Haaga-Helia
University of Applied Sciences

Development of a serverless web-application for large scale IoT platform administration

Yulia Ageeva

Bachelor's Thesis
Degree Programme in Business Information Technology
2019

Haaga-Helia
University of Applied Sciences

Abstract

13/05/2019

Author(s)

Yulia Ageeva

Degree programme

Business Information Technology

Report/thesis title	**Number of pages and appendix pages**
Development of a serverless web-application for large scale IoT platform administration	**40**

The main goal of the thesis was to study serverless technologies and gain practical experience through development of a web application for a Finnish company called Helvar. The result of the thesis is a functioning web application created for the administrators of the company's IoT platform to register new building sites, view and modify the existing sites, manage user access rights and grant or revoke user permissions to specific sites.

Theoretical framework was composed focusing on learning about cloud technologies proposed by the commissioning party, such as AWS Lambda, Cognito, DynamoDB, API Gateway, S3 and CloudFormation. Node.js was used as the programming langauge for the backend implementation and React.js for the frontend implementation. Theoretical study was followed by the design and planning stage for the application which consisted of UI design, REST API design and finally the overall architecture of the application including the backend. Final stage was the implemenation of the application based on the planned architecture.

In addition, the serverless technolgies for running a web application were explored and identified as very beneficial for applications which are sporadically used but require high availability.

As part of this work, agile methodlogies for software development activities were explored as well and Kanban was selected as the development methodology, resulting in efficient development prcoess and on time development of the application.

Finally, the thesis concludes by discussing the challenges faced during the development of the web application, the recommendations for further development of the application and a review of the development process.

Keywords

AWS, web application, serverless, Node.js, React, cloud architecture

Table of contents

Terms and Abbreviations

Amazon S3	Amazon Simple Storage Service
API	Application Programming Interface
AWS	Amazon Web Services
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheet
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IoT	Internet of Things
I/O	Input/Output
JS	JavaScript
JSON	JavaScript Object Notation
npm	Node Package Manager
REST	Representational State Transfer
SASS	Syntactically Awesome Style Sheets
SDK	Software Development Kit
SSD	Solid State Disk
SQL	Structured Query Language
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
UUID	Universally unique identifier
WIP	Work In Progress
XML	eXtensible Markup Language

1 Introduction

Nowadays companies try to automate as many manual processes as possible and fortunately there are plenty of tools and ways to do it. Using serverless computing gives the opportunity to scale on demand, control costs and enable faster development times along with removing the necessity of planning for and procuring servers and other IT infrastructure beforehand.

The commissioning party of this project is a Finnish software & electronics manufacturing company called Helvar. It specializes in providing commercial lighting control solutions and digital services for smart buildings. To be able to provide the digital services and capabilities to leverage data to provide better experience to its customers, the company has developed a cloud based IoT platform known as Lighting Intelligence Platform.

While the platform can serve multitudes of connected building across the globe, the administration of this platform is a manual process at Helvar which can be error prone and does not scale with the growing demand for the platform. To solve the problem of manual administration tasks and streamlining the process of registering new building sites, users and giving them appropriate access rights, the company was looking for developing a web application through which platform administrators can carry out the aforementioned tasks. Their proposition was to use Amazon Web Services for developing a serverless backend and a REST API which the front-end can utilize to interact with the backend. React.js was proposed as the technology for developing the front-end of the application.

The reason why this project was chosen by the author is own interest in learning new technologies, gaining experience in applying theoretical knowledge and own practices to the web application implementation as well as understanding the software development life cycle in a professional environment.

This chapter includes presentation of the commissioning party, explanation of project's objectives and project's process description.

1.1 Presentation of commissioning party

Helvar is a Finnish company which was founded in 1921, focusing on the lightning components and systems since 1970's. The headquarters are located in Espoo, Finland and company has global competence centre for the lighting control systems business in Dartford, United Kingdom. Helvar employs around 272 people and the turnover was about 76 million euros in 2016. (Helvar, 2019).

Helvar offers various lightning solutions to the customers, such as lighting management solutions, programmable solutions, room solutions, simple switching solutions, luminaire based solutions and installation of lightning control. (Helvar, 2019).

The thesis project has been started to resolve following issues in the company, turn existing manual process in faster and less error prone automated one, which has been resulting so far in lower customer satisfaction and also improve security issues regarding data storage. The manual process is slower to do and becomes a bottleneck and requires the information to be entered and changed in many different places.

The information relates to building data which should be entered into the platform before the site can be connected to the IoT platform, users in the platform then should be assigned access to the created site so they can access the gathered data and analytics, API keys need to be created in the platform which allow end users or developers to access the information from sites and access policy for each API key needs to be maintained as well. The application being developed as part of the work aims to provide means of accomplishing the above-mentioned actions from a single point via an easy to use web application.

1.2 The objectives

The objectives of this project are to gain deeper knowledge of React and Node JS technologies, as well as services provided by AWS through professional development and delivery of the web application corresponding to the commissioning party's requirements. The target of the project is to deliver a working web application to the company together with documentation regarding the source code.

The following concrete, measurable results would be created in the project: user guide, software components.

The aim of the project is to learn about cloud services and serverless technology based on the services provided by AWS, as well as how to utilize them together, in addition writing source code in Node.js and React.js are going to improve programming skills of the writer of the thesis.

1.3 Scope

The scope of the project includes studying theoretical framework, developing the web application according to the commissioning party's requirements. Within the scope of web application, front-end UI, backend implementation and developing a RESTful API are encompassed.

The operation and functionality of the commissioning party's IoT platform itself is not in the scope of this thesis.

1.4 Process description

The process of writing the thesis was split into two main parts. After acquiring requirements for the web application from the commissioning party, the first part included reading documentation about technologies required for implementing the project, concentrating especially on previously unknown technologies. The second part consisted of planning the development and implementation process and then carrying it out.

The first part of the process was carried through literature review, various literature resources, online resources in form of articles, tutorials and documentation were gathered and reviewed.

The implementation part of the project was carried out using agile development methodology, as the work is carried out by the author alone, Kanban methodology was determined to be the most suitable option. Kanban method makes it easier to track the to-do, done and in-progress items, it also provides the ability to prioritize the most important tasks and a mechanism for communicating the exact state of development progress to relevant stakeholders. The tool called Trello was used to manage the implementation items and maintain the Kanban board. The tool was used to assist the writer of the thesis in keeping the progress of tasks to-do and also tracking the ongoing items, as well as monitoring the order of the items that needed completion before the other tasks could be started.

The outcome of the project is the tool for the commissioning party that would automate current manual work and improve security issues, the final web application would be a result of different components working together smoothly and error-free. In addition, implementation of the project would result in gaining the knowledge and experience of the real-life processes related to the development of a web application.

2 Theoretical framework

This chapter would be focusing on the introduction of services and technologies that would be used in the project. All of them were proposed by the commissioning party and many of them are extensively used in other different company's projects.

2.1 Serverless Architecture

Serverless architecture also known as serverless computing allows the application to be built and run in a such way that the third-party provider takes care of infrastructure management. The concept of virtual servers has already been known since the early 2000s, but the main breakthrough in cloud computing happened with the launch of Amazon's EC2 in 2006. (Gurturk 2017).

Currently many known IT companies are trying to offer own cloud services, but the main players on the market are Amazon, Microsoft and Google. (Roberts 2018). Therefore, such a time-consuming task as scaling, provisioning, managing and patching of the servers is taken over by a service provider, freeing up resources and concentrate on solving the business problem. (Sbarski 2017). Payment for the serverless services is calculated according to the consumed compute time.

2.2 Amazon Web Services

All services provided by Amazon focus on simplicity and robustness. AWS cloud services are usually integrated with other internal services, which makes it easier to use. The benefits of serverless computing are:

- No server management – when the application is deployed to the server, it would require such resources as CPU, memory and hard disk even though it is not running. Furthermore, those resources should be provisioned and planned, as well as required software and hardware installed. (Rady 2016, 34). With serverless there is no necessity to have person in charge to maintain servers, the development process is much less time consuming.
- Flexible scaling – Serverless computing scales applications automatically and doesn't require to precisely estimate traffic demand in advance. Necessary changes could be done very efficiently and quickly in the AWS console. (Rady 2016, 34). Such flexibility provides optimized availability and costs.

- No idle capacity – At the time when the code is not running, there is no charge, so the total costs for running an application is much lower in comparison to the application running on own servers.
- High availability – No server management and flexible scaling make sure that the application stays available at any time.

2.2.1 AWS Lambda

Lambda is a compute service and the main component of serverless architecture. It would execute provided function code on demand from few requests a day to thousands per second, a function could be invoked multiple times simultaneously. Once again it doesn't need any servers' scaling or provisioning. Code is run virtually for any type of web application or backend service. AWS Lambda supports following languages through the use of runtimes: Node.js, Python, Ruby, Java, Go, .NET, for other languages a custom runtime could be implemented. (Amazon Web Services, 2019).

The steps to run the code in Lambda typically includes authoring code, then uploading code packages to AWS Lambda and then monitoring and troubleshooting. According to AWS Lambda documentation (2019), regardless of the chosen programming language the code written for a Lambda function should adhere to the following AWS Lambda 's core concepts:

- Handler - is a method that would be called by AWS Lambda to start the execution of Lambda function. It could be thought of as main method in regular applications. The event data which causes the trigger is passed as the first parameter and processed by the handler. The second parameter is context and third optional parameter is call-back.
- Context – is a second parameter that is passed to the handler as a context object. Information about properties available via a context object, such as remaining time for the execution of Lambda function, request ID, function version and etc.
- Event – is a JSON object which is passed to the handler object and contains the information about the source and the event that triggered the Lambda function.
- Logging – logging statements could be used in Lambda to utilize another service by AWS, CloudWatch Logs.
- Exceptions – an error parameter could be passed to call-back function to check the information about failed execution.

- Concurrency – concurrent executions are restricted across all functions in one region to 1000. Concurrency usage could be monitored through Amazon CloudWatch.

As mentioned before, the payment is calculated based on the number of requests, time of execution and the amount of memory allocated, therefore to keep it cheap the code running in Lambda should be quickly executable. By using Lambda, the company can concentrate on spending more time to create more features for the users which would result in better user experience. (Sbarski 2017).

2.2.2 AWS Cognito

Amazon Cognito is the service for authentication and authorization to web and mobile applications. The registration and login system can be implemented with Cognito entirely, but it also can be integrated to authenticate users through public identity providers or using already existing authentication process. (Sbarski 2017). Authentication with external identity providers require support of Security Assertion Markup Language(SAML) or OpenID Connect. Cognito service is serverless and doesn't require users' database management.

Amazon Cognito has two main features user pool and identity pool, which also can be used together for authentication and user access granting. User pool is a directory of stored signed-up and signed-in users. It also supports integration with social identity providers such as Facebook, Twitter and Amazon. Identity pools allow guest users to receive temporary AWS credentials and authenticate users who obtained a token to access other AWS services, for example DynamoDB and S3. (Amazon Cognito 2019). To access different AWS services and resources the IAM role with specified rules could be chosen for each user or group of users. (Sbarski 2017).

2.2.3 AWS DynamoDB

Amazon DynamoDB is a NoSQL database that is fully managed and easy to scale. The main purpose of using it is to concentrate on the application itself eliminating administrative work of operating a distributed NoSQL database. With Amazon's DynamoDB there is no need for hardware provisioning, software patching, setup and configuration of NoSQl database. (Baron, Baz, Bixler, Gaut, Kelly, Senior & Stamper 2017, 177). All data tables created are stored on solid state disks (SSDs) and are replicated in across multiple Availability zones withing an AWS Region for durability protections and hign-availability. (Niranjanamurthy, Archana, Niveditha, Jafar & Shravan 2014). Amazon CloudWatch is used for

monitoring database's capacity and optimize scaling capabilities. (Amazon CloudWatch 2019).

Main benefits of using DynamoDB are high level of performance, simple scale out scheme and enhanced reliability. Tables created with DynamoDb could be scaled to store a virtually unlimited number of items with steady low-latency execution. The security over the database is controlled with access rights and permissions for users and administrators. The IAM service is integrated in DynamoDB to restrict access to items and actions on tables through permissions using policies. (Baron, Baz, Bixler, Gaut, Kelly, Senior & Stamper 2017, 177-187).

2.2.4 AWS API Gateway

Amazon API Gateway is a service to create scalable and secure REST APIs and WebSockets to access AWS or other web services, and data stored in AWS Cloud. API Gateway could be presented as interface between the client applications (web, mobile, desktop) and AWS services and websites. Lambda functions are integrated within the API Gateway to allow more secure or private communication with services. (Sbarski 2019). API Gateway gives an option to enable cross-origin resource sharing (CORS) support, which depends on the http requests the created API gets. It would require additional setup of the required headers to support CORS. (Amazon Web Services 2019)

REST APIs created in API Gateway execute standard HTTP methods like GET, PUT, POST, DELETE and PATCH. WebSocket APIs created in API Gateway implement a stateful communication between frontend and AWS services or an HTTP endpoint. (Amazon Web Services 2019).

2.2.5 AWS S3

Amazon Simple Storage Service (Amazon S3) is one of the earliest services introduced by Amazon and it provides a scalable cloud storage that virtually protects, stores and retrieves unlimited amount of data for different use cases such as websites, mobile applications, nearline archive, disaster recovery, backup and recovery, IoT devices and big data analytics. (Amazon Web Services 2019). S3 allows organizations to store any number of objects and access them via API reachable over HTTP-based protocols. To work with S3 Management Console, the command-line interface, SDK wrapper are used, as well as third party tools. (Wittig 2016). Amazon S3 provides REST based minimalistic API –create/delete bucket, read/write/delete objects, responses are formatted in JSON. (Amazon S3, 2019).

The main Amazon S3 Concepts include:

- Buckets-a container for storing the object
- Objects – is a combination of some metadata and the data itself. (Wittig 2016).
- Keys –object's global unique identifier
- Versioning- keeps multiple versions of an object in the same bucket
- Regions – the geographical region where the created bucket would be stored
- Amazon S3 Data Consistency Model – meaning that PUTS for new objects are first checked for existence with key name, then written in an S3 bucket and then are available for read requests. (Baron, Baz, Bixler, Gaut, Kelly, Senior & Stamper 2017, 21-30).

Default bucket encryption provides server-side or client-side encryption for all objects stored in the bucket. Amazon S3 provides different levels of policy securities for common operations on the bucket, individual objects or subset of objects. Once again IAM service could be used in S3 as well to control access that user or user groups have on different elements/components of S3 bucket. (Amazon Web Services 2019).
S3 has range of storage classes depending on the frequency of data accessibility, such as S3 Standard for frequently accessed data and Amazon Glacier for rarely accessed data. (Baron, Baz, Bixler, Gaut, Kelly, Senior & Stamper 2017, 38).

2.3 JS components(React and Node)

In the project Javascript library React would be used for building the front end of the application and Node.js for the backend. This chapter shortly describes the main concepts of React and Node.js.

2.3.1 React

React has been one of the most popular JavaScript libraries nowadays. It is developed by Facebook and released in 2013. It is great for handling the view layer for mobile and web applications, since it allows to divide UI into components.

React relies on 4 concepts such as:

- Components –is an isolated chunk of UI and can be defined as a function or class component.
- JSX(JavaScript XML) allows to write HTML tags inside JavaScript. It doesn't necessarily have to be used, but using JSX makes React simpler and easier to read.

- State – is a concept and represents a lifecycle of React component.
- Props –stands for properties and is similar to global object or variable, which would be passed as an argument to a React component.

React-DOM(Document Object Model) is a package provided by React to let the developer efficiently manipulate DOM elements of the web page. Virtual DOM is another concept presented in React and it represents a copy of an actual DOM and allows a browser to use less resources as a response to the change done a page. When there is a change to a component, meaning that the state of the component has changed, Virtual DOM would be updated and batched updates would be sent to the actual DOM for repainting or re-rendering the changes at once. (Copes 2019).

One of the most import methods in React-dom package is render function - ReactDOM.render(). It takes up to three parameters: element, container and callback (being an optional parameter) and examines this.props and this.state. The render function returns either a reference to a component or null for stateless component. (React.js Facebook Inc. 2019).

When React application is built there are few options provided for styling the application, such styling with CSS, SASS/SCSS, bootstrap or by using different tools or third party React components. Material UI is an open source project that provide React components to implement Google Material Design. It was chosen for the project implementation due to popularity, therefore it has well-tested components and a lot of documentation about implementation of different elements in to the application. (Material-UI Documentation 2019).

2.3.2 Node.js

Node.js is an open source platform and a runtime environment for JavaScript. Node.js has many benefits, one of which is being memory efficient. Applications created with Node.js are executed as single threaded application, but could also handle multithreaded tasks by none blocking I/O events. (Node.js Foundation 2019).

Node.js is one of the languages natively supported by AWS Lambda. It doesn't require compilation of the source code, Node.js application could be zipped, uploaded and run on lambda along with all the required modules. Being one of the most popular languages, it enables wide support resources, which makes troubleshooting easier. Logically when the front-end is developed in the same programming language as a backend, in given case JavaScript, it is easier to maintain and do handover to other development team.

2.4 Cloud formation

CloudFormation is a service provided by Amazon Web Services which supports the infrastructure as code paradigm and enables provisioning and configuration of all required services and components. It allows the users to model their required infrastructure in simple text files which can easily and securely be deployed to the AWS to get all the required services for the application up and running. The ability to model one's infrastructure in a simple text files helps in meeting organizational compliance, standardization & easier problem solving. (AWS CloudFormation 2019).

CloudFormation service also enables the possibility to automate deployment and support continuous integration and continuous deployment activities. It allows for packaging infrastructure of an application together with its source code which allows software developers to maintain, review and version control the infrastructure just as they would with the source code. (AWS CloudFormation 2019).

The service also provides the rollback functionality i.e. in case of an error during deployment, it can roll back the applied changes to bring the infrastructure back to the pre-deployment stage. CloudFormation supports YAML and JSON as the notations which can be used for writing CloudFormation files, the service also provides ready-made templates which include the most commonly used services and configuration patterns. The files can be uploaded to AWS S3 storage buckets which can then be utilized from the AWS web console, API or command line tools for deployment. (AWS CloudFormation 2019).

2.5 Kanban Development Methodology

Kanban is a methodology that evolves around process visualization and helps to identify bottlenecks and optimizing the workflow. Kanban as term comes from Japan and stands for visual sign or card. It comes under Lean method, which was initially used in manufacturing process and originated within Toyota Production System. Even though Kanban was initially used on a factory floor, its' principals are widely applied in software companies. (Wester 2016).

The main properties of Kanban include:

- Visualize the workflow – the workflow process is broken down into steps, and each step is represented with a column name on a special board and each task or item is written on a card at the initial stage and then placed to the board in accordance to its current state. The more complicated the process is the more columns there would be.

- Limit Work In Progress (WIP) –the number of items in progress at each step of the process are limited and the new item is added to the stage only when there is space for it.
- Manage and Improve Flow – identify the problem areas where the flow is interrupted, analyse the issues and apply the improvements to keep the flow smooth. (Klipp 2019).

There are various software tools available on the market to support Kanban methodology, additionally it could be used just on white board in any part of the office. Kanban is adopted by many agile teams and provides a good amount of flexibility for planning and reprioritizing work items in the backlog in comparability to fixed-length iterations in scrum. (Atlassian 2019).

The other benefit of Kanban is having cycle time as one of the workflow metrics, measuring the average to complete one task, which allows to optimize the process by predicting how much time similar task could take in the future. (Atlassian 2019).

As there could be only specific number of tasks at one stage, there are fewer bottlenecks in the workflow and they are easier to determine and faster to solve. Applying and analysing various metrics is another component of Kanban, which improves team's efficiency and effectiveness, using various Kanban software tools makes it even easier to generate and monitor. (Tutorials Point 2016).

3 Design and planning

This chapter describes the steps followed to plan the development of the web application, concentrating on creation of the overall architecture, designing the APIs and creating basic user workflows for front-end screens. Prior to starting the implementation of this step of the web application all the related tasks were recorded in to-do column of Kanban software application and prioritized accordingly.

3.1 Overall architecture

The figure (Figure 1) below depicts the overall application architecture and shows the interaction between the various components and services involved.

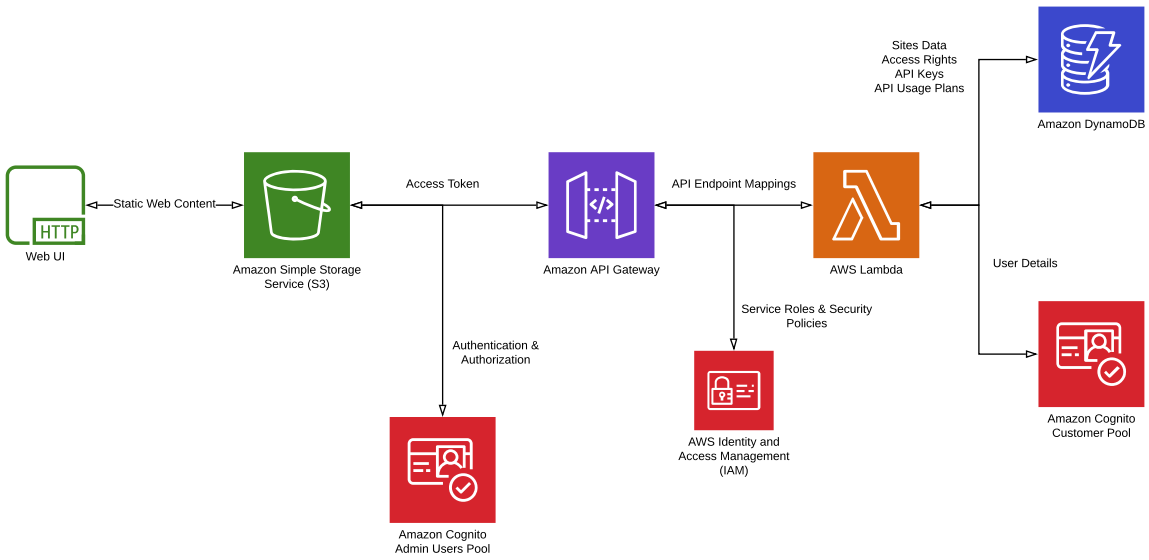

Figure 1. Overall Application Architecture

The front-end of the application is hosted in an Amazon Web Services S3 bucket which is configured to serve static web content, this allows the application to be served out to end-users via HTTP/HTTPS without requiring a web-server. In essence, a web-server is still involved but its leveraged from the web hosting capability of Amazon Storage Service S3 extending the serverless nature of the application to frontend as well.

The dynamic content shown to the end user in the frontend and the actions admin users can carry out such as creating new sites or assigning access rights to users are handled via RESTful API calls to the API hosted in Amazon API Gateway service. All API endpoints are defined in the API gateway service with mappings to their corresponding lambda functions. The access to API Gateway is restricted and requires a valid security token to be present for each call to the API.

Authentication and authorization in the web application are handled via the Amazon Cognito service which is integrated to the frontend for access control, the service is also utilised for generating the security tokens which are used for API access.

Lambda service shown in the architecture provides the computing resources required in the backend, each API endpoint is mapped with lambda function which is called in an event driven manner as the users access the corresponding end points. All lambda functions in the application are written in node.js and interface with the services on customer's platform to modify database tables or user data in Cognito service.

AWS identity and access management service is used for managing service roles in the application architecture and each component shown in the figure has a corresponding role with an access policy attached. This access policy determines which other services and resources the component can interact with and what actions it can perform.

The database used in the application is called Amazon Dynamo DB, it is a managed no-SQL database which the application interacts to create new records or modify existing entries. The development of the database is not in the scope of this thesis work as it is already created and managed by the client's platform. Similarly, to database the application also interacts with the Cognito service on client's platform to manage user access rights to various sites.

3.2 API design

As the web application developed as part of this thesis work is based on serverless pattern which means there is no traditional webserver needed to run the application, this along with the many benefit brings about a few interesting problems as well.

One of the problems is on how to secure the communication with the client's platform databases, in the traditional model this could easily be solved by using server-side code i.e. the code for communication to database along with the keys is stored on the server and client-side code only contains the rendering code which displays the information to user. In the serverless world, this is not so straightforward as the webserver which in the case of this application is AWS S3 only allows static web-hosting which means it can serve up files over HTTP/HTTPS but all execution happens on the client side i.e. web browser.

In order to overcome the aforementioned problem, it was decided that a RESTful API would be developed as part of this work which can be utilized by the front-end of the application to interact with client's platform databases. The access to API would be secured

and would require a valid security token to be included with each request for the API to process the request. All API endpoints along with their use case are described below.

3.2.1 Sites

This is first top-level resource in the API and is used for representing all sites which are connected to the commissioning party's IoT platform. A site can be a stand-alone building, a collection of buildings or a sub-area of a building, in general a site can be considered as any space which has one or more of client's devices installed and connected to their platform. The REST methods supported by the '/sites' endpoint are GET and POST.

A restful call using the GET methods returns the list of all sites existing in commissioning party's platform, as the list can grow to be very large over time, the API endpoint supports pagination and returns a paging token with the response if there are more results present than the sent ones, the pagination token can then be included in the next request to get the next page of results from the API.

POST method on the sites resource is used for creating new sites in the platform, the details of the site are presented as JSON object in the request body. Each site has a unique identifier in the form of UUID in commissioning party's platform, this identifier is automatically generated when the POST method is called on the 'sites' end point and it is stored in the database on commissioning party's platform along with the information provided by the users.

3.2.2 Sites/{Site_Id}

{Site_Id} is a sub-resource of sites endpoint and it is a dynamic endpoint which is created at run-time to represents each individual site as an API endpoint. The endpoint provides information and operations for a single site. The REST methods supported by this endpoint are GET, PUT and DELETE.

GET method is used for returning all information about the individual site represented by the identifier present in the URL path as a JSON object. The object contains information on site's unique identifier, physical address, site name, floors and other building data relevant for commissioning party's smart building platform.

PUT method is used for editing information for an already existing site. The modifications required to the site are sent as a JSON object in the request body and the API upon success modification in the database returns the modified sites object.

DELETE method on the endpoint is used for deleting an existing site from the platform, the site identifier is fetched from the URL path. Once the request is processed and requested site is successfully deleted the API returns the confirmation message along with the identifier of the deleted site.

3.2.3 Users

Users is the second top-level resource in the designed API, it is used for representing the commissioning party's platform users in the API as a resource. A user is considered to be any customer of Helvar having access to the platform. There is no need for creating new users in the platform by the administrator as the client's platform provides the capability to their customers to sign up and create their own accounts as the users of the platform.

Although the customers can sign up as users by themselves, an administrator is required to allocate access rights to the user which will determine which site or sites the user can access, by default the user has access to no sites at the time of signing up. The process to determine to which sites the user should have access is an internal process of the commissioning party's organization and it is determined by the administrator of the platform accordingly using the web application tool developed as part of this work.

Users endpoint supports only GET method for the resource as no other operations are needed for this resource. A GET call to the users endpoint returns the list of all users in the platform and as such list can be quite large, the endpoint supports paging mechanism and provides pagination tokens for navigating through the results.

3.2.4 Users/{User_Id}

This resource is a sub-resource of 'users' endpoint and it's a dynamic resource as well. It represents each individual user represented by the unique identifier which is assigned to each user when they sign up on commissioning party's platform.

The endpoint supports only GET method for fetching the details of the user like their username, email, access rights etc. The API endpoint returns a JSON object containing the user details.

3.2.5 Users/{User_Id}/access-rights

Access-rights is a sub resource of 'users/{user-id)' endpoint and it represents the access rights associated with the user, these rights determine which sites the user can access on the platform. The endpoint supports GET, POST and DELETE methods for the resource.

The access rights are modeled as a JSON object and contains the unique identifier for each site the user has access to.

GET method on the resource is used for returning the access-rights associated with the user having the identifier present in the URL path. The response from the API is returned as a JSON object.

POST method is used for creating new rights or modifying existing access rights of the user. The API endpoint would automatically create the access-rights entry in to the database if it doesn't exist previously and in case it already exists in the database, It will modify the existing record with the new access-rights object sent as the request. The endpoint accepts the access rights as JSON object present in the request body.

DELETE method is used for removing all associated access rights from the user. The endpoint doesn't require any body or query parameters. Upon successfully removing the access-rights, the API returns the identifier of the user whose access rights have been deleted.

3.2.6 Keys

The last top-level resource in the API is called 'keys', it represents the API Keys which are used for accessing data and resources from commissioning party's IoT platform. The API keys allows users who don't want to use the platform provided UI but rather have access to data to integrate with other system in the building or get data into their own monitoring systems.

Each API key is unique in the platform with a unique identifier, a custom name and its own set of permissions. An API key needs to be associated to a usage plan, a usage plan in the commissioning party's platform determines the usage limits for the API such as amount of calls, throughput, data transfer metrics etc. Each API key also has associated permissions, which determine which site's data can be accessed by using the API key. This endpoint supports GET and POST methods.

GET method returns the list of all active keys in the platform, as the list can be quite big, the endpoint supports paging mechanism via including a pagination token in the response. The response from the API contains a JSON object with an array of all Keys represented as JSON object along with their details.

POST method is used for creating new API keys in the platform, the required details are expected by the endpoint to be included as JSON object in the request body. The API endpoint automatically creates a new usage plan with default values provided by the commissioning party and associates the newly created key with the usage plan. As the response, the newly created API key is returned along with its details as a JSON object.

3.2.7 Keys/{Key_ID}

{Key_Id} is a dynamically created resource at the run time to represent each individual key which exists in the commissioning party's platform. It is a sub resource of the top-level Keys resource. The endpoint returns information about the API key and the supported method by the resource are GET, PUT and DELETE.

GET method returns the API key details and the associated permissions with the key. PUT method is used for editing information about an existing key referred by the key identifier present in the URL path parameter. Finally, the DELETE method as apparent from the naming is used for deleting API key referred by the identifier.

3.3 Workflow and use cases for the front-end

To facilitate the development process of the web application workflows for user's sign-in and sign-out actions were created together with use cases for the users and sites view pages.

Based on theoretical knowledge of AWS Cognito, it was decided that the best way to access the service from the front-end securely would be retrieving the token after successful authorization and saving it to React's session Storage, after that the user would be redirected to the Sites page. In case of failed authentication an error message would be displayed to the user. The workflow for user's authentication was created based on previously mentioned details, see figure 2.

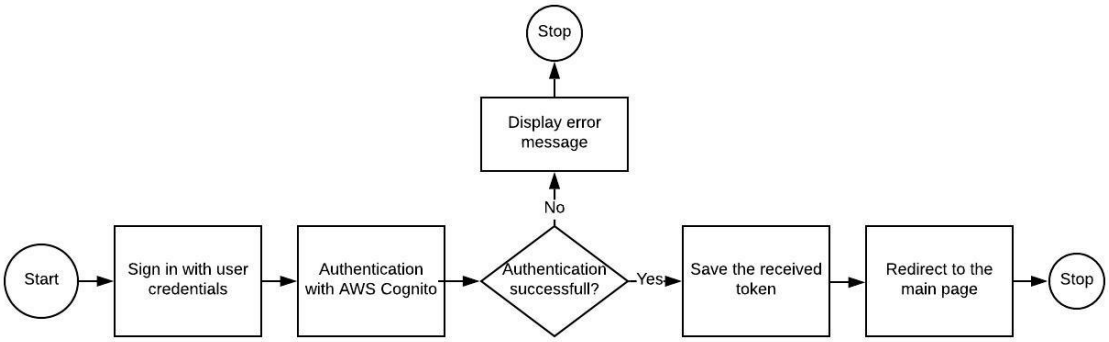

Figure 2. Workflow of sign-in process

To sign out from the application, the user would have to click the sign out button which would clear the storage and invalidate the token and the user would be redirected back to sign-in page, see the workflow of sign-out process on figure 3.

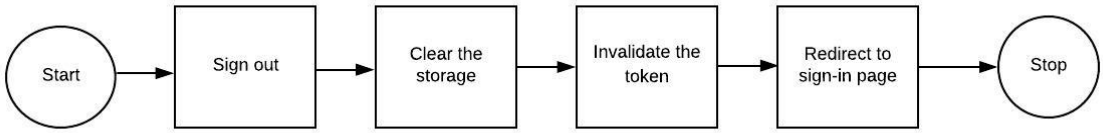

Figure 3. Workflow of sign-out process

To represent the system's functional requirements, the use case diagrams were created for sites and users pages. Modeling the intended application's functionality, listing possible use cases and expected behavior of the system. See figure 4 and 5 for more details.

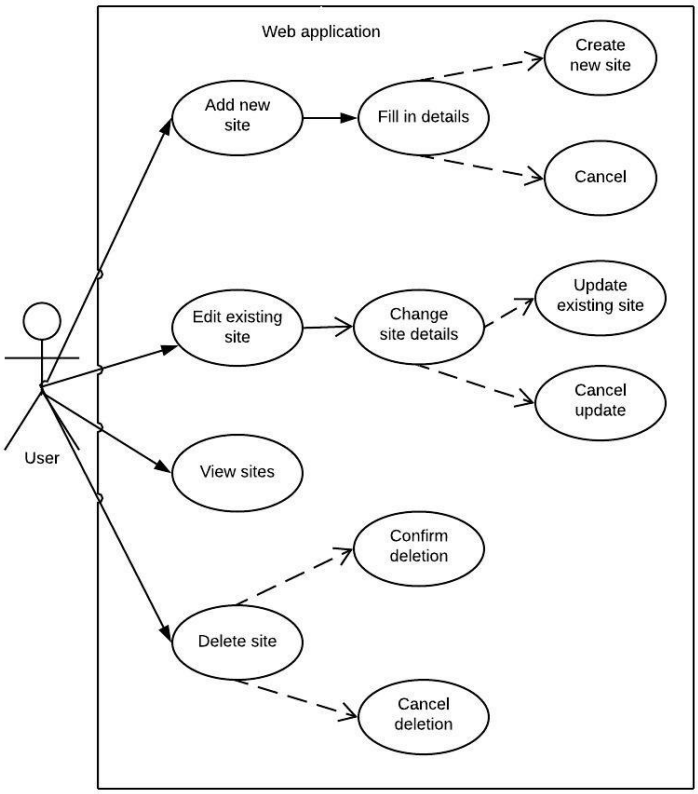

Figure 4. Use case diagram for Sites page

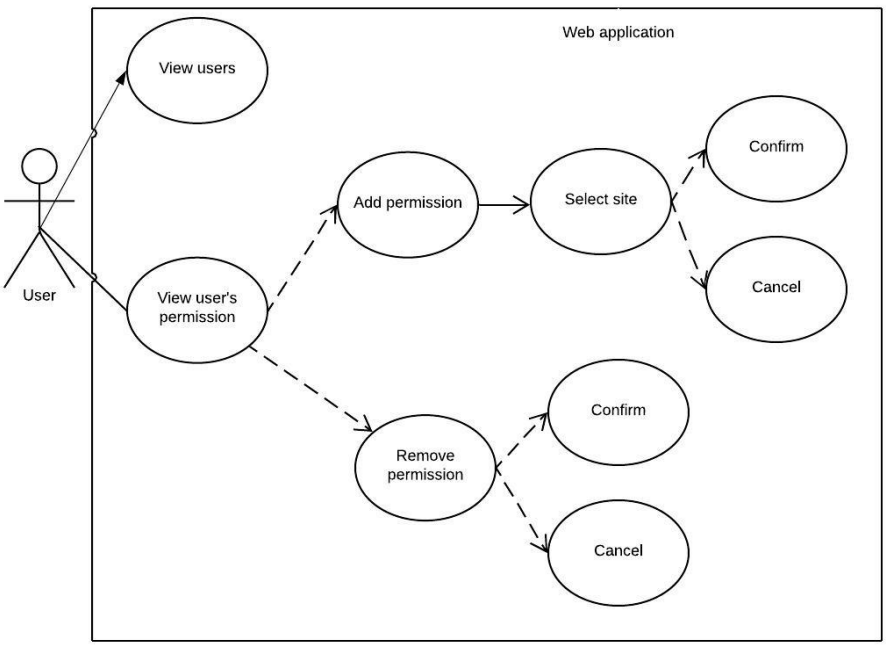

Figure 5. Use case diagram for Users page

4 Implementation of the Web application.

This chapter describes the implementation process of the web application in stages as for the front-end and the backend and explains completed steps to create the working product. During each stage the outcome was confirmed with the commissioning party. Kanban software tool was used to plan implementation part of the thesis, the tasks in the to-do column were added to already existing ones and afterwards analysed and prioritized to create the most efficient work flow for the process.

4.1 Setting up the environment

As mentioned before, the commissioning party have presented their requirements for the tools to be used for developing and deploying the application. So, there was no comparative research done to choose one or another tool. The commissioning party already has a stable development environment set up with the help of AWS services. After getting theoretical knowledge about AWS services to be used in the project, the credentials were acquired for the development environment to learn and practice required services in real account. Otherwise to create AWS account, the credit card should be linked during sign-up process, even though nothing is going to be charged if the services are used with the free tier policies set out by AWS.

Since the commissioning party already has the platform in production on AWS, the databases required for creating, editing and deleting sites and users already existed, which didn’t require any modifications.

The backend Node.js code was written directly in Lambda service, so it didn’t require any additional set up, as well as API Gateway. All amazon services are managed from their own web console or command line tools.

React.js was used for writing front-end code and required installation to own computer, as it was decided to use the material-UI framework for front-end to have the same look and feel as the other web applications developed by the commissioning party, this component library had to be installed as well. The following steps were completed to install the framework:

- Install Node.js by downloading binaries from node.js website and installing them on local computer. When Node is downloaded to the computer it automatically includes npm. Npm later on is required to install libraries and frameworks, such as React.js.
- Create React-app with the command *npx create-react-app appName*, which is a build setup with no configuration.
- Install react-router-dom with the command npm install –save react-router-dom to create DOM bindings for React Router in the application.
- Install material-ui library and dependencies with the command *npm install @material-ui/core*
- Install aws-amplify library with the command npm install –save aws-amplify to use this library for user authentication with AWS Cognito

Versions used during the development process of the web application: Node.js v.9.4.0 and npm v.6.8.0.

The source code was version controlled via Git repositories, the repositories were hosted in the commissioning party's git servers. Git was used directly from command-line and for code reviews and repository management.

4.2 Front-end implementation

After the development environment was prepared and initial application infrastructure was created, the development was set in three iterations: implementation of Sites view page, implementation of Users page and implementation of Sign-in page as per commissioning party's requirements.

Sign –in page's purpose is to provide capability of signing-in already existing admin users, the number of users is small including few selected people from client's support team and logistics who would have access to the tool.

The Users page lists all the active users of the platform in a tabular format, including their username, email and status, in addition it has a button to show permission for a specific user, which would open the pop-up and display the list of sites, the specific user has access to and allow to grant access permissions for additional sites which already exist in commissioning party's platform it. The same pop-up menu can be used for removing access to sites as well.

The Sites page lists all the sites in a table, displaying information like site's id, name, country and address, in addition there are two buttons for modifying the site details or deleting one. An Add site button is provided on the page as well which would allow to create a new site in the platform.

The front-end project code is structured in three main directories which are src, node_modules, public and three other files called package-lock.json, package.json and README.md. The actual source code of the application is located in 'src' folder, which has the following substructure: components, views and static folder and few other files including App.js, index.js and index.html. Views folder contains jsx files for each UI view page and the components folder includes all the components used to build up the application itself. Since React components are reusable they could be placed in different jsx files, for example in this project the NavBar component is reusable and is used in both Sites and Users views.

As the web-application created with React.js is a single page application, the main component is called 'App' which allows us to place and work with other components in the context of DOM (Document Object Model). The rendering of React elements into a root DOM node happens when the App and 'root' are both passed to the ReactDOM.render() function, see figure 6. React DOM compares all existing elements and their children and if there is a difference in the state, DOM would execute the required updates to refresh the views.

```
ReactDOM.render(<App />, document.getElementById('root'));
```

Figure 6. Rendering the main App component into a root DOM

To navigate between views, react-router-dom was installed as a node module and since we have 3 different page views, there are 3 Routes path defined, as shown in the figure 7. The default page with route matching "/" is taking user to sign in view, with the route "/main" the applications would show the sites view and with the route "/users" the users view would be displayed.

```
<Switch>
<Route exact path="/" component={LoginPage} />
<Route path="/main" component={SitesPage} />
<Route path="/users" component={UsersPage} />
</Switch>
```

Figure 7. Route path for existing components.

Initially it was planned to have a side-bar menu to switch between page's view but since there are only two views available when the user is signed in, the implementation has been modified so that the navigation bar would have clickable button with text, either. Users or Sites next to the sign out button and allow the user to switch from one view to another.

The state within the components was managed without state-managed libraries, this.state and this.props were used in the source code when the output of render should be influenced. Through setState() the state is changed and component is re-rendered. Calls to setState are asynchronous therefore it is recommended to pass an updater function. This approach is used when new items are added, edited or deleted in the sites table or permissions are updated in users table. For example, once the selected site is deleted the user should see updated list of sites in the table, therefore we call loadSites() function inside the delete function to fetch the updated list of sites, to set the new state for the sites array and render the page to display updated data. See the sample code in the figure 8 below.

```
loadSites = () =>{
    fetch(URL,
      { method: 'GET',
        headers:{'Authorization': this.state.requestToken},
      })
    .then(res => res.json())
    .then((resData) => {
        this.setState({sites: resData.Items})}
    );
}

deleteSite= (value) => {

                fetch(URL+'/'+value,
                  {method: 'DELETE'
                  headers:'Authorization': this.state.requestToken,

                  }),
                .then(res => {
                this.loadSites()
                })
                .catch(err => console.error(err))
}
```

Figure 8. Example of using state within Sites component

As discussed in the theory part, Material-UI library was chosen for utilising various material design elements in the application, which allows to keep the look of elements such as table, buttons, header, navigation menu in the same style in the application, this results in a better-looking UI and provides an enhanced user experience (UX) via a consistent look and behaviour.

The usage of Material-UI components was in most of the cases with their default configurations, but sometimes the default colours had to be overwritten, so 'MuiThemeProvider' component was used to customize the theme and was placed to wrap the component that needs to be changed. For example, the palette is one of the main theme configuration variables and it maps colour intentions. Primary colour intention is used for primary interface elements and the default colour is blue. Figure 9 displays the code for changing the colour of primary Material-UI palette variable.

```
const redTheme = createMuiTheme({
  palette: {
    primary: {
        main: '#b71c1c',
    }
  },
```

Figure 9. Changing primary color of Material-UI component.

Securing the access to web application was the last step in the functional implementation of the front-end. Once user pool was created and Cognito service was configured to connect to the API endpoints, the source code in Sign-in component was added to connect to the API endpoint using aws-amplify JavaScript library, in addition to that, the config.js file was created to store the following details related to AWS Cognito service: REGION, USER_POOL_ID, APP_CLIENT_ID. To load the configuration file Amplify.configure() is called in index.js component.

Calling 'Auth.signIn' method initiates the sign-in process to request a session from Amazon Cognito. In case of successful authentication, the user object is returned and contains amongst other data the 'tokenId' required for further interaction with the implemented API endpoints. In case of failed authentication, a pop-up window is shown to the user with the corresponding error message.

As the solution for token storage and retrieval the sessionStorage was chosen. It seemed to be an appropriate option, since it was allowing to imitate the following behaviour: the user is logged in into the application and user session is saved until the browser window is closed. AWS-Amplify also provides the ready to use method 'Auth.signOut', which would revoke the authentication token.

4.2.1 Sign-in page

The purpose of the sign-in page is to allow already existing users to login to the web application, the number of users for the tool is small consisting of a few selected people from client's support team and logistics who would have access to the tool.

The sign-in page view is the landing page of the application. The implementation of the page was broken into two stages, first was creating the UI and putting components' structure in place and second stage included creating functionality of receiving credentials and authorizing them to log in and be directed to the main page with sites view. Figure 10 displays the user's view of sign-in page.

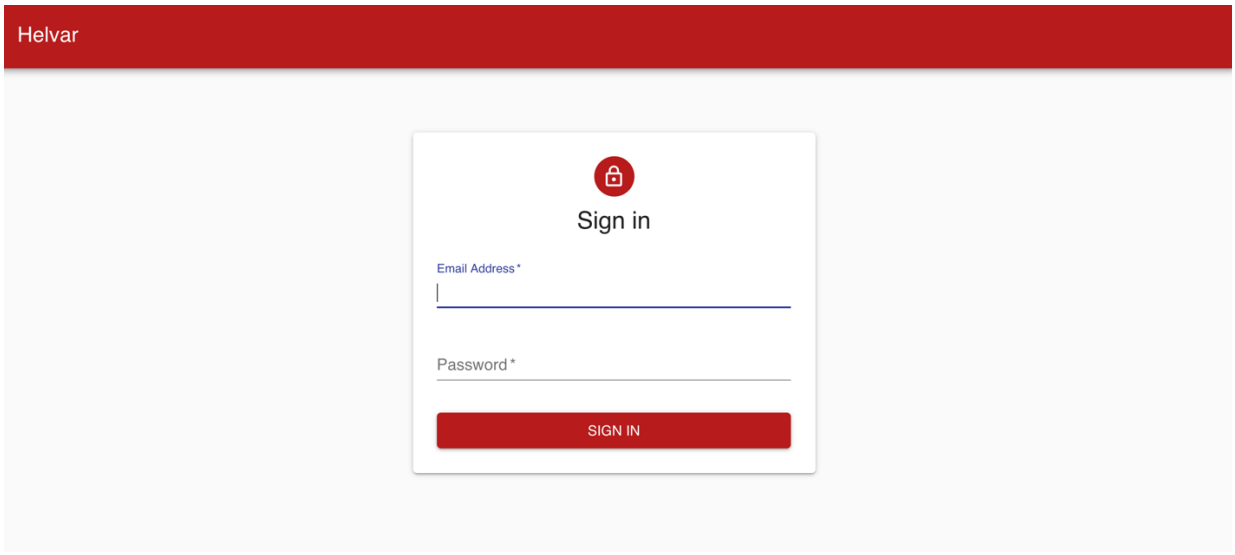

Figure 10. The view of sign-in page

4.2.2 Sites Page

Main page is the view of all sites in a tabular format with possibility to add, edit or remove sites. The navigation bar at the top of the page has two buttons Users and Sign out, so the user can switch to another view of Users table or sign out from the application. Sites' and users' views are similar and have similar components structure, for example SitesPage view has three different components, such as NavBar, Sitelist and AddSite, see figure 11.

```
import React from 'react';
import Header from '../../components/Header/header.jsx';
import NavBar from '../../components/NavBar/NavBar.js';
import Sitelist from '../../components/Sites/Sitelist.js';
import AddSite from '../../components/Sites/AddSite.js';


class SitesPage extends React.Component {
  render() {
    return (
    <div>
      <NavBar />
        <div>
          <AddSite />
        </div>
      <div>
        <Sitelist />
      </div>
    </div>
  );
  }
}

export default SitesPage;
```

Figure 11. The components' structure on the example of SitesPage

'Sitelist' component includes different Material UI components, the main ones are table, buttons and dialog and functionality to interact with the created REST API. Once the user gets redirected to the main page, 'componentDidMount' function is invoked which instantiates the network request by loading data from the endpoint.

To add new site the button 'Add Site' is available for the user on top of the page before the table of sites. AddSite is a separate component and contains the dialog form for adding a new site, see figure 12. The functionality to interact with corresponding API endpoint is included in the component as well.

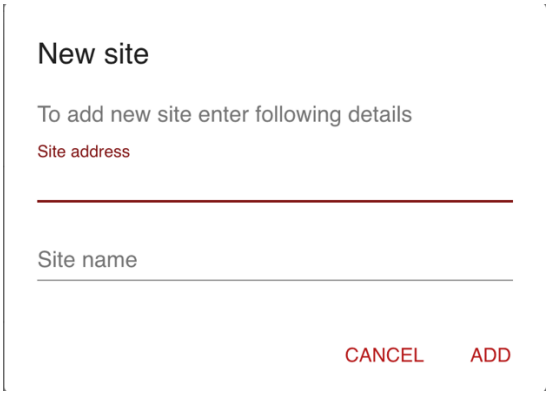

Figure 12. The diaolg window for adding a new site.

The functionality of editing and deleting existing sites is accessible for the user through the buttons with corresponding names for each specific site displayed in the table. The

functionality includes the logic of interacting with REST API through fetch function and defining the call method together with headers and body. Edit functionality is placed as child component of ‘Sitelist.js’ and delete functionality is included in the main Sitelist component.

4.2.3 Users page

Users page view is quite similar to the Sites page view, it uses mainly the same Material-UI components such as NavBar, table, buttons and dialog. The functionality is implemented in similar way as well. The user can access the users page by clicking the button Users on sites page view. The users page view displays the table of existing sites’ users and the navigation bar at the top which has two buttons Sites and Sign out. Each user listed in the table has a ‘Show Permissions’ button, when clicked would open a dialog form and display the sites that this specific user is allowed to access. The sites are listed in a tabular format, each site has a corresponding ‘Remove Permission’ button and above the table there is ‘Add permission’ button to add new permissions for the site.

Users component has a ‘Permissions’ component as a child component, the functionality of both components is implemented through fetching the data from REST APIs and manipulation of the state of different elements within components in a similar fashion to the previously described components.

4.3 Backend implementation

4.3.1 Developing Lambda functions

As discussed in the theory section, Lambda functions should be concise and quick to execute, therefore for each API point the functions were created separately.

As per client’s requirements, the following methods were required for extracting data from users table: GET all users, GET/DELETE specific user, POST a new user. As for the sites information, following methods were used: GET all sites, POST a new site, PUT/DELETE /GET specific site.

Initial step was to create a Lambda function for each API endpoint. Even though there are templates available for creating functions, they were not utilized in this project. The required functions were created with runtime language Node.js 8.10. AWS provides SDK to use DynamoDB document client for querying and scanning tables and provides simple ways to manipulate items in the tables. Below is a sample code for getting all items from one table with DocumentClient where scan corresponds to get method, see figure 13.

DocumentClient was used in each Lambda function which are invoked for each request to the API endpoints.

index.js

```
1  console.log('function starts');
2
3  const AWS = require('aws-sdk');
4  const docClient = new AWS.DynamoDB.DocumentClient({region: 'eu-west-1'});
5
6  exports.handler = function(event, context, callback){
7      console.log('processing event: %j', event);
8
9      let scanningParameters = {
10         TableName: 'TableName',
11         Limit: 100 //maximum result of 100 items
12     };
13
14     //In dynamoDB scan looks through your entire table and fetches all data
15     docClient.scan(scanningParameters, function(err,data){
16         if(err){
17             callback(err, null);
18         }else{
19             callback(null,data);
20         }
21     });
22 }
```

Figure 13. Sample code of Lambda function to get all sites.

Once the Lambda function is written it can be directly tested, which appeared to be quite handy for debugging and caching the initial errors, but not enough to track the errors which may occur when more services are integrated or for the errors in request mapping, which forwards all the user provided request parameters such as body and query parameters to the lambda function.

4.3.2 API Implementation

React.js utilizes created API with fetch methods, which is one of the option available in React.js for invoking remote API endpoints and fetching information from these via HTTP/HTTPS requests. The API calls were used to fetch data from the commissioning party's platform about sites and users which is displayed in the developed web application. The main logic for REST API was implemented through Lambda functions and API Gateway service in AWS using node.js.

AWS API Gateway refers to endpoints as resources, so needed end points were created through AWS console under the root resource, such as /sites and /users. There are some basic configurations offered before creating each resource, such as configure the resource as proxy or enable API Gateway CORS. Every resource may have child resources

where the individual item from the table could be requested, for example for users the child resource is user-id, see the figure 14. The project required to have two main resources: sites and users. Sites therefore had one child resource: site-id and resources had a child resource user-id, which had its' own child resource access-rights.

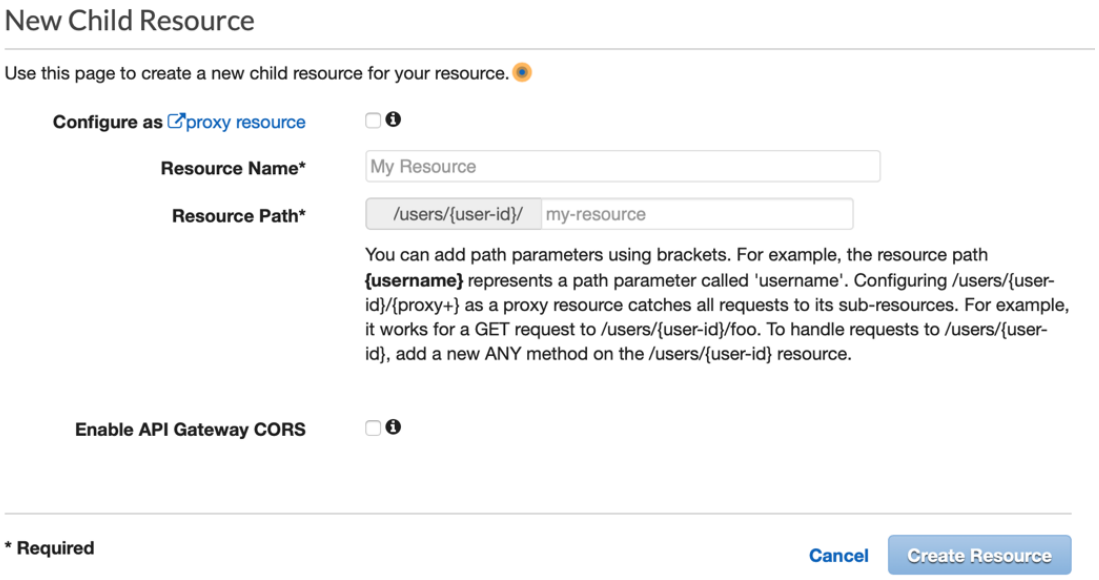

Figure 14. Creation of child resourse for users in API Gateway

Each resource and child resource have their own REST methods. According to client's requirements for resource sites following methods were created: GET and POST, for its' child resource site-id the methods were DELETE, GET and PUT, for the users resources the method was GET, for its' child resource user-id – GET, DELETE, POST, PUT and for the last child resources access- rights GET. API Gateway configuration for resources listed above is represented in figure 15, only the methods mentioned are supported.

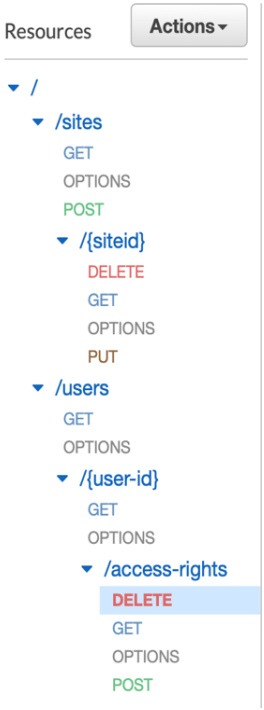

Figure 15. Structure of resources and methods for the project in AWS API Gateway

When the requests from the frontend are made to the endpoints the API Gateway connects those requests to Lambda functions, the configuration of API Gateway for sites resource and GET method is demonstrated in figure 16.

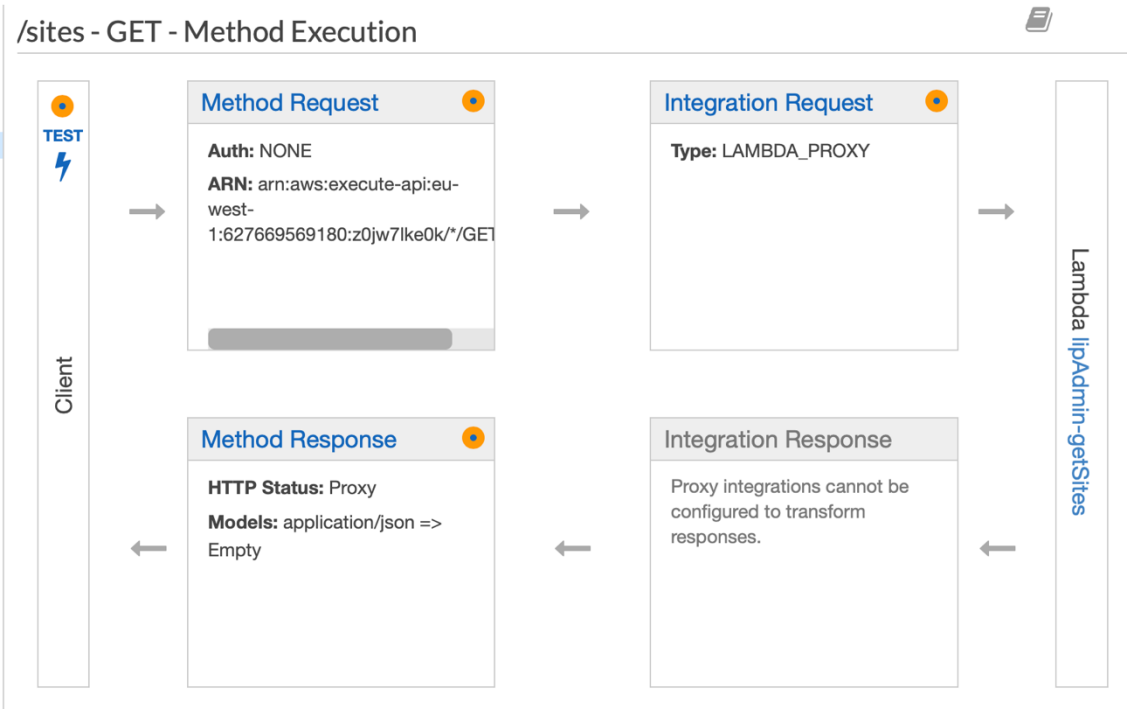

Figure 16. API Gateway configuration for GET method for /sites endpoint

Methods can be easily created in console and during setup offer different integration types, in given case Lambda function integration was used. When creating the method with Lambda integration the Lambda function name should be given, as the lambda functions were written prior in the project, typing the name in the field would suggest it for choosing as the target function.

API Gateway allows to configure directly what would be the method's request and response header and body, but since our Lambda functions contains that already, this configuration is left empty.

4.3.3 Backend Security

Securing the backend was achieved by a combination of encryption, authentication and authorization techniques. Encryption was enabled by using the REST API over HTTPS which uses TLS to encrypt the messages being sent to the API, this would ensure the frontend was always communicating with the backend API to which the public certificate belongs to. Using HTTPS also ensure that attacks such "man in the middle" can be prevented as the communication will be encrypted with the public certificate of the API and can only be decrypted by the API endpoints having access to the private certificate.

Authentication was handled by utilizing the AWS service called Cognito. AWS Cognito provides two main ways for handling authentication which are 'User Pool' and 'User Federated Identities' as discussed in theory section. As the application under development is for internal use only and the tasks carried out are business confidential therefore it was decided that Cognito user pools were the best fit as they provide for the application the ability to create and manage its own users. In this case, a few users from the commissioning party's organization who will have administration rights to the platform would be created a user profile manually at the time of deployment. Users are not allowed to sign up for new accounts by themselves and this remained out of scope for the thesis work.

The setup of user pools in AWS Cognito was handled via web console and the changes made were also captured in a CloudFormation file which would allow the commissioning party to deploy the Cognito user pools in their environment. During the setup of user pools, the mandatory fields and value types which are required for the user profile are marked along with any additional steps required for signing in such as MFA (multi-factor authentication) or SMS verification as well as the minimum password length and other password characteristics such as the use of numeric and special characters.

Once the user pool and test users were setup, the next step was to create an ‘Application Client’ in AWS Cognito service, the application client refers to the applications which will use the user pool for user authentication, each application has its own unique client id and secret key to ensure only authorized application can utilize the user pool. At this point the token expiry period is also set which means once the token is expired, the user must sign in again to obtain a new token.

To secure the access to API, AWS API Gateway service which is used to host the API provides a feature called Authorizers. This allows for the setup of own authorization functions which are executed for each call to the API to verify if the request is legitimate. The use of AWS Cognito as the authorization function is supported as a built-in function in the API Gateway and the it was used as the authorizer for this application.

When setting up the authorizer, the AWS Cognito user pool identifier was provided which tells the function about which user pool should be used to validate the provided token, next the request header which will be used by the front-end to provide the user token was setup. The header was called “Authorization” which is a very commonly used header for this purpose and is built in by default in many of the http client libraries. Finally, a token validator was setup in the authorizer function, this is not the final validator but the validation is based on a regular expression to verify the information provided under the Authorization header follows the expected format and type before passing it on to AWS Cognito to check the validity of the token. The set up for the authorizer is shown in the figure 17. This initial step helps in avoiding sending malformed tokens to the authentication function and also makes sure no attacks such as “SQL injection” can be performed on the authorization database.

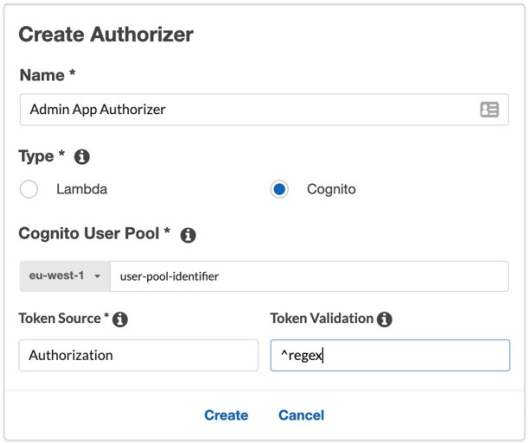

Figure 17. Set up of the authorizer for AWS Cognito

After the setup of Authorizer had been completed, the final step in securing the API endpoints was to enable the use of the created method on each API resource. As authorization is supported out of box by the API Gateway and the built in Authorization function is used, no special steps were required for integration and the setup was simply selecting the authorizer as enabled in method request setting for each resource and pointing it to the newly created authorizer method for validation, see figure 18.

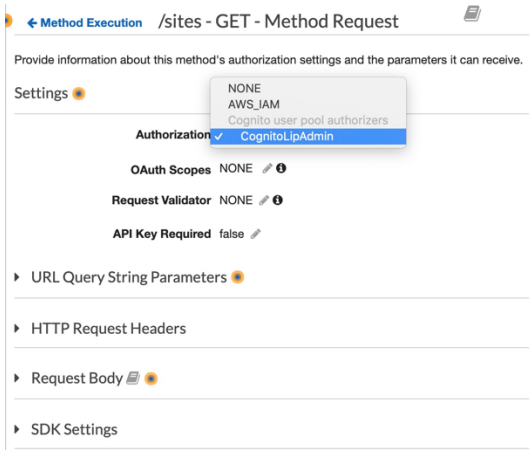

Figure 18. Enabling authorization function for GET sites API resource.

5 Discussion

This chapter discusses the final project result and implementation, evaluating the thesis process and author's own learning, as well as challenges faced during the process and their solutions, in addition the maintenance and future development ideas were proposed to the commissioning party.

5.1 Review

The use of cloud services has been recently on the rising trend and many companies are getting interested in switching from traditional server-based approaches to serverless or have already done it, as well as JavaScript frameworks such as React and backend technologies like Node.js are gaining more popularity for their ease of use. These services have plentiful of resources available about documentation, different libraries and plugins which further make them an attractive choice for development.

One of the objectives for writing the thesis was to gain deeper knowledge of AWS, React and Node.js technologies and gain practical experience through developing the web application, keeping in mind that knowledge received would help the author in her future career development and job search, referring to the recent job market requirements. The objective was carried out in a successful way, but more practice would be required to get more fluent with the technologies mentioned above.

Another objective was to deliver the web application to the commissioning party corresponding to their requirements that would replace the current manual process of registering new building sites, users and giving them appropriate access rights with an easy to use web application as an administration tool. This objective can also be evaluated as successfully implemented, since the product was delivered to the commissioning party and was described as easy to use as well as conforming to all the listed requirements. The application was tested by one of the future users of the tool and the basic flow was checked.

The theoretical framework was conducted through the literature review methodology and was presented to the readers before empirical part of the thesis to introduce the main concepts of the technologies being used in the implementation of the web application. The planning and development parts of the thesis were carried out with Kanban methodology and the list of items was tracked in the software tool. Kanban tool is great approach to be used in a team, but in the particular case it had to be adjusted and modified for the needs of the author and some parts like using metrics and analytics for the improvement of the

process workflow were omitted. The tool was used to visualize the process flow and list all the tasks required for implementation of the web application. This allowed to keep all the items in a logical order and identify the remaining tasks that still need to be completed.

During creation of the front-end, the workflows and use cases were created to validate required functionality with the commissioning party. Once the functionality was implemented into the frontend, it was verified again with the commissioning party to validate the logic and user flows.

The assistance and availability of the commissioning party ensured successful implementation of the project's goal, resulting in the delivery of the working web application conforming to the initial requirements.

5.2 Challenges and solutions

One of the biggest challenges faced during the thesis work, was the sheer amount of new technologies to learn and implement during relatively short time period, the author was unfamiliar with most of AWS services before starting the work. React.js was used by the author in a few school projects and Node.js was used previously for one school project.

Author's personal interest in the topic and provisioning of AWS testing environment for learning purposes by the commissioning party as well as the good availability of the commissioning party to answer the questions and provide clarifications during the development and implementation process helped to overcome the challenge of learning and using new technologies.

Another challenge faced during the implementation process on front end side was the removal of one Material-UI component used in Sign-in form of the application in the next major release versions. The possible solutions were offered on the official website of Material-UI and their migration guide was used to complete required modifications. Fortunately, that required very little time to fix.

On the backend side, there were a few technical challenges faced as well, one of them was related to the use of asynchronous programming model of Node.js which was used for implementing the backend lambda functions. Lambda functions as described earlier are on demand compute functions, which are deployed during runtime when they are invoked and are decommissioned when execution is over. When using an asynchronous programming model this needs to be considered and all function calls should be handled either via JavaScript Promise feature or marking the functions as asynchronous and using

the await command for them to finish processing. This was a challenge at the start which resulted in some debugging issues as the functions kept ending the execution before the results were returned from the subsequent function calls but these issues were resolved by using the aforementioned techniques.

5.3 Future development

There are more features that are planned to be implemented into the application by the commissioning party in relation to other administration activities required for the platform. Currently the application deals only with the site creation and managing user rights but in future it is envisioned that the commissioning party would be able to manage fine grained access control for users such as letting them control only a floor or a part of the floor in a building instead of the whole site.

From technical perspective, it is recommended to use the Redux library in react for front-end development as when the amount of various views and components will grow in the application, there would be a need for maintaining a global state in the application. Such a state can then be shared across the components to maintain the information which should be used by multiple components or different pages.

For security improvements, it is recommended to implement the use of multi factor authentication for platform administrators which provides better protection and access-control than password-only solutions. Any MFA would be useful for the application but it is recommended to the use email or SMS for the MFA code distribution to maintain the user experience and keep the application as simple as possible.

References

Amazon CloudFormation, 2019. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-ug.pdf. Accessed: 28 April 2019.

Amazon CloudWatch, 2019. URL: https://aws.amazon.com/cloudwatch/. Accessed: 03 April 2019.

Amazon Web Services, Inc and/or its affiliates. 2019. Amazon API Gateway Developer Guide. URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf. Accessed: 03 April 2019.

Amazon Web Services, Inc and/or its affiliates. 2019. Amazon Cognito Developer guide. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-dg.pdf. Accessed: 03 April 2019.

Amazon Web Services, Inc and/or its affiliates. 2019. Amazon Simple Storage Service: Developer Guide. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf. Accessed: 03 April 2019.

Amazon Web Services, Inc and/or its affiliates. 2019. AWS Lambda: Developer Guide. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf. Accessed: 07 April 2019.

Atlassian Agile Coach. 2019. URL: https://www.atlassian.com/agile/kanban. Accessed: 02 May 2019.

Baron J, Baz H., Bixler T., Gaut B., KellyK., Senior S., Stamper J. 2017. AWS, Certified Solutions Architect Official Study Guide. John Wiley & Sons, Inc. Indianapolis, Indiana.

Copes, F. 2019. The React Handbook. 2019. URL: https://medium.freecodecamp.org/the-react-handbook-b71c27b0a795. Accessed: 07 April 2019

Ghauri, P. & Grønhaug, K. 2010. Research methods in business studies. 4th ed. Pearson Education. Harlow.

Gurturk, C. 2017. Building Serverless Architectures. Published by Packt Publishing.

Helvar. 2019. URL: https://www.helvar.com/en/. Accessed: 15 April 2019.

Klipp, P. Getting started with Kanban. 2019. URL: https://kanbanery.com/ebook/GettingStartedWithKanban.pdf. Accessed: 02 May 2019.

Material-UI Documentation. 2019. URL: https://material-ui.com/. Accessed: 10 April 2019.

Niranjanamurthy M., Archana U., Niveditha K., Jafar S., Shravan N. 2014. The Research Study on DynamoDB – Nosql Database Service. IJCSMC, Vol. 3, Issue. 10, October 2014. URL: https://www.academia.edu/8818008/The_Research_Study_on_DynamoDB_NoSQL_Database_Service. Accessed: 03 April 2019.

Node.js Foundation. 2019. URL: https://nodejs.org/en/about/. Accessed: 10 April 2019.

Rady, B. P. 2016. Serverless Single Page Apps. Pragmatic Bookshelf.

React.js Facebook Inc. 2019. URL: https://reactjs.org/. Accessed: 10 April 2019

Roberts, M. Serverless Architectures. URL: https://martinfowler.com/articles/serverless.html. Accessed: 17 March 2019.

Sbarski, P. 2017. Serverless Architecture on AWS: With examples using AWS Lambda. Manning Publications.

Tutorials Point. Kanban. 2016. URL: https://www.tutorialspoint.com/kanban/kanban_tutorial.pdf. Accessed: 02 May 2019.

Wester, J. What is Kanban. 2016. URL: http://www.everydaykanban.com/what-is-kanban/. Accessed: 02 May 2019.

Wittig, M. 2016. Introducing the Object Store:S3. URL: https://cloudonaut.io/introducing-the-object-store-s3/. Accessed 03 April 2019.

Table of figures