Aleksandr Semjonov

Implementing serverless web application backend using AWS Lambda

Bachelor's thesis
Information Technology

2019

South-Eastern Finland
University of Applied Sciences

Author (authors)	**Degree**	**Time**
Aleksandr Semjonov	Bachelor of Engineering	May 2019
Thesis title Implementing serverless web application backend using AWS Lambda		46 pages 2 pages of appendices
Commissioned by Matti Juutilainen		
Supervisor Matti Juutilainen		

Abstract

This work investigated the serverless application design using AWS Lambda provided by Amazon as a cloud platform of choice. Theoretical part described both the industry standard approach to servers and the upcoming serverless technology, briefly touching on the history of the former and providing an in-depth explanation and introduction of the latter. The advantages and disadvantages of serverless application design was discussed in detail, without a significant focus on AWS over other cloud platform providers.

In the practical part, a small-scale real-world application specification was introduced. The application in question was a supplementary backend for a previously created website, providing a service of private messaging to existing users, while being invisible for the end-user. Step-by-step instructions are provided to implement a similar application using AWS Lambda and Node.js. Relevant code snippets and configuration files were included and explained in detail.

As a result, a working application was produced, tested and deployed to cloud environment. A list of encountered problems and possible solutions was provided, and monetary benefits of serverless were discussed. The goal of the work was accomplished.

Keywords

Serverless, web development, backend, JavaScript, AWS Lambda, Node.js, cloud architecture

CONTENTS

1 INTRODUCTION

“A new model, called serverless computation, is poised to transform the construction of modern scalable applications” (Hendrickson et al. 2016). The introduction of applications developed with serverless cloud technologies has a significant chance to change the way both programmers and information technology engineers think about the backend infrastructure. While implementation details are abstracted by well-documented public APIs, developers are free to spend their valuable time on the business needs of their application. At the same time, reliance on highly developed and feature-rich services, provided by a cloud service provider, allows higher degree of flexibility in the future.

Of course, no new technology is ever able to enter the market without flaws in its design. According to Hendrickson et al. (2016), serverless has been well-known since at least 2016, and during the years major platform providers have had the chance to make their serverless solutions available to public while iterating on and improving the concept. If we investigate the technology at this time, most of the issues have either been completely resolved, or the workarounds have been found, and any other problems encountered today are likely to be among the list of core drawbacks of serverless. Thus, it is a good time to investigate whether or not serverless is worth the attention it’s getting.

The goal of this work is to investigate both the potential benefits and drawbacks of a serverless design approach and utilize it in a practical setting to determine whether or not the real-world results will match theoretical expectations. The tools of choice for this work are AWS Lambda by Amazon, with Serverless Framework as a build/deployment tool and JavaScript with Node.JS as the programming language. None of these technologies is a requirement to use serverless nowadays. However, according to Waterworth (2018), they are the most established players in their respective fields.

2 DEVELOPMENT TOWARDS SERVERLESS ARCHITECTURE

Before diving deep into the concept of a serverless application, it is important to understand some concepts that are common to the industry standard approach – servers. Since the early years of computer science and first networked computers, the IT engineers have gained large amounts of valuable experience that has become the base knowledge to develop modern Internet infrastructure. Even the standard approach nowadays has a number of concepts in common with the serverless architecture, namely the idea of maximizing the abstraction and virtualization up to the point of completely removing the constraints of a single physical machine, and these topics are the ones to be discussed first here.

2.1 Abstraction and virtualization

“The rapid pace of innovation in datacenters and the software platforms within them is once again set to transform how we build, deploy, and manage online applications and services. In early settings, every application ran on its own physical machine. The high costs of buying and maintaining large numbers of machines, and the fact that each was often underutilized, led to a great leap forward: virtualization. Virtualization enables tremendous consolidation of services onto servers, thus greatly reducing costs and improving manageability.” (Hendrickson et al. 2016)

The concept of virtualization has been incredibly important in both computer science and information technology. Programmers use this concept while designing the architecture of their code all the time, cloud engineers provide services based on the concept, and even networking engineers find the benefits of virtualization attractive. It would not be possible to design systems as complex as we have today without the abstraction of concepts and virtualization of objects.

Whenever a person uses some cloud service, they interact with a virtualized environment without even realizing that this is the case. Google, Amazon, Apple, Microsoft and other cloud providers would not be able to offer their services on such a scale without virtual machines running in their data centres around the world. These VMs need to interact with the shared storage which is abstracted away and instead presented as a simple-to-use interface, while also performing security checks to ensure that the information is available only to authorized parties. These steps happen automatically and are completely invisible to the end-user.

Every device connected to the world-wide-web requires a unique identifier, and when the system was designed, the commonly used Internet protocol standard became known as IPv4. There are only 3,706,452,992 public IP addresses available, whereas in 2017 there were over 20 billion devices connected to the Internet (Statista, 2019). As the IPv6 protocol has not yet been able to replace its predecessor, the world relies on the network address translation (NAT) to solve a problem, and this approach qualifies as abstraction as well.

The operating systems we commonly use in our daily life are built with many different layers, and each one of them abstracts some information from the layers above and below. The highest level is an application. It is not and it should not be aware of the details of hardware it is running on. It only interacts with the OS layer using the provided API. Then, even though the operating system is in control of the hardware, it doesn’t need to interact with it directly. There are device drivers which handle the precise control and they provide some sort of communication method for the OS to give commands and data. These drivers then translate the commands into a set of messages that are understandable by this specific piece of hardware and send them to the correct place in the memory. The abstraction rule also works backwards, and the drivers do not need to be aware of any applications that are running in the operating system, and they only need to interact with the latter.

2.2 The way we used to do it

While high levels of virtualization became an industry standard in the recent years, the classical approach to the service backend, which relies heavily on the dedicated physical servers running in a datacenter, is still in use in many legacy applications. These machines may or may not run clusters of VMs that provide some layer of abstraction and separation for the services running there, but in the end, it all comes down to the fact that there are physical machines which can malfunction, which require proper cooling and other types of maintenance. All of those problems need to be handled by the owner of the servers, and it is also important to consider secondary services, like the backup, storage and databases. As with any approach, there are pros and cons, and it is important to understand all the details.

One of the most valuable advantages that the “old way” provides is the full control over most of the operations. We have absolute power to use our devices to whatever purpose we feel is necessary. We can put their whatever load is required and the cost of running the servers will not change significantly. At worst, increased load requires higher cooling capacity.

There have been servers deployed around the globe even before the World Wide Web became what it is today, and the development of the best practices has also begun at that time. During the years hundreds of practices and recommendations have been proposed, tested and either discarded or adopted in common use. When deploying a server nowadays it is relatively easy to run through a full checklist of all the things that are to be done in order for the installation to be run reliably and securely, and the time has proven that these practices are actually useful.

Following from the previous point, the time-tested approach to servers means that it is relatively easy to find experienced people to deploy, support and maintain the servers. Senior DevOps engineers may have decades of experience in the field, which effectively guarantees high level of security and reliability on the servers they install. Also, some degree of familiarity with the servers is important even for the software engineers, meaning that the developers will have some understanding of the underlying system architecture when coding a new feature or a brand-new service.

While the traditional approach provides a significant number of advantages, one can find a counter-argument for every one of them. Full control over the physical machines means that any problems the hardware encounters have to be thought of in advance and the failover has to be planned for that case. In some situations, it might even mean duplicating the whole service and putting the secondary machine into a hot failover state, which means that this piece of hardware is not performing any useful work, while still consuming the same amount of cooling and electricity resources as the primary hardware.

Keeping the server software and hardware up-to-date, properly cooled and running implies maintenance costs. These can be either direct, like in the case of cooling and electricity, or indirect, for example, in the time of DevOps team, which tends to be the major part of all the server maintenance.

Creating a new instance of the service or installing a new machine to scale up the production is typically a time-consuming task. It may take days or weeks for the new hardware to arrive and just as much time to install all the necessary software while making sure that the existing environment is not affected by the installation. In case of serverless approach, all of these delays can be reduced to the scale of minutes or even seconds.

One may think that there are also containers, which are a comparably new solution to a similar problem. However, containers have a significant disadvantage over the serverless applications, and that is time to boot. A container takes seconds to get up and running itself, and only after that the actual service inside starts to initialize. It can be completely unacceptable level of delay for the time-critical applications. Serverless applications, even though not the perfect solution for that use cases themselves, provide a much better experience.

3 INTRODUCTION OF SERVERLESS

In terminology used by Amazon, serverless applications work on a level of Lambda functions, which contain and run the code created by the user. The Lambda functions are self-contained entities that are able to communicate with each other and other services using the event system. The design model utilizing the Lambda functions can also often be called the Lambda model.

“The Lambda model has many benefits as compared to more traditional, server-based approaches. Lambda handlers from different customers share common pools of servers managed by the cloud provider, so developers need not worry about server management. Handlers are typically written in languages such as JavaScript or Python; by sharing the runtime environment across functions, the code specific to a particular application will typically be small, and hence it is inexpensive to send the handler code to any worker in a cluster. Finally, applications can scale up rapidly without needing to start new servers. In this manner, the Lambda model represents the logical conclusion of the evolution of sharing between applications, from hardware to operating systems to (finally) the runtime environments themselves.” (Hendrickson et al. 2016)

Despite the fact this design mentality literally claims that no servers are involved, it is not exactly true. The physical machines running the service will always be there, but the way we think about the servers is completely different. Instead of considering that there is a machine we have to manage that has a certain CPU with some amount of RAM and persistent memory, we have our code just running somewhere. There are core limitations for CPU cores, RAM and execution time, which are mentioned in chapter 5, but for most use cases, if we need more resources, we just have more resources. It happens instantly and automatically, without any intervention from the programmer. It only affects the budget at the end of the month.

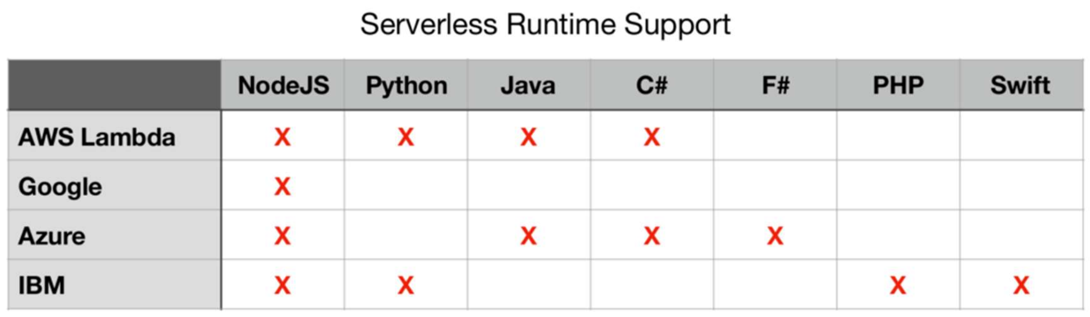
But aside from the core concept, there are other ideas which also define serverless. In general, with the serverless app we outsource as many services and infrastructure to the platform provider as possible, and these services often include authorization, authentication, accounting, database control, storage, media processing and others. These types of services are required to be implemented in most applications, but their general codebase is often very similar. Using an existing implementation that has been polished to the point of near-perfect state is a logical step to reduce the development time and costs, while also increasing reliability of the service we’re building.

The serverless architectures are often complex networks of different services that interlink with each other and the business logic of our application. These networks are entirely event-driven, meaning that only an interaction from a user or another system will trigger any changes in the application. If there are no events to process, i.e. the application is not used during the night-time, or the staging deployment is inactive after working hours, the system is softly shut down to prevent any excess billing.

At the moment of writing this thesis, we may choose one of the four main serverless platform providers: Google Cloud Platform, AWS Lambda by Amazon, Azure Functions by Microsoft and OpenWhisk by IBM. All of them have their own advantages and disadvantages, but the AWS Lambda was the first service, launched in 2014, that defined serverless as we know it today.

Even though nowadays a programmer is effectively required to know multiple programming languages to work with large and complex products currently on the market, there is always a preference for one or the other language. It may be a project requirement, like using C for the highest possible performance, or collective consensus that this specific language works best for that project. Regardless of the reason, the main programming language for the application is an important factor when choosing a serverless platform. Waterworth (2018) provides a useful table to help choose the best combination of a cloud platform and the programming language.

Table 1. Serverless runtime support (Waterworth 2018)



Serverless Runtime Support

	NodeJS	Python	Java	C#	F#	PHP	Swift
AWS Lambda	X	X	X	X			
Google	X						
Azure	X		X	X	X		
IBM	X	X				X	X

As can be seen from the table above, not every platform supports every programming language, and while all major players in the field have support for Node.JS – backend-oriented JavaScript environment – most other technologies are specific to one or two of the platform providers.

3.1 Common serverless components in the AWS cloud platform

As mentioned by Roberts (2018), a platform that provides serverless capabilities typically also provides an array of supporting services to allow the intended use of the serverless code. These services include authorization, authentication, logging, storage and other APIs. While not being exactly the same across the platforms, these services are similar enough that a list of basic components available on AWS will give a good overview of their counterparts available on other platforms.

- **AWS Lambda** is the core component that contains our custom business logic and runs the code in the cloud. Lambda supports a variety of features itself like automatic version control and automatic deployment.
- **AWS Step Functions** is the state machine that orchestrates the serverless workflow. It is able to coordinate the Lambda functions and manage the internal state of the running process. Using the Step Functions the Lambda functions can easily be decoupled and abstracted away to increase the modularity of our architecture.

- **Amazon API Gateway** is an essential component for any application that relies on REST API. The gateway is able to accept and route incoming requests, while also managing access control, monitoring and versioning.
- **Amazon DynamoDB** is a persistent NoSQL database that stores the current state of the application. When used in conjunction with DynamoDB Streams, Lambda functions can be invoked nearly instantly after a database state is modified.
- **Amazon Simple Storage Service (S3)** is a storage service for static resources like HTML web-pages or media files. These resources can also be made available to users using a content delivery network, and Amazon provides CloudFront to do just that.
- **Amazon Cognito** is a user authentication and authorization service. It is able to support user sign-in, login, SSO (Single Sign-on) and data synchronization when necessary.

In addition to the mentioned components, AWS provides 52 more services in many areas, such as game development, machine learning, IoT and others (Amazon 2019). While using them is not a requirement for serverless, their inclusion may be beneficial if the services' use-cases match the ones of the application being built.

3.2 Example architecture

As a part of AWS documentation (Amazon 2019), Amazon provides a diagram that illustrates a simple infrastructure built on their platform, pictured on Figure 2 below. The structure described by Amazon is not a requirement or a system limitation, but only a guideline of the way their systems and services are intended to be used in the classic use-case. The structure is to be expanded upon or completely replaced if it is unsuitable for the current use-case.

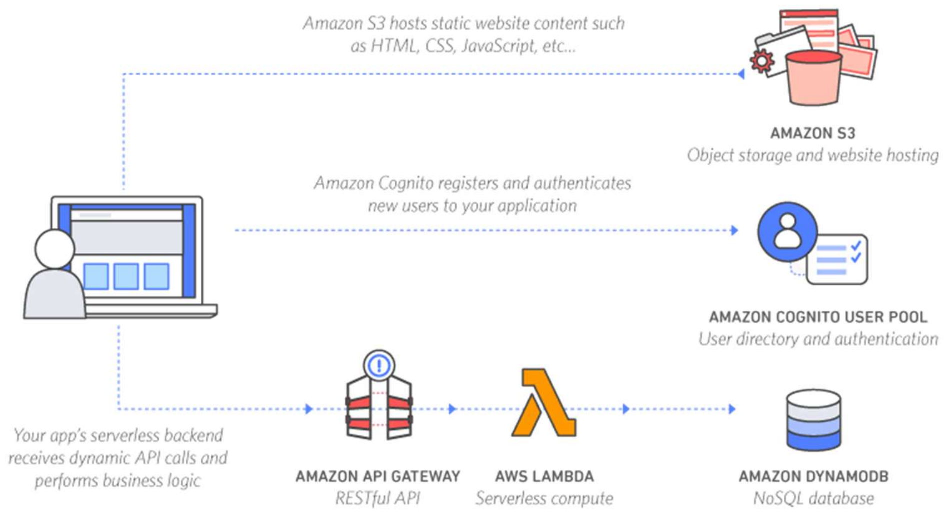

Amazon S3 hosts static website content such as HTML, CSS, JavaScript, etc...
AMAZON S3
Object storage and website hosting
Amazon Cognito registers and authenticates new users to your application
AMAZON COGNITO USER POOL
User directory and authentication
Your app's serverless backend receives dynamic API calls and performs business logic
AMAZON API GATEWAY
RESTful API
AWS LAMBDA
Serverless compute
AMAZON DYNAMODB
NoSQL database

Figure 1. Example Serverless architecture (source: Amazon)

Generally speaking, any static content is expected to be hosted in an S3 bucket where the users have direct access to. In case some of the files need to be protected, for example, require user authentication, it can be done in S3 as well. User authentication is to be handled using Cognito directly, without invoking the Lambda functions at this step. Only after the user has a valid session, they are expected to make calls to Lambda functions performing the business logic of the application. The requests always travel through API Gateway that handles load-balancing and caching, and the functions have access to DynamoDB as a storage engine.

4 BENEFITS OF SERVERLESS

The serverless computing model comes with a significant number of solutions for problems that have been a defining factor in the system design over the last decades. They mostly relate to the costs and maintenance factors, but the serverless approach is also capable of changing the whole way the developers look at the servers, creating some entirely new patterns and practices. It has incredible potential to allow smaller businesses to run their operations with the same efficiency as old and established enterprises, thus in turn, creating healthier competition in the industry that has been dominated by the big players for years. However, we should take a look at all the benefits and their impact one by one. This chapter is based on “Serverless Architectures” article (Roberts 2018).

4.1 Reduced operational costs

“Serverless is at its most simple an outsourcing solution” (Roberts 2018). Instead of developing our own systems with all the up-front costs for the datacenter rent, hardware, operating systems and other software, we are now able to click a number of buttons in a friendly user interface and to get our code running. A few extra clicks will also enable us to use an infinitely-scalable database with built-in protection and a large set of features that we may or may not use in the future, as well as a complete authentication and authorization solution that seamlessly integrates with all the other services. All of this has happened while we didn’t pay anything, because there is no usage of that service skeleton yet.

As the service is developed and released to the public, the customer base increases from the size of a single development team to hundreds, thousands, maybe even millions of users, and there is no need for us to invest in new servers, new storage, improved networking, clustering technologies or failover mechanisms. All of them have already been handled for us, and the only thing that is required from the developers is to increase the usage thresholds, if they have been setup at some point. The bill size will increase with the number of customers and requests they make to our service, but there is no need to maintain anything but the actual software used, thus minimizing the operational costs. It may even be possible to completely avoid creating a separate dev-ops team, and as people consume a high percentage of all the businesses’ monthly costs, this can lead to a significant level of money savings. Additionally, in case that a service is no longer needed for whatever reason or has lost all the users, it can be shut down easily and instantly, without the need to manage the hardware that has lost its purpose.

4.2 Economies of scale

One of the problems that a software-development startup will encounter is the cost of doing the operations on their own. Initial investments are costly, and in order to maximize the user satisfactions, the servers should always have extra capacity reserved in case of a usage spike. The number of users also typically will fluctuate depending on the time of day, rising during the working hours and dropping nearly to zero during the night. Global enterprises, however, do not suffer from that problem. As their services are available world-wide, the number of users throughout the day is more stable, thus making the hardware work more evenly. Additionally, if the enterprise already has a massive server cluster setup for some service, it may be reasonably easy to run an extra VM with a new service for testing purposes. The experimentation is, once again, quite costly for the new players in the field.

With the serverless approach even the start-ups are able to utilize high efficiency that is provided by economies of scale. All the “heavy-lifting” has been done by the service provider (Amazon, Google, Microsoft), and we are able to use their platforms as if they were our big server clusters. Granted, we are still required to pay for our usage, but in comparison with the costs we would have if we ran every service on our own hardware, the benefits are significant.

If the new start-up companies are allowed easier access to the field, they can stimulate competition even with their limited resources, and some amount of healthy competition is required to move the progress forward. It used to be the case that big enterprises are able to win by pure power and efficiency of the way they do operations, making their services both cheaper and better for the end users. However, by utilizing the serverless approach it is possible to match those parameters even without investing billions into high-performance hardware clusters.

4.3 Reduced development costs

Another interesting consequence of introducing serverless design into an application development process is the reduced development cost, both in terms of time and money. A team of developers is often required to spend their valuable time for programming not the actual business logic, but the supporting code which is not required in case of a serverless application. However, in case of Amazon, Google or other platform providers, they have already invested their time and effort in designing robust and feature-rich systems that we are now able to use for solving our problems.

Another part of the same idea would be the fact that up to a certain point we can even assume that any problem with our own service is caused by a hidden bug or a miscalculation in our business logic, while the supporting serverless services are operating normally. The chances of a fault on the platform provider’s side are incredibly low, and any high-scale outage will sound alarm bells all around the world, making it a well-known occurrence. The confidence in the supporting services makes the actual service we develop easier to debug, save time, effort and, in turn, money.

4.4 Global delivery

Currently it is common to use all sorts of global content delivery networks (CDNs) to offload some of the servers’ traffic onto more powerful dedicated machines controlled by another company. It can help tremendously with both latency for the user and the peak load our own servers need to handle. However, with the serverless application design the concept of CDN is built into the core design of the design method and, in turn, our application itself.

The code we create for a serverless app is automatically distributed around the world and the containers with that code may be running in a datacenter that is closest to the user. While this may not be the default behaviour, there is nothing that will really prevent that. Distributed operations, just like CDNs, will help decrease latencies for the user and make sure their experience is optimal.

4.5 Scalability

One of the key benefits for serverless applications is the ability to easily scale both up and down with the number of users and/or requests that hit our service.

This process is completely automated and new instances will be created when the demand is high and destroyed when the active user amount drops back down. The ability to scale easily makes serverless app a perfect solution for some use cases that include, for example, bursty workloads, irregular requests or fluctuating user amount.

The code that we are running in the cloud can also scale both horizontally and vertically. In the current context, vertical scaling means that every single instance has access to more resources for their task, and horizontal scaling is a term describing the ability to run multiple instances in parallel. This way we can serve a theoretically limitless number of users while dedicating more than enough resources for each. There are some fundamental limitations for serverless computing however, but those are mentioned in better details in the drawbacks section.

4.6 Environmental impact

“Gartner (2019 has long talked about the "80% rule": that 80 percent of IT budgets get spent simply "keeping the lights on", this survey seeks to wrap some clarity around that. According to McKinsey and Company, typical servers in business and enterprise datacenters deliver between five and 15 percent of their maximum computing output on average over the course of the year.” (Kepes 2015)

If this statement is accurate, anywhere between 85% and 95% of computing power available to privately owned servers around the world is completely wasted. The servers are required to have high performance margins to accommodate for possible performance spikes. However, if we were able to combine all the wasted computational power, it would be possible to run much fewer servers around the world, saving high amount of energy and, in turn, reducing the amount of fuel burned to provide that energy.

5 DRAWBACKS OF SERVERLESS

As with any other technology in the market, serverless has a number of disadvantages that need to be considered before locking yourself down for this system design approach. This chapter is based on “Serverless Architectures” article (Roberts 2018).

5.1 Vendor lock-in

One of the first things that comes to mind with the serverless applications is the vendor lock-in. When an application is developed for a specific vendor, be it Amazon, Microsoft, Google or other, it may be problematic to then move that application to another platform. The core ideas of serverless are similar across the board, but some of the features provided by platform providers may differ significantly.

Even though there are some design patterns and methods to prevent the lock-in, it also prevents us from using some of the features of our platform of choice, moving more of the logic into our code which in turn partially defeats the purpose of serverless, which is to delegate as much work as possible to the platform owner.

5.2 Shared hardware

Even in serverless applications there are still servers. And these servers are highly virtualized environments which are being used by hundreds of thousands of users at once. This may cause significant problems for multiple reasons, these being, for example, performance, security and reliability.

The performance of all the users will degrade, if just one of them consumes a large portion of processing time. It is unlikely to happen due to the core limitations, which are mentioned in more details in chapter 5.3, and just because of the sheer amount of power the servers are equipped with. But, for smaller service providers and/or smaller server clusters it may come into effect.

Even though the platform owners make their best to abstract away any sort of hardware they actually use, in case of a configuration error, hardware failure, network failure, or some other unforeseen circumstance, it is possible that there will be an information leak between different applications. They share the same host and the same physical memory modules, meaning that in some very rare cases the information leak is possible. It touches the aspect of security, but it’s yet another reason to consider not switching all the applications to serverless design model.

Inherently, the serverless application is running in the cloud datacenter which is mostly out of our control. It means that this approach may not be viable for some critical applications, for example military or governmental projects with high security requirements. We need to trust a specific platform provider to trust them with all our data and all our source code, but if for one reason or another we are unable to do so, the serverless applications are not a viable solution. The development, deployment, debugging and everything else is done over the network which is reasonably reliable nowadays, but man-in-the-middle attacks are still a thing and such an attack may compromise the whole project.

5.3 Core limitations

Any cloud platform, would it be serverless application or any other type of service, it has a set of core limitations. As the main focus of this work is on AWS Lambda service, we can take a look at their upper execution limits as a guideline an example.

Table 2. AWS Lambda execution limits (Amazon 2019)

Resource	Limits
Memory allocation	3008 MB
Ephemeral disk capacity ("/tmp" space)	512 MB

Number of file descriptors	1,024
Number of processes and threads (combined total)	1,024
Maximum execution duration per request	300 seconds
Invoke request body payload size (RequestResponse/synchronous invocation)	6 MB
Invoke request body payload size (Event/asynchronous invocation)	128 KB

Even with these values it is important to understand that the limits are not necessarily a negative thing. They force us to think and design in a certain manner, in this example it is to build smaller, more specialized Lambda functions that will, in turn, make our application more flexible. If we think about Lambda in the way it is intended to be thought about, those limits become absurdly high and we are unlikely to even hit any of them. Still, it is important to keep them in mind and in case something goes wrong with the service, it may be caused by a mistake or a design flaw that would cause the Lambda instance to hit one of the limits.

As a side note about the importance of the limitations, limits have a role in making sure that the money is not drained from the product owner's pockets for nothing. The code will fail from time to time, especially in the development phase, and the errors can cause massive memory leaks, hanged execution, high CPU usage or some other undefined behaviour. As the code is running on the cloud platform, the system has no way of knowing if that type of behaviour is intended or it is caused by a typo in the code. Hard limits will mitigate some of the monetary damage that can occur in that case, making them a drawback, but a debatable one.

5.4 Start-up latency

As any system, Lambda functions need some time to boot up. This delay time can vary from individual milliseconds to tens of seconds, and it depends on multiple factors: language used, amount of code, dependency lists, configuration and so on. After the initial cold start, the subsequent requests will be served significantly faster as Lambda is stored in the memory and re-used multiple times. However, it will be retired after a period of inactivity, which is 5 minutes in case of Amazon. (Amazon 2019)

This fact can either be of no consequence for an actively used application that will only experience a cold boot after a code update or a manual restart, or it can cause significant problems for an application that is only used from time to time while requiring near-instantaneous response. This issue, however, is recognized by the service providers and is a subject of continuous improvement making it a smaller concern over time, but still an issue worth discussing.

5.5 Potentially low flexibility

As mentioned in the chapter 4, the best practice of the serverless application design is to use as many services implemented by the platform provider as possible, while focusing internal development efforts on the business logic of the application. While this approach is an efficient one in most cases, there is always that one extra use-case where the existing service is not suitable. If there is no way that the application is able to mitigate that issue by using other service or by somehow modifying the existing behaviour, it may be possible that the platform provider doesn’t have a suitable service and there is a need to implement that functionality using internal development power.

However, especially if the service in question is a critical one, like authentication or access control, all the other services are often designed in a way to allow easy integration with a built-in authentication service, while not having any compatibility with an externally developed one. This may create a challenge that leads to the development team creating more and more services for internal use, which defeats the entire purpose of a serverless application. If that is the case, the platform that is being used only creates problems instead of helping solve them.

6 IMPLEMENTING SECONDARY BACKEND USING SERVERLESS

In the modern world where website constructors like Wordpress or Wix are quite common, an interesting problem may arise. While the developer has access to the front-end, the back-end is completely out of their control. It is not problematic until the service provides all the necessary features, but as soon as the developer decides to do anything non-standard, they immediately run into trouble. If they don’t have access to the back-end, implementing any complex feature becomes nearly impossible, but that can still be solved with the help of serverless.

This thesis will focus on implementing a “supporting back-end”, following the idea that a website is served from the primary server, but some functions are outsourced to this service. Supporting back-end needs to provide a mechanism for secure authentication and its API needs to be as simple as possible. The exact function of that service is not critical for the topic of this work, but in our case, it will provide a secure messaging system.

Let us imagine a theoretical website where users are free to publicly post their thoughts and opinions. Any user on the website needs to be authenticated, but other than that all the content is public. Later, the owner of the site wants to add a feature to add private content to the posts, targeted to specific users. They can be, for example, warnings for the users that break the rules or just extra information for administrator users only.

6.1 Technical specification

To be more specific with the requirements, the following requirements must be met for the service to be considered finished:

- Safe and secure way to authenticate the user
- Ability to send a message to a single user, multiple users or a user group
- Ability for the user to read all messages sent to them

Going into even more detail, a sent message is exchanged for an UUID, which is then embedded into the message. From the primary back-end's point of view, this is just a part of the post that doesn't have any special meaning, but the custom front-end code will detect the code, contact the supporting back-end and either replace the code with the message body or hide it, if the user doesn't have access.

In practice, these specifications mean that there will be three endpoints available for the front-end to call as follows:

- `/login`
  Required parameters: username, application token, secret
  Returned value: Authenticated JSON web token or error code
- /message/get
  Required parameters: authentication token, message id
  Returned value: Message body and list of receivers
- /message/send
  Required parameters: authentication token, receiver, message
  Returned value: Created message id or error code

All communication is performed using the JSON message format. Authentication token is sent in the Authorization HTTP header in the same format as the OAuth authorization header, word "bearer" followed by a space character and a token:

6.2 Selecting the technology

Even at this point in the development of AWS Lambda, there are multiple ways to approach the development of a serverless application. Official guides from Amazon propose a "drag-and-drop" way, showing screenshots of the AWS console UI and pointing to the correct buttons to click (Amazon 2019). However, this type of deployment is prone to errors, requires lots of manual labor to set everything up and creates problem when creating multiple deployment stages, i.e. development, staging and production.

One alternative to AWS console is Serverless Framework (Serverless Framework 2019). It is a command line tool that offers a standardized way to develop serverless applications on all major platforms, including Amazon. With Serverless Framework the infrastructure is described as a single configuration file which is then translated into a format suitable for the target platform, which is CloudFormation in case of Amazon. Serverless Framework simplifies the deployment significantly, essentially reducing the work to the following command:

```
$ serverless deploy
```

Under the hood, this command creates a .zip archive with all the Lambda functions and their dependencies, uploads it to the S3 bucket and initiates a CloudFormation stack update. After the execution is finished, the service is deployed to Amazon servers and is available on a randomly generated URL. The simplicity and convenience of the deployment are the reasons this work uses Serverless Framework over alternatives.

6.3 Installing dependencies

As this project is based on Node.JS technology, dependency management is done via the NPM system. All dependency modules are installed from the public repository using the following command:

```
$ npm install {package-name}
```

The only three packages we need are the serverless framework itself, JSON web token library and a hashing library to safely store the passwords. I have decided to use bcrypt algorithm, but it may need to be upgraded to a more secure option when it becomes obsolete. The package names are as follows:

```
Hashing library: bcryptjs
Web token library: jsonwebtoken
```

Please note that the recommended way to install Serverless framework is to use global mode, adding –g flag to the install command. The resulting command looks as follows:

```
$ npm install –g serverless
```

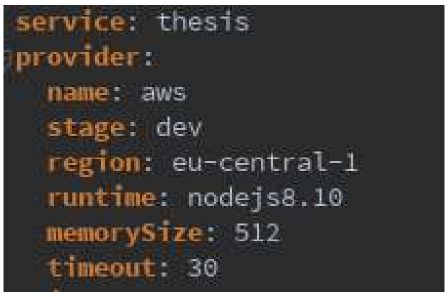
Next, the Serverless Framework. A first step to deploy a service using Serverless Framework is to connect it to the AWS account. That requires creation of a new IAM user, enabling programmatic access to said user and downloading the secret access keys. Then, the following command will configure the framework to use the credentials globally:

```
$ serverless config credentials --provider aws --key KEY --secret
SECRET
```

Global configuration is arguably the most convenient available option; however, the alternatives provide better security and flexibility. Please refer to the official website for more information.

https://serverless.com/framework/docs/providers/aws/guide/credentials/

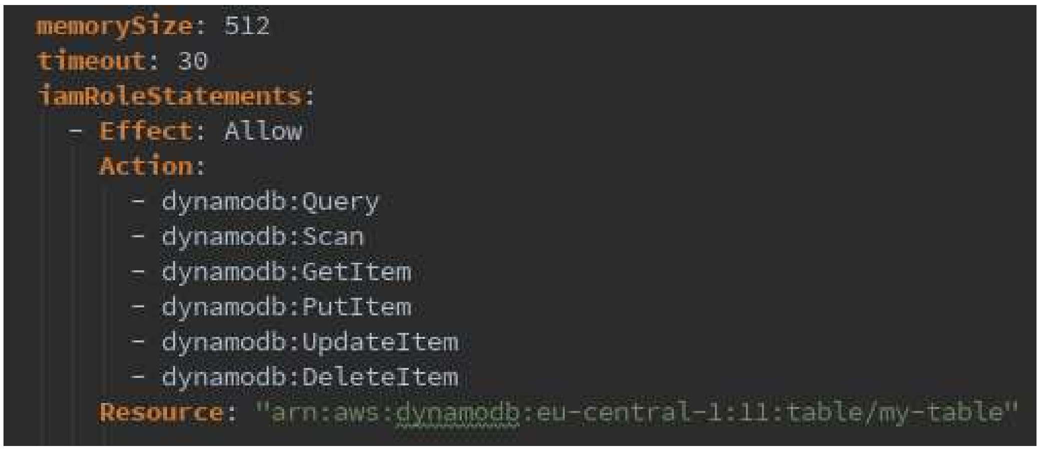
The core of Serverless Framework is **.yml** configuration file that contains full description of used resources and services. During the deploy process, this information is translated into appropriate format and transferred over to CloudFormation, which in turn provisions and initializes the requested services and resources. A typical configuration file may begin as shown in Figure 3:



```
service: thesis
provider:
  name: aws
  stage: dev
  region: eu-central-1
  runtime: nodejs8.10
  memorySize: 512
  timeout: 30
```

Figure 2. Serverless Framework minimal configuration

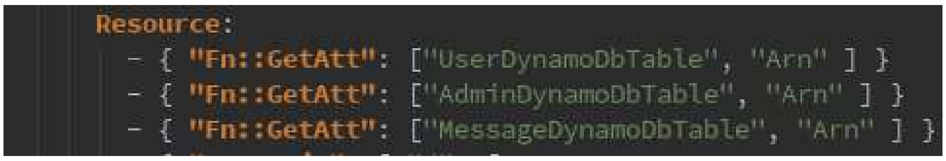
With those lines in place an application may already be deployed, however, it will not provide any functionality. The next step in configuration is to setup the DynamoDB database which will store all the data of the service. The first part of the setup is to create appropriate permissions to access the database, as pictured in Figure 4.



```
memorySize: 512
timeout: 30
iamRoleStatements:
  - Effect: Allow
    Action:
      - dynamodb:Query
      - dynamodb:Scan
      - dynamodb:GetItem
      - dynamodb:PutItem
      - dynamodb:UpdateItem
      - dynamodb:DeleteItem
    Resource: "arn:aws:dynamodb:eu-central-1:11:table/my-table"
```

Figure 3. DynamoDB permission configuration

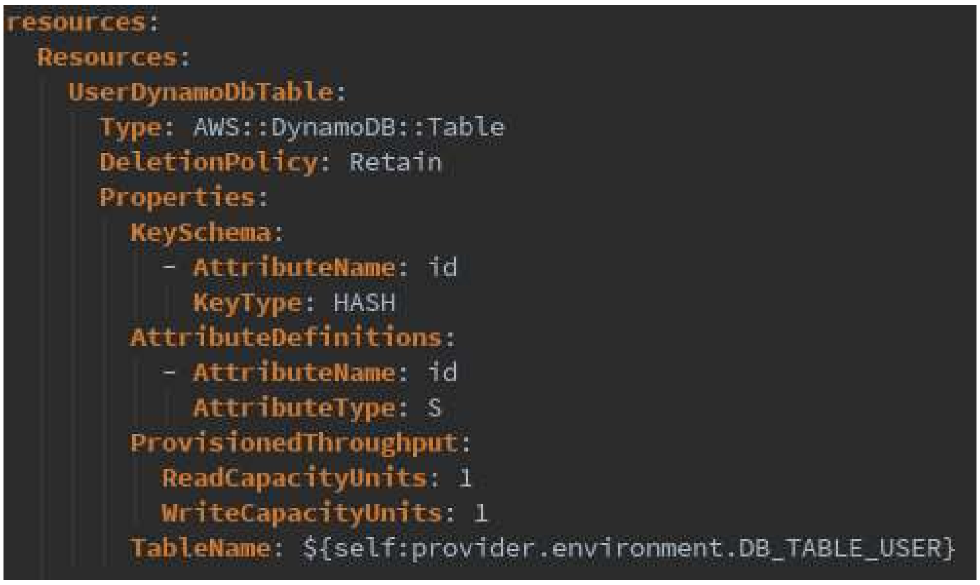
The code in Figure 4 allows the table with given **ARN**, which is AWS unique resource identifier format, to be queried and updated as required, but an issue here is the fact that this identifier is unknown until we create the table. A solution is to look up the ARN dynamically, as shown in Figure 5.



```
    Resource:
      - { "Fn::GetAtt": ["UserDynamoDbTable", "Arn" ] }
      - { "Fn::GetAtt": ["AdminDynamoDbTable", "Arn" ] }
      - { "Fn::GetAtt": ["MessageDynamoDbTable", "Arn" ] }
```

Figure 4. Obtaining dynamic DynamoDB table IDs

`{...}DynamoDbTable` are identifiers declared below in the config file. Now the permissions are set up and we can move forward to creating the tables. The segment demonstrated in Figure 6 declares one of the required tables, specifically `UserDynamoDbTable` that will contain the registered users' data. The declaration is located on the same hierarchy level as `service or provider`, meaning the segment does not require indentation.



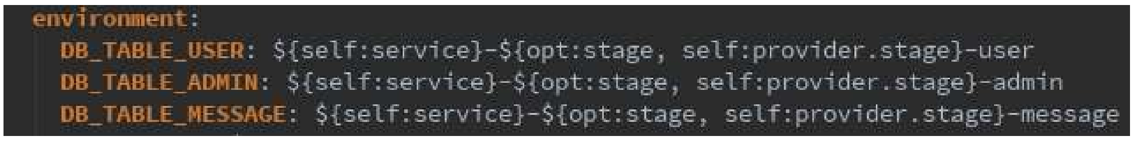
```
resources:
  Resources:
    UserDynamoDbTable:
      Type: AWS::DynamoDB::Table
      DeletionPolicy: Retain
      Properties:
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TableName: ${self:provider.environment.DB_TABLE_USER}
```

Figure 5. DynamoDB table definition

The TableName parameter is an alias declared in the “environment” section under “provider”. This alias will change depending on the service name (declared on first line of the config file) and the selected deployment stage. The values those aliases evaluate to are as follows:

```
DB_TABLE_USER = thesis-dev-user
DB_TABLE_ADMIN = thesis-dev-admin
DB_TABLE_MESSAGE = thesis-dev-message
```

Figure 7 demonstrates the code snippet that declares the aforementioned aliases.

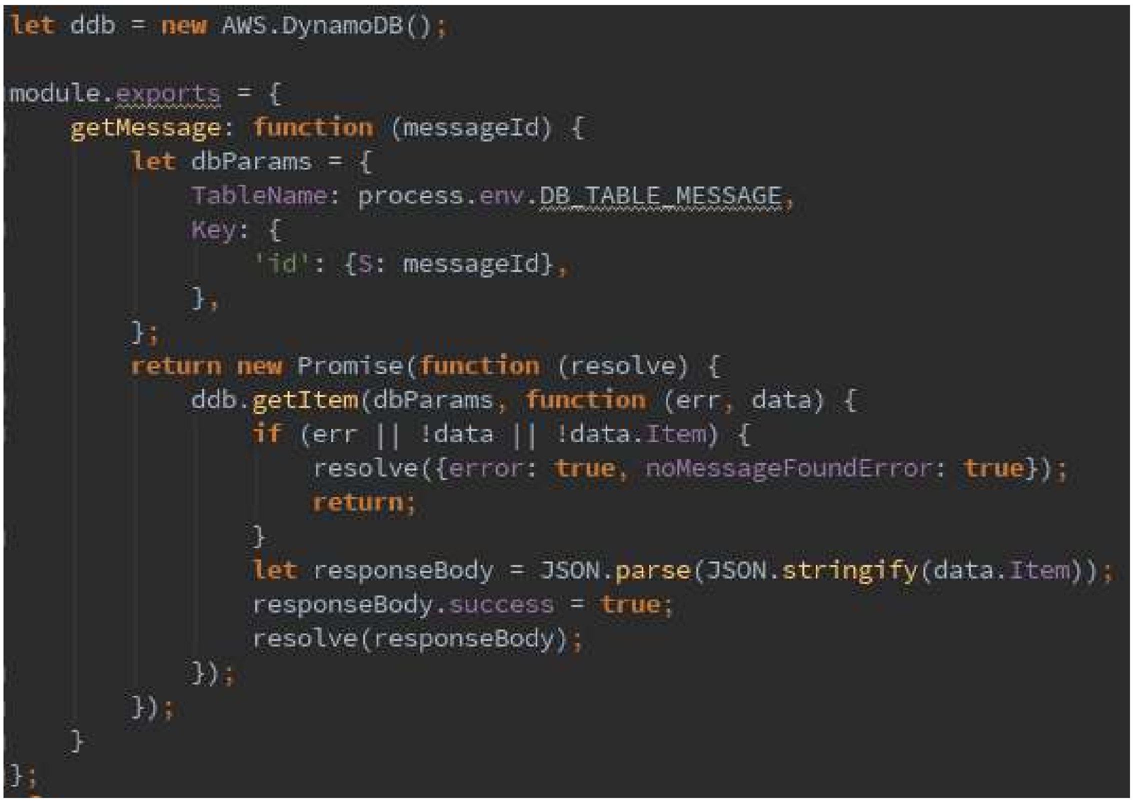


```
environment:
  DB_TABLE_USER: ${self:service}-${opt:stage, self:provider.stage}-user
  DB_TABLE_ADMIN: ${self:service}-${opt:stage, self:provider.stage}-admin
  DB_TABLE_MESSAGE: ${self:service}-${opt:stage, self:provider.stage}-message
```

Figure 6. Global environmental variables

The only change from this definition to two other tables we create is the identifier and `TableName`. Only a single column, id, is declared on the table. DynamoDB requires to declare the keys that are used for searching. Because the table is able to store arbitrary data for each combination of keys, there is no need to specify every single column at this stage.

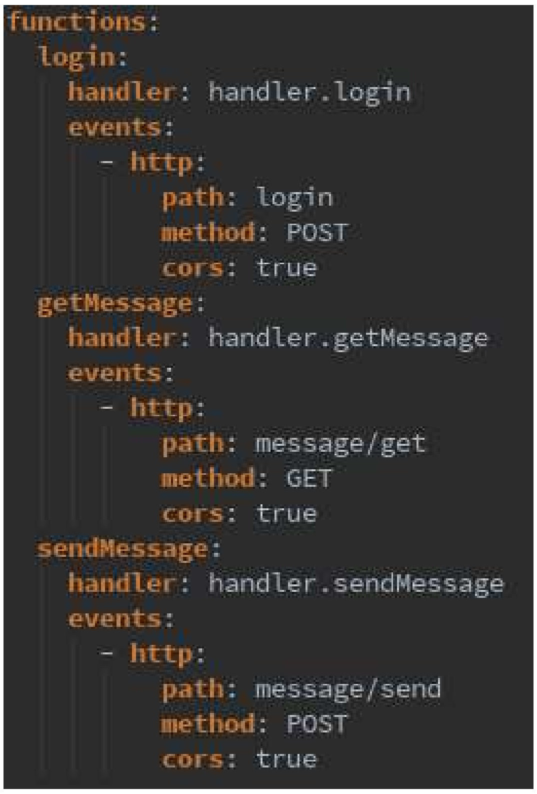
From now on, the three tables are accessible from the Lambda code. An example to read a row from one of the tables may look as the following Node.JS snippet, pictured in Figure 8. In the final version of the code attached to this thesis, the table names are defined as environmental variables in the .yml configuration file.



```
let ddb = new AWS.DynamoDB();

module.exports = {
    getMessage: function (messageId) {
        let dbParams = {
            TableName: process.env.DB_TABLE_MESSAGE,
            Key: {
                'id': {S: messageId},
            },
        };
        return new Promise(function (resolve) {
            ddb.getItem(dbParams, function (err, data) {
                if (err || !data || !data.Item) {
                    resolve({error: true, noMessageFoundError: true});
                    return;
                }
                let responseBody = JSON.parse(JSON.stringify(data.Item));
                responseBody.success = true;
                resolve(responseBody);
            });
        });
    }
};
```

Figure 7. DynamoDB table access example

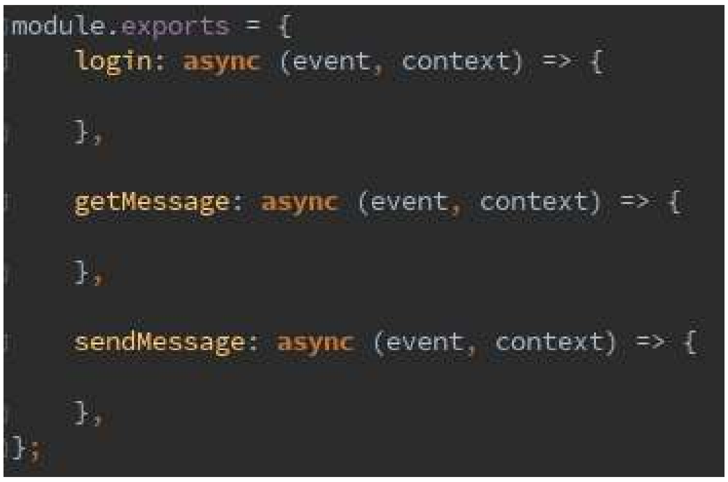
Finally, the actual endpoints need to be declared and implemented. The code snippet in Figure 9 declares three endpoints described in the section 6.1. The snippet is located on a root level of configuration file, without indentation.



```yaml
functions:
  login:
    handler: handler.login
    events:
      - http:
          path: login
          method: POST
          cors: true
  getMessage:
    handler: handler.getMessage
    events:
      - http:
          path: message/get
          method: GET
          cors: true
  sendMessage:
    handler: handler.sendMessage
    events:
      - http:
          path: message/send
          method: POST
          cors: true
```

Figure 8. Lambda function definition

This definition assumes that the actual implementation is located in the module `handler`, which can be created by using the code snippet in Figure 10, located in handler.js file.



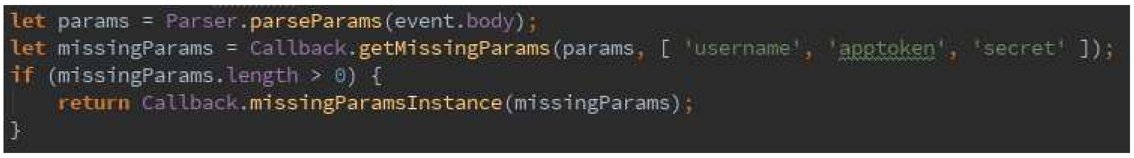
```javascript
module.exports = {
    login: async (event, context) => {

    },

    getMessage: async (event, context) => {

    },

    sendMessage: async (event, context) => {

    },
};
```

Figure 9. Minimal handler.js file example

Figure 10 demonstrates the minimal implementation of the endpoint handlers which will compile and run without producing any errors. Obviously, they will not produce any meaningful results either, but this is a starting point.

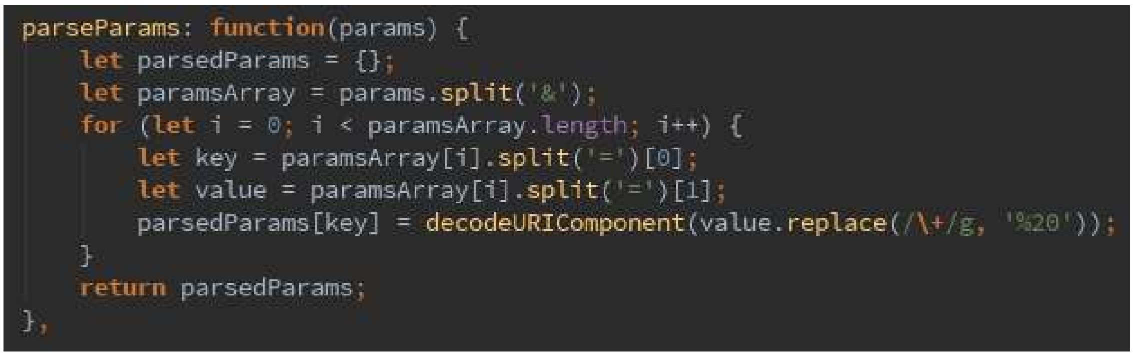
6.4 Validating the requests

The best way to develop a secure online system is to first assume that every single user is a potential attacker, and then design the system in such a way that nobody will be able to exploit it, unless they have explicit permission to use the resources. It is also safe to assume that some part of the requests that will hit the server will be invalid, thus making it necessary to validate a request before executing any business logic. The minimal validation should at least make sure that all the required parameters are present in the request, and reject it if they are not. In this project, the following code validation flow is followed for every endpoint, as demonstrated in Figures 11 to 13:



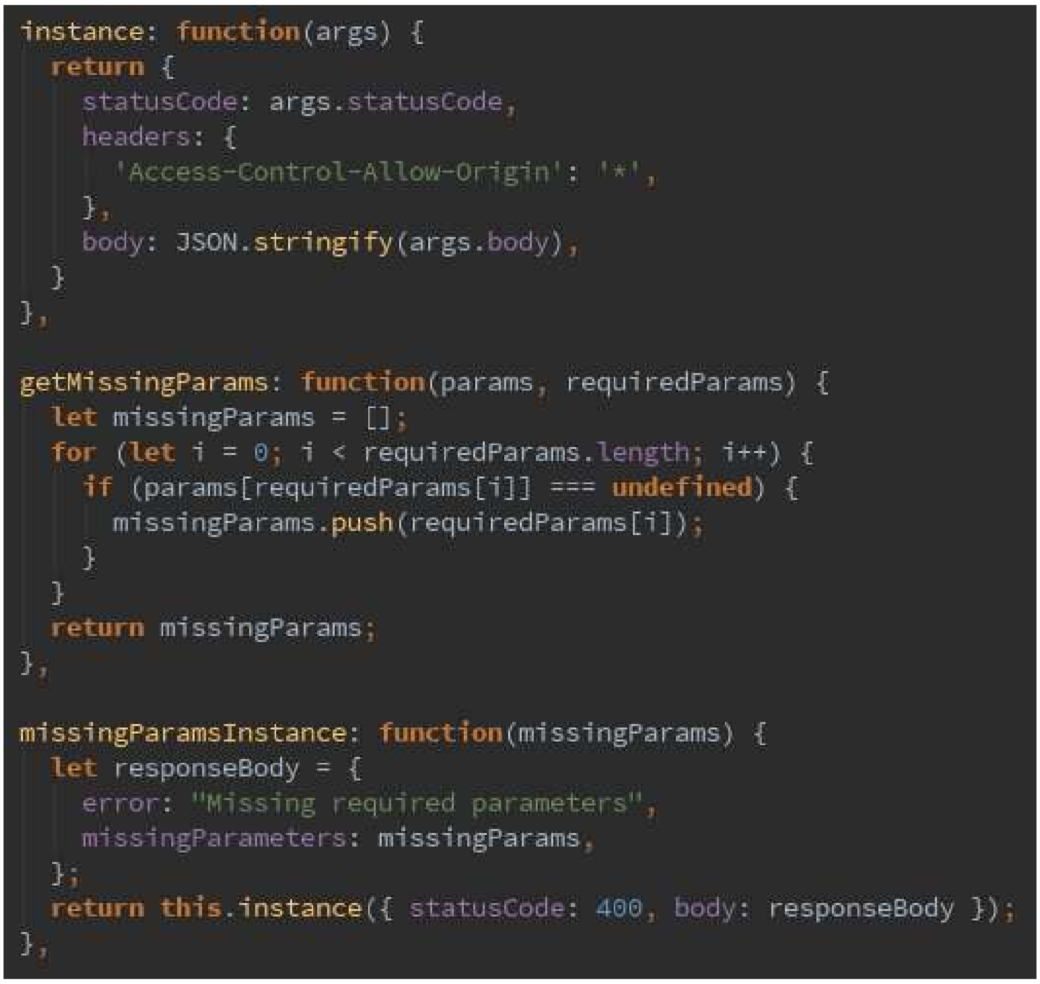
```
let params = Parser.parseParams(event.body);
let missingParams = Callback.getMissingParams(params, [ 'username', 'apptoken', 'secret' ]);
if (missingParams.length > 0) {
    return Callback.missingParamsInstance(missingParams);
}
```

Figure 10. Request validation logic



```
parseParams: function(params) {
    let parsedParams = {};
    let paramsArray = params.split('&');
    for (let i = 0; i < paramsArray.length; i++) {
        let key = paramsArray[i].split('=')[0];
        let value = paramsArray[i].split('=')[1];
        parsedParams[key] = decodeURIComponent(value.replace(/\+/g, '%20'));
    }
    return parsedParams;
},
```

Figure 11. Parameter parsing logic in Parser.js file



```
instance: function(args) {
  return {
    statusCode: args.statusCode,
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
    body: JSON.stringify(args.body),
  }
},

getMissingParams: function(params, requiredParams) {
  let missingParams = [];
  for (let i = 0; i < requiredParams.length; i++) {
    if (params[requiredParams[i]] === undefined) {
      missingParams.push(requiredParams[i]);
    }
  }
  return missingParams;
},

missingParamsInstance: function(missingParams) {
  let responseBody = {
    error: "Missing required parameters",
    missingParameters: missingParams,
  };
  return this.instance({ statusCode: 400, body: responseBody });
},
```

Figure 12. Helper functions in the Callback.js file

Additionally, Callback.instance(…) is used as an abstraction layer for the response sent by an endpoint. Due to the nature of this service, the requests will always arrive from different origin, making it necessary to specify a custom header for the communication to proceed.

6.5 Implementing user authentication

Amazon provides an authentication service by the name of AWS Cognito (Amazon 2019), and it is the recommended way to handle user authentication. When this project had started, I expected to be using Cognito. However, the use-case of the service didn't match the requirements set above. This has forced me to eventually switch away from Cognito in favor of custom login flow implementation.

AWS Cognito, while being incredibly flexible and powerful, is tailored in a specific way. I didn’t manage to find any conclusive evidence to back up the following claims, but I believe that it is the user who is supposed to communicate with Cognito, not the server on their behalf. In other words, the service expects the user to contact it directly, to register or to exchange username/password combination for authentication token, which should then be used to authenticate on the actual service this user is trying to access.

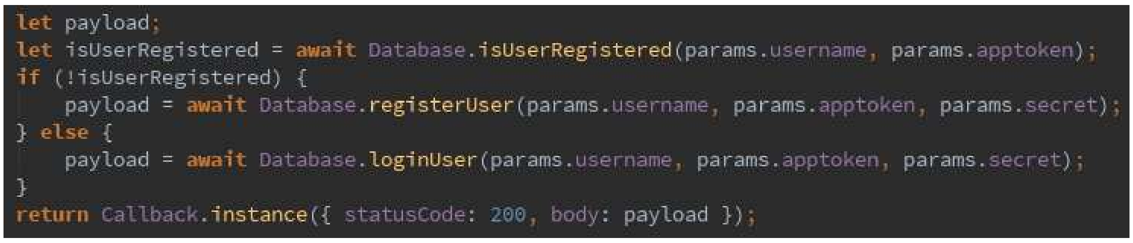
This approach does make sense as this reduces the amount of logic needed to be programmed to the serverless application. However, it doesn’t work in all cases. One of the requirements I have described above is to keep the API as simple as possible. Only a single login endpoint should be provided, that handles registration behind the scenes. In Cognito that is either not possible or extra complicated, and extra security measures - like email verification - are problematic to disable. Being aware of multiple services and endpoints would make the front-end code more complicated, which is not the direction I am willing to take the project; thus, I have decided to switch away from Cognito to the custom authentication implementation.

Due to the fact the supporting back-end doesn’t have access to the main user database, it needs to maintain a separate list of all the user accounts. One possibility for that is to hook into user registration, but the method this work describes is simple and stateless. Whenever a user loads the website, custom code checks if the authentication is present by trying to load an authentication token from the cookie. If the cookie is not available or the token has expired, then a call to /login endpoint is made.

Application token, which is referred to as ‘apptoken’ in the code, is used to distinguish multiple websites that may use the same shared service. In the backing database, the usernames are stored in the following format:

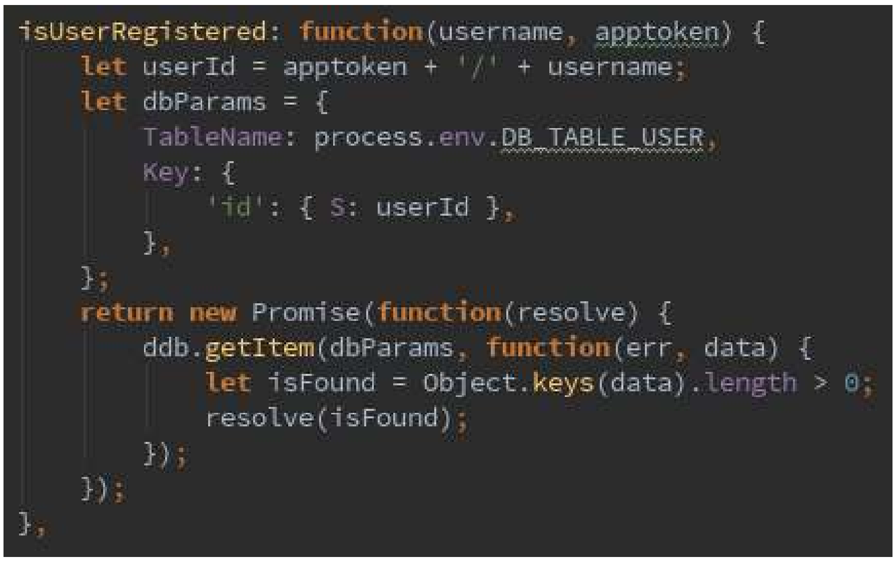
```
apptoken/username
```

This guarantees that even if two users have the same username on separate services, it will not cause a name collision in the database. Figures 14 to 17 depict the full code responsible for the authentication flow.



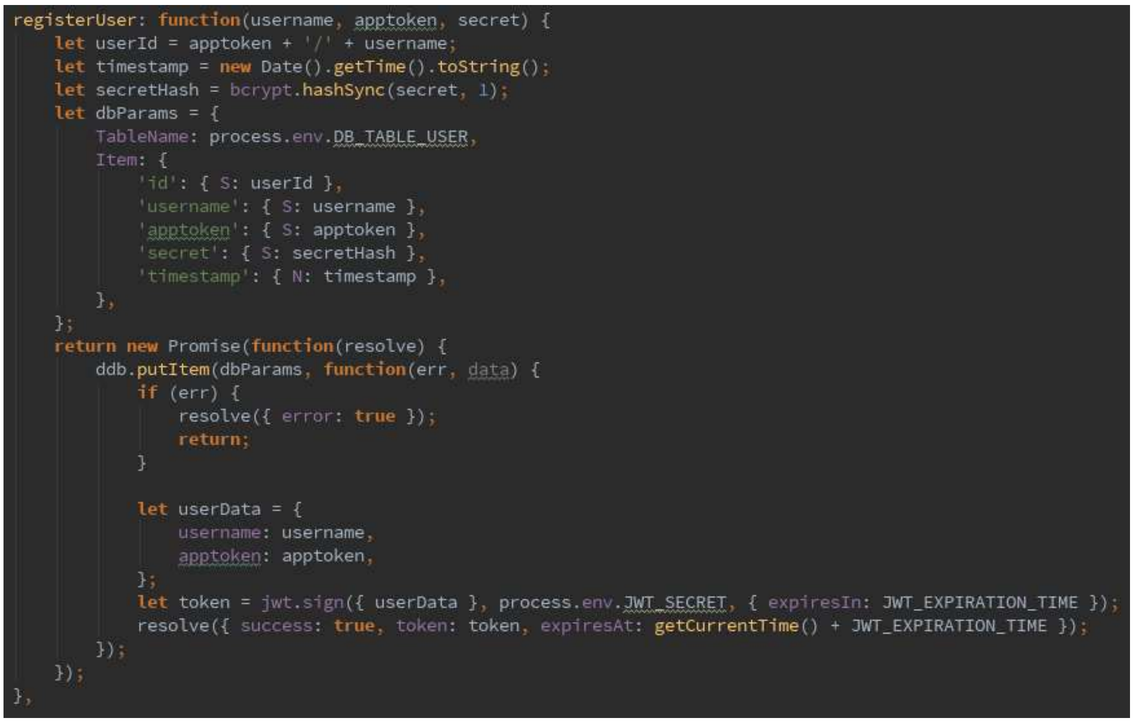
```
let payload;
let isUserRegistered = await Database.isUserRegistered(params.username, params.apptoken);
if (!isUserRegistered) {
    payload = await Database.registerUser(params.username, params.apptoken, params.secret);
} else {
    payload = await Database.loginUser(params.username, params.apptoken, params.secret);
}
return Callback.instance({ statusCode: 200, body: payload });
```

Figure 13. Business logic for /login endpoint



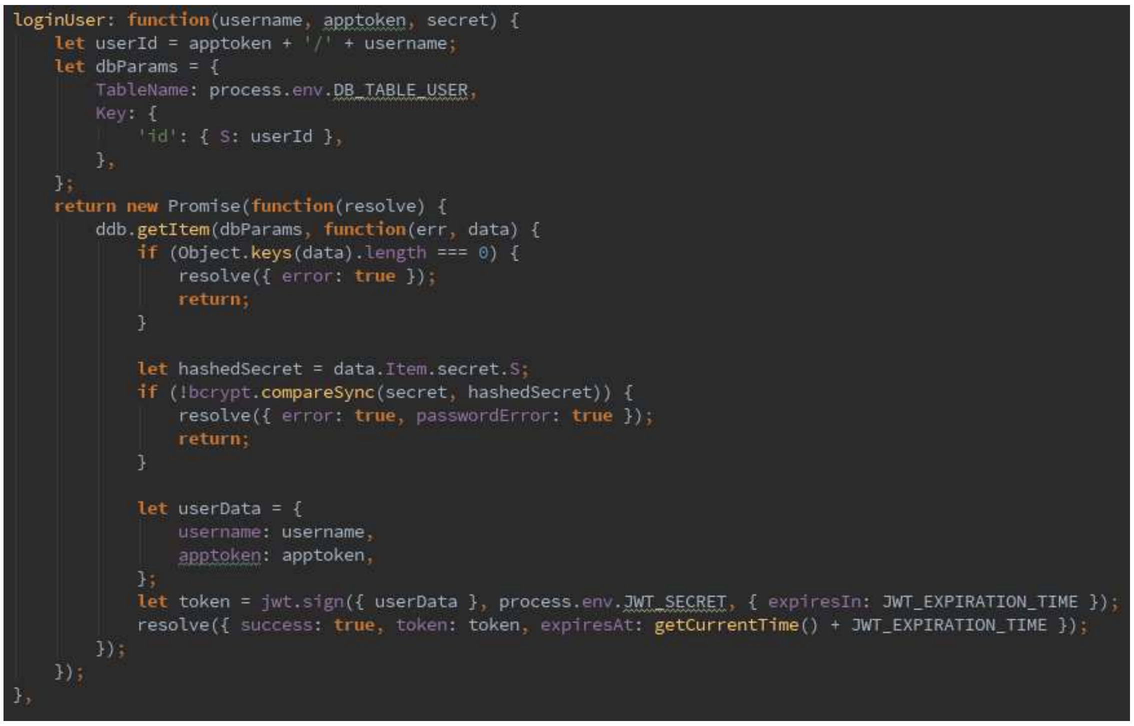
```
isUserRegistered: function(username, apptoken) {
    let userId = apptoken + '/' + username;
    let dbParams = {
        TableName: process.env.DB_TABLE_USER,
        Key: {
            'id': { S: userId },
        },
    };
    return new Promise(function(resolve) {
        ddb.getItem(dbParams, function(err, data) {
            let isFound = Object.keys(data).length > 0;
            resolve(isFound);
        });
    });
},
```

Figure 14. User credentials check in Database.js file



```
registerUser: function(username, apptoken, secret) {
    let userId = apptoken + '/' + username;
    let timestamp = new Date().getTime().toString();
    let secretHash = bcrypt.hashSync(secret, 1);
    let dbParams = {
        TableName: process.env.DB_TABLE_USER,
        Item: {
            'id': { S: userId },
            'username': { S: username },
            'apptoken': { S: apptoken },
            'secret': { S: secretHash },
            'timestamp': { N: timestamp },
        },
    };
    return new Promise(function(resolve) {
        ddb.putItem(dbParams, function(err, data) {
            if (err) {
                resolve({ error: true });
                return;
            }

            let userData = {
                username: username,
                apptoken: apptoken,
            };
            let token = jwt.sign({ userData }, process.env.JWT_SECRET, { expiresIn: JWT_EXPIRATION_TIME });
            resolve({ success: true, token: token, expiresAt: getCurrentTime() + JWT_EXPIRATION_TIME });
        });
    });
},
```

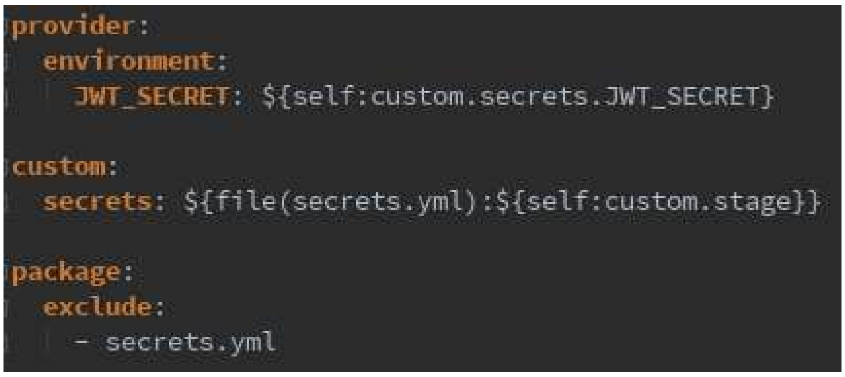
Figure 15. User registration function in the Database.js file



```
loginUser: function(username, apptoken, secret) {
    let userId = apptoken + '/' + username;
    let dbParams = {
        TableName: process.env.DB_TABLE_USER,
        Key: {
            'id': { S: userId },
        },
    };
    return new Promise(function(resolve) {
        ddb.getItem(dbParams, function(err, data) {
            if (Object.keys(data).length === 0) {
                resolve({ error: true });
                return;
            }

            let hashedSecret = data.Item.secret.S;
            if (!bcrypt.compareSync(secret, hashedSecret)) {
                resolve({ error: true, passwordError: true });
                return;
            }

            let userData = {
                username: username,
                apptoken: apptoken,
            };
            let token = jwt.sign({ userData }, process.env.JWT_SECRET, { expiresIn: JWT_EXPIRATION_TIME });
            resolve({ success: true, token: token, expiresAt: getCurrentTime() + JWT_EXPIRATION_TIME });
        });
    });
},
```

Figure 16. User login function in the Database.js file

Creating and validating a JSON web token requires the use of a secret, which is loaded dynamically from a file. The secret keys differ for staging and production environments, and they are also excluded from the packaging step, except for one key used in given environment. The following snippet demonstrates the relevant part of the serverless framework configuration file:



```yaml
provider:
  environment:
    JWT_SECRET: ${self:custom.secrets.JWT_SECRET}

custom:
  secrets: ${file(secrets.yml):${self:custom.stage}}

package:
  exclude:
    - secrets.yml
```

Figure 17. Secret key file configuration example

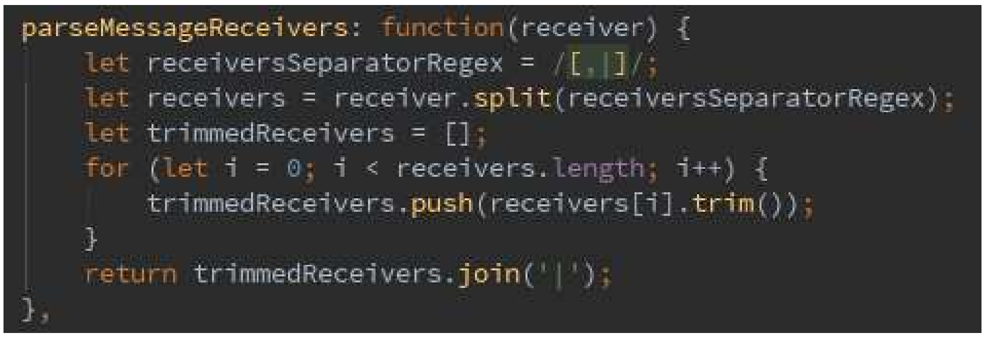
The code listing attached to this work does not include the “secrets.yml” file. On the other hand, “secrets-example.yml”, provides a template that this file follows. Creating a copy of the example file, renaming it to “secrets.yml” and inserting generated values is sufficient for the service to run normally.

6.6 Sending a message

As the front-end code is out of scope of this work, at this point sending a message can be described as a following set of steps:

- Validate a request
- Parse the list of receivers and message body from the payload
- Save the message object into database
- Respond with the message unique ID

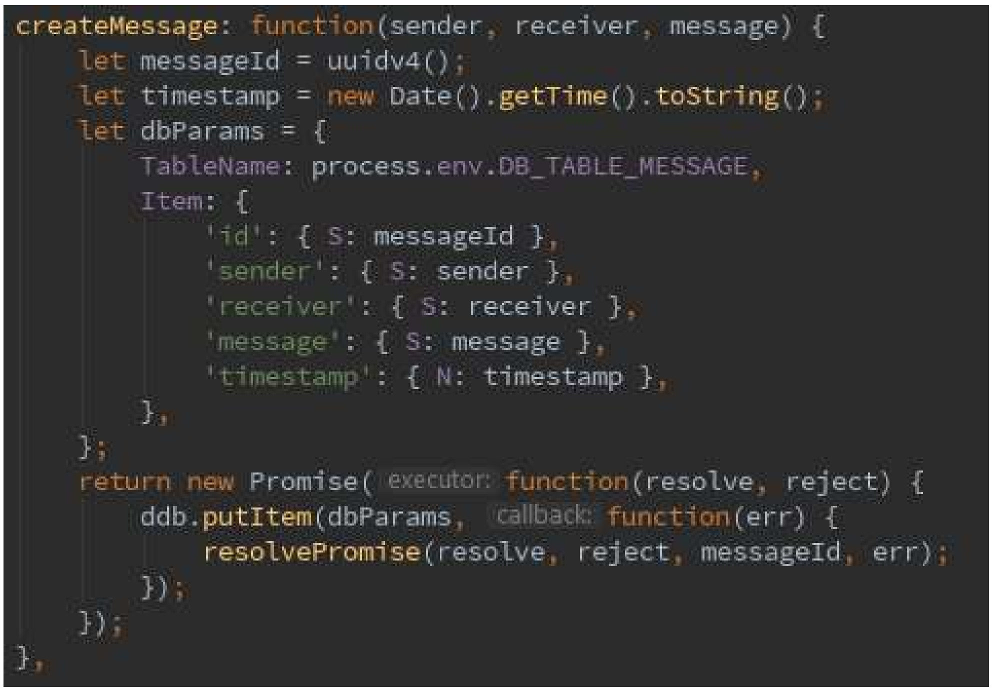
As the request validation has already been covered, I will assume that the request is already valid and authenticated. The list of receivers is a comma separated list of usernames that may also contain spaces. A simple way to parse them into a list is to use a regular expression and JavaScript built-in split method, as demonstrated in Figure 19.



```
parseMessageReceivers: function(receiver) {
    let receiversSeparatorRegex = /[,|]/;
    let receivers = receiver.split(receiversSeparatorRegex);
    let trimmedReceivers = [];
    for (let i = 0; i < receivers.length; i++) {
        trimmedReceivers.push(receivers[i].trim());
    }
    return trimmedReceivers.join('|');
},
```

Figure 18. Parse message receivers function in the Parser.js file

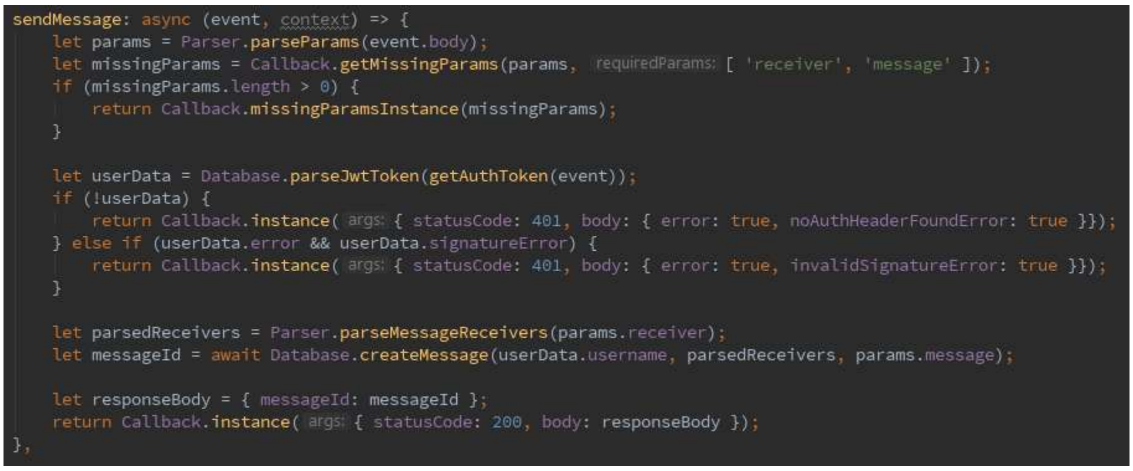
The function depicted above returns a string which is guaranteed to contain only usernames separated by a vertical line character. This character is not legal in the username in my use-case, but if it was, the escaping would be necessary at this step as well. In Figure 20, the message creation function is demonstrated.



```
createMessage: function(sender, receiver, message) {
    let messageId = uuidv4();
    let timestamp = new Date().getTime().toString();
    let dbParams = {
        TableName: process.env.DB_TABLE_MESSAGE,
        Item: {
            'id': { S: messageId },
            'sender': { S: sender },
            'receiver': { S: receiver },
            'message': { S: message },
            'timestamp': { N: timestamp },
        },
    };
    return new Promise( executor: function(resolve, reject) {
        ddb.putItem(dbParams,  callback: function(err) {
            resolvePromise(resolve, reject, messageId, err);
        });
    });
},
```

Figure 19. Message creation function in the Database.js file

Message creation is similar to user creation, as it uses the same DynamoDB API. A version 4 UUID is generated as a message id, and the same id is exposed on the client-side. In the figure above, `timestamp` is the current time, `sender` is the username of the user that sends a message, `receiver` is the vertical line separated list of receiver usernames, and `message` is the actual text of the message. Full endpoint code is pictured in Figure 21:



```javascript
sendMessage: async (event, context) => {
    let params = Parser.parseParams(event.body);
    let missingParams = Callback.getMissingParams(params, requiredParams: [ 'receiver', 'message' ]);
    if (missingParams.length > 0) {
        return Callback.missingParamsInstance(missingParams);
    }

    let userData = Database.parseJwtToken(getAuthToken(event));
    if (!userData) {
        return Callback.instance( args: { statusCode: 401, body: { error: true, noAuthHeaderFoundError: true }});
    } else if (userData.error && userData.signatureError) {
        return Callback.instance( args: { statusCode: 401, body: { error: true, invalidSignatureError: true }});
    }

    let parsedReceivers = Parser.parseMessageReceivers(params.receiver);
    let messageId = await Database.createMessage(userData.username, parsedReceivers, params.message);

    let responseBody = { messageId: messageId };
    return Callback.instance( args: { statusCode: 200, body: responseBody });
},
```

Figure 20. Message sending endpoint in handler.js file

The unique ID returned to the client is to be embedded somewhere in the page with a unique tag that a parser can later use to request the message body back from the server if the user has valid authentication to access it. The details of this implementation are, again, out of scope of this work, so I will assume that from this point the messages are available in the database and the front-end is able to request them from the server.

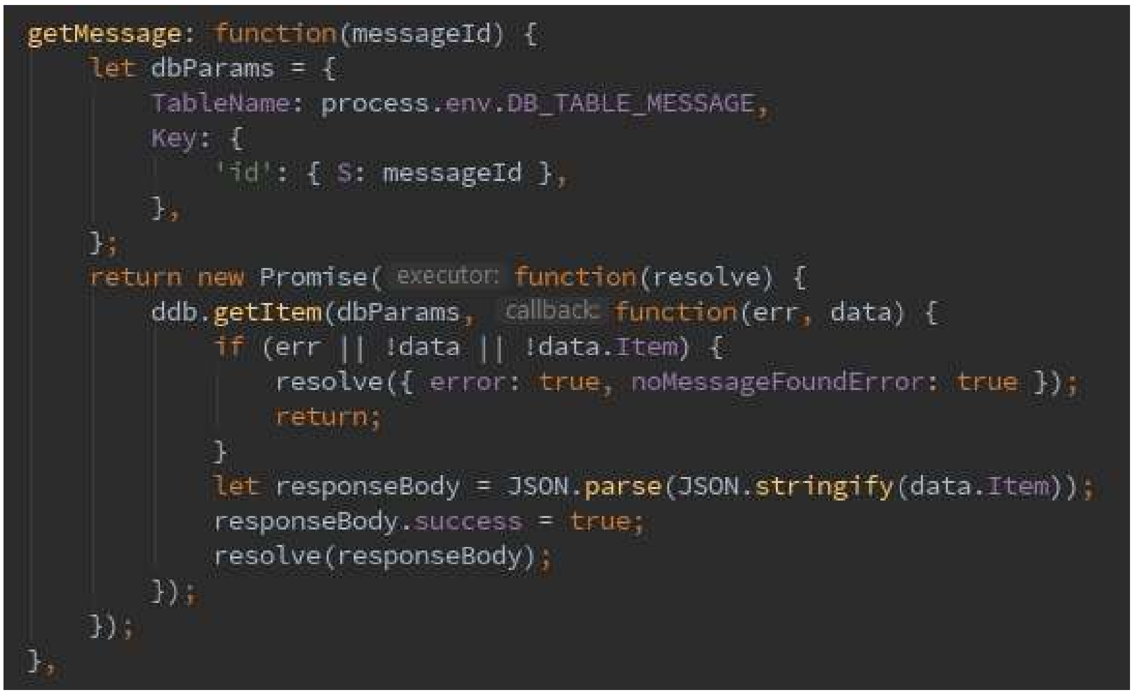
6.7 Receiving a message

After the message is created, it needs to be fetched again. Similar to sending a message, receiving a message can be described as a short set of steps, but with extra effort added to ensure security:

- Validate a request
- Fetch the message object from the database
- Check if the authenticated user is one of the message receivers
- Check if the authenticated user is registered as administrator
- Respond with the message text and list of receivers

For access checks, the user has to either be a message receiver or an administrator. Administrators have implicit access to all messages sent through the service, so if one of the checks passes, then the access is granted.

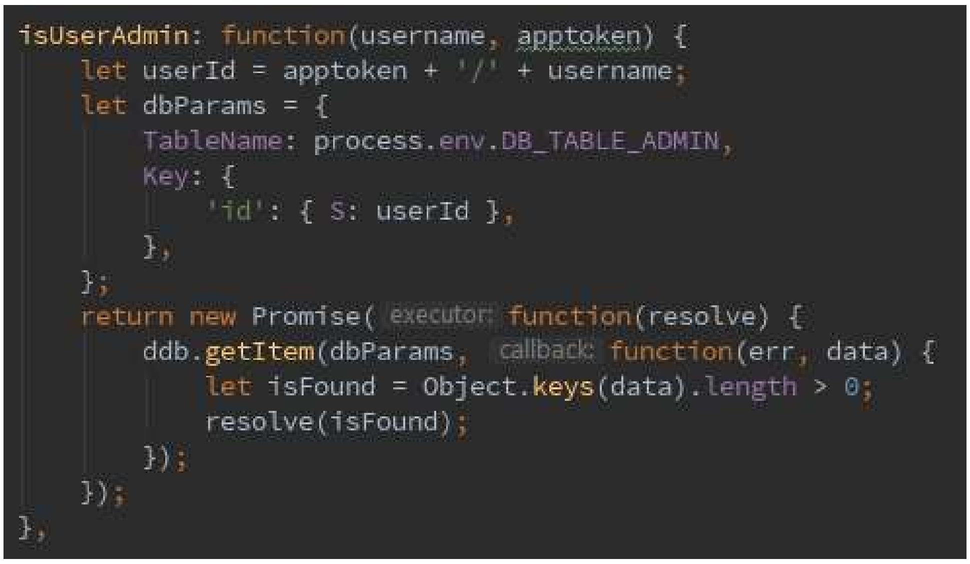
To fetch the message from the database, the helper function is again similar to the one used to create it, with extra error handling added in case the requested message ID is invalid. The helper function in question is shown in Figure 22.



```
getMessage: function(messageId) {
    let dbParams = {
        TableName: process.env.DB_TABLE_MESSAGE,
        Key: {
            'id': { S: messageId },
        },
    };
    return new Promise( executor: function(resolve) {
        ddb.getItem(dbParams,  callback: function(err, data) {
            if (err || !data || !data.Item) {
                resolve({ error: true, noMessageFoundError: true });
                return;
            }
            let responseBody = JSON.parse(JSON.stringify(data.Item));
            responseBody.success = true;
            resolve(responseBody);
        });
    });
},
```

Figure 21. Message fetching function in Database.js file

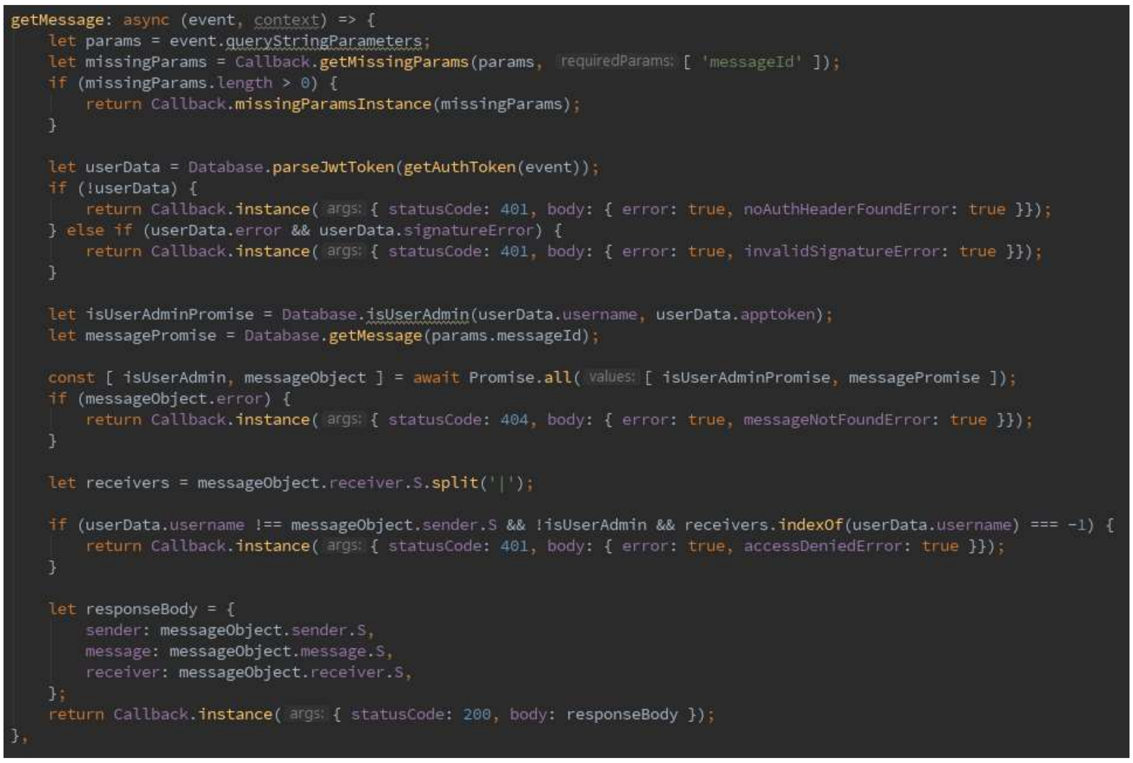
As the receiver information is contained within the message object, we first need to fetch the object from the database to check if the user has access to it, but we don't need the message to check if the user is an administrator. In the current implementation, administrator users have an entry in a separate DynamoDB table, so we need to fetch that information as well. As two database calls are asynchronous, it is beneficial to actually perform them simultaneously to save a significant amount of time. Figure 23 shows a call that checks whether or not the user is currently an administrator:



```
isUserAdmin: function(username, apptoken) {
    let userId = apptoken + '/' + username;
    let dbParams = {
        TableName: process.env.DB_TABLE_ADMIN,
        Key: {
            'id': { S: userId },
        },
    };
    return new Promise( executor: function(resolve) {
        ddb.getItem(dbParams,  callback: function(err, data) {
            let isFound = Object.keys(data).length > 0;
            resolve(isFound);
        });
    });
},
```

Figure 22. User privilege check function in Database.js file

If the user is an administrator, or their username is present in the message object as one of the receivers, then we can safely return them the content of the message. The list of receivers is also available in the response, as this is not a private information if the user can read the message. The full source code of the endpoint is available in Figure 24.



```
getMessage: async (event, context) => {
    let params = event.queryStringParameters;
    let missingParams = Callback.getMissingParams(params,  requiredParams: [ 'messageId' ]);
    if (missingParams.length > 0) {
        return Callback.missingParamsInstance(missingParams);
    }

    let userData = Database.parseJwtToken(getAuthToken(event));
    if (!userData) {
        return Callback.instance( args: { statusCode: 401, body: { error: true, noAuthHeaderFoundError: true }});
    } else if (userData.error && userData.signatureError) {
        return Callback.instance( args: { statusCode: 401, body: { error: true, invalidSignatureError: true }});
    }

    let isUserAdminPromise = Database.isUserAdmin(userData.username, userData.apptoken);
    let messagePromise = Database.getMessage(params.messageId);

    const [ isUserAdmin, messageObject ] = await Promise.all( values: [ isUserAdminPromise, messagePromise ]);
    if (messageObject.error) {
        return Callback.instance( args: { statusCode: 404, body: { error: true, messageNotFoundError: true }});
    }

    let receivers = messageObject.receiver.S.split('|');

    if (userData.username !== messageObject.sender.S && !isUserAdmin && receivers.indexOf(userData.username) === -1) {
        return Callback.instance( args: { statusCode: 401, body: { error: true, accessDeniedError: true }});
    }

    let responseBody = {
        sender: messageObject.sender.S,
        message: messageObject.message.S,
        receiver: messageObject.receiver.S,
    };
    return Callback.instance( args: { statusCode: 200, body: responseBody });
},
```

Figure 23. Message fetching endpoint in handler.js file

At this point all the specification requirements have been met. The user can register, send a message and then receive a message. The service is ready to be deployed. As a final step, the front-end needs to be configured with the endpoint URLs, which will be randomly generated during the first deployment of the service. The Serverless Framework provides the URL for each endpoint separately, and the front-end needs to know those links to access the service.

When the deployed service becomes more heavily used, it might be a good idea to take a look at provisioned capacity for the database tables, as the load spikes will affect the user experience and loading times on the website. Additionally, multiple environments are a must-have in case of Serverless. Any testing has to be done after deploying somewhere, and, of course, development version should be kept separate from production. Thankfully, Serverless Framework does provide tools to handle multiple environments without any significant development overhead.

7 DISCUSSION

Developing the application using a completely new technology is not a smooth experience. Aside from the issues described in Chapter 5, there have been other problems that have cropped up during the development. It is important to pay attention to those problems as well while evaluating whether or not serverless approach is something worth working with.

7.1 Slow log delivery

The process of creating a new application always consists of three main stages: thinking, typing, debugging. A programmer jumps rapidly from one to another, and the debugging is always the one with the most inherit frustration. There are many tools that help with debugging, but the most basic is the one we rely on a lot. When something is wrong, the first things to look at are the logs. The Serverless Framework provides a nice API that is able to attach to a running function and display the entire output. However, the information is not displayed immediately. There is a significant delay between the event happening and the log appearing on the screen, and it can take up to 17 seconds, while averaging at about 15.

The following chart in Figure 25 demonstrates a difference between execution time and log delivery time in milliseconds with 10 identical POST requests:

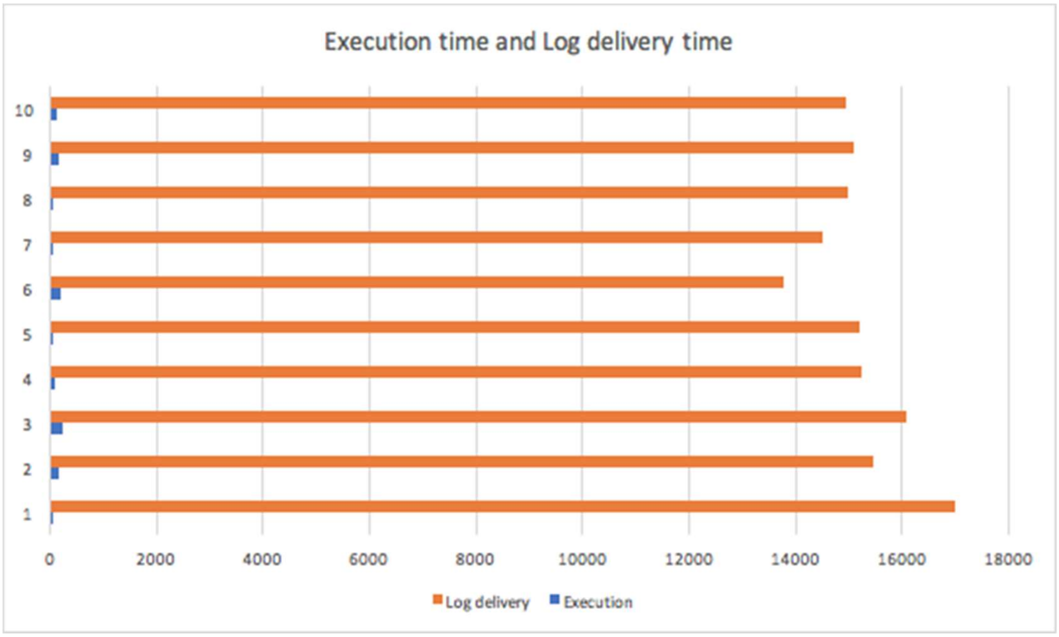


Figure 24. Execution time vs log delivery time chart

Over time these 15-second delays accumulate and are able to seriously impact the speed of development. The exact cause of the delay is unknown, as Amazon doesn’t disclose their internal infrastructure, but the fact that sometimes the logs are delivered immediately may indicate that the delays are happening due to the high demand on some of the servers.

7.2 Inconsistent execution time

One of the core principles of serverless application design is the pay-for-use business model. More precisely, with Lambda it means that some cost is associated with a request itself and execution time. The billing is discrete at 100ms intervals, meaning that a request that takes 1ms to complete and the one that takes 95ms to complete cost the same amount of money. The problem, however, arises when we are unable to predict the execution time of our code.

The following example, depicted in Figure 26, is recorded using the NodeJS application deployed in dev stage on AWS servers. The function being tested performs a reading operation from DynamoDB table with less than 10 lines, performs data comparison and data conversion operations and returns some result. All requests contain identical data.

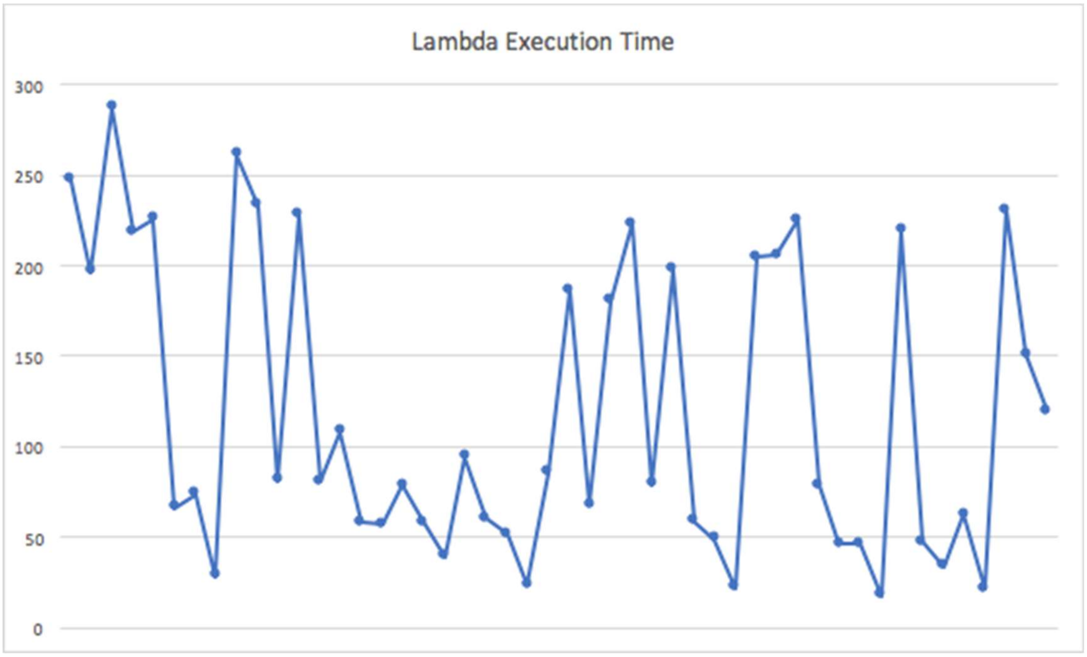


Figure 25. Lambda function execution time graph

The execution time is inconsistent and the difference between consecutive requests may be up to 250ms, which is significant if the service relies on real-time code execution for critical operations. The most likely scenario that would explain the observed behaviour is the fact that the code is running on the shared hardware, and the moments of peak load correspond to higher execution time, while lower load periods result in lower execution time.

7.3 Monetary price of an error

Developing a normal server application typically involves low risk. The development environment is running on the developer’s personal machine and any configuration changes, possible crashes or data corruptions will only affect their own setup. While working with AWS, the situation is different though. The recommended and simplest way to develop a serverless application is to deploy it to the cloud, but that involves an inherited risk, because cloud computation time costs money.

With typical usage it is difficult to exceed the limits of the free tier, which, at the moment of writing, allow up to 1 million API Gateway calls, 1 million Lambda function calls per month and 18,600 read and write capacity unit-hours for

DynamoDB. However, the code doesn’t always behave in a way we expect, and it is possible to do a mistake that will result in these limits being exceeded.

Personally, I found the capacity units of DynamoDB a confusing measure and by mistake increased throughput capacity of my tables way higher than they should have been, expecting the costs to be consistent with other pricing models of Amazon, which is “pay for what you use” (Amazon 2019). However, a day later I found out that I accumulated 136,000 both read and write capacity unit-hours, which way exceeds the limit of the free tier, resulting in USD 130 bill, while the service I am working on has no users.

I have been able to resolve the issue after contacting the customer support, but situations like this remind us about the dangers associated with working on somebody else’s infrastructure. I would never be charged any money for deploying a service in any state on my personal hardware, but it did happen with Amazon.

7.4 Monetary benefits

However, it would be unfair not to mention the benefits that serverless design provides. At the moment of writing, a version of the application is deployed to production with a small set of customers, and the operational costs are even lower than expected. With over 300 registered users and about 50 active everyday users, the total running costs of the service are below EUR 1 per month, being mostly covered by the Amazon free tier.

According to the price breakdown, easily accessible from the AWS management console, most of the costs comes from API Gateway service, which handles every single request that hits the servers. On average, there are about 2,000 requests per day, which include cache and load-balancing, already provided as a part of API Gateway package. While not being anywhere large enough to be called a highly loaded service, this application is already sizable enough to see the benefits of developing an application based on AWS Lambda.

8 CONCLUSION

On such small scale the application hits the “sweet spot” of serverless benefits, already providing a significant amount of load that probably requires a dedicated machine while not yet having acquired a number of users large enough to justify the investment for a dedicated server in a datacenter. As mentioned in the Section 7.4, the operational costs of the service that scale will most likely be in single-digit euros, which is affordable even for most students. As a result, new business opportunities will be created as the entry cost will be lowered.

At the same time, however, serverless has a number of disadvantages that must be considered before going that route, especially for a large enterprise. Vendor lock-in in particular can prove problematic as the prices of the platform holder fluctuate and the monetary benefits may dwindle over time. The application itself has to also be built around the infrastructure, not the other way around, which may require extra training or research time for developers, which, of course, translates into expenses for the employer. As a verdict, the serverless application design itself is not a gamechanger for every single developer, but it most definitely creates more options in a market segment that has been lacking since the inception of the Internet, namely small businesses or private entrepreneurs.

In any case, the goal of this work was completed successfully. The project with given specification was completed and made available to the public, and at the moment of writing is operational with a healthy number of users and minimal running costs. Personally, I may consider using serverless for my future projects as well, thanks to the experience obtained during the implementation of this thesis, and I only expect the technology to become more widely adopted over time, as the benefits it provides are truly unique, regardless of the possible drawbacks.

REFERENCES

Amazon. 2018. Serverless Application Lens. WWW document. Available at: https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf [Accessed 03.05.2019]

Amazon. 2019. AWS Lambda Limits. WWW document. Available at: https://docs.aws.amazon.com/lambda/latest/dg/limits.html [Accessed 22.04.2019]

Gojko, A. 2017. Designing for the Serverless Age. Live presentation. Available at: https://www.youtube.com/watch?v=w7X4gAQTk2E [Accessed 03.05.2019]

Hendrickson, S., Sturdevant S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. 2016. Serverless Computation with OpenLambda. WWW document. Available at: https://www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf [Accessed 02.05.2019]

Herron, D. 2018. Building REST services with Serverless framework, in Node.js, AWS Lambda and DynamoDB. WWW document. Available at: https://blog.sourcerer.io/building-rest-services-with-serverless-framework-in-node-js-aws-lambda-and-dynamodb-765beada2c57 [Accessed 03.05.2019]

Kepes, B. 2015. 30% Of Servers Are Sitting "Comatose" According To Research. WWW document. Available at: https://www.forbes.com/sites/benkepes/2015/06/03/30-of-servers-are-sitting-comatose-according-to-research/#7052339459c7 [Accessed 22.04.2019]

Roberts, M. 2018. Serverless Architectures. WWW document. Available at: https://martinfowler.com/articles/serverless.html [Accessed 03.05.2019]

Serverless Framework. 2019. Build apps with radically less overhead and cost. WWW document. Available at: https://serverless.com/ [Accessed 16.05.2019]

Statista. 2019. Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). WWW document. Available at:

https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ [Accessed 16.09.2018]

Wagner, T. 2016. Serverless Design Patterns with AWS Lambda. Presentation. Available at: https://chariotsolutions.com/wp-content/uploads/2016/04/Serverless-Design-Patterns-with-AWS-Lambda-Philly-ESE.pdf [Accessed 03.05.2019]

Waterworth, S. 2018. Which Serverless Platform Should You Use? WWW document. Available at: https://www.instana.com/blog/which-serverless-platform-should-you-use/ [Accessed 19.09.2018]