

Serverless backend migration from JavaScript to TypeScript



Bachelor's thesis

Information and communications technology, Riihimäki

Spring, 2022

Jani Koskela

Koulutus

Tiivistelmä

Kampus

Tekijä Jani Koskela

Vuosi 2022

Työn nimi Pilven palvelimettoman taustajärjestelmän muuntaminen JavaScriptistä
TypeScriptiksi

Ohjaajat Toni Laitinen

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli muuttaa Lambda-funktioista koostuva pilven palvelimeton Node.js-taustajärjestelmä, TypeScriptiksi. Lambda-funktiot oli kirjoitettu JavaScriptillä ja käyttivät Amazon Web Service -pilvipalvelun palveluita koodin käyttöönottoprosessin automatisointiin.

Taustajärjestelmä on käytössä Maranet-sovelluksella. Maranet on websovellus, mobiililaitteelle ja tietokoneelle. Sovellus on käytössä maanrakennusyriyksillä. Se helpottaa yrityksiä kirjaamaan maanrakennustöitä. Sovelluksessa on käytössä ReactJs selainpuoli.

Opinnäytetyössä käydään läpi teknologiat, joita käytetään muuntotyössä, Lambda-funktion käyttöönotossa ja Lambda-funktion testaamisessa. Näihin teknologioihin kuuluvat AWS palvelut, Node.js, TypeScript, JavaScript, CI/CD ja tietokannat.

TypeScript-muunnos teki taustajärjestelmästä tukevamman ja helppolukuisemman. Muunnos auttaa koodia ehkäisemään virheitä kokoamisvaiheessa. Selvempi koodi auttaa nopeamman koodin tuottamista tulevaisuudessa.

Avainsanat Lambda-funktiot, Node, Serverless, AWS, TypeScript

Sivut 37

Name of Degree Programme

Abstract

Campus

Author Jani Koskela

Year 2022

Subject Serverless backend migration from JavaScript to TypeScript

Supervisors Toni Laitinen

ABSTRACT

The goal of the thesis was to migrate serverless Node.js backend consisting of Lambda functions to use TypeScript. Lambda functions were written with JavaScript and were using the Amazon Web Service cloud provider's services, to automate the code deploy process.

The backend is used by Maranet application. Maranet is a web application for mobile and desktop. Application is used by earthwork companies. It helps companies to create earthwork jobs. The application uses ReactJs frontend.

The thesis goes through technologies used in the migration process, the Lambda deployment, and testing the Lambda function. These technologies are AWS services, Node.js, TypeScript, JavaScript, CI/CD, and databases.

The TypeScript migration made backend more robust and easier to read. The migration helps the code to prevent errors on compile time. Clearer code helps to develop code faster in the future.

Keywords Lambda functions, Node, Serverless, AWS, TypeScript

Pages 37 pages

Contents

Terminology	
1 Introduction.....	1
2 Serverless.....	1
2.1 Serverless types	2
2.2 Advantages.....	2
2.3 Disadvantages	2
2.4 Comparing to on-premise	3
2.5 Cloud Service Providers	3
3 Technologies.....	4
3.1 Database and MySQL	4
3.2 Amazon Web Services.....	5
3.2.1 AWS Lambda	5
3.2.2 Amazon S3.....	6
3.2.3 Amazon Cloudformation	6
3.2.4 Amazon Cognito	6
3.2.5 Amazon API Gateway	7
3.2.6 Amazon RDS	8
3.2.7 Amazon Cloudwatch	8
3.3 JavaScript	8
3.3.1 ECMAScript.....	9
3.3.2 Node.js and NPM	9
3.3.3 Promise and async/await	11
3.3.4 Deno	13
3.4 CI/CD	13
3.5 TypeScript.....	14
4 Maranet Application.....	15
5 Development.....	18
5.1 My initial thoughts	18
5.2 Lambda function	19
5.3 CI/CD pipeline	22
5.4 Installing TypeScript and type packages	23

5.5	Typing.....	25
5.5.1	Classes	25
5.5.2	Interface	27
5.5.3	Pre-made types from packages.....	28
5.6	Deploying function with CI/CD.....	29
5.7	Testing the Lambda function	31
6	Conclusion	33
	References.....	35

Figures

Figure 1.	Lambda function configuration.....	5
Figure 2.	HTTP request from the client to retrieve, create or alter data in the database.....	7
Figure 3.	NPM init command.....	10
Figure 4.	package.json file with installed packages.....	11
Figure 5.	Promise and async await example	12
Figure 6.	Wrong type in a function call.	14
Figure 7.	Compiling the TypeScript file and it creates a JavaScript file.....	14
Figure 8.	Create order page.....	16
Figure 9.	Web page report.	17
Figure 10.	Lambda handler function flowchart.....	19
Figure 11.	Validate input function call.	20
Figure 12.	Get user role return object.....	21
Figure 13.	Handler function.....	22
Figure 14.	tsconfig file	24
Figure 15.	Classes.....	25
Figure 16.	Class imports.....	26
Figure 17.	New instance of request body.....	26
Figure 18.	New instance of settings	26

Figure 19. User.d.ts file and User interface.....	27
Figure 20. Importing User interface	27
Figure 21. User interface used for type definition	28
Figure 22. NodeRequire type from @types/node.....	28
Figure 23. SignUpResponse type	29
Figure 24. TypeScript function in AWS Lambda	30
Figure 25. TypeScript Lambda function stack events.....	31
Figure 26. Created user	32
Figure 27. Created user in Amazon Cognito user pool.....	32
Figure 28. Lambda execution in Amazon Cloudwatch	33

Terminology

Stack	Single unit of resources
AWS	Amazon Web Services
Backend	Also known as server-side. Part of the software. Not accessible for users. Responsible for storing the data.
Frontend	Also known as client-side. Part of the software. The user interface that allows user to interact with the application.
API	Application Programming Interface. Allows two applications to communicate with each other.
Cognito user pool	Group of users saved in the Amazon Cognito.

1 Introduction

In today's world the serverless model has gained a lot of popularity due to its cost efficiency, easy maintainability, and scalability. Because cloud service providers handle things like databases and servers. It offers an easy way to execute the code automatically, saving time and money.

The goal of this thesis is to migrate a serverless backend consisting of Lambda functions to use TypeScript instead of JavaScript. Using TypeScript should increase functions' readability and make them more robust and easier to maintain.

In addition to migrating the function to use TypeScript, the CI/CD will be modified to handle compiling TypeScript file to JavaScript to avoiding unnecessary files in AWS Lambda package.

The thesis's theory section goes through serverless use in Node.js backend and covers advantages and disadvantages of the serverless architecture and Amazon Web Service technologies in use.

The serverless backend is used in Maranet that is an application made by SW-TECH Oy that is mostly meant for mobile usage. Maranet is used by construction businesses as an easy way to log worksite events and create reports.

2 Serverless

Serverless is an execution model for the cloud, where a cloud service provider is responsible for managing and maintaining the servers. Serverless apps are stored in a container, waiting to be called when needed.

The serverless means serverless for consumers, but there are still physical servers for a cloud service provider to manage. (Red Hat, 2017)

2.1 Serverless types

There are different types of serverless, IaaS (*Infrastructure-as-a-Service*) where the cloud service provider handles infrastructure, and the rest is self-handled. CaaS (*Container-as-a-Service*) is a service where also the operating system is handled along with the infrastructure. In PaaS (*Platform-as-a-Service*) also containers and the runtime are handled by the cloud provider along with the infrastructure and the operating system. FaaS (*Function-as-a-Service*) that leaves developers only to handle writing the code because all the rest is already managed by the cloud provider.

When using serverless, the more the cloud provider handles the easier and faster it gets to deploy a new code, but it will also cost more to let the cloud provider handle all those services.

2.2 Advantages

The serverless architecture really shines in that it is much faster to deploy the code. The cloud provider charges only for used services, so it removes need to pay for the useless server space. The serverless' only pay for the usage model is very cost effective when software is fairly new and has unstable usage time. (Cloudflare, n.d -a)

In addition to the serverless' cost effectiveness, it also provides an easy way to scale the software when the usage increases.

2.3 Disadvantages

Because altering code in serverless environment means uploading the code to cloud provider service, testing the code might be bit trickier. (Cloudflare, n.d -b)

Cost efficiency also means since the code is not running all the time, the first 'boot up' might take more time, this is referred as a 'cold start', but if the code is running regularly, it is referred as a

'warm start'. In this case the 'cold start' is more costly since the execution time is longer.
(Cloudfare, n.d -b)

2.4 Comparing to on-premise

On-premises type is an alternative for the serverless. In on-premises everything is self-managed. The serverless might be easier and faster, but when the application gets big enough and has a lot of traffic it might get very expensive. So eventually it might be more cost efficient to use On-premise than the serverless. (Intellias, 2021)

Two big advantages of on-premises solution are that because the data is stored locally, the sensible data can be kept inside of own company, and it is always accessible because an online connection is not needed. (Kemper, 2021)

2.5 Cloud Service Providers

The three biggest cloud service providers are AWS, Microsoft Azure, and Google Cloud Platform. Together they are accounted for 61% of the cloud service market in 2011. AWS is the most popular, covering 31%, Microsoft Azure is the second, covering 22% and Google cloud is the third, covering 8%. (Canalys, 2021)

When thinking about setting up the serverless development environment, any of these cloud service providers is a viable option. They all have their pros and cons. For example, AWS has a superior compute capacity, but Microsoft and Google have other very popular services for the easy integration. (BMC, 2021)

With AWS being the most popular with the most services, Google Cloud Platform being the least popular of the three with the least services and Microsoft Azure being the middle ground. In the end it comes down to preference when choosing the cloud provider. If something specific is needed, then there might be the need for looking bit more in depth which one to choose.

In Maranet's case the reason for choosing AWS was that AWS was very mainstream at the time. When other businesses starting to migrate towards it, this was a great way to start learning about AWS services.

Because Microsoft Azure and Google Cloud Platform are out of the scope of this thesis, only AWS services are covered.

3 Technologies

Technologies used in the thesis are explained in this chapter. These technologies consist of multiple AWS services, the databases, JavaScript, TypeScript, and CI/CD. These technologies are used in the backend, before and after the migration process.

3.1 Database and MySQL

Applications data is stored in a relational database located in Amazon RDS and it is using MySQL to manage and update the database.

The relational database model means that tables can have connections with other tables. The relationship with tables is established through keys. Every table has a unique primary key which can be used as a mother table's foreign key to connect tables together. A table can have only one primary key, but many foreign keys if it has many relations with children tables. (TechTarget, 2021)

MySQL is an open-source RDBMS (*relation database management system*). MySQL is owned by Oracle. It is the most popular RDBMS compared to its alternatives, like PostgreSQL and MariaDB. It uses SQL to create, update, delete, and retrieve the information from a relational database. (Oracle, 2022)

SQL or Structured Query Language is a query programming language created by IBM for communicating with any SQL compatible database. (Tutorialspoint, n.d)

3.2 Amazon Web Services

3.2.1 AWS Lambda

AWS Lambda is a FaaS (*Function-as-a-Service*) service in Amazon Web Services for executing event-driven Lambda functions. The functions can be triggered by calling them from a web application, a mobile application, or other AWS services. Lambda functions can be written with many programming languages such as JavaScript, Java, python, and many more. Lambda functions are created through the AWS console by configuring the basic information for the function displayed in Figure 1. The code itself is uploaded as a .zip file or retrieved from the S3 location.

Figure 1. Lambda function configuration.

Function name
Enter a name that describes the purpose of your function.

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime [Info](#)
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Architecture [Info](#)
Choose the instruction set architecture you want for your function code.

x86_64
 arm64

Permissions [Info](#)
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

► [Change default execution role](#)

Because of the serverless architecture, Lambda functions are very cost efficient and an easy way to execute a code. AWS Lambda charges by the execution time and lets developers set customizable execution concurrency, and limits the memory used by the function. (Amazon Web Services, 2021-a)

3.2.2 Amazon S3

Amazon S3 or Amazon Simple Storage Service is a storage for storing, retrieving, and analyzing data. S3 has resources known as buckets and objects. The S3 storage is created by creating a S3 bucket and uploading objects to the bucket. Buckets have a maximum data limit of five terabytes. In case of running out of storage, multiple buckets can be used. (Amazon Web Services, 2021-b)

S3 is also used by big companies, since it is much easier, scalable and more cost efficient to let AWS handle storing or backup their data. This is why AWS S3 has big customers such as Autodesk and Siemens. (Amazon Web Services, 2021- c)

3.2.3 Amazon Cloudformation

Amazon Cloudformation is a service for automating and managing the AWS resources. It uses template files written in YAML (*yet another markup language*) or JSON (*JavaScript Object Notation*). Templates are used for specifying which resource is changed and how. Even though JSON is a more common format, YAML is more readable and easier to format which has made it a primary choice for the Cloudformation template files.

The Cloudformation template is deployed by executing change set in AWS Cloudformation console. In case changes want to be reviewed before executing it, “—no-execute-changeset” can be used. (Amazon Web Services, 2021-d)

3.2.4 Amazon Cognito

Amazon Cognito is a user management, an authentication, and an authorization service. When a user signs in through a user pool located in Cognito, if successful it receives a token that can be used for authenticating and authorization for other AWS services. (Amazon Web Services, 2021-e)

3.2.5 Amazon API Gateway

Amazon API Gateway is a service for handling REST, HTTP and WebSocket APIs. (Amazon Web Services, 2021-f)

Software regarding this thesis is using REST API, this means HTTP and WebSocket APIs will not be covered in this section.

Retrieving a data from a database or sending a data to a database starts with a client sending request to an API. When API Gateway receives a request, it handles that request and routes it to an integrated AWS Lambda function through the correct endpoint. After the Lambda function finishes the request, it returns a response back to the client whether request was successful or failed displayed in Figure 2.

If API Gateway route has an authorized attached, it must check if the user making a request has a valid token. It connects to Amazon Cognito through an app client id and checks if the user is authenticated. (Awati, 2021)

Figure 2. HTTP request from the client to retrieve, create or alter data in the database.



3.2.6 Amazon RDS

Amazon Relational Database Service or Amazon RDS is a service for outsourcing relational database to AWS. It is easy to set up and maintain, offering great scalability, security, and automatic patching and back up. Amazon RDS supports the most common database engines, MySQL, PostgreSQL, Amazon Aurora, MariaDB, Oracle and SQL server. (Amazon Web Services, 2021-g)

3.2.7 Amazon Cloudwatch

Amazon Cloudwatch is a logging and monitoring service in Amazon Web Services. It can be set up to monitor AWS resources by creating a log group. It can be also set to monitor resource point of the log groups ARN (Amazon Resource Name). The log groups can be set up to expire from range one day and 120 months or if specified to never expire.

Cloudwatch provides metrics and logs for monitoring execution times, execution durations, errors during executions, executions costs, and many more. A threshold can be specified for Cloudwatch and if the threshold is exceeded, it triggers an alarm and can execute an action. (Amazon Web Services, 2021-h)

3.3 JavaScript

JavaScript is a programming language created for developing more complex web pages by executing scripts on a web browser. It allows the user to interact with the web page and adds features to the web page that HTML or CSS cannot do. (Megida, 2021)

JavaScript was created in 1995 by Netscape and it was originally called, Mocha, then LiveScript and finally JavaScript. Even though JavaScript is named after Java, it does not actually have many similarities with it but was named JavaScript because Java was quite popular at the time of the creation of JavaScript. (DeGroat, 2019)

JavaScript is a good choice for developers with today's technologies. It can be used to create desktop, web, and mobile apps. It has many frameworks to help with these applications, such as React, React Native and Angular. Even though JavaScript was made for the client-side development, it has also gained lot of popularity in the server-side development since Node.js release. (Simplilearn, 2021)

3.3.1 ECMAScript

ECMAScript or ES for short is a standard for scripting languages such as JavaScript. ECMAScript editions are named as ES followed by number like ES1, ES2 etc. (skaytech, 2020). Newest ECMAScript version is ES12, which was released in 2021 June. In case the newest ES version is preferred for use, it can be referred dynamically as ES.Next.

3.3.2 Node.js and NPM

Node.js is a JavaScript runtime environment for executing the code outside of a web browser. It is built on the Chrome's V8 JavaScript engine. Node.js is asynchronous, meaning it can make multiple concurrent API calls, unlike synchronous which must wait for the previous call to have finished before making a new one. (Node.js, n.d)

Node package manager, commonly known as NPM is a package manager for JavaScript. Packages speeds up the developing process with offering already created solutions for technologies used in the project. (NPM, n.d). NPM packages can be installed with `NPM install <package name>`.

To start a Node.js project it must be initialized with `'npm init'` on command line and configure project information displayed in Figure 3. After the configuration it generates `package.json`, containing information about the project, such as installed dependencies and a version number displayed in Figure 4. In case the NPM package is not needed in production build and is only needed in development environment `'npm install'` can be flagged with `'save-dev'`. With the first `npm install` command the project generates a `package-lock.json` file and the `node_modules` folder

containing the defined packages from package.json. “The goal of package-lock.json file is to keep track of exact version of every package that is installed so product is 100% reproducible in the same way even if packages are updated by their maintainers” (nodejs, n.d).

Figure 3. NPM init command

```
package name: (npmtest)
version: (1.0.0)
description: node.js project for test purposes
entry point: (index.js)
test command:
git repository:
keywords:
author: Jani Koskela
license: (ISC)
About to write to C:\Users\JK\Desktop\npmtest\package.json:

{
  "name": "npmtest",
  "version": "1.0.0",
  "description": "node.js project for test purposes",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jani Koskela",
  "license": "ISC"
}
```


Figure 4. package.json file with installed packages.

```
{
  "name": "npmtest",
  "version": "1.0.0",
  "description": "node.js project for test purposes",
  "main": "index.js",
  ▶ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Jani Koskela",
  "license": "ISC",
  "dependencies": {
    "mysql": "^2.18.1"
  },
  "devDependencies": {
    "jest": "^27.3.1"
  }
}
```

3.3.3 Promise and async/await

A promise is an object in asynchronous programming that is expected to have a value in the future, but not at the creation moment. Promises can be resolved using Async/Await which makes code feel more synchronous. Async functions returns a promise either, fulfilled, or rejected and pauses the function to await a promise displayed in Figure 5. (Singh, 2020)

Figure 5. Promise and async await example

```
● 6  v  const promise = new Promise((resolve, reject) => {
7      const condition = true
8  v   if (condition) {
9  v       setTimeout(() => {
10          resolve("promise resolved") // fulfilled
11          }, 3000)
12  v   } else {
13       reject('promise rejected') // rejected
14   }
15   })
16  v  const asyncExample = async () => {
17  v   try {
18       const awaitExample = await promise
19       console.log(awaitExample)
20  v   } catch (err) {
21       console.log(err)
22   }
23   }
24   asyncExample()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ node index.js
promise resolved
```

Promise has three different states:

- Pending: meaning promise has been made
- Fulfilled: meaning promise is kept and returned value
- Rejected: meaning promise has been broken

3.3.4 Deno

Deno is a runtime environment for JavaScript and TypeScript. Just like Node.js it was made by Ryan Dahl, but according to Ryan Dahl, he created Deno to fix what he thought as the biggest weaknesses in Node. (MacManus, 2020)

Some differences with Deno and Node are that with Deno the packages are imported with URL and cached to the hard drive, saving the trouble of using NPM packages and avoiding big node modules folders. Also, Deno is much more secure heavy than Node as it requires permission or use of security flags when executing the scripts. (Sugandhi, 2022)

In the end it is unlikely that Deno would replace Node.js or make Node.js obsolete. They will likely coexist and eventually Deno might be more popular, but that is a long way from now. Node.js is still very popular and in use in many projects. It has plenty of documentation already written, while Deno is still new and this 'cutting edge' technology.

3.4 CI/CD

CI/CD, also known as continuous integration and continuous delivery/deployment, is a way to automate the software delivery and automate running tests. It provides an easy and efficient way to deliver new code.

The first step in the CI/CD is continuous integration. After pushing the code into the code repository, automation server like Jenkins CI could be used for checking the code for errors and give feedback to the developer, about which tests have passed or failed.

The second step can be continuous deployment or continuous delivery. In continuous deployment the process is fully automated when the code is pushed into a repository, meaning that the CI/CD pipeline needs good test automation so the code with bugs or errors does not get to the production build. In continuous delivery there might be a manual step, such as deploying a workflow from GitHub actions to AWS. (Red Hat, 2018)

3.5 TypeScript

TypeScript is a JavaScript superset developed and maintained by Microsoft. TypeScript adds typing to JavaScript. Because JavaScript does not have strict typing, it often has errors or bugs at the runtime and TypeScript removes that possibility. Because of the TypeScript's static checking, it detects errors during the compile-time as displayed in Figure 6. Because of this early error handling, it makes codebase clearer and more robust.

Figure 6. Wrong type in a function call.

```
const typeScriptTestFunction = (stringVariable: string) => {
  console.log(stringVariable)
}
typeScriptTestFunction(1000)
```

Argument of type 'number' is not assignable to parameter of type 'string'. ts(2345)

[View Problem](#) No quick fixes available

TypeScript and JavaScript share syntax and the TypeScript code compiles to JavaScript displayed in Figure 7, so good knowledge of JavaScript helps writing TypeScript. (TypeScript, n.d)

Figure 7. Compiling the TypeScript file and it creates a JavaScript file.

```
TS index.ts > ...
1  const typeScriptTestFunction = (stringVariable: string) => {
2    console.log(stringVariable)
3  }
4  typeScriptTestFunction('Compiling TypeScript it creates JavaScript file')
```

```
JS index.js  x
JS index.js > ...
1  var typeScriptTestFunction = function (stringVariable) {
2    console.log(stringVariable);
3  };
4  typeScriptTestFunction('Compiling TypeScript it creates JavaScript file');
```


But with all these good features that TypeScript has, it must be remembered after compiling TypeScript is just plain JavaScript. Even though it has fewer errors, it is still not 100% error free. (Stempniak & Świstak, n.d)


4 Maranet Application


Maranet is a web application for mobile and desktop. It allows earthwork companies to create orders displayed in Figure 8, on site. Orders can be created based on pre-created or new information.


Addition to creating orders, it can also create Excel and pdf reports for logging and billing purposes displayed in Figure 9. It provides an easy way to assign tasks for the employees withing the business or subcontractors.


Figure 8. Create order page.



Toimintaloki


Kuljetus


Kaivuutyö





Tuntiajo


Tarvikkeet


Henkilötyö

Kirjaa kaivuutyö

Kirjaa kaivuutyön tiedot.

 Jani Koo  1.12.2021  18:07:19

Työksianto

TYÖNTEKIJÄ

Jani Koo ✕

+ Valitse
Syötä käsin

ASIAKAS

Thesis test ✕

+ Valitse
Thesis test ✕

TYÖMAA

Thesis test ✕

+ Valitse
Thesis test ✕

KONE

Thesis test ✕

+ Valitse
Thesis test ✕

LISÄVARUSTE

...

+ Valitse
Syötä käsin

Lisätiedot

Lisätietoja kaivuutyöhön liittyen.

KOMMENTTI

Thesis test ✕

TOIMITUSPÄIVÄ

1.12.2021

ALOITUSAIKA

^ ^
00 : 00

Figure 9. Web page report.

Tietojen vienti
Näyttöraportti
Laskuliite

Näyttöraportti

Näyttöraportti hakee kirjaukset valittujen tietojen perusteella ja listaa ne sivun alalaitaan.

Rajaus:	Määrä <input checked="" type="radio"/>	Aikaväli <input type="radio"/>
Järjestys:	Laskeva <input checked="" type="radio"/>	Nouseva <input type="radio"/>
Määrä:	<input style="width: 100%;" type="text" value="25"/>	
Asiakas:	<input style="width: 100%;" type="text" value="Thesis test"/>	

Hae tiedot

Tiedot	€	Tyyppi	Asiakas	Työmaa	Kirjaaja	Toimitus ▾
↗ Avaa	<input checked="" type="checkbox"/>	Kaivuutyö	Thesis test	Thesis test	Koo Jani	01.12.2021
↗ Avaa	<input type="checkbox"/>	Kuljetus	Thesis test	Thesis test	Koo Jani	27.11.2021

Maranet's frontend is written with JavaScript using ReactJs framework. Backend is made with Node.js with the help of AWS Lambda functions.

Maranet's user authentication is handled with the Amazon Cognito that has user pool with registered users and related information. AWS Lambda functions and applications frontend are stored in S3 buckets.

5 Development

In this chapter I discuss some of my initial thoughts before the migration process. The Lambda function and custom-made packages are explained here as well. The typing process consists of installing the TypeScript, creating types for the code, and installing pre-made types.

After the typing process, the automated Lambda function deployment script must be modified for TypeScript. The script should compile the typed file and deploy that file to AWS Lambda successfully.

After the successful deployment, the function will be tested with Insomnia. It should return the successfully created user object in database and user in Amazon Cognito user pool.

After the development process, I will share my thoughts about the problems in the process. I will also discuss some of the benefits and downsides about the migration.

5.1 My initial thoughts

Thesis's development section includes some research and some functional parts. Before I start the migration of the backend, there are some questions I hope to have an answer and the end of the thesis

- Is TypeScript worth it in the project this size or should it be used only in bigger applications?
- Does the backend use the TypeScript for all its potential?
- Are interfaces and classes worth using?

Even if the answer for all these questions were no, is the clearer codebase alone worth the migration? With the clearer codebase it should help new developers study the code faster.

5.2 Lambda function

AWS Lambda handler flow, displayed in Figure 10, starts when a client-side request is made and contains the request body. The request body is validated with the custom made validate input package displayed in Figure 11. After validating the request body, the code execution continues to the user authorization with a custom made get user role package that fetches roles and allowed actions for the role, from the database and returns object displayed in Figure 12 to the handler function. After the authorization is successful, the handler calls aws-sdk built in method signUp and creates user to the Cognito user pool. If the user creation in Cognito is successful, Cognito sends response containing a tid which is used as the primary key in addUserToDb function, that inserts user to database table. After the successful database operation, the function returns promise to the handler function which is resolved and returned as a JSON string containing the user information to the client-side.

Figure 10. Lambda handler function flowchart.

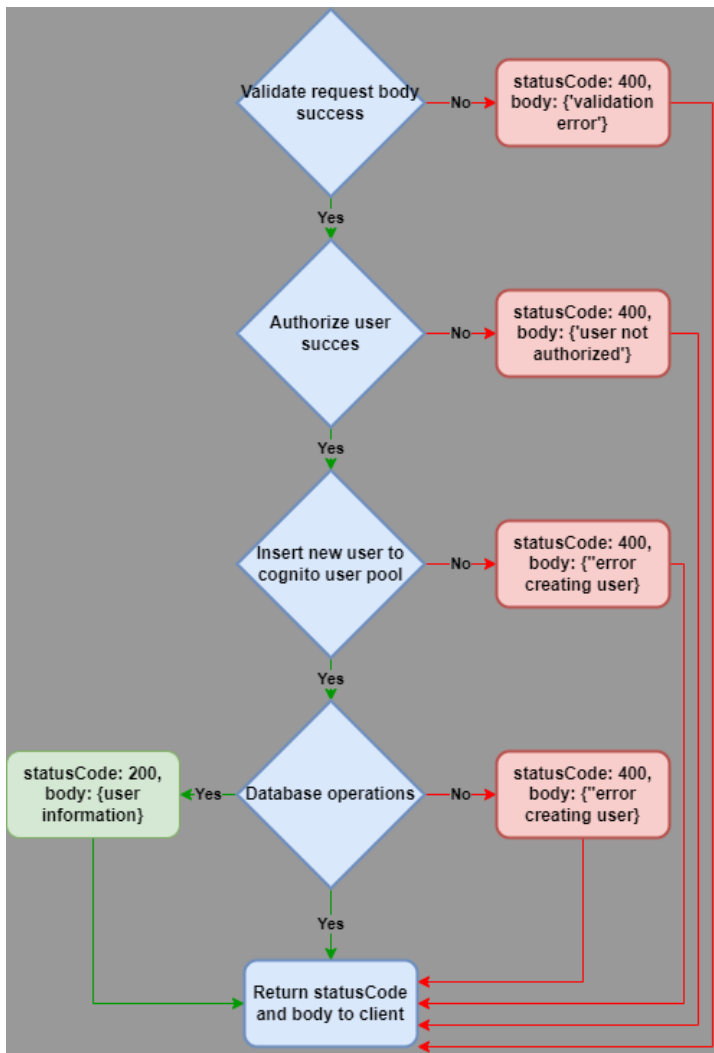


Figure 11. Validate input function call.

```

await validate_input({
  email: reqBody.email,
  name_first: reqBody.firstname,
  name_last: reqBody.lastname,
  phone_number: reqBody.phone_number,
  password: reqBody.password,
  entry_item_id: reqBody.role,
  id_array: reqBody.usergroups
})
  
```

Figure 12. Get user role return object

```
return {  
  cognito_id: results.cognito_id,  
  aspnet_id: results.aspnet_id,  
  business_id: results.business_id,  
  role_ids: rolesResults.map(item => item.role_id),  
  firstname: results.firstname,  
  lastname: results.lastname,  
  phone: results.phone,  
  business_name: results.business_name,  
  default_usergroup_id: results.default_usergroup_id,  
  default_customer_group_id: results.default_customer_group_id,  
  actions: actions.map(item => item.action)  
}
```

To be noted that if the user does not have a valid Cognito token and the auth is set in AWS API Gateway for that route, the Lambda function will not trigger because the API Gateway stops it.

The first argument of the handler function displayed in Figure 13, is an event which has information about the event that triggered the invocation. The second argument is context which has information about the invocation, the function, and the execution environment. (Amazon Web Services, 2021-i)

Figure 13. Handler function

```

const AWS_main = async (event, _context) => {
  try {

    const reqBody = typeof (event.body) === 'string' ? JSON.parse(event.body) : event.body
    const reqHeaders = event.headers || {}
    console.log('BODY', reqBody)
    console.log('HEADERS', reqHeaders)

    const settings = get_settings()

    > await validate_input({...
    })

    let cognito = new AWS.CognitoIdentityServiceProvider()

    > const params = {...
    }

    const user = await get_user_role(reqHeaders.authorization, settings.db_port, settings.db
    console.log('USER', user)

    let cognitoUser
    let data
    if (user.actions.includes('create_user')) {
      cognitoUser = await cognito.signUp(params).promise()
      data = await add_user_to_db(settings, reqBody, user, cognitoUser.UserSub)
    } else {
      throw new Error('User not authorized')
    }
    return {
      statusCode: 200,
      body: JSON.stringify(data)
    }
  }

  catch (error) {
    console.error(error)
    return {
      statusCode: 400,
      body: JSON.stringify({
        message: 'Error creating user'
      })
    }
  }
}

```

5.3 CI/CD pipeline

CI/CD uses GitHub actions, meaning every Lambda function has its own workflow which is GitHub's equivalent for the CI/CD job. When the Lambda function is ready to be deployed and executed in the client-side, Lambda function's version number must be changed for the

Cloudformation to recognize the changes and update the stack. The deployment happens in GitHub actions. The Lambda function's deploy environment must be specified with test, dev, or prod and GitHub secrets must be configured. Lambda function's Cloudformation template can retrieve a needed configuration parameters for the function.

5.4 Installing TypeScript and type packages

The original plan is that TypeScript is not needed when executing function in AWS Lambda. Since the TypeScript file compiles to plain JavaScript in the end, it can be installed as a development dependency using `--save-dev` flag along with the `npm install` command. With successful installation it should now exist under the `devDependencies` in the `package.json` file.

TypeScript's compiler options can be modified in `tsconfig.json` displayed in figure 14. Mostly default settings can be used, but in the config file couple of settings are changed. The target is set as "ESNext" this means that compiler uses the newest ECMAScript standard that this version of TypeScript supports.

Figure 14. tsconfig file

```
{
  "compilerOptions": {
    "target": "ESNext",
    "module": "commonjs",
    "moduleResolution": "node",
    "alwaysStrict": true,
    "allowUnusedLabels": false,
    "allowUnreachableCode": false,
    "noImplicitAny": true,
    "noImplicitReturns": true,
    "noImplicitThis": true,
    "noFallthroughCasesInSwitch": true,
    "noUnusedParameters": true,
    "strictFunctionTypes": true,
    "strictNullChecks": true,
    "strictPropertyInitialization": false,
    "allowJs": false,
    "noUnusedLocals": true,
    "outDir": "dist"
  }
}
```

Setting `noImplicitAny` is set as `true`, so the code always throws an error if “any” is used as a type. the `outDir` settings specifies the output directory where the `index.js` will be compiled to.

Along with TypeScript there is couple good packages from DefinitelyTyped for Node.js and AWS Lambda, so using `npm i @types/node --save-dev` and `npm i @types/aws-lambda`. Those two packages have prebuilt types for easier typing.

5.5 Typing

This section goes through typing process. Typing was done using classes, interfaces, and pre-made types from npm packages.

5.5.1 Classes

Request body and environment settings function types were set within a class and exported from their own files displayed in Figure 15.

Figure 15. Classes

```

requestBody.ts M x
classes > requestBody.ts > ...
1  export class RequestBody {
2      email: string
3      firstname: string
4      lastname: string
5      phoneNumber: string
6      password: string
7      role: string
8      usergroups: Array<number>
9
10     constructor(
11         email: string,
12         firstname: string,
13         lastname: string,
14         phone: string,
15         password: string,
16         role: string,
17         groups: Array<number>
18     ) {
19         this.email = email
20         this.firstname = firstname
21         this.lastname = lastname
22         this.phoneNumber = phone
23         this.password = password
24         this.role = role
25         this.usergroups = groups
26     }
27 }

settings.ts x
classes > settings.ts > Settings > constructor
You, last month | 1 author (You)
1  export class Settings {
2      clientId: string | undefined
3      dbPort: string | undefined
4      dbHost: string | undefined
5      dbUser: string | undefined
6      dbPassword: string | undefined
7      dbName: string | undefined
8
9     constructor(
10         clientId: string | undefined,
11         dbPort: string | undefined,
12         dbHost: string | undefined,
13         dbUser: string | undefined,
14         dbPassword: string | undefined,
15         dbName: string | undefined
16     ) {
17         this.clientId = clientId
18         this.dbPort = dbPort
19         this.dbHost = dbHost
20         this.dbUser = dbUser
21         this.dbPassword = dbPassword
22         this.dbName = dbName
23     }
24 }

```

In Settings where the environment settings are typed, it is unknown if parameter is string or undefined so it must receive both types with '||' character.

Classes are imported in the index.ts file displayed in Figure 16. The new instance of the class is created, and the object properties passed as arguments to the classes displayed in Figure 17 and Figure 18.

Figure 16. Class imports

```
import { RequestBody } from './classes/RequestBody'  
import { Settings } from './classes/Settings'
```

Figure 17. New instance of request body

```
const body = new RequestBody(  
  reqBody.email,  
  reqBody.firstname,  
  reqBody.lastname,  
  reqBody.phoneNumber,  
  reqBody.password,  
  reqBody.role,  
  reqBody.usergroups  
)
```

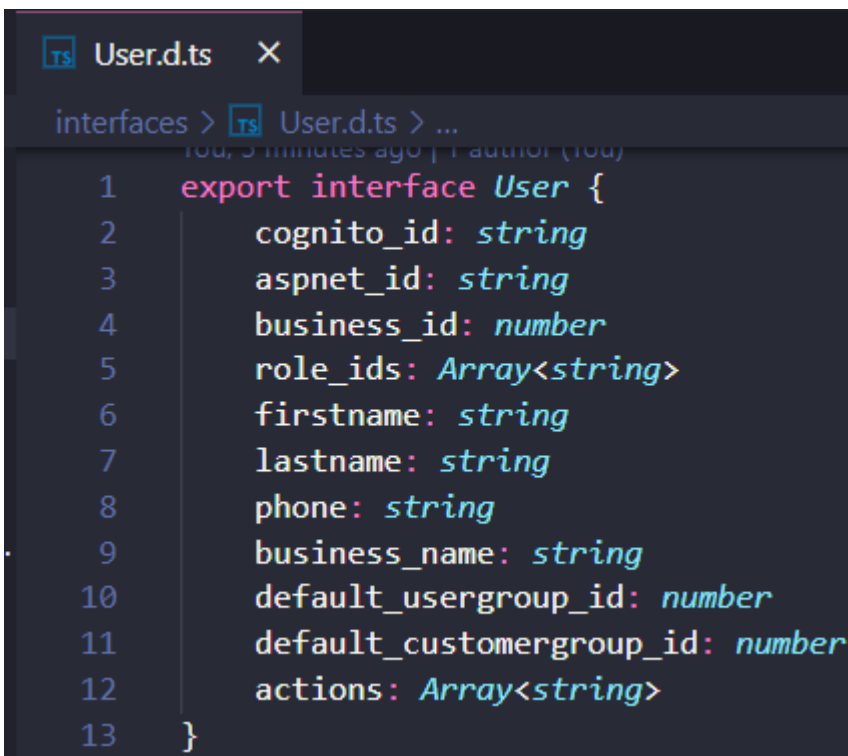
Figure 18. New instance of settings

```
const settings = new Settings(  
  process.env.client_id,  
  process.env.db_port,  
  process.env.db_host,  
  process.env.db_user,  
  process.env.db_password,  
  process.env.db_name  
)
```


5.5.2 Interface

Another way to type an object is an interface which is used for `getUserRole` function's return object. It is defined in its own `User.d.ts` file and it is purely used for typing and does not compile to JavaScript unlike classes. Looking at the return object, which was already displayed in Figure 12, it is easy to set types for all properties of the object. After creating the interface, it is exported from `User.d.ts` displayed in Figure 19 and imported in the `index.ts` file displayed in Figure 20.

Figure 19. `User.d.ts` file and `User` interface



```
TS User.d.ts X
interfaces > TS User.d.ts > ...
1 export interface User {
2     cognito_id: string
3     aspnet_id: string
4     business_id: number
5     role_ids: Array<string>
6     firstname: string
7     lastname: string
8     phone: string
9     business_name: string
10    default_usergroup_id: number
11    default_customer_group_id: number
12    actions: Array<string>
13 }
```

Figure 20. Importing `User` interface

```
import { User } from './interfaces/User'
```

After successful interface import, the interface is used as a type for the function call return displayed in Figure 21.

Figure 21. User interface used for type definition

```
const user: User = await getUserRole(
  reqHeaders.authorization,
  settings.dbPort,
  settings.dbHost,
  settings.dbUser,
  settings.dbPassword,
  settings.dbName
)
```

5.5.3 Pre-made types from packages

Npm package `@types/node` provides many type definitions for Node.js and it automatically sets type for 'require' displayed in Figure 22. When using TypeScript with Node.js, `@types/node` are essential to install.

Figure 22. NodeRequire type from `@types/node`

```
You, 14 seconds ago | 1 author
'use strict'
const AWS = require(
  var require: NodeRequire
  (id: string) => any
```

When typing the AWS Lambda handler function, `@types/aws-lambda` provides many good types for the handler function parameters and the handler function return. For the parameters, event and context, the `APIGatewayProxyEvent` and the `Context` types are imported from the `aws-lambda` package and the `APIGatewayProxyResults` for the function return.

For `aws-sdk`'s own method for creating the user to Cognito user pool, `aws-sdk` has built in type `SignUpResponse` displayed in Figure 23.

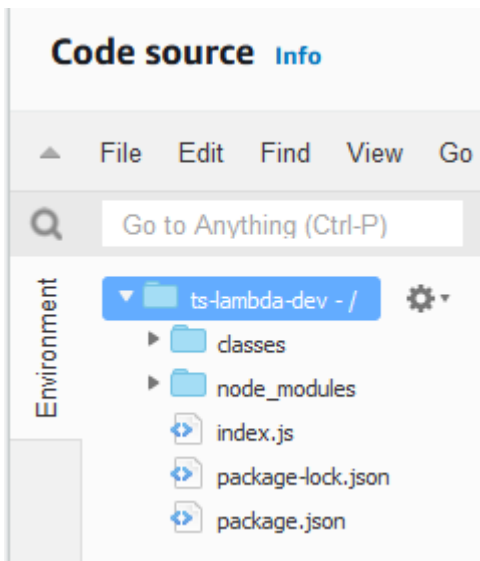
Figure 23. SignUpResponse type

```
export interface SignUpResponse {  
  /**  
   * A response from the server indicating that a user registration has been confirmed.  
   */  
  UserConfirmed: BooleanType;  
  /**  
   * The code delivery details returned by the server response to the user registration request.  
   */  
  CodeDeliveryDetails?: CodeDeliveryDetailsType;  
  /**  
   * The UUID of the authenticated user. This isn't the same as username.  
   */  
  UserSub: StringType;  
}
```

5.6 Deploying function with CI/CD

Because the deployment script used in CI/CD is purely for deploying functions' written in JavaScript, the script must be modified. The original folder structure included the test files, which can be easily removed when compiling the files. When deploying the function, it requires the development dependencies, so the file can be compiled to JavaScript using TypeScript compiler. The JavaScript file is compiled to dist folder, and package.json and package-lock.json will be copied to folder. Node_modules are installed during the script. First scripts move to dist folder and then it executes command 'npm install --only=prod' meaning it only installs production packages, avoiding the development ones. After the successful installation, the script uploads the folder to AWS Lambda displayed in figure 24.

Figure 24. TypeScript function in AWS Lambda



The function deployment also creates a route to Amazon API Gateway. Amazon API Gateway provides an authorizer for the route, so if the API call comes from different user than one in the the Amazon Cognito user pool, it gives an 'Unauthorized' error.

Amazon Cloudformation creates a stack for the Lambda function and shows timestamp for every step during the stack creation. It can be viewed in AWS console and in the function's Cloudformation stack events displayed in figure 25.

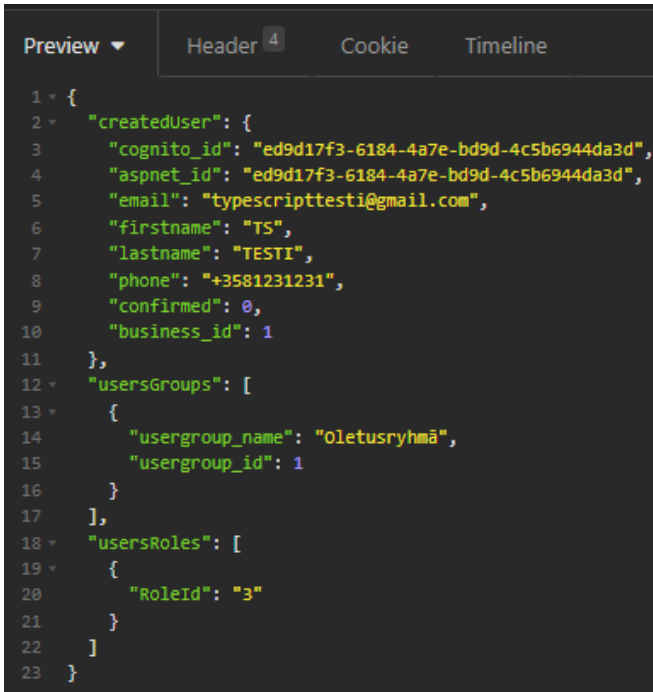
Figure 25. TypeScript Lambda function stack events

Events (15)				
<input type="text" value="Search events"/>				
Timestamp	Logical ID	Status	Status reason	
2022-05-22 22:20:41 UTC+0300	maranet-ts-lambda-dev	CREATE_COMPLETE	-	
2022-05-22 22:20:40 UTC+0300	LambdaFunctionInvokePermission	CREATE_COMPLETE	-	
2022-05-22 22:20:33 UTC+0300	ApiRoute	CREATE_COMPLETE	-	
2022-05-22 22:20:33 UTC+0300	ApiRoute	CREATE_IN_PROGRESS	Resource creation Initiated	
2022-05-22 22:20:32 UTC+0300	ApiRoute	CREATE_IN_PROGRESS	-	
2022-05-22 22:20:30 UTC+0300	ApiIntegration	CREATE_COMPLETE	-	
2022-05-22 22:20:30 UTC+0300	ApiIntegration	CREATE_IN_PROGRESS	Resource creation Initiated	
2022-05-22 22:20:30 UTC+0300	LambdaFunctionInvokePermission	CREATE_IN_PROGRESS	Resource creation Initiated	
2022-05-22 22:20:29 UTC+0300	LambdaFunctionInvokePermission	CREATE_IN_PROGRESS	-	
2022-05-22 22:20:29 UTC+0300	ApiIntegration	CREATE_IN_PROGRESS	-	
2022-05-22 22:20:28 UTC+0300	LambdaFunction	CREATE_COMPLETE	-	
2022-05-22 22:20:20 UTC+0300	LambdaFunction	CREATE_IN_PROGRESS	Resource creation Initiated	
2022-05-22 22:20:19 UTC+0300	LambdaFunction	CREATE_IN_PROGRESS	-	
2022-05-22 22:20:15 UTC+0300	maranet-ts-lambda-dev	CREATE_IN_PROGRESS	Transformation succeeded	
2022-05-22 22:20:12 UTC+0300	maranet-ts-lambda-dev	CREATE_IN_PROGRESS	User Initiated	

5.7 Testing the Lambda function

The lambda function is tested with the Insomnia REST API client. The Lambda function requires a JSON object for request body. So, using the correct endpoint and parameters in object, user creating is successful and the function returns the created user as a response and creates the user in Amazon Cognito user pool displayed in Figure 26 and Figure 27.

Figure 26. Created user



```
1 {
2   "createdUser": {
3     "cognito_id": "ed9d17f3-6184-4a7e-bd9d-4c5b6944da3d",
4     "aspnet_id": "ed9d17f3-6184-4a7e-bd9d-4c5b6944da3d",
5     "email": "typescripttesti@gmail.com",
6     "firstname": "TS",
7     "lastname": "TESTI",
8     "phone": "+3581231231",
9     "confirmed": 0,
10    "business_id": 1
11  },
12  "usersGroups": [
13    {
14      "usergroup_name": "Oletusryhmä",
15      "usergroup_id": 1
16    }
17  ],
18  "usersRoles": [
19    {
20      "RoleId": "3"
21    }
22  ]
23 }
```

Figure 27. Created user in Amazon Cognito user pool

ed9d17f3-6184-4a7e-
bd9d-4c5b6944da3d

Enabled UNCONFIRMED

typescripttesti@gmail.com

Lambda execution can be monitored in Amazon Cloudwatch. It provides information about the execution time and things logged in the code displayed in Figure 28.

Figure 28. Lambda execution in Amazon Cloudwatch

```

Message

No older events at this moment. Retry

START RequestId: ff809175-872a-48ed-b6a9-c8370e7e9b21 Version: $LATEST
2022-05-30T16:10:31.182Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO BODY { email: 'typescripttesti@gmail.com', firstname
2022-05-30T16:10:31.210Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO HEADERS { accept: '*/*', authorization: 'eyJraWQiOiJ
2022-05-30T16:10:31.230Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO typescripttesti@gmail.com
2022-05-30T16:10:33.090Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO connection closed
2022-05-30T16:10:33.091Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO USER { cognito_id: 'ae940676-50e8-4c13-ba42-80771040
2022-05-30T16:10:33.983Z ff809175-872a-48ed-b6a9-c8370e7e9b21 INFO Connection closed
END RequestId: ff809175-872a-48ed-b6a9-c8370e7e9b21
REPORT RequestId: ff809175-872a-48ed-b6a9-c8370e7e9b21 Duration: 2806.60 ms Billed Duration: 2807 ms Memory Size: 128 M
No newer events at this moment. Auto retry paused. Resume

```

6 Conclusion

The custom-made package for retrieving the user's role should have been typed already in itself. When the typing is done to more than one function, it adds repetition to code and takes more time for no reason. But the user's role package typing was done in the code itself to stay in the scope of the thesis.

In this case, using classes do not bring many more benefits than using interfaces. I used classes for some objects and interfaces on some. In my opinion, interfaces would have worked just as well, because in the code we care purely about the types. In case objects were built in the backend, classes could have more impact.

In the end TypeScript made the code base more robust and clearer. It is a great way to see that the objects are correct, and it makes sure all the parameters are in the object on compile time. It provides help for new developers jumping in midway, to figure out the objects faster without needing to study the database or the frontend code too much when data types are already written down. In my opinion, if the backend is well made, the size of project should not matter when typing the code. It does not require too much extra time to add types where needed.

I did convert unit tests to TypeScript file but did not see the benefits to add types for them. In this case the request body is already typed and validated withing the handler, so testing that in unit tests would not bring that many more benefits.

References

Canalys. (29.7.2021). *Global cloud services market Q2 2021*.

<https://www.canalys.com/newsroom/global-cloud-services-q2-2021>

Intellias, (25.4.2021). *On-Premises vs Cloud Computing: Pros, Cons, and Cost Comparison*

<https://intellias.com/cloud-computing-vs-on-premises-comparison-guide/>

BMC. (1.10.2021). *AWS vs Azure vs GCP: Comparing The Big 3 Cloud Platforms*

<https://www.bmc.com/blogs/aws-vs-azure-vs-google-cloud-platforms/>

TechTarget. (2021, June). *Relational Database*

<https://www.techtarget.com/searchdatamanagement/definition/relational-database>

Tutorialspoint. (n.d). *SQL – Overview*

<https://www.tutorialspoint.com/sql/sql-overview.htm>

Oracle. (2022). *What is MySQL?*

<https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html>

Red Hat. (31.10.2017). *What is serverless*.

<https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless#overview>

Red Hat. (31.1.2018). *What is CI/CD*.

<https://www.redhat.com/en/topics/devops/what-is-ci-cd>

Kemper, F. (11.8.2021). *On-Premise vs Cloud: Advantages and Disadvantages*

<https://www.empowersuite.com/en/blog/on-premise-vs-cloud>

Cloudflare. (n.d-a) *Why use serverless? | Pros and cons of serverless*

<https://www.cloudflare.com/learning/serverless/why-use-serverless/>

Cloudflare. (n.d-b) *Why use serverless? | Pros and cons of serverless*

<https://www.cloudflare.com/learning/serverless/why-use-serverless/>

Amazon Web Services. (2021-a) *AWS Lambda*. <https://aws.amazon.com/lambda/>

Amazon Web Services. (2021-b) *Amazon S3*. <https://aws.amazon.com/s3/>

Amazon Web Services. (2021-c) *Amazon S3 customers*.

<https://aws.amazon.com/s3/customers/>

Amazon Web Services. (2021-d) *AWS Cloudformation*

<https://aws.amazon.com/cloudformation/>

Amazon Web Services. (2021-e) *Amazon Cognito*

<https://aws.amazon.com/cognito/>

Amazon Web Services. (2021-f) *Amazon API Gateway*.

<https://aws.amazon.com/api-gateway/>

Amazon Web Services. (2021-g) *Amazon Relational Database Service (RDS)*.

<https://aws.amazon.com/rds/>

Amazon Web Services. (2021-h) *Amazon Cloudwatch*.

<https://aws.amazon.com/cloudwatch/>

Amazon Web Services. (2021-i) *AWS Lambda function handler in Node.js*

<https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html>

Awati, R. (2021, June). *Amazon Cognito*. Retrieved 21.11.2021 from:

<https://searchaws.techtarget.com/definition/Amazon-Cognito>

Megida, D. (29.3.2021). *What is JavaScript? A Definition of the JS Programming Language*.

<https://www.freecodecamp.org/news/what-is-javascript-definition-of-js/>

DeGroat, T.J. (19.8.2019). *The History of JavaScript: Everything You Need to Know*

<https://www.springboard.com/blog/data-science/history-of-javascript/>

Simplilearn. (28.10.2021). *Top 10 Reasons to Learn JavaScript*.

<https://www.simplilearn.com/reasons-to-learn-javascript-article>

skaytech. (14.6.2020). *History of ECMA (ES5, ES6 & Beyond!)*

<https://dev.to/skaytech/history-of-ecma-es5-es6-beyond-lpe>

NPM. (n.d) *About npm*. <https://www.npmjs.com/about>

Node.js. (n.d) *About Node.js*. <https://nodejs.org/en/about/>

MacManus, R. (2.11.2020). *How Node.js Is Addressing the Challenge of Ryan Dahl's Deno*

<https://thenewstack.io/how-node-js-is-addressing-the-challenge-of-ryan-dahls-deno/>

Sugandhi, A. (1.1.2022). *What is Deno, and Difference Between Deno & Node.js*

<https://www.knowledgehut.com/blog/web-development/what-is-deno-difference-between-deno-nodejs>

TypeScript. (n.d) *Typescript for the New Programmer*

<https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>

Imjoff, J. (11.3.2012). *How to explain callback in plain english? How are they different from calling one function from another function?*

<https://stackoverflow.com/questions/9596276/how-to-explain-callbacks-in-plain-english-how-are-they-different-from-calling-o/9652434#9652434>

Stempniak, A. & Świstak, T. (n.d) *What is TypeScript? Pros and Cons of TypeScripts vs Javascript.*

<https://www.stxnext.com/blog/typescript-pros-cons-javascript/>

Singh, A. (27.8.2020) *Callback vs Promises vs Async/Await*

<https://www.loginradius.com/blog/async/callback-vs-promises-vs-async-await/>