rules.createdAt], [user_rules].[updatedAt] AS [user_rules.updatedAt], [user_rules].[ruleId] AS [user_rules.ruleId], [user_rules].[userId] AS [user_rules.userId] FROM [rules] AS [rul]
    INNER JOIN [user_rules] AS [user_rules] ON [rules].[id] = [user_rules].[ruleId] AND [user_rules].[userId] = 1;
ADMIN
Executing (default): DELETE FROM [devices] WHERE [id] = N'40'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices];
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices] WHERE [devices].[id] = N'41';
Executing (default): SELECT [id], [name], [username], [email], [password], [createdAt], [updatedAt] FROM [users] AS [users] WHERE [users].[id] = 1;
Executing (default): SELECT [rules].[id], [rules].[name], [rules].[createdAt], [rules].[updatedAt], [user_rules].[createdAt] AS [user_rules.createdAt], [user_rules].[updatedAt] AS [user_rules.updatedAt], [user_rules].[ruleId] AS [user_rules.ruleId], [user_rules].[userId] AS [user_rules.userId] FROM [rules] AS [rul]
    INNER JOIN [user_rules] AS [user_rules] ON [rules].[id] = [user_rules].[ruleId] AND [user_rules].[userId] = 1;
ADMIN
Executing (default): DELETE FROM [devices] WHERE [id] = N'41'; SELECT @@ROWCOUNT AS AFFECTEDROWS;
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices];
Executing (default): SELECT [id], [deviceId], [devStatus], [freeDescription], [createdAt], [updatedAt] FROM [devices] AS [devices] WHERE [devices].[id] = N'42';
Executing (default): SELECT [id], [name], [username], [email], [password], [createdAt], [updatedAt] FROM [users] AS [users] WHERE [users].[id] = 1;
Executing (default): SELECT [rules].[id], [rules].[name], [rules].[createdAt], [rules].[updatedAt], [user_rules].[createdAt] AS [user_rules.createdAt], [user_rules].[updatedAt] AS [user_rules.updatedAt], [user_rules].[ruleId] AS [user_rules.ruleId], [user_rules].[userId] AS [user_rules.userId] FROM [rules] AS [rul]
    INNER JOIN [user_rules] AS [user_rules] ON [rules].[id] = [user_rules].[ruleId] AND [user_rules].[userId] = 1;
ADMIN
Executing (default): DELETE FROM [devices] WHERE [id] = N'42'; SELECT @@ROWCOUNT AS AFFECTEDROWS;

```

APPENDIX 22: Reflected state change in Redux DevTools after single device deletion



APPENDIX 23. Delete all devices function test in Postman



APPENDIX 24. Database table 'devices' after removal of all devices via UI

The screenshot displays the SQL Server Enterprise Manager interface. The left pane shows the 'testing-database' database with the 'devices' table selected. The right pane shows the 'devices' table structure with columns: id, deviceid, devstatus, freedescription, createdat, and updatedat. The bottom pane shows the 'Services' list, including 'devices', 'user_roles', 'roles', 'users', and 'users_roles'. The 'devices' service is highlighted, and its status is 'Stopped'.

Database: testing-database\sql database.windows.net | testing-database | dbo | tables | devices

Table: devices

Columns: id, deviceid, devstatus, freedescription, createdat, updatedat

Services:

- testing-database\sql database.windows.net
- console (411 ms)
- devices (403 ms)
- user_roles (439 ms)
- roles (411 ms)
- users (400 ms)

Log:

```
FROM [testing-database].dbo.devices t
[2021-09-07 15:48:11] 37 rows retrieved starting from 1 in 99 ms (execution: 73 ms, fetching: 26 ms)
SELECT TOP 501 t.*
FROM [testing-database].dbo.devices t
[2021-09-07 15:56:50] 37 rows retrieved starting from 1 in 96 ms (execution: 70 ms, fetching: 26 ms)
SELECT TOP 501 t.*
FROM [testing-database].dbo.devices t
[2021-09-07 15:57:59] 37 rows retrieved starting from 1 in 100 ms (execution: 74 ms, fetching: 26 ms)
SELECT TOP 501 t.*
FROM [testing-database].dbo.devices t
[2021-09-07 16:02:19] 37 rows retrieved starting from 1 in 101 ms (execution: 75 ms, fetching: 26 ms)
SELECT TOP 501 t.*
FROM [testing-database].dbo.devices t
[2021-09-07 16:11:23] 32 rows retrieved starting from 1 in 96 ms (execution: 74 ms, fetching: 22 ms)
SELECT TOP 501 t.*
FROM [testing-database].dbo.devices t
[2021-09-07 16:14:56] 0 rows retrieved in 88 ms (execution: 76 ms, fetching: 12 ms)
```


APPENDIX 25. Deletion of all devices from the state reflected in Redux DevTools

