Higher Education Diploma in Software Engineering with emphasis
in Web Programming
September 2022

Creating a Serverless Application Using the Serverless Framework and React

Deploying a serverless back-end to different cloud providers

Emma Edlund

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Higher Education Diploma in Software Engineering with emphasis in Web Programming. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author:
Emma Edlund
E-mail: emeu17@student.bth.se

University advisor:
Emil Folino
Department of Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Serverless applications have become more popular in recent years. Server configuration and usage is hidden away from the developer. It is marketed as a way of decreasing development time and decrease the complexity in development. The application is automatically scaled based on usage and the cost is based on actual time and amount of resources run. A framework can be used to speed up development as the cloud infrastructure can be treated as code. It makes it easier to deploy the same application to multiple serverless providers and simplify the deployment process regardless of underlying platform.

The terms Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) are two common terms used in the field of serverless. FaaS divides an application into functions, making it possible to re-deploy a single function without having to re-deploy the application as a whole. BaaS services offered by serverless providers, for example authentication and databases, can speed up development of an application.

This thesis investigates the differences between serverless and traditional server-oriented development. An empirical study is conducted where the Serverless Framework is used to deploy a serverless application to three providers; AWS, Azure and Google. It investigates the possibility to utilize BaaS services compared to third party solutions for database and authentication. The thesis aims to find out which provider is most suitable for a smaller back-end API that connects to a front-end React application. By comparing similarities and differences in configuration and deployment it aims at answering to what extent code can be reused between the serverless providers.

The result shows that the Serverless Framework by default is aimed at AWS. The Azure and Google project requires a function plugin to work with each providers functions and events. In AWS a back-end application was created which contained the BaaS services DynamoDB for database and Cognito User Pool for authentication. Azure and Google documentation described very few BaaS services and it was not possible to implement any BaaS services. MongoDB Atlas was used to create a third party solution to a database. This was implemented on the Azure and Google project. JWT Token and bcrypt were used to handle authentication in a non-proprietary way and this solution was implemented in AWS.

The thesis concludes that AWS is the most suitable cloud provider to create a smaller back-end API through the Serverless Framework. AWS has the most extensive documentation and provides support for most events. BaaS services can make development faster but does require more vendor specific configurations. This makes it difficult to reuse code between providers. Using the non-proprietary solutions JWT token and MongoDB makes reuse of code quite simple and there are only minor configurations that differ between the providers.

Keywords: Serverless, Serverless Framework, AWS, Azure, Google

Contents

Abstract **i**

1 Introduction **2**
1.1 Background . . . 2
1.1.1 The Serverless Framework . . . 3
1.2 Scope of the thesis . . . 3
1.3 Motive and value . . . 4
1.4 Outline . . . 4

2 Research Questions **5**
2.1 Research Question 1 . . . 5
2.2 Research Question 2 . . . 5
2.3 Research Question 3 . . . 5
2.4 Research Question 4 . . . 6

3 Method **7**
3.1 Research questions . . . 7
3.2 Literature search . . . 7
3.3 Empirical study . . . 8
3.3.1 Serverless Framework . . . 8
3.3.2 React application . . . 9
3.3.3 Limitations . . . 10

4 Literature Review **12**
4.1 Introduction . . . 12
4.2 Earlier theses works . . . 12
4.3 What is serverless? . . . 12
4.4 How to use serverless? . . . 13
4.5 Strengths and limitations . . . 14
4.5.1 Function versus application . . . 14
4.5.2 Scalability and cost . . . 15
4.5.3 Managed services . . . 15
4.5.4 Developer velocity . . . 15
4.5.5 Cold starts . . . 16
4.5.6 Security and permissions . . . 16
4.5.7 Other concerns . . . 16

Abstract 1

5 Results and Analysis **17**
5.1 Serverless framework implementation 17
5.1.1 General 17
5.1.2 AWS 18
5.1.3 Azure 22
5.1.4 Google 25
5.2 Non-proprietary implementations 27
5.2.1 Authentication 28
5.2.2 Database 29
5.3 Research Question 1 30
5.4 Research Question 2 31
5.5 Research Question 3 34
5.6 Research Question 4 35

6 Conclusions and Future Work **38**
6.1 Conclusions 38
6.2 Future Work 39

Bibliography **39**

Chapter 1

Introduction

The serverless concept is marketed as a way of decreasing development time of an application and at the same time decrease the complexity in development. Many of the stated benefits of going serverless, such as scalability and pay-as-you-go, are advantages when choosing serverless for a smaller back-end application. Both Function-as-a-Service (FaaS) and Backend-as-a-Service (Baas) are concepts that might take some time to get familiar with for developers new to serverless. FaaS means that the application is divided up into smaller functions which usually serve a single purpose each - such as creating a post in a database. These functions can much easier be re-deployed independently. Using BaaS services from the serverless providers means less coding and the possibility to use the expertise from the provider to set up things such as databases and user authentication. It does however create a vendor lock-in where it might be difficult to switch to another provider. The required configurations and knowledge in the services used will be provider specific and switching to another provider would require learning a new service and configurations.

1.1 Background

Serverless is a term that has become more popular in recent years [43] [37]. It describes a new way of publishing applications where the server usage is hidden away from developers. Code is run on-demand and automatically scaled based on usage. Billing is based on the actual time the code is running. Deployment, resource allocation and autoscaling are all left to the cloud resource provider. Some common providers of serverless computing are Amazon Web Services (AWS), Microsoft Azure and Google Cloud [20].

Two common terms within serverless are FaaS and BaaS. BaaS gives developers access to provider-managed services such as authentication, cloud-accessible databases and file storage. FaaS allows developers to deploy their own code (functions) on servers or containers managed by the cloud provider. These two terms are sometimes used intertwined but refer to the degree of control and flexibility the developer has over the deployed code. BaaS results in less control over the code used and a greater amount of vendor lock-in but could make development of an application faster [45].

It can be quite complex to set up and configure a serverless application as the learning curve to use the providers resources can be quite steep. Configuration of the serverless application could in worst case scenario be more complex than the actual application [38]. There is also an increased vendor lock-in which makes it important

to carefully choose a provider.

1.1.1 The Serverless Framework

A framework can make deployment of serverless applications easier [75] [38]. The Serverless Framework is a free, open-source framework launched in 2015. At first it only supported deployment to AWS but can nowadays deploy the same code to many different serverless providers including Google Cloud and Microsoft Azure [18]. By writing a single configuration file the framework creates the cloud infrastructure dependencies described in the file. Infrastructure can be treated as code which makes scaling easier and faster to implement. Using the Serverless Framework can be a way to simplify and standardize the configuration for serverless microservices [39]

It can be a steep learning curve for developers new to working with deployments in the cloud as it requires other types of configurations and ways of structuring the code [18]. Also, as soon as the deployment has been made and technologies has been adopted to the provider in question, it is difficult to change provider [74]. The Serverless Framework adds an abstraction layer on top of the serverless platforms. This is a step towards simplifying the deployment process regardless of underlying platform and a move towards a more standardized way of creating serverless applications in the future [2].

The Serverless Framework is in this thesis used to create a backend API using Node.js. In order to produce a complete web application the React framework is used to create a front-end web application. Javascript is thus used for both the back-end and front-end development. Because React is the most used Javascript framework [73] it was chosen to build the front-end. The front-end will interact with the serverless back-end API through HTTP requests.

1.2 Scope of the thesis

This report will focus on describing the difference between serverless and traditional server-oriented application development. By building a back-end application on different cloud providers, containing the same routes and services, it also aims at finding out which one is the most suitable option for those with little experience in serverless applications. The providers in question are three of the largest serverless providers [74]; AWS, Google Cloud and Microsoft Azure. The Serverless Framework will be used to deploy the code to the providers. Even though parts of the code can be reused across platforms the necessity of configuring vendor-specific functions still remains. Each provider also has its own set of BaaS services that will be used for authentication and database storage. The usage of BaaS will be analyzed in comparison to using third party options for authentication and database storage.

Creating an account and managing security configuration is still required at all three vendors but deployment should occur through the Serverless Framework. Generous permissions (administrator/contributor) will be given to the Serverless Framework to make deployment as easy as possible. Restricting access to minimum requirements would in reality be something to further consider, in order to keep the application as secure as possible.

The React Application will be a basic web application with connection to the back-end and possibility for sign up and signing in users and saving data to a database. Extra services will not be considered.

1.3 Motive and value

The interest in serverless applications has increased and so has the number of providers offering these kind of services [74]. The product is marketed as a way of decreasing development time of an application and at the same time decrease the complexity in development [20]. This report will examine if the techniques has become mature enough for smaller application development and if it has a chance of replacing traditional back-end development. It will include practically implementing serverless applications. As mentioned in the article *Serverless Applications: Why, When, and How?* [20] there is a need for more empirical studies about serverless use to guide software developers in building serverless solutions.

Lately serverless frameworks have gained popularity and decreased the threshold for serverless configurations. However, the question remains if the threshold for setting up the FaaS and BaaS within a serverless framework is still too steep to be feasible for smaller applications/projects? The report can be of value for anyone considering the usage of serverless. It will investigate if any of the providers is worth investing development time and money in.

1.4 Outline

This thesis is built around the four research questions introduced in Chapter 2. The method in Chapter 3 describes how the research should be conducted in order to answer the questions; through a literature review and an empirical study. The literature review defines the term serverless and puts it in a perspective; when and how could it be used? It is also input to research question 1 - how serverless differs from server-oriented solutions. The empirical study in Chapter 5 utilizes the Serverless Framework to set up serverless applications in AWS, Azure and Google. It analyses the Serverless Framework documentation and the implementations to find out if code can be re-used between the different providers. It also aims at finding out which provider is most suitable for developers new to the serverless development and how the providers implementation in the Serverless Framework differ from each other. Finally the result is summarized and conclusions are disclosed in Chapter 6.

Chapter 2

Research Questions

This thesis focuses on introducing serverless application basics and exploring the Serverless Framework. It aims at comparing deployments made to the three serverless providers: AWS, Azure and Google. A back-end API will be created in the Serverless Framework utilizing the same type of services from different providers.

2.1 Research Question 1

What are the differences between serverless and traditional server-oriented solutions for a web application?

A literature study will be conducted in order to investigate the area of serverless. It intends on finding out which strengths and limitations serverless has compared to traditional server-oriented development. The empirical study is then used as input to discuss how the literature findings correlate to the practical work implemented in this thesis.

2.2 Research Question 2

What are the main differences between the chosen serverless providers Amazon Web Service Lambda, Google Cloud Functions and Microsoft Azure Functions in the Serverless Framework in regard to Function syntax, BaaS services and price?

Serverless applications are built on FaaS and even though the Serverless Framework is used there are differences in code structure when deploying functions to the different providers. What are common denominators and what differences exists? The selection of BaaS and costs of running a serverless application differ between the providers, this will be investigated as it could be a reason to chose a specific provider. The research question will be answered through the empirical study and with data from each provider.

2.3 Research Question 3

Without earlier experience in serverless – which of the providers is most suitable for new development of a smaller back-end API, to connect to a front-end React application, using the Serverless framework?

- *Easiest to get started, least configuration*
- *The application should contain authentication and a database. It will investigate the possibility to use BaaS for these services as well as non-proprietary options.*

This report will through an empirical study examine if the techniques has become mature enough for smaller application development and if it has a chance of replacing traditional back-end development. By comparing vendor specific BaaS services and non-proprietary solutions for database and authentication it will also explore the advantages and disadvantages of both options.

2.4 Research Question 4

Does the Serverless Framework support easy reusage of code across different serverless providers and if so to what extent?

Being able to deploy the same application to different providers decreases vendor lock-in. This thesis investigates the possibility to deploy the same back-end application to the three different providers; AWS, Azure and Google. The empirical study will be the foundation to examine what code needs to be rewritten in the back-end and what code can be reused between the providers when using the Serverless Framework. The back-end code contains both functions and configuration files which will likely both need to be altered between the providers.

Chapter 3

Method

3.1 Research questions

The research questions will be answered through a literature study and practical implementation of serverless applications in the Serverless Framework.

A literature study will be conducted in order to investigate the area of serverless and answer the **RQ1** comparing serverless and traditional server-oriented development. **RQ2**, comparing the three cloud providers on the Serverless Framework, will in part be answered by comparing documentation in the Serverless Framework and data from the different providers.

RQ3, **RQ4** and part of **RQ2** will be answered by an empirical study to compare the three cloud providers deployments through the Serverless Framework. It will be conducted by creating a serverless back-end (for a React front-end webpage) in AWS, Google Cloud and Microsoft Azure using the Serverless Framework. **RQ1** will use the empirical study to discuss the serverless implementation compared to literature findings.

3.2 Literature search

The area of serverless applications is still quite new and developing rapidly [74]. Cloud providers offering serverless services are increasing and evolving.

The literature search includes the Diva portal in order to find out what earlier exam works has been conducted in the area of "serverless" and "serverless application" [38] [3]. BTH library's own search tool has been used to further examine what studies, books and articles have been released in the area of "serverless" and "serverless application" globally. By re-iteration from a recent article in the serverless area [20] more theses (published by Chalmers) connected to the field were found through the library web page of Göteborgs Universitet [75] [43].

As this area is developing rapidly, only references from 2019 and onwards will be considered. Finally, the Serverless Framework and the serverless providers that will be examined in the report all have their own guides and information that will be utilized.

3.3 Empirical study

The back-end for the application will be based on the serverless concept and should contain storing of the application data (through a database) and authentication of users. It will utilize the Serverless Framework to deploy a serverless Node.js application to AWS, Azure and Google. The command `serverless create -help` can be typed in the Serverless Framework CLI [49]. This provides a list of all available templates in the framework. For all three providers a Node.js application has been developed and these are built on the templates *aws-nodejs*, *azure-nodejs* and *google-nodejs* respectively. The back-end should expose the following routes through an HTTP API:

- **/test**: a test route to GET data (a simple message string) from the back-end
- **/user**: create a new user (user and password) in the database through a POST request
- **/user/login**: login an already existing user through a POST request
- **/auth**: route only accessible to logged in users, requires a token to access through a GET request. Will then return data (a simple message string).

The front-end will be a React application utilizing the serverless part. The serverless back-end will be deployed on each providers server. The front-end is run locally throughout the project. During testing of the created routes the API testing tool Postman has been utilized.

This will be a proof-of-concept to see if the Serverless Framework can facilitate easy deployment across multiple serverless providers. It will also explore how easy the front-end can interact with a serverless back-end. The creation and deployment of the serverless applications will follow the Serverless Framework's documentation [68] with aid of each vendors documentation [1] [35] [41] when specific vendor configurations are needed.

3.3.1 Serverless Framework

Basic concepts

All configurations of functions, events and services can be kept in a single file called *serverless.yml* which should be stored in the root folder of the project. This is the file that the Serverless Framework uses to deploy everything at once when the command `serverless deploy` is used. An application can contain multiple services but in this thesis a **service** is the same as all the functions, events and resources described the *serverless.yml*-file. A *serverless.yml*-file could in larger projects contain multiple services, each having their own specified functions, events and resources.

The code of a serverless application is deployed and executed in **functions**, which are independent units. Functions are generally written to perform a single job such as retrieving a user from the database or performing a scheduled task.

Events are what triggers functions to execute. Events are connected to resources such as an http-request, a schedule rule, a new file upload etcetera. The Serverless

Framework can be used to automatically create the infrastructure needed for an event and configure the function to listen to that event.

Resources are infrastructure components used by functions, such as a DynamoDB table or a S3 bucket, both in AWS. The Serverless Framework can be used to deploy AWS resources.

Plugins are used to extend the Serverless Framework. Some important plugins are *serverless-azure-functions* and *serverless-google-cloudfunctions* which enables support of Azure and Googles functions in the Serverless Framework. The *serverless-offline*-plugin can be used to run functions locally.

These concepts and more can be further explored in the introduction pages to the Serverless Framework [67].

CLI

The Serverless Framework CLI is installed with the command:

```
npm install -g serverless
```

Some important commands used in the CLI are the following (note that *serverless* can be abbreviated to *sls* in all commands):

- `serverless` - can be used to create a new service (project), log-in to the cloud provider and, if chosen, deploy the project at once.
- `serverless create -t <provider>-nodejs -p <appName>` creates a new nodejs service on provider (aws/azure/google) with the name *appname*. This is the template used in this thesis.
- `serverless deploy` - deploys the whole project to the cloud provider
- `serverless deploy function -f <function name>` - deploys a specific function in the project to the cloud provider
- `serverless remove` - removes the whole project from the cloud provider

3.3.2 React application

The front-end web page, Figure 3.1, utilizes the serverless back-end developed in AWS, Azure and Google Cloud. It will be developed using the React Framework. A link needs to be provided in the file *vars.js* to the base URL of the deployed serverless application. The web page will then interact with the back-end through GET and POST requests. The Test, Register, Login and Auth-links in the navbar will utilize these requests. See the GitHub repository that has been set up [25].

The React application will be a basic web page with a nav bar containing links to the different pages. Any difficulties during the development is solved using React's own documentation [44]. The Test-page will present information retrieved from a GET-request to the chosen provider. The Register and Login pages will use POST-requests to register or login a user. A token will be retrieved when a successful login is performed. With this token it will be possible to access a message, through a GET request, when clicking on the Auth-link in the navbar.

A second version of the front-end is also written and available on GitHub [26]. It is based on the original version mentioned above. Design-wise it looks the same but it contains different code for the login and register-functions, based on the usage of AWS Cognito. These are not based on fetch-requests to access the back-end http but instead communicates with the AWS Cognito User pool directly, see Chapter 5.1.2 for further description of this implementation.

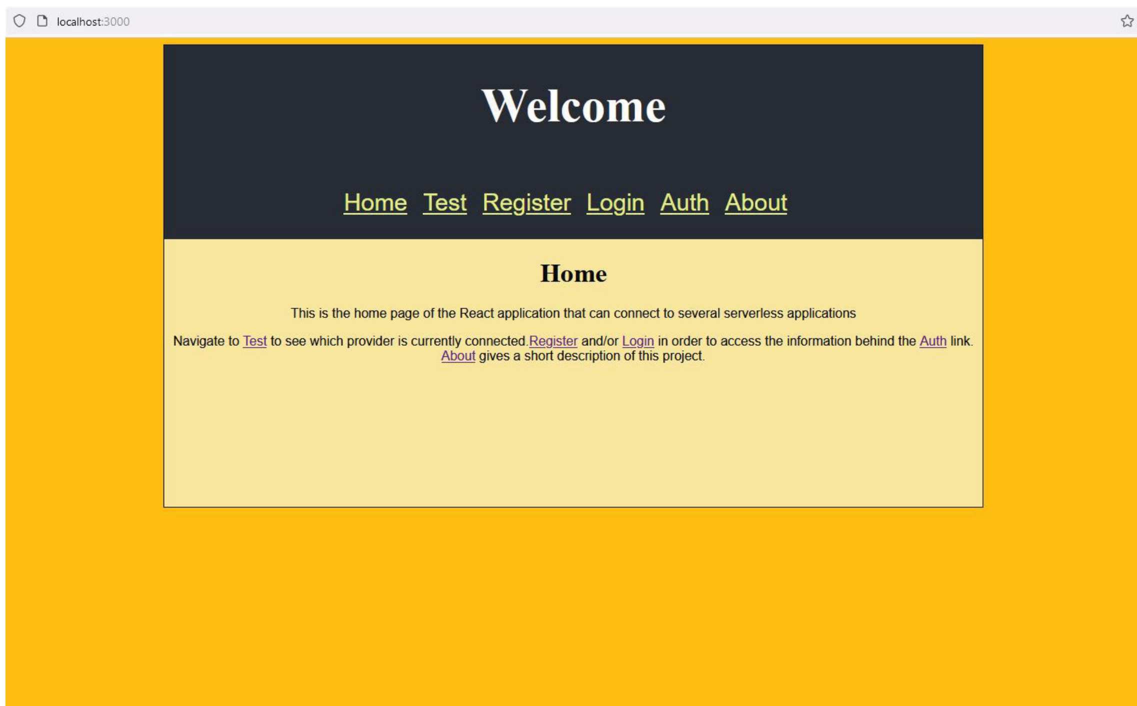

Figure 3.1: Basic React Application utilizing the serverless back-end from AWS, Google and Azure

3.3.3 Limitations

The front-end React application is only tested locally but would for production be deployed online to reach users of the application.

There are many aspects to consider regarding permissions on all three providers, in order to avoid security threats. In this thesis security is not considered in any larger extent, however it is further discussed in the Literature review, Chapter 4. During the empirical study of the thesis the Serverless Framework is allowed admin or contributor permissions whenever that is required to make deployment as easy as possible.

Documentation used to set up the serverless applications is solely from the Serverless Framework and its connected resources (Guides, Youtube channel, Github repositories) and with the aid of the three cloud providers own documentation. There are many guides on the internet discussing serverless application set up, configuration, deployment etcetera. This thesis only follows the standards set by the Serverless Framework and AWS, Azure and Google documentation whenever referred to through the Serverless Framework documentation.

The Serverless Framework has recently released a new type of product called *Serverless Cloud* [65] which is a fully integrated back-end and front-end project compatible with many front-end frameworks, among others React. This will not be used in this project as this is a service built solely on AWS technology [62].

Chapter 4

Literature Review

4.1 Introduction

The literature review will explore what earlier theses works has been conducted in the area of serverless. It will also explore what serverless is, compare it to traditional server-oriented solutions and discuss its strengths and weaknesses.

4.2 Earlier theses works

Earlier theses works either focus on the AWS or the Azure platform specifically [38] [75] [3]. The thesis *Multi-level FaaS Application Deployment Optimization* [75] analyzes the performance when implementing a self-created framework for deployment to AWS. It discusses FaaS in general and how it is affected by deployment through a framework. Both the theses *Architectural Implications of Serverless and Function-as-a-Service* [3] and *Evaluation of "Serverless" Application Programming Model* [38] analyzes the impact of cold starts on serverless applications and the conclusion is that it does affect the response time. The second thesis [38] has a section discussing AWS deployment through frameworks. The Serverless Framework is mentioned and compared to AWS specific frameworks such as AWS Sam. It concludes that the Serverless Framework is one of the most mature technologies for AWS deployments.

Another thesis, *Serverless Development Trends in Open Source: a Mixed-Research Study* [43] discusses current serverless trends. The thesis has analyzed serverless open-source projects on Github to gain insight in use cases, the complexity of the project and architectural patterns. It concludes that the primary programming languages for building serverless applications are Javascript and Python and that the majority of the analyzed projects use AWS or Azure to deploy serverless applications.

No theses have been found discussing the Serverless Framework from a broader perspective, that is by deploying serverless applications to more than one provider.

4.3 What is serverless?

The term serverless can be explained in many ways and from different perspectives. Two interpretations of what the serverless term actually means are:

"Serverless computing is any computing platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the

time the code is running." [20]

"An architectural approach to software solutions that relies on small independent functions running on transient servers in an elastic runtime environment." [2]

The first description tells us that developers do not need to be concerned with server usage and configurations, it is handled by the cloud provider. Both descriptions talk about server allocation increasing and decreasing depending on the amount needed at this very instant. This has to do with one of the often stated advantages of going serverless - you only pay for the amount of code actually being run. If your application is inactive during a period there are no costs occurring. Transient servers means that the actual machines hosting the code are temporary instances. They are spun up when needed and only active during function invocation. Small independent functions indicate that it is built on FaaS, Function as a Service, which is described in the Introduction chapter.

4.4 How to use serverless?

The largest provider of serverless solutions on the market is AWS, with Azure at second place [20]. Serverless applications can be developed directly using each providers dashboards or command-line tool but this will require manually configuring all services and how they should work together. Common services include file storage, databases, messaging and authentication [20]. A framework can manage all the infrastructure for you, which will automate the configurations and make scaling and maintenance more manageable [39].

In this thesis the Serverless Framework [68] is used to deploy serverless applications to some of the most popular vendors; AWS, Azure and Google [43]. An advantage of using a multi cloud deployment service such as the Serverless Framework is the possibility to decrease vendor lock-in or to provide cross-cloud redundancy. Azure partnered up with the Serverless Framework to implement a multi cloud solution for AWS and Azure deployments [17]. This project was finished in the year of 2019 and has not received any updates since then. But the idea was to build cross cloud solutions for best-of-breed services, which would be another benefit of using a multi cloud serverless framework.

If the goal is to only deploy to a single provider, AWS has multiple frameworks of its own, such as Firebase and Amplify, to manage the AWS infrastructure in a serverless application [19]. It can help a developer focus their skills on a specific providers offerings. In the case of Amplify the framework can help develop both the back-end and front-end part of the serverless application. The Serverless Framework has recently developed a similar solution called *Serverless Cloud* which is also based on AWS infrastructure [65].

Table 4.1: Strengths and limitations of serverless versus traditional server-oriented solutions

	Serverless	**Traditional server-oriented solutions**
Function vs application	+ Easier to deploy changes without affecting the whole application - Execution time of functions are limited	+ No time limit on execution of code - Changes requires redeploying the whole application
Scalability	+ Scales automatically	- Scaling takes time and requires adding more servers
Cost	+ Pay for what you use	- Usually pay for more than the actual usage in order to have additional resources.
Managed Services	+ Already developed services can quickly be added - Vendor lock-in	Not available
Developer velocity	+ No network configuration needed + BaaS can speed up development	- Managing network configurations requires time and knowledge - No BaaS available
Cold start	- Running a function can require additional time	+ Servers always running, cold start not an issue
Security and permissions	+ Expert competence by using cloud providers security solutions - Need to rely on the cloud providers solutions - May require more roles and permissions	+ May require less roles and permissions - Security is more up to the developers own knowledge
Debugging	- May be difficult to reproduce errors	+ Could be less complex
Best practices	- Not available	+ Many best practices available

4.5 Strengths and limitations

4.5.1 Function versus application

Before serverless, an application was seen as a unit of deployment. In serverless it is common to look at smaller and more finely grained model of individual functions [40]. Thus the FaaS way of thinking is important, each function typically describes a small part of an entire application. Using small, independently deployed services increase the flexibility of the system. Parts of the system can more easily be updated or replaced without affecting the rest of the system [2]. This is possible with microservices but the Serverless models allow smaller functions to be uploaded which offers even more independence [74]. Before the serverless solution these small functions meant an increased need for managing network connections, security controls, request routing,

and general stability between all the independent parts of your solution. Serverless solutions takes care of managing these aspects.

Serverless applications are event-driven and cloud-based. The execution time of functions is typically limited (for example 15 min for AWS Lambda functions). As the servers are transient the functions are not constantly active. The FaaS platforms listens for events that instantiate the functions. Events can be something like a client request, a file transfer or a data stream. Examples of applications which might not benefit from going serverless are those with long-running functions, such as machine learning training or long-running algorithms. They might have timeout problems and its constant workloads might result in higher costs compared to running on traditional server solutions with virtual machines or container runtimes [74].

4.5.2 Scalability and cost

This is one of the important features of serverless. There is no need to plan for future capacity. If you have a serverless application that becomes popular overnight, then the cloud provider will handle the load by scaling up seamlessly. The fact that it scales seamlessly means that you only pay for the amount of resources used. [18] [40] [74]

When using traditional server-oriented solutions customers pay for generously large machines to avoid running out of resources. That means costs are paid independently f"r the actual usage of the system [74]. The thesis *"Evaluation of "Serverless" Application Programming Model"* [38] concludes that current operational costs are easy to overview but it is more tricky to predict future costs. But also that prices are usually lower than predicted.

4.5.3 Managed services

BaaS typically refers to managed services such as databases and authenticaiton [19]. In serverless applications they are seamlessly scaled just like the core functionality of the application. That means the serverless technology promotes heavy usage of managed services. Each cloud provider has their own set of Baas services which means that it can be difficult to switch to another provider. Vendor-specific databases, API gateways or other native technologies increase the vendor lock-in. The Serverless Framework allows the deployment of the same code to different serverless providers but once the code is deployed (and a specific cloud providers technology adopted) it is hard to move data to another provider [74].

4.5.4 Developer velocity

Thanks to the fact that servers are handled by someone else, developers can focus on writing code and worry less about the network configurations. The vast amount of BaaS offerings mentioned above can increase development velocity. Developers need little or no knowledge of the back-end implementation in order to implement BaaS features [19]. This means less code to maintain as well. As mentioned in section 4.5.1, developers can deploy and, in particular, re-deploy a single serverless

function without needing to deploy the whole system [20], which makes maintaining the application easier.

4.5.5 Cold starts

Performance is a highly debated subject in the serverless field [40] [20] [3] [38]. Because functions are not always active they need to activated on usage. This requires some time if a new function container needs to spin up. The users will spend time waiting for the application to respond (that is, a delay is introduced) due to the *cold start*. Some cloud providers already provide solutions to reduce the cold start, such as Amazon Cloudflare, and there are work-arounds to keep functions *warm* [74] [40]. Cold starts will eventually be optimized as close to zero as possible [40]. But even at present, applications with latency requirements adopt a serverless strategy as shown by a survey over current serverless implementations [20].

4.5.6 Security and permissions

As the cloud providers core competency is to maintain their services, they supply something that is most likely more elaborate and secure than you could have built yourself. As they supply these services for thousands of customers they have already fixed many issues and edge cases [19].

Adapting to the serverless paradigm however means relying on strangers for the infrastructure, as described in the book *Learning Serverless* [39]. It means giving them permissions to spin up costly infrastructure and take care of critical components of a system. Leaked credentials can quickly come to hackers attention, who can for example create costly cloud infrastructure to mine for cryptocurrency on your billing account. Permissions are a major part of the architecture of cloud applications. Both serverless applications and applications utilizing their own servers need to separate permissions between development and production environments. For serverless applications it might be necessary to further decrease permissions to the role deploying your services. For example the role used to create a database table should not be allowed to read or write from them, and perhaps not be allowed to delete the created tables. The book *Learning Serverless* [39] concludes that permissions are a major part of the architecture of cloud applications and that configuration for your cloud provider can be more powerful and dangerous than your application code.

4.5.7 Other concerns

- Complex debugging: a dynamic runtime means it can be difficult to reproduce errors. It might be complicated to trace a user action throughout the system. According to the book *Learning Serverless* [40] this is due to an incorrect usage of serverless. Used correctly a serverless approach should give more understanding of core functionality of the system.

- Best practices: The serverless technology is rapidly evolving and improving. Even if the serverless community benefits from using cutting-edge technology it struggles to invent and adopt best practices. [40]

Chapter 5

Results and Analysis

5.1 Serverless framework implementation

5.1.1 General

The Serverless Framework is primarily aimed at deployment on the AWS platform. That is what their documentation describes by default [50]. There are certain sections describing configurations and deployment on Google [59] and Azure [55]. Table 5.1 presents and overview of what has been tested through the Serverless Framework at each provider. More details about the implementations are discussed in the following sections.

When deploying a serverless application it is possible to develop, deploy, test, secure and monitor the application in the Serverless Framework Dashboard. This is a SaaS (Software as a Service) solution with a graphical interface to keep track of all deployments [66]. It makes development and deployment easier as everything can be done through the Serverless dashboard and there is no need to learn the specific providers interface. Configuring data for the services or reading log outputs for the deployed functions can be done through the dashboard [52]. The dashboard so far only allows deployment and monitoring of AWS applications. They need to be Node.js or Python applications and deployed in certain regions [66].

The routes described in Section 5.1 has the format **/route-path**. This implicitly describes the route **baseurl/route-path** where the baseurl is the deployed applications base URL.

Table 5.1: Overview of which implementations that has been tried at each cloud provider using the Serverless Framework.

	AWS	**Azure**	**Google**
Account	Free Tier, 12 months	Free, 1 month without credit card	Free tier, 3 months
BaaS	Same as AWS	Fewer than Azure	Fewer than Google
Database	DynamoDB	Cosmos DB	No
Authentication	Cognito	No	No
Dashboard	Yes	No	No
JWT	Implemented	-	-
MongoDB	-	Implemented	Implemented

5.1.2 AWS

Getting started

A good starting point for the Serverless Framework is the guide *Getting Started With Serverless Framework*. As mentioned above the Serverless Frameworks documentation is by default aimed at AWS and so is the guide to get started with the framework [52]. It describes how to create an AWS account and install the Serverless Framwork. It further explains how to set up a Node.js application and how to monitor it in the Serverless Dashboard. At the end of the guide an http-endpoint has been created and connected to a DynamoDB database for persistent storage of data. The guide contained an error as the wrong code was presented when the function should be extended with *createCustomer*. The correct implementation could however be found on the Github repository (linked to in the guide). Following this guide a **/test** route can be created that can be called as a GET request to return the message "*This is a test route on AWS!*". A functions route is set in the *serverless.yml*-file under the *functions*-section, under path:

```
functions:
  testFunction:
    handler: src/functions/testFunction.testFunction
    events:
        - http:
            method: get
            path: /test
```

AWS account

The AWS account mentioned in the section above is an AWS Free Tier account [6]. It gives the user access to a free account for 12 months and with many features free even after the first year. 25 GB of storage in a DynamoDB and 1 million free Lambda requests per month is for example always free. It also includes 5 GB of storage in S3 during the first 12 months.

BaaS Services

The number of BaaS offerings from AWS documented on the Serverless Framework are overall the same as what AWS offers, see Figure 5.1 and the Serverless AWS Event documentation [51] compared to AWS own event documentation [7]. The BaaS services offered includes the DynamoDB database and Cognito User Pool used for authenticating users. Both will be further discussed below.

Database

The guide to get started with AWS, section 5.1.2, includes an example of using the DynamoDB. Searching through guides and tutorials [72], most of the AWS guides utilizing a database are using the DynamoDB. This is therefore the preferred choice as there are many examples available.

Figure 5.1: AWS Events. The left column shows all events listed on AWS while the right column shows all AWS events listed on the Serverless Framework

User authentication

As can be seen in Figure 5.1 the BaaS services of AWS includes the Cognito User Pool which is described in the event section in the Serverless Frameworks documentation [51]. A user pool is a user directory in AWS Cognito. It provides sign-up and sign-in services, user directory and user profile management, a UI to sign in users and sign-in through Facebook, Google and other social networks [4].

Implementation

The full implementation is available in a GitHub Repository [21].

The Serverless Framework documentation on Cognito (authentication) is limited and further directs to AWS own documentation [4]. The Serverless Framework has a set of guides and tutorials [72] which includes a guide on how to implement a Cognito User Pool [8]. The guide sets up a Cognito User Pool and App Client with two secured endpoints, one GET and one POST found on the route **/user/profile**. The user pool will contain the user data while the app client is used to connect to

a front-end application. The two created endpoints (routes) can only be accessed by an already registered user. The guide shows how to manually set up a user in the User Pool on AWS own site [5]. When the user is manually added to the user pool, the hosted UI in the user pool can be used to acquire a token to access the secured routes.

The above guide about Cognito was used as a starting point. This is all that is needed in the back-end for the user pool and the two secured routes to be functional. But the front-end needs to connect to the Cognito User Pool in order to add new users and sign-in existing users. As mentioned before, the Serverless Frameworks page on the Cognito User Pool further directs to AWS own documentation [4]. Reading about the sign up lambda trigger functionality there is a Sign-up (in this case the Javascript version, also available for Android and iOS) tutorial on how to set up the registration, login and much more in the front-end with the aid of the *Authentication with amazon-cognito-identity-js* SDK [9]. With this guide the front-end can be set up to include registration and sign-in of users, utilizing use case 1 and use case 4 in the guide.

This method of accessing the back-end does not use the routes /**user**/**login** or /**user** for login and registration. Instead it uses the SDK mentioned in combination with the Cognito User Pool ID and the App Client ID, which can be found in the Cognito User Pool dashboard [5]. In this thesis only registration and login was set up, but the Cognito user registration as a rule requires a validation of the user. When a user has registered a new profile, an email is sent with a validation code. This needs to be validated before the user profile is activated. In this thesis the email has been approved manually through the Cognito user pool dashboard [5]. Either one more page needs to be added to the front-end to validate users (use case 2 in the mentioned guide [9]), or the back-end needs to be configured to not require the validation step. The second option should be possible but requires further research in the AWS documentation [4].

Because the front-end is not run on the same domain as the back-end, CORS need to be configured. Trying to access the routes in the front-end without configuring CORS results in the error message: *CORS header 'Access-Control-Allow-Origin' missing*. Using the Serverless Frameworks guide to the HTTP API event [61], the section *CORS Setup* explains how to add the code *cors: true* to the provider section of the *serverless.yml*-file.

The getting started guide [52] was used to add a DynamoDB database to the project. A /**test**-route was created to present all the customers in a DynamoDB collection. In the guide this route was given the index path / instead of /**test**. A /**customer**-route was created to POST new customers to the DynamoDB collection for testing purposes. This /**customer**-route had been given the index /-path in the guide. The secured GET-route from the Cognito guide /**user**/**profile** was changed to be found on the /**auth**-path to conform with the Method (see Chapter 3) described in this thesis. The auth route returns a text message that can only be accessed by a logged in user.

Serverless Dashboard

As described in section 5.1.1 the Serverless Framework has a dashboard which works with AWS deployments. The deployed application can thus be monitored through this, see Figure 5.2. This gives an overview of which functions that have been deployed. Any errors invoking the functions can easily be traced and handled without learning AWS own setup and logging. If the application is not already published to the dashboard it can be done by running the command *serverless* in the same folder as the *serverless.yml*-file. This will add the lines *org: <org name>* and *app: <app name>* to the *serverless.yml*-file [66].

Figure 5.2: AWS Dashboard when the serverless application is deployed

5.1.3 Azure

Getting started

The Serverless Framework has its own section to document using Azure through the Serverless Framwork [55]. A good starting point is the quick start guide [53]. It explains how to create a free Azure account and how to set up a basic Nodejs API with HTTP-requests. It describes the installation process of Azures own CLI and how to set up authentication to the Azure subscription where applications should be deployed. Following this guide a **/test** route can be created that can be called as a GET request. The route path is set in the *Functions*-section of the *serverless.yml*-file with the *route* command:

```
functions:
  test:
    handler: src/handlers/test.initialTest
    events:
      - http: true
        methods:
          - GET
        route: test
        authLevel: anonymous
```

Some minor issues encountered during the set up and deployment of a project in Azure are:

- Running functions locally (offline) requires an older Node version which was solved by running nvm in order to set an older version when testing out functions offline (node version 14.0.0 works). To download templates and deploy the application Node version above 16 was required.
- If a service principal is set up to use Azures own CLI (described in the quick start guide) it creates an account for this principal. This account needs permission to deploy resources to Azure. Permissions can be configured through the Azure Portal [12] in the Access control (IAM). During this project the role *Contributor* was assigned to the service principal account. It grants full access to manage all resources but does not allow the account to grant access to others.

Azure account

The Azure account mentioned in the section above is an Azure Free Account [13]. It gives the user access to a free account for 12 months with many popular services included. Some services are always for free. It gives the user $200 Azure credits to use the first month. A credit card is needed after the first month in order to keep using the Azure Portal [12]. To give some examples of free services, 1 million requests to Azure functions per month will always be free and 400 request units per-second to Cosmos DB is free for the first 12 months.

BaaS Services

The number of BaaS (back-end as a Service) offerings from Azure documented on the Serverless Framework are fewer than what Azure actually offers, see Figure 5.3 and the documentation of Serverless Frameworks Azure events [64] compared to Azures own documentation of events [11]

Supported bindings

This table shows the bindings that are supported in the major versions

Type	1.x	2.x and higher[1]
Blob storage	✓	✓
Azure Cosmos DB	✓	✓
Azure SQL (preview)		✓
Dapr[3]		✓
Event Grid	✓	✓
Event Hubs	✓	✓
HTTP & webhooks	✓	✓
IoT Hub	✓	✓
Kafka[2]		✓
Mobile Apps	✓	
Notification Hubs	✓	
Queue storage	✓	✓
RabbitMQ[2]		✓
SendGrid	✓	✓
Service Bus	✓	✓
SignalR		✓
Table storage	✓	✓
Timer	✓	✓
Twilio	✓	✓

Azure
- Overview
- CLI Reference
- Events
  - Overview
  - HTTP
  - Timer
  - Queue Storage
  - Service Bus
  - Event Hubs
  - Blob Storage
  - Cosmos DB
  - Other Bindings

Figure 5.3: Azure Events. The left column shows all events listed on Azure while the right column shows all Azure events listed on the Serverless Framework

Database

As can be seen in Figure 5.3 there is a section on how to implement Cosmos DB. This is a database that through the Serverless Framework can handle inserts and updates but not deletions [56]. The lack of deletion-handling is not an issue as this project focuses on adding new users and logging in the created users. Thus all that will be required are GET and POST requests to retrieve or insert information in the database.

User authentication

There is no event description for handling user authentication in Azure using the Serverless Framework, see Figure 5.3. There is a short paragraph on connecting to other events in Azure [55] but this would require thoroughly investigating the Azure documentation. A quick overview lists a built-in authentication option in Azure called Easy Auth [10]. This would require configurations through the Azure Portal or through a script in the Azure CLI, as no documentation on how to implement it through the *serverless.yml* file is available.

Implementation

The actual implementation is available in a GitHub Repository [22], which is further built upon in section 5.2 when a MongoDB database is added.

The database has been set up using the documentation mentioned above, [56] and Azure's own documentation for connecting to Cosmos DB [14]. From the Serverless Frameworks documentation it is not clear if it is enough to state the Cosmos DB under the function and event in question in the *serverless.yml* file. This is all that is described in the Serverless Frameworks documentation. During deployment of the AWS serverless application, section 5.1.2, the database needs to be stated under a Resource section in order to create a database resource.

Deploying the Azure application using only the information from the Serverless Documentation yields no callable route for interacting with the database. Neither is a Cosmos DB created. As mentioned above it might be necessary to add a Resource section but there is no documentation on to how to implement this for Azure applications. There was a minor change to the code compared to the Serverless Frameworks documentation as an error occured during deployment: `cosmosDB bindings: Error: Binding direction not supported`. This was fixed by setting *CosmosDB: true* in the *serverless.yml*-file as stated in this Github issue written on the Serverless Framework Azure function plugin [28].

A new try to set up the database manually on the Azure Portal [12] was performed with the aid of Azures documentation [14]. Azures documentation describes creating a database through a script and this has been performed manually through Azure's portal as described below.

Deploying a serverless application to Azure using the Serverless Framework creates a new Resource group, see Figure 5.4, with the *service*-name stated in the *serverless.yml*-file.

Accessing this resource group it is possible to press *create* to create a new Cosmos DB resource with the *Core (SQL) - Recommended* option. This creates a database in the same resource group as the deployed serverless application. When the Cosmos DB resource is created one can access it and find the *connection string* which should be entered in the *connectionStringSetting* option in the *serverless.yml* file of the serverless application.

When deploying the changes it still yields no callable route to interact with the database. Logs are difficult to find, a thorough search yields one way which is to access a specific function in the Resource group by clicking on the deployed Function App → Functions → <function name> and then press on successful or failed exe-

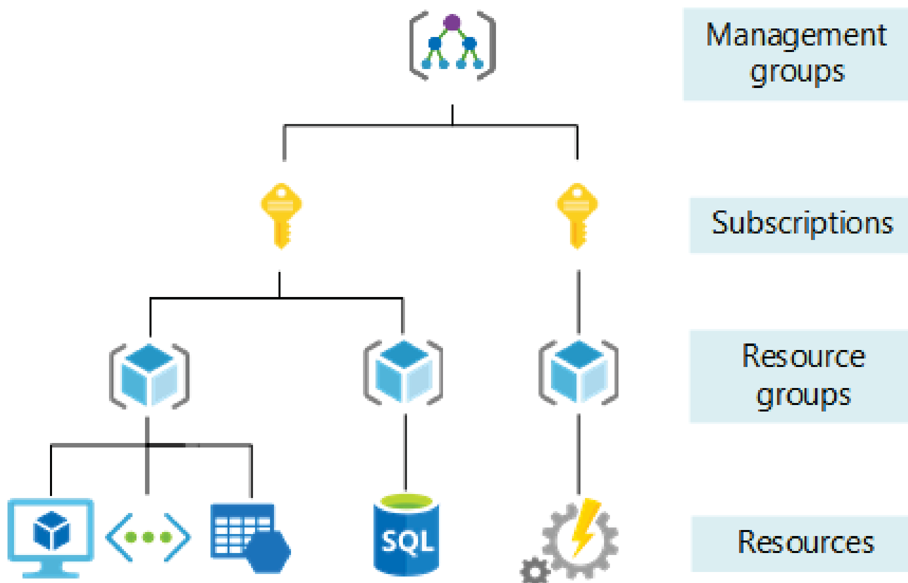

Figure 5.4: Azure Management Levels and Hierarchy. A subscription can contain multiple Resource groups and each Resource groups can contain multiple resources [16]

cutions to see more information. Since calling the new route yields a 404-response the route is not callable and cannot be found in the function logs. Removing the cosmos-db configurations and setting the route to a simple GET-route works fine but adding the cosmos-db configuration makes the route fail somehow. Searching through the Serverless Frameworks own forum there are more developers struggling to configure the Cosmos DB [47] [46] [48]. None of the forum posts have resolved the issues of connecting to Cosmos DB. The Github repository containing the Serverless Azure function plugin (used for connecting to functions and events in Azure) is looking for maintainers and no commits has been performed during 2022 and only three commits during 2021 [54]. A database connection to Azure could therefore not be set up in the Serverless Framework at the moment.

The only route that has been set up in Azure through the Serverless Framework is the **test**-route, which returns a simple message when a GET-request is sent. It can be implemented using the quick start guide [53] described in the *Getting started* section above. Further investigation into implementing a database and authentication using non-proprietary solutions are described in Section 5.2.

5.1.4 Google

Getting started

The Serverless Framework has its own section on how to use Google to deploy applications [59]. A good starting point is the quick start guide [58]. It describes setting up a basic Nodejs API with HTTP-requests. It further links to a section describing how to create a Google Account, setting up a project and enabling necessary APIs [57]. In this project the Google Account has been used directly and no Service Ac-

count has been created. A JSON credentials keyfile is created on Google and stored in the project in order to deploy the serverless application to the correct project. Following this guide a /**test** route can be created that can be called as a GET request. The route path is set in the *Functions*-section of the *serverless.yml*-file with the *name* command:

```
functions:
  test:
    name: test
    handler: test
    events:
      - http: path
```

Changes made which are not described in the quick start guide are:

- Accessing the GET route after deployment results in the error message `Your client does not have permission to get URL ....`. This can be fixed by accessing the project in Google Cloud and searching for functions. On the function in question a permission can be set for *allUsers* to access it [36].
- Setting up a custom name on a Google route requires the addition of the parameter *name* in the function, otherwise the route name will be quite long: *<service-name>-<environment(dev)>-<function name>*.

Google Cloud Account

The Google Account mentioned in the section above creates a free tier account. It gives access to $300 in free credits to use the first 90 days. Many products are offered for free up to a certain limit of usage even after the 90 days. This free limit usage does not expire so long as the usage is below the threshold. Google Cloud can be used for free as long as needed. The free tier includes a 5 GB standard storage, 1 GB of storage in the Firestore database, 2 million invocations per month of Cloud Functions and many more products [31] [34].

BaaS Services

The number of BaaS offerings from Google documented on the Serverless Framework are fewer than what Google actually offers, see Figure 5.5 and the documentation of Serverless Frameworks Google events [70] compared to Google's own documentation of events [30]

Database

No documentation exists of how to access databases through Google Events in the Serverless Framework. Google Cloud has an overview of different databases that can be used through their services [33]. Only Cloud Firestore is supported through functions in Google at the moment, as could be seen in Figure 5.5. But this database is not mentioned in the Serverless Frameworks documentation.

Currently, Cloud Functions supports events from the following providers:

- HTTP
- Cloud Storage
- Cloud Pub/Sub
- Cloud Firestore
- Firebase (Realtime Database, Storage, Analytics, Auth)
- Cloud Logging—forward log entries to a Pub/Sub topic by creating a sink.

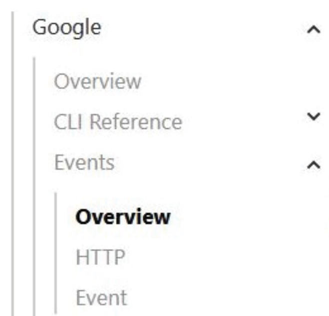

Figure 5.5: Google Events. The left column shows all events listed on Google while the right column shows all Google events listed on the Serverless Framework (only HTTP and Event)

The Github repository containing the Serverless Google Cloud Functions Plugin (used for connecting to functions and events in Google) is just like Azures plugin looking for maintainers [71]. There are however recent commits from the current year of 2022 and multiple commits from the end of 2021.

User authentication

There is no description of how to handle user authentication when deploying to Google through the Serverless Framework. There is a section regarding Authentication on Google Clouds documentation [29]. This requires installing additional libraries to handle authentication. As this is not connected to the Serverless Framework it will not be further investigated.

Implementation

The actual implementation is available in a GitHub Repository [23], which is further built upon in section 5.2 when a MongoDB database is added.

The only route that has been set up in Google through the Serverless Framework is the /**test**-route, which returns a simple message when a GET-request is sent. It can be implemented using the quick start guide [58] described in the *Getting started* section above. Further investigation into implementing a database and authentication using non-proprietary solutions are described in Section 5.2.

5.2 Non-proprietary implementations

The use of BaaS services creates a vendor lock-in. To create a more general back-end solution that can be deployed to all three providers, authentication and database-handling can be handled by third party providers. Authentication can for example be set up using Json Web Tokens. MongoDB Atlas can be used as a database. This database is cloud based and can be accessed by all three serverless applications.

5.2.1 Authentication

Json Web Token in AWS

A GitHub repository is available with the implementation [24].

By using a custom authorizer such as JWT a more general solution to authentication can be created. It handles authentication entirely through code. This sub-section describes how it can be implemented in AWS using a guide from the Serverless Framework. No such guide exists for Azure or Google. To limit the work in this thesis it has only been implemented in AWS.

The Serverless Framework has both its own set of guides and tutorials [72] and a Youtube-channel with courses. The Youtube-channel includes a course for Full Stack Application development in AWS [69]. The course covers basic understanding of the Serverless Framework and the concept of Serverless applications. From video number 15 onward it goes through setting up a rest API in AWS. It then adds a DynamoDB database to save persistent data (user name and password) and jsonwebtoken to handle JWT tokens for logged in users. Bcryptjs is used to encrypt passwords. Following this guide it is clear that there are a lot of configurations to be done to get the whole application to work together. Since this is a rapidly changing field (the videos are from September 2019) such things as the following warnings occur when deploying the application to AWS:

- `"serverless-pseudo-parameters" plugin is no longer needed.`
- `Starting with next major version, "http.request.schema" property will be replaced by "http.request.schemas"`
- `Resolution of lambda version hashes was improved with better algorithm, which will be used in next major release.`

These are all deprecation warnings and did not affect the deployment of the serverless application. Setting up this type of application from scratch without a thorough guide to follow might demand a great deal of time and require a lot of trial and error. Such things as implementing the npm module bcrypt is apparently difficult on AWS Lambda functions. Thus the module bcryptjs is used instead as the course describes.

At some points in the Youtube course material there were some misspelling. Either this was corrected in the video or the correct solution could be found in the Github repository connected to the course [27]. In order for the serverless back-end to accept the front-end request, headers were added in the return-statements from the handler files (where the actual function code resides). The following information had to be included, which is described in the AWS API Gateway guide [63] (section Enabling CORS):

```
headers: {
          'Access-Control-Allow-Origin': '*',
          'Access-Control-Allow-Credentials': true,
},
```

In summary, the course explains how to add the routes /**user** for registering a user and saving the data to a DynamoDB, /**user**/**login** for logging in an existing user and /**validate** which is not a public route as it should be used as a pre-route to check if a user is logged in and has a valid token.

Another route was added to the back-end, /**auth**. This is a secure route that could only be accessed by already logged in users. It utilizes the **validate**-function created in the course to be run before the auth-function is called. It is set up using the guide about the AWS API Gateway [63] (section HTTP Endpoints with Custom Authorizers). This creates a complete back-end with user registration, log in, a test-route accessible by all and a route only accessible by authenticated users.

5.2.2 Database

MongoDB in Azure and Google

The Serverless Framework contains a guide on how to use MongoDB with the Serverless Cloud [60]. Although the Serverless Cloud is different from the Serverless Framework, the part describing how to set up MongoDB Atlas and connect to it are the same. MongoDB has its own guide on how to get started, which can be used to accomplish the set up [42]. As no database connection was established in the Azure and Google projects (see section 5.1.3 and 5.1.4), these guides have been used on both the Azure and Google deployment described above. The earlier mentioned GitHub repositories were updated with these changes, see [22] and [23].

Both the Azure and Google projects used to connect to MongoDB are based on what was created in Section 5.1.3 and 5.1.4 respectively. Another route called */mongodb* was created which can be used to POST an entry to the MongoDB database.

An account is created on MongoDB Atlas and a shared cluster is created, which is free to use. It is possible to choose provider and Azure is chosen (with region Netherlands) to try out with the serverless Azure deployment. Another shared cluster is created with Google as provider (and region Belgium) to use with the severless Google deployment (and to try out deployment to two providers through MongoDB Atlas). The chosen regions makes the cluster free to use. The setup of a cluster creates a new database. A new network access from ip 0.0.0.0/0 opens up for all ip addresses to establish connection with the database. A database user is created to connect with the database and perform actions. After that it is possible to connect to the cluster by clicking on the database in question and click on *connect*. By marking the checkbox *Include full driver code example* a complete code example can be retrieved for connecting to the database. This code and the code from the Serverless Cloud guide [60] are used to create a function which adds an entry (a name) in the database *test* and collection *devices*.

A difference noted during deployment to Azure versus Google is that the deployment to Google caught an error already during deployment. The first trial with Azure resulted in a deployed application that did not work as intended. The route connecting to the database returned a 500-response (database error). As mentioned earlier Azure was difficult to troubleshoot. When deploying the same code to Google it complained already during deployment about a forgotten "async" statement. After correcting this issue the code was deployed to Google without any problems but

the route was still not working properly. By searching for *function* at the top of the Google Cloud Console [32] page, the option *Cloud Functions* can be found. This gives an overview of all deployed functions and each function has an overview of the number of requests received and any errors that has occurred. It was easy to find the issue; the *mongodb* npm-package had not been installed. Correcting this resulted in a working route and deploying the same code to Azure worked as intended.

5.3 Research Question 1

What are the differences between serverless and traditional server-oriented solutions for a web application?

This section discusses the benefits and drawbacks of going serverless and compares it to the findings in the empirical study conducted.

The literature review revealed many benefits of going serverless, compared to traditional server-oriented solutions. There is no need to manage network connections, request routing, handle stability between parts of the system nor worry about scalability. The serverless provider handles all of this. The cost of a serverless application is generally lower as the cost is based on actual usage. If nothing is run then there is no cost. If the application needs to scale up, the cost goes up. In traditional server-oriented solutions customers tend to have larger machines than necessary to avoid running out of resources. In this thesis the applications were small enough to be run on free accounts where the cost was not an issue. During the empirical study the serverless applications were easy to spin up in the cloud on each provider and no issues occurred related to servers. Since only free tier accounts were needed there were no costs related to running the serverless applications.

The architecture of a serverless application is similar to a microservice solution. It is built up by many functions. Each function usually has one single task to perform. It makes it easier to deploy or update a single function instead of the whole application. There are however limitations as to how long a function is allowed to run. In this project the functions were small and only ran for a couple of seconds at most, thus the serverless architecture would be a good choice. The idea of dividing up the application into many functions creates an easier deployment structure. This did however not always work out as expected. Sometimes a re-deployment of an already existing function (and not the whole application) did not update the function. Smaller changes such as which status code an http-request would return did not update the code when deployed. If this was due to the Serverless Frameworks short-comings or the specific vendors system has not been further investigated. The solution has been to either re-deploy the whole application or to remove and then deploy the function anew.

Many serverless applications rely on BaaS services such as databases, authentication, file management etcetera. It can decrease time to market of a product as services are already developed and only need to be applied to the new application. Since the servers are not the developers issue it might be a time saver and decrease the development time. As the Serverless Framework is still mainly focused on AWS it was possible to implement BaaS services through the framework aimed at AWS

deployment. But for Azure and Google this was not possible. If the developer has chosen to work with a specific provider and set up the BaaS services through the specific providers dashboard (or framework) this would likely not be an issue. This would however result in greater vendor lock-in as described earlier in the thesis, see for example section 4.5 discussing strengths and limitations of going serverless.

One potential issue of going serverless are the cold starts. If a function is not active for a while there are no active containers. When the function needs to be activated again it is called a cold start. A container needs to spin up and this results in a longer waiting time for the user before the application responds. The cold start is constantly being decreased through different provider improvements but is something to consider if the latency requirements are rigorous. In this thesis the latency is not an issue but can be noticed by a somewhat longer waiting time when first calling a function.

The serverless field is still new and lacking best practices and solutions. Different providers have slightly different solutions and it is rapidly evolving. This is quite noticeable especially when using a framework for deployment. Both the providers services and the frameworks updates are frequent and sometimes the framework has not kept up with the providers solutions. The Google deployment for example had to be updated to Nodejs version 14 to work correctly, but the template from the Serverless Framework still used version 12. It can be tricky to find complete documentation for the latest standard as some guides and examples on the Serverless Framework are older and thus it's best practices are not valid anymore.

Security and permissions are two important details when going serverless. Many security issues are handled by the provider especially when using BaaS services. This can be a benefit as they are experts on their provided services and can constantly work on improving security. It also means that the developer has to trust the vendors implemented solutions. In order to deploy, run and manage serverless applications, functions, users etcetera there are many permissions to handle as mentioned in Section 4.5. This is an important aspect that has not been further investigated in this thesis but is an area that is probably large enough to be handled in a thesis of its own.

5.4 Research Question 2

What are the main differences between the chosen serverless providers Amazon Web Service Lambda, Google Cloud Functions and Microsoft Azure Functions in the Serverless Framework in regard to function syntax, BaaS services and price?

Table 5.2 summarizes the differences in function syntax between AWS, Azure and Google. BaaS services and pricing are also discussed below. In summary BaaS services are mainly supported by AWS and not Azure and Google. For this small application development a free tier account is enough and pricing is therefore not an issue. A noticeable difference in the free tier is that Google provides two million function invocations while the others only allow one million.

One reason to use a framework to deploy serverless applications is to decrease the number of configurations required. Using the Serverless Framework makes it

Table 5.2: Differences in function syntax between the three providers

	AWS	**Azure**	**Google**
Route path naming	path	route	name
Nested routes	Supported	Supported	No
HTTP event available	Two events	One event	One event
Method (GET/PUT etc)	Specified in serverless.yml	Specified in serverless.yml	Specified in handler-file
Support	Supported by default	Requires an Azure plugin	Requires a Google plugin
Extra		authLevel needs to be specified	

possible to deploy to many different providers. It is however noticeable that the Serverless Framework, which started out supporting only AWS, is still mainly aimed at AWS deployments. The documentation is by default aimed at AWS deployments. It means that the framework supports AWS Lambda functions as a standard. For Azure Functions or Google Functions a plugin is used to support the implementation of these providers functions and events.

Comparing the three providers syntax in the Serverless Framework, see Figure 5.6, 5.7 and 5.8 there are many similarities. These are all basic functions handling a GET HTTP request. All three begin with a function name. This name is for example used to deploy the single function using `serverless deploy function -f <function name>`. All three points to a *handler* which is where the function code is located. The event in all three cases is http which handles a GET request.

```
functions:
  testFunction:
    handler: src/functions/testFunction.testFunction
    events:
        - http:
            method: get
            cors: true
            path: /v1/test
```

Figure 5.6: AWS serverless.yml functions setup

```
functions:
  test:
    handler: src/handlers/test.initialTest
    events:
      - http: true
        methods:
          - GET
        route: v1/test
        authLevel: anonymous
```

Figure 5.7: Azure serverless.yml functions setup

The route path is */v1/test* for the AWS and Azure project and */test* for the Google project. This is specified in the *path* field for AWS, *route* field for Azure and *name* field for Google. Google does not support nested routes such as */v1/test* in the *serverless.yml*-file [70] (see HTTP event).

```
functions:
  test:
    name: test
    handler: test
    events:
      - http: path
```

Figure 5.8: Google serverless.yml functions setup

For AWS the *http* event was used in the JWT authentication project but the newer *httpApi* event was used when the Cognito User Pool was created. One difference is that *cors* has to be specified in the older *http* event. This has to be added in the function to handle requests from the React front end, see Figure 5.6. In the newer *httpApi* event *cors: true* can be added to the *provider* section and apply to all functions at once.

The *method* is stated as GET in both AWS and Azure while this is specified in the functions handler-file in Google.

The Azure function needs to specify the *authLevel*. An anonymous function means that it does not require any authentication to access. The authLevel can be used in conjunction with an API key to restrict access to certain routes [15].

The supported number of BaaS services (events) for AWS in the Serverless Framework are many. AWS utilizes the *resource section* in the *serverless.yml*-file to handle spinning up resources such as databases and authentication services.

In contrast to AWS, there are few BaaS services available for both Azure and Google in the Serverless Framework. More BaaS are described for Azure than Google but the documentation is quite limited for both providers. It was difficult to work with events in both the Azure and Google project because of the lack of documentation. Neither database nor authentication using BaaS were either supported or could be implemented in this thesis.

The cost of running a serverless application differs between the providers. For a small application the free tiers are enough, and the cost will therefore not be an issue. If the application grows and exceeds the free usage the cost will depend on which service that is exceeded. Is it the number of function invocations? The number of database requests or perhaps the database storage size? Each provider has their own ways of calculating prices for number of function invocations, running times, space usage and resource allocations. One noticeable difference was that Google Cloud provides 2 million function invocations per month for free while AWS and Azure provides 1 million.

So how could a developer make sure to get the best price? One way would be to use a serverless framework with as few vendor specific services as possible. That way it would be possible to switch provider in case the price is better somewhere else. But using for example MongoDB to avoid AWS, Azure or Google's own database solutions would mean depending on a third party. Even though MongoDB provides a serverless solution which should scale seamlessly it still requires depending on a provider (MongoDB in this case) and of course their pricing.

5.5 Research Question 3

Without earlier experience in serverless – which of the providers is most suitable for new development of a smaller back-end API, to connect to a front-end React application, using the Serverless framework?

- *Easiest to get started, least configuration*
- *The application should contain authentication and a database. It will investigate the possibility to use BaaS for these services as well as non-proprietary options.*

As described in section 5.1 above it is clear that AWS has the most complete documentation on the Serverless Framework. It also has the most guides and tutorials. For both Azure and Google the documentation is thin but the Github repository for the Google plugin seems to retrieve more updates than the equivalent Azure repository. It was also noted that deployment to Google seemed to catch more errors already during the deployment phase compared to Azure, see section 5.2.2. This might mean that it would be better to choose Google over Azure for future deployments, but as mentioned below if the developer already has some experience in one of the providers tools it would probably be wise to stick to that provider.

Easiest to get started with on the Serverless Framework is AWS. There are many getting-started guides and examples of how to implement databases, authentication and much more. By adding a *resource*-section in the *serverless.yml*-file the Serverless Framework can configure and spin up resources such as a DynamoDB database and Cognito User Pool to handle authenticating users. All the documentation is by default aimed at AWS and there are specific sections on how to deploy to Azure or Google. These sections are small and in general further directs to each vendors specific documentation. If a developer has earlier experience in Azure or Google it might be easier to implement the Serverless Framework with the specific providers product. It does however seem like deployment of BaaS services to Azure or Google would require some configurations through the providers own CLI or dashboard and could not be spun up only through the *serverless.yml* configuration file.

The fact that deployment and monitoring/troubleshooting can be done through the Serverless Frameworks own dashboard means that AWS own dashboards barely need to be used. That makes it easy to get started with AWS on the Serverless Framework. Google and Azure deployments had to be monitored through each providers own dashboard. It meant more work getting to know each vendors specific navigation, ways of monitoring functions and configuration of services. Very little work was needed to monitor AWS deployments as this could be done through the Serverless frameworks dashboard.

It can be a steep learning curve for developers new to working with deployments in the cloud and this was noticeable also when working through the Serverless Framework. Even though AWS was easiest to get started with it did require reading through parts of the AWS documentation in addition to the Serverless Frameworks own documentation. Working with FaaS and configuring, deploying and troubleshooting in the cloud also takes some time to get used to. The fact that the serverless providers handles scaling of the application and that the cost is based on actual usage is a

strong advantage of serverless. This could be a reason to deploy a smaller application in a serverless version, especially if the intention is to reach many users over time.

AWS would be the recommended choice. As to using proprietary solutions such as DynamoDB and Cognito User Pool - AWS own database and authentication service - it will depend on which degree of vendor lock-in feels comfortable. These BaaS services could make deployment faster but will make it difficult to switch to another provider in the future. It is difficult to rank Azure or Google as the second choice after AWS. Azure's documentation in the Serverless Framework is somewhat more detailed than Google but Google deployments seemed to catch more errors in the conducted empirical study. Neither of the providers BaaS services were possible to deploy using the Serverless Framework. As mentioned earlier the choice between Azure and Google would come down to earlier experience in either of the providers tools.

5.6 Research Question 4

Does the Serverless Framework support easy reusage of code across different serverless providers and if so to what extent?

To summarize the answer to Research Question 4:

- Reusage of code across different serverless providers will not be possible when BaaS services are introduced.
- Sticking to non-proprietary solutions instead of BaaS services, such as JWT for authentication and MongoDB for database, makes reusage of code managable.
  - Configurations such as where functions are saved and configurations of the *serverless.yml* file are slightly different between the providers
  - Requires more coding and the solution will be more dependent on the developer

As most of the configurations are handled by the *serverless.yml*-file this is the main part that needs to be adjusted in order to deploy the same project to all three providers. Google however requires a JSON credentials keyfile saved in the project in order to authenticate and deploy serverless application. There are some smaller differences in the configurations (*serverless.yml*) between the providers such as different function attributes and naming conventions, see Research Question 2. The *provider*-section needs to be changed to AWS/Azure/Google and runtime environment could differ. During this project both AWS and Azure were run on nodejs12 (which was the original template choice) while the Google project was run on nodejs14 as a TextEncoder error would otherwise occur.

The actual handler of each function (the code run when a function is invoked) have different standards as to where they are saved depending on the provider. The general structure is however similar, see Figure 5.9, 5.10 and 5.11. There are minor code variations as different guides have been used to set up the handlers. The AWS

```
module.exports.testFunction = async (event, context) => {
  return {
    statusCode: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
      'Access-Control-Allow-Credentials': true,
    },
    body: JSON.stringify({
      message: 'This is a test route on AWS!',
      input: event,
    }),
  };
};
```

Figure 5.9: AWS basic GET route

```
module.exports.initialTest = async function (context, req) {
    context.log('Request to test route.');

    context.res = {
        // status: 200, /* Defaults to 200 */
        body: 'This is a test route on Azure!',
    };
};
```

Figure 5.10: Azure basic GET route

project includes CORS so the functions return statement includes headers to allow all origins.

The Serverless Framework is advertised as a way of avoiding vendor lock-in and provide easy reusage of code across providers. If specific BaaS services are used it will not be possible to transfer code between providers. This creates a vendor lock-in with proprietary solutions that cannot easily be transferred to another provider - it would require migrating data from one serverless provider to similar services at another serverless provider. It does however provide advanced solutions without a lot of coding. The AWS implementation of a database and authentication shows that the serverless application requires few lines of code. But as the Azure implementation of a database showed, it can be difficult to troubleshoot. The code is short and concise but is rapidly evolving as the Serverless Framework is updated. There are for example different ways of referencing resources and some of the older ways will not be supported much longer by the framework. There are many potential issues that makes the application and its related resources fail. Luckily the Serverless Framework has an active community and forum for resolving issues. Unfortunately it is mainly focusing on AWS and it might not be possible to take an AWS solution and implement it on Azure or Google. AWS supports many more features in the Serverless Framework, such as the *Resource*-section in the *serverless.yml*-file. This is an example of a feature that is not supported by Azure and Google and could therefore only be implemented in AWS projects.

```
exports.test = (request, response) => {
    response.status(200).send('This is a test route on Google Cloud!');
};
```

Figure 5.11: Google basic GET route

If a more general authentication and database solution is used (in this thesis MongoDB database and JWT for authentication), then the Serverless Framework facilitates easy reusage of code across providers. Most of the code in the *serverless.yml*-file for a back-end HTTP API are the same with some minor changes. The *provider*-section in the *serverless.yml*-file needs to be updated. Minor changes in the *functions*-section such as the name of a route path (called *path* in AWS, *route* in Azure and *name* in Google) need to be updated, see Research Question 1.

Adding a MongoDB database requires an initial configuration on MongoDB's own website as described above in section 5.2.2. The actual code for each route (the handler file) would be very similar between the different providers. But what happens if MongoDB is unavailable but the serverless application on for example AWS is running as it should? This type of error handling needs to be addressed as an additional dependency is introduced.

JWT for authentication was implemented in the AWS project and can be added to the Azure and Google project. This was not done in this thesis to limit the amount of work. Adding JWT to Azure and Google would require adding registration and log-in of users, similarly to the AWS implementation. The JWT token would have to be handled differently than in the AWS project where an IAM policy was created. This is specific to AWS as provider. Npm modules were used to handle the tokens and the encryption (bcrypt). AWS had issues handling the npm module *bcrypt* so *bcryptjs* was used instead. Perhaps bcrypt or bcryptjs might work better than the other if implementing the solution in Azure or Google. As mentioned before it is sometimes difficult to troubleshoot as some errors might not occur during deployment but can be monitored through Azure or Google's own dashboard as discussed in Chapter 5.

The implementation of JWT authentication should be quite straight forward for someone who has earlier experience in utilizing JWT. JWT is an open, industry method frequently used for authentication and there are many guides available online. This solution is however more dependent on the developer. Using a BaaS service such as Cognito User Pool means the developer acquires AWS security expertise for handling its users data. Using a proprietary solution such as Cognito User Pool (AWS) to authenticate users does however require more configurations in the front-end. In this thesis, node modules related to AWS and Cognito were used to access the user pool and add or sign in users. The code had to be modified in the front-end as a fetch through an http-request was not used. Instead the Cognito User Pool was directly accessed. It was not easy to find all the required configurations in AWS own documentation and it took time to go through documentations and determine what was needed. A solution using fetch was found in the AWS documentation but it only discussed sign-in of users and not registration.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Many of the stated benefits of going serverless, such as scalability and pay-as-you-go, would be an advantage for a smaller back-end application. Both Function-as-a-Service (FaaS) and Backend-as-a-Service (Baas) are concepts that might take some time to get familiar with for developers new to serverless. Serverless is marketed as a way of decreasing development time of an application and decrease the complexity in development. For developers new to serverless, configurations would most likely require more time than the actual coding.

This thesis compared deployment in AWS, Azure and Google using the Serverless Framework. The conclusion is that AWS would be the most suitable provider to create a smaller back-end application. The Serverless Framework is by default aimed at AWS and contains support for most AWS events. Thus BaaS services can with the aid of documentation and tutorials be implemented by developers new to serverless. In this thesis DynamoDB database and Cognito User Pool for authentication were implemented. The Serverless Dashboard makes monitoring and troubleshooting easier, there is little need to learn AWS's own tools.

It is more difficult to find documentation, tutorials and support for deployments to Azure and Google through the Serverless Framework. It was not possible to set up any BaaS services in Google and Azure since they were not available in Serverless Framework. Basic http routes were created on all three providers and function syntax could be compared. There were many similarities but each had their own way of stating the route path and Azure for example required an *AuthLevel* on each function to determine if an API key was required or not.

An advantage of using BaaS services is that less code needs to be managed. But it makes it difficult to deploy the same application to another provider. Most code cannot be reused and it would require migrating database and users to another providers BaaS services. The thesis concludes that using a third-party solution for authentication and database results in less vendor lock-in than BaaS solutions. Third-party solutions would still require changes in the configuration file *serverless.yml* and there are some changes required in the function configuration, mentioned above. The handler files (with the actual code) should not require much change in order to be deployed to another provider. Using BaaS services back-end might require more coding or usage of proprietary solutions front-end. This was the case when implementing Cognito for authentication in AWS.

6.2 Future Work

This thesis focused on deploying a serverless back-end API to different providers through the Serverless Framework. There were many similarities between the deployments but also differences to be aware of. Best practices in serverless are constantly adopted to new solutions. AWS is currently the most popular choice for serverless applications but this might change over time. Multiple large vendors might over time streamline code structure and the usage of BaaS services which might decrease vendor lock-in. Conducting a similar study as this thesis in a few years time might render a different result. Perhaps the deployment configurations would be more similar even without the usage of a framework. The Serverless Framework (or a new framework) might be updated to better support other providers than AWS. Another angle would be to implement the same non proprietary solutions for database and authentication on all three providers. By examining the code it could tell how large part of the code base that can be reused between the providers.

Security and permissions are two important areas when working with serverless applications. These are important parts of the serverless architecture and it would be interesting to compare how this is handled by different vendors.

The Serverless Dashboard supports monitoring and troubleshooting of AWS deployments. The solutions to logging and troubleshooting had many differences between the Serverless Dashboard, Azure and Google. Logging can be a difficult to gain a deeper insight into and has not been further explored in this thesis. Perhaps an investigation into this area could help set guidelines for a more generic solution.

Bibliography

[1] Amazon AWS, "The aws serverless documentation," https://aws.amazon.com/serverless/, accessed: 2022-04-02.

[2] M. Amundsen, *What Is Serverless?*, 1st ed. O'Reilly Media, 2020, ch. 1.

[3] O. Andell, "Architectural implications of serverless and function-as-a-service," Master's thesis, Linköping University, 2020.

[4] AWS, "Amazon cognito," https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html, accessed: 2022-04-10.

[5] ——, "Amazon cognito user pools," https://console.aws.amazon.com/cognito/home, accessed: 2022-05-30.

[6] ——, "Aws free tier," https://aws.amazon.com/free, accessed: 2022-06-02.

[7] ——, "Aws provider events," https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-property-function-eventsource.html, accessed: 2022-05-30.

[8] ——, "Serverless auth with aws http apis," https://www.serverless.com/blog/serverless-auth-with-aws-http-apis, accessed: 2022-04-10.

[9] AWS Amplify, "Amazon cognito identity sdk for javascript," https://github.com/aws-amplify/amplify-js/tree/master/packages/amazon-cognito-identity-js, accessed: 2022-05-31.

[10] Azure, "Authentication and authorization in azure app service and azure functions," https://docs.microsoft.com/en-us/azure/app-service/overview-authentication-authorization, accessed: 2022-04-20.

[11] ——, "Azure functions triggers and bindings concepts," https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp, accessed: 2022-05-30.

[12] ——, "Azure portal," https://portal.azure.com/, accessed: 2022-04-15.

[13] ——, "Build in the cloud with an azure free account," https://azure.microsoft.com/en-us/free/, accessed: 2022-06-02.

[14] ——, "Create an azure function that connects to an azure cosmos db," https://docs.microsoft.com/en-us/azure/azure-functions/scripts/functions-cli-create-function-app-connect-to-cosmos-db, accessed: 2022-04-20.

[15] ——, "Httptrigger.authlevel method," https://docs.microsoft.com/en-us/java/api/com.microsoft.azure.functions.annotation.httptrigger.authlevel?view=azure-java-stable, accessed: 2022-06-16.

[16] ——, "Management levels and hierarchy," https://docs.microsoft.

com/en-us/azure/cloud-adoption-framework/ready/azure-setup-guide/organize-resources#management-levels-and-hierarchy, accessed: 2022-04-20.

[17] ——, "Multicloud solutions with the serverless framework," https://docs.microsoft.com/en-us/azure/architecture/example-scenario/serverless/serverless-multicloud, accessed: 2022-04-12.

[18] N. Dabit, *Full Stack Serverless*, 1st ed. O'Reilly Media, 2020, ch. Introduction.

[19] ——, *Full Stack Serverless*, 1st ed. O'Reilly Media, 2020, ch. 1.

[20] S. Eismann and J. Scheuner Et al, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, 2020.

[21] Emma Edlund, "Aws node http api with cognito authorizer," https://github.com/emeu17/serverless-aws-cognito, accessed: 2022-06-16.

[22] ——, "Serverless azure project," https://github.com/emeu17/serverless-azure, accessed: 2022-06-16.

[23] ——, "Serverless google project," https://github.com/emeu17/serverless-google, accessed: 2022-06-16.

[24] ——, "Thesis aws serverless backend," https://github.com/emeu17/serverless-aws, accessed: 2022-06-16.

[25] ——, "Thesis react application," https://github.com/emeu17/serverless-react, accessed: 2022-06-16.

[26] ——, "Thesis react application with cognito," https://github.com/emeu17/serverless-react-cognito, accessed: 2022-06-16.

[27] Gareth McCumskey, "Fullstack course," https://github.com/serverless/fullstack-course, accessed: 2022-04-15.

[28] Github issues, "cosmosdb bindings: Error: Binding direction/name/-databasename/collectionname not supported," https://github.com/serverless/serverless-azure-functions/issues/507, accessed: 2022-04-15.

[29] Google Cloud, "Authentication overview," https://cloud.google.com/docs/authentication, accessed: 2022-04-20.

[30] ——, "Events and triggers," https://cloud.google.com/functions/docs/concepts/events-triggers, accessed: 2022-05-30.

[31] ——, "Free tier products," https://cloud.google.com/free, accessed: 2022-06-02.

[32] ——, "Google cloud console," https://console.cloud.google.com, accessed: 2022-04-25.

[33] ——, "Google cloud databases," https://cloud.google.com/products/databases, accessed: 2022-04-25.

[34] ——, "Google cloud free program," https://cloud.google.com/free/docs/gcp-free-tier, accessed: 2022-06-02.

[35] ——, "Google cloud serverless documentation," https://cloud.google.com/serverless, accessed: 2022-04-02.

[36] ——, "Using iam to authorize access," https://cloud.google.com/functions/

docs/securing/managing-access-iam, accessed: 2022-04-20.

[37] Google Trends, "Interest last five years in term "serverless"," https://trends.google.com/trends/explore?date=today%205-y&q=serverless, accessed: 2022-08-27.

[38] A. Grumuldis, "Evaluation of 'serverless' application programming model: How and when to start serverless," Master's thesis, KTH, Skolan för elektroteknik och datavetenskap (EECS), 2019.

[39] J. Katzer, *Learning Serverless*, 1st ed. O'Reilly Media, 2020, ch. 5.

[40] ——, *Learning Serverless*, 1st ed. O'Reilly Media, 2020, ch. Introduction.

[41] Microsoft Azure, "Microsoft azure serverless documentation," https://azure.microsoft.com/en-us/solutions/serverless/, accessed: 2022-04-02.

[42] MongoDB, "Start with guides," https://www.mongodb.com/docs/guides/?_ga=2.14748918.1353163014.1654192754-14025039.1643312442, accessed: 2022-05-20.

[43] A. Pavlov, S. Ali, and M. T, "Serverless development trends in open source: a mixed-research study," Master's thesis, Chalmers University of Technology and University of Gothenburg, 2019.

[44] React, "Getting started," https://reactjs.org/docs/getting-started.html, accessed: 2022-04-02.

[45] Red Hat, "What is serverless," https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless, 2017, accessed: 2022-02-24.

[46] Serverless Forum, "Azure cosmos db," https://forum.serverless.com/t/azure-cosmos-db/11605, accessed: 2022-04-16.

[47] ——, "Azure cosmosdb yml configuration," https://forum.serverless.com/t/azure-cosmosdb-yml-configuration/12951, accessed: 2022-04-16.

[48] ——, "Azure serverless and cosmodb," https://forum.serverless.com/t/azure-serverless-and-cosmodb/11484/3, accessed: 2022-04-16.

[49] Serverless Framework, "Aws - create," https://www.serverless.com/framework/docs/providers/aws/cli-reference/create, accessed: 2022-05-30.

[50] ——, "Aws provider documentation," https://www.serverless.com/framework/docs/providers/aws, accessed: 2022-05-20.

[51] ——, "Aws provider events on the serverless framework," https://www.serverless.com/framework/docs/providers/aws/guide/events, accessed: 2022-05-20.

[52] ——, "Aws provider getting started," https://www.serverless.com/blog/getting-started-with-serverless-framework, accessed: 2022-04-05.

[53] ——, "Azure functions - quickstart," https://www.serverless.com/framework/docs/providers/azure/guide/quick-start, accessed: 2022-04-10.

[54] ——, "Azure functions serverless plugin," https://github.com/serverless/serverless-azure-functions, accessed: 2022-05-30.

[55] ——, "Azure provider documentation," https://www.serverless.com/

framework/docs/providers/azure, accessed: 2022-05-20.

[56] ——, "Cosmosdb trigger," https://www.serverless.com/framework/docs/providers/azure/events/cosmosdb, accessed: 2022-04-15.

[57] ——, "Google - credentials," https://www.serverless.com/framework/docs/providers/google/guide/credentials, accessed: 2022-04-20.

[58] ——, "Google - quick start," https://www.serverless.com/framework/docs/providers/google/guide/quick-start, accessed: 2022-04-20.

[59] ——, "Google provider documentation," https://www.serverless.com/framework/docs/providers/google, accessed: 2022-05-20.

[60] ——, "How to use mongodb with serverless cloud," https://www.serverless.com/blog/how-to-use-mongodb-with-serverless-cloud, accessed: 2022-05-20.

[61] ——, "Http api (api gateway v2)," https://www.serverless.com/framework/docs/providers/aws/events/http-api, accessed: 2022-05-31.

[62] ——, "Introducing serverless cloud," https://www.youtube.com/watch?v=0lGNFFQt5No, accessed: 2022-06-02.

[63] ——, "Rest api (api gateway v1)," https://www.serverless.com/framework/docs/providers/aws/events/apigateway, accessed: 2022-04-17.

[64] ——, "Serverless azure functions events," https://www.serverless.com/framework/docs/providers/azure/events, accessed: 2022-05-30.

[65] ——, "Serverless cloud - full stack development," https://www.serverless.com/cloud, accessed: 2022-06-02.

[66] ——, "Serverless dashboard," https://www.serverless.com/framework/docs/guides/dashboard, accessed: 2022-05-01.

[67] ——, "Serverless framework concepts," https://www.serverless.com/framework/docs/providers/aws/guide/intro, accessed: 2022-06-05.

[68] ——, "The serverless framework documentation," https://www.serverless.com/framework/docs, accessed: 2022-04-02.

[69] ——, "Serverless full stack application course," https://www.youtube.com/watch?v=XTJImzRH8aY&list=PLIIjEI2fYC-BZliSOIhWUqiiwadhCvewg, accessed: 2022-04-15.

[70] ——, "Serverless google cloud functions events," https://www.serverless.com/framework/docs/providers/google/events, accessed: 2022-05-30.

[71] ——, "Serverless google cloud functions plugin," https://github.com/serverless/serverless-google-cloudfunctions, accessed: 2022-05-30.

[72] ——, "Serverless guides and tutorials," https://www.serverless.com/category/guides-and-tutorials, accessed: 2022-04-12.

[73] State of JavaScript, "Front-end frameworks," https://2021.stateofjs.com/en-US/libraries/front-end-frameworks, 2021, accessed: 2022-02-22.

[74] D. Taibi, J. Spillner, and K. Wawruch Et al, "Serverless computing - where are we now, and where are we heading?" *IEEE Software*, vol. 38, no. 1, pp. 26–31, 2020.

[75] J. Zhang, "Multi-level faas application deployment optimization," Master's thesis, Chalmers University of Technology, 2021.

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden