How serverless architecture is transforming the cloud

Pavol Krajkovič

Software Architecture,
Intelligent software systems
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava

December 17, 2018

Abstract

Serverless architecture is a cloud execution model, that rises as a new trend in web development and cloud infrastracture. In serverless architecture, developers can freely allocate resources to developing the bussiness model of the application and delegate the infrastructure and operations of servers on a third party provider.

In the document, we will look at the transformation of a web application consisting of multiple components to a new serverless architecture hosted on the cloud. The main focus will be on the bussiness logic of the application and so the client side application would be intact. The application consists of many use cases which may hinder the transformation process to a fully serverless platform. Therefore, it is upon us to prepare a plan to overcome this issues. The code of the application will be of stripped application to showacse the usage of Terraform.

For this project, a repository was created that captures the proof of concept work done on the sample application https://github.com/pavol6999/as-terraform-serverless.

1 Introduction

Serverless is yet another abstraction of software design coming from the cloud providers. Complex monolithic applications turned into a lot of different microservices, the microservices were containerized, containers were orchestrated and so on. Now coming to serverless functions, as if the whole application logic was split into atomic pieces scattered on the cloud. The term serverless computing shows a rising trend as per google trends seen in figure 1. It is a prominent paradigm executed in cloud infrastracture and infrastructure operations.

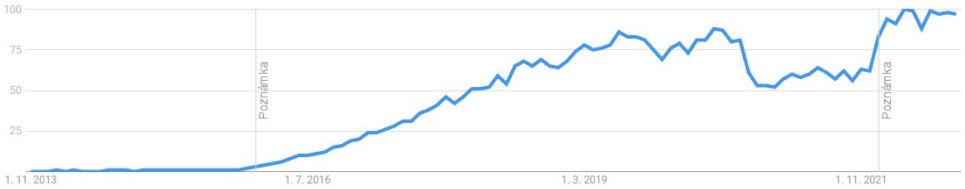

Figure 1: Google trends showing the trend of term "serverless" around the world from year 2013 up to year 2022 [2]

In serverless architecture, the whole infrastructure is delegated to a third party cloud provider, who takes care of the server system as a whole. Therefore, developers of an application hosted on a cloud, can allocate more resources into the functionality of the actual application, which may bring up a topic of NoOps architecture [12], if the trend prevails.

In this document, we will look at how we can transform the business logic, i.e. the server part, of the web application into a serverless environment. The aim of the output of the document is to strive for full integration of the server part and thus its potential redundancy as a separate system for specific use cases. The code of the application will be of stripped application to showacse the usage of terraform.

In section 2 we will briefly analyze the concept of serverless paradigm and its trending tools like Function-as-a-Service (FaaS in short). Then we proceed to the general analysis of our problem in section 5. After initial analysis of the approach, in section 5 is the core of the document being the lightweight implementation of our approach of transforming the server side of the application. We will further look at serverless tool tool in section 6.

2 Concept of serverless computing

Serverless aside from the infrastructure approach can be decomposed into two major objects, being the trigger and an action [18]. The trigger, or an "event", can be of any source such as an API call from a client, an event from a message broker or an internal call from solution's other components. An action on the other hand is the function that goes live after certain conditions are met, i.e. trigger completion. Serverless hosts many solutions and for functions we can think of a runtime function, a database call, authorization process or steam processing event [19]. The general structure of a serverless composition can be seen in the figure 2.

The main advantage of serverless computing is its cost, as the model follows the pay-as-you-go paradigm. Serverless functions act only when triggered and therefore they do not accumulate cost while in the idle mode. On

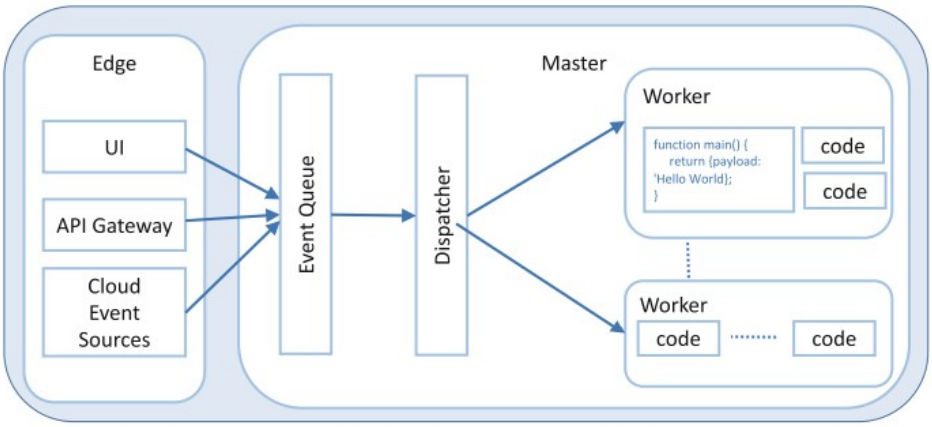

Figure 2: Serverless structure of trigger and a service [8]

other hands, there are services in the serverless paradigm which need to be always up and running such as a database [22].

For another benefit we can assume the on demand scaleability of serverless services. When the application grows due to increasing trend and needs to handle more requests concurrently, running an instance can fast reach its limits. We can overcome this issue with two options. The first is to optimize the software to utilize hardware resources more effectively. This option is expensive and not always possible; thus, in the majority of the cases the second option is chosen, to increate hardware resources, which exactly is done in serverless paradigm [22, 7].

As a motivation for choosing this topic, we can consider the growing trend of topics dealing with serverless methods, whether for server or client parts of the software in the cloud community [15].

It is also a middle-step from achieving greater overview of edge functions, which act as a further installment of FaaS (Functions-as-a-Service) on the cloud. In general, a further abstraction, serverless computing, in cloud development is an interesting yet a complex topic.

3 Function as a Service

Function as a Service is a relatively newer concept that aims to offer developers the freedom to create software functions in a cloud environment easily. It is the main building block of serverless platforms. In this method, the developers will still create the application logic, yet the code is executed in stateless compute instances that are managed by the cloud provider [6]. There is usually a misconception of serverless and FaaS, where people interchangeably use these two terms. As aformentioned in section 1, serverless consists of many aspects like serverless databases, message brokers, video

processors and so on. FaaS is only the most trending solution from this concept.

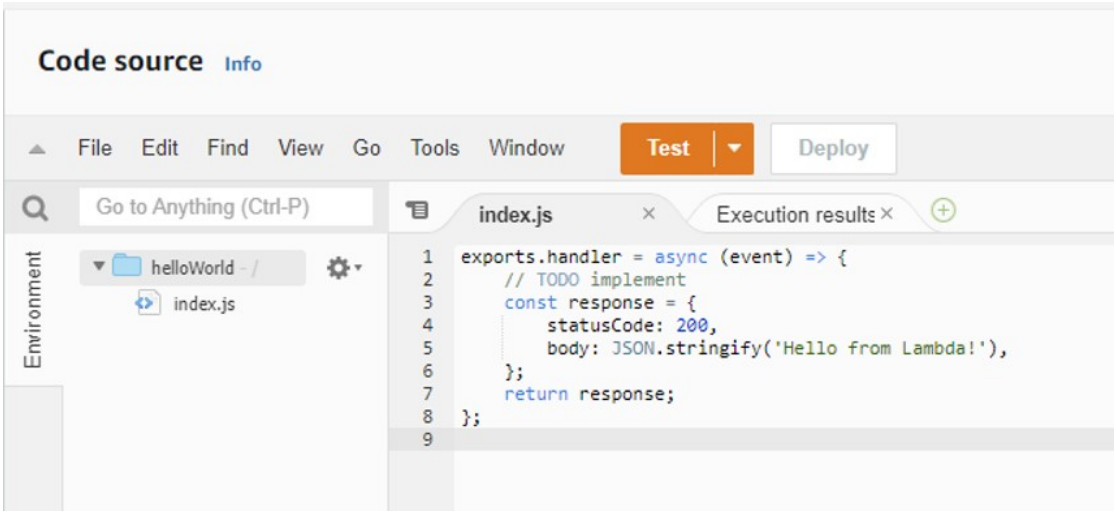

Figure 3: A function in AWS Lambda

A resource which fall into the term of Funcion-as-a-Service may well be the AWS Lambda, mantained by the Amazon Web Services. AWS Lambda is a serverless, event-driven compute service that lets the user run code for virtually any type of application or backend service. [20]. An example of how the internals of this resource looks, can be see in figure 3, showing the easy to use environment to publish code which functions as a serverless function.

4 Infrastructure as Code

While keeping in mind the fact that the serverless application is hosted generally on a third party cloud provider, the infrastructure operators must establish a consistent and manageable inventory of serverless resources.

Creating a resource on a third party cloud provider platform is done via graphical or command line interface. In a bigger application platform, it may be tedious and time consuming for a infrastructure operator to create, delete or configure the serverless components manually.

To overcome this issue, it is recommended [16] to use IaC (Infrastructure-as-Code) tools to keep the infrastructure state of resources in a structured code.

resource ”aws_lambda_function” ”test_lambda” {
  filename = ”lambda_function_payload.zip”
  function_name = ”lambda_function_name”
  role = aws_iam_role.iam_for_lambda.arn

```
  handler = "index.test"
  runtime = "nodejs14.x"
}
```

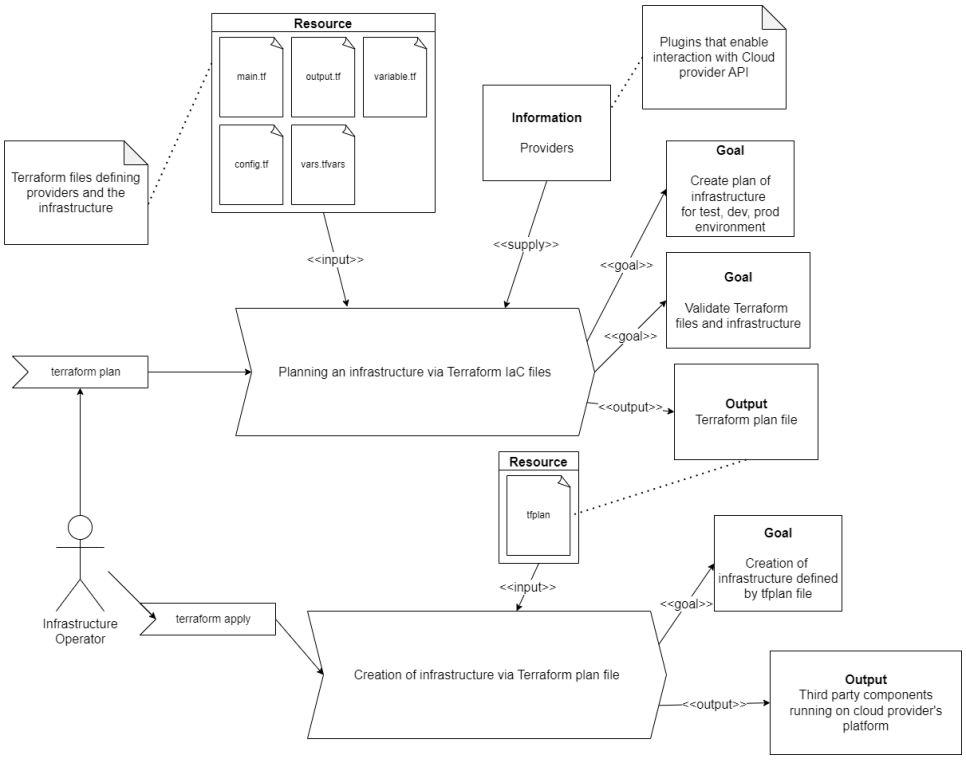

Figure 4: Terraform workflow showing the planning and creation of infrastructure

One example of an IaC tool is Terraform, in Terraform an infrastructure operator creates objects in code, in which each object represents a cloud resource. We can see an example of a serverless resource, in this case AWS lambda, being written in Terraform code. We can see a diagram of Terraform workflow in figure 4

5 Case study: From Monolith to Serverless

This section will consist of two main logical parts. At first, we will briefly describe the choice of application and then, we will propose a method of how we should handle the transformation of backend server service to a serverless platform.

When monolithic applications get bigger, the time spent managing, deploying and upgrading the application gets bigger. At one point, it is not cost effective and the monolithic application becomes a legacy software re-

placed with a newer and better option [10]. A proposed solution to battle the ineffectiveness is to transform this monolith into couple serverless components hosted on the cloud, thus reducing complexity and cost of maintaining the monolith.

We have two example approaches to transforming the monolith being the hybrid transformation and the total transformation [11, 22, 17].

In hybrid transformation, the creation of new features in a monolithic application is shifted into serverless. Therefore, the application is a hybrid of a monolith and serverless components. Also in this type of migration, the application is decomposed into layers and these layers are also transformed into serverless components. This type of transformation can be done on applications of any scale. A decomposition of application can be seen in figure 5.

In total transformation, a whole new project is started composing the application from scratch in serverless environment. This type of transformation is done on applications of low to mid scale level [11, 22, 17].

5.1 Application used

For this project, we have used an application from FIIT STU. It is a web application consisting of a frontend, backend application written in Typescript framework and a non relational database. The reason behind choosing this project is because of it's simple yet different use cases. As we have previosly said, we will propose a transforming process of the server side of the application, in this case it is the backend service, database and the file storage.

The application lets us save video metadata to a database, save documents to a file system and create new information tables. The services are containerized via Docker.

5.2 Transformation of layers

Our proposed transformation of a web application to a serverless framework builds on layers of components, mentioned in section 5.1. The server side application's internal components can be seen in figure 6. In our example, each component is a so called layer in terms of transformation. From the figure we can see that, the layers are main components of a general server application.

In a total layer-by-layer transformation, we should transform each layer individually and map the internal components into serverless components hosted on the cloud. Since it is not in production, we can start with whatever layer we choose to do so.

Each layer is to be written in terraform code and as a subsection dedicated to the said layer. For the cloud provider we have chosen Amazon Web Services. The idea behind this transformation is to change the application

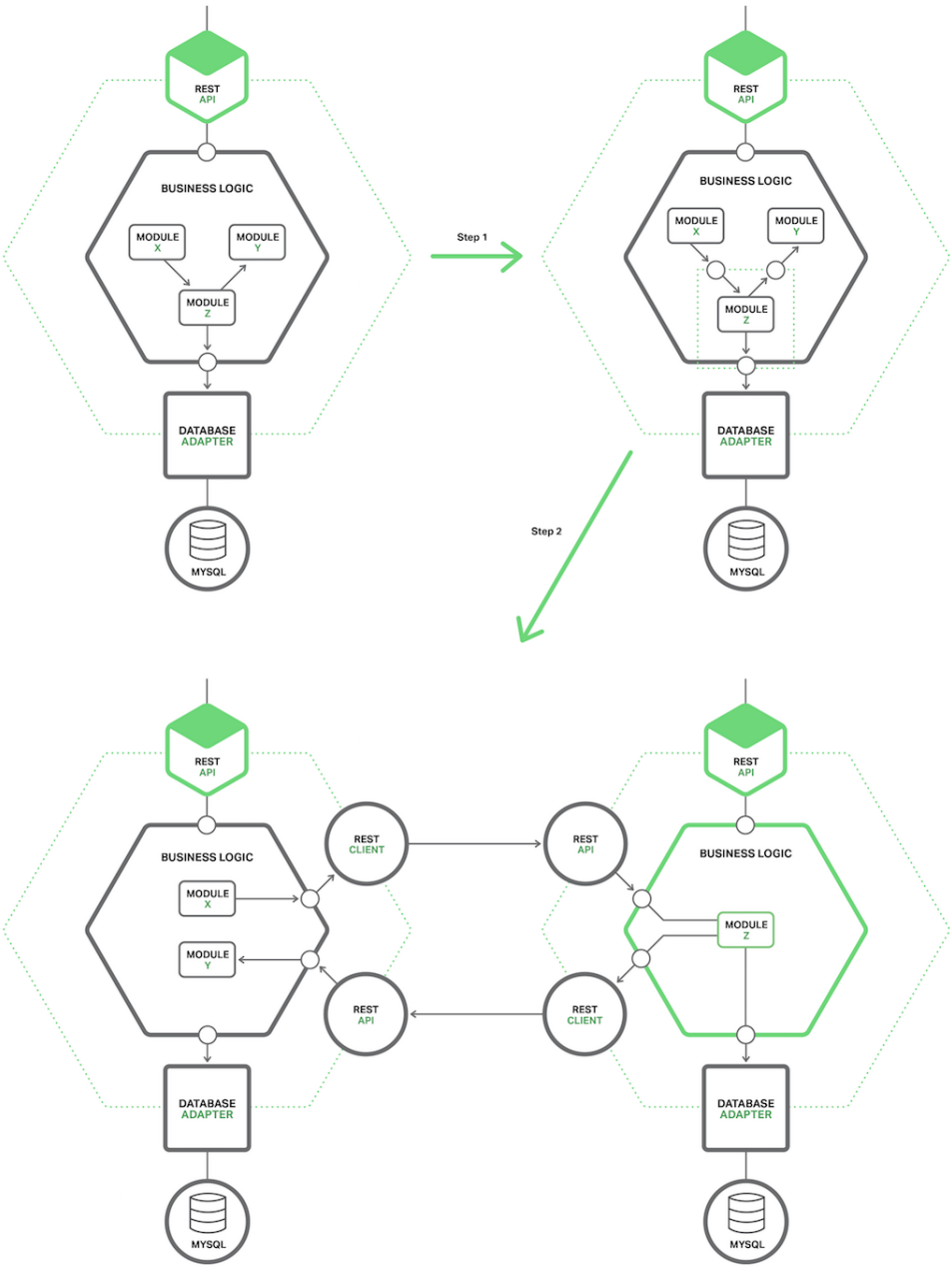

Figure 5: Process of breaking up logic of monolithic application [22]

layer by layer, we may first start with the database layer exchanging the on-premise database for a managed database in the cloud.

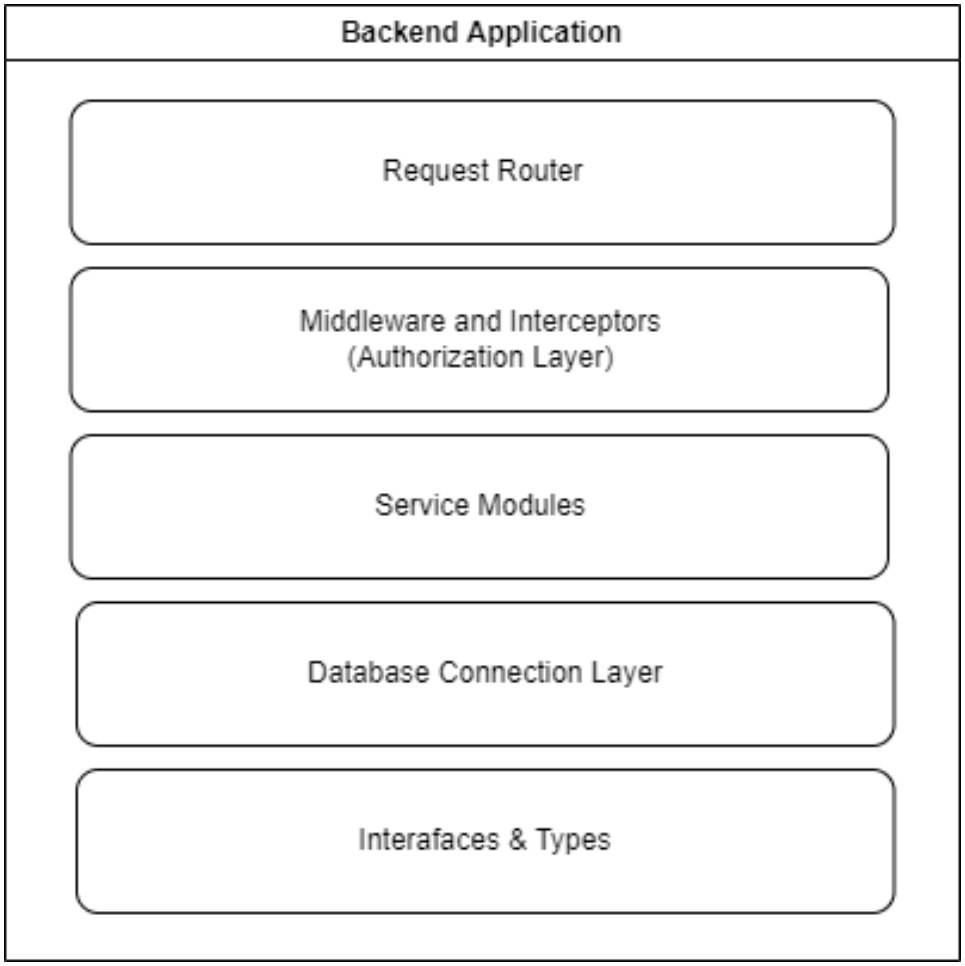

Figure 6: Layers of a monolithic application

5.2.1 Database layer

We first start with the database layer as this is the least complex task. What needs to be done is to create a cloud serverless database and rewrite the connection string in the application.

Database used in the application was MongoDB as it is a lightweight database for storing documents, in our case metadata. The database was used to store news, video links, document metadata and user administrators.

MongoDB provides a serverless option on the cloud, creating a cluster of three replicas hosted on AWS Cloud Provider [3]. Therefore, we have chosen this option. It is also natively deployed on a specified region in AWS, after creating the database, we acquire from the Terraform output connection string which is then passed to further serverless components. The edited diagram after the switch from on-premise database to serverless database

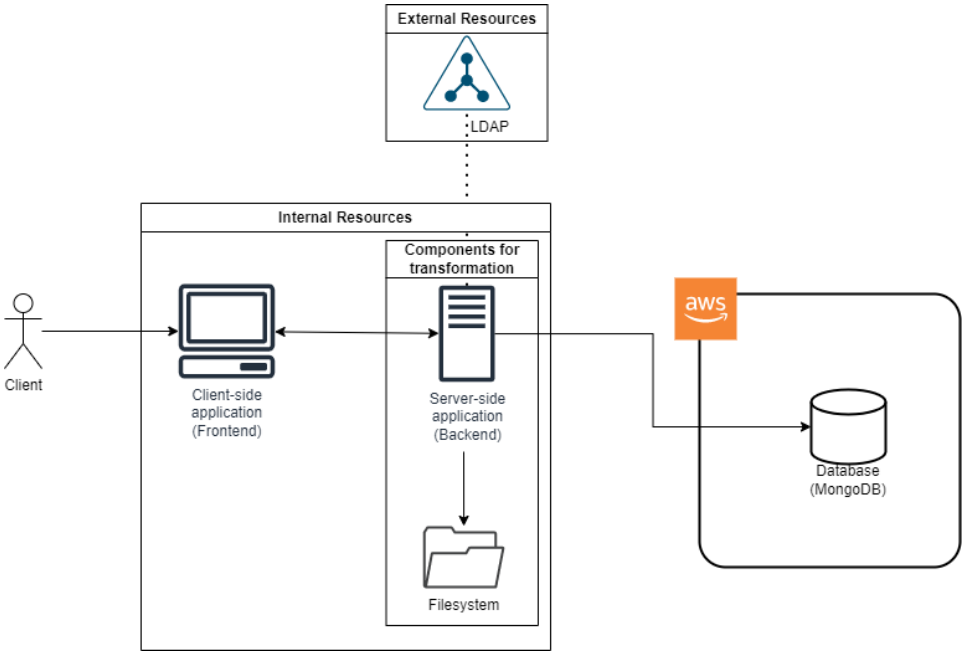

Figure 7: Application architecture after switching DB source

can be seen in figure 7.

5.2.2 Authorization and Authentication

This section consists of two key parts, the first one being the external authentication that is done on the web application using external Lightweight Directory Access Protocol (LDAP) server running behind a Virtual Private Newtork (VPN) and second part being the internal authorization done via JWT tokens.

LDAP protocol is a standard that enables the connectivity to a distant Active Directory which stores information about users. The client is given access and obtains the required information if the credentials provided by the user match the credentials associated with their core user identity that are maintained within the LDAP database (attributes, group memberships, or other data). The client is prevented from accessing the LDAP database if the provided credentials do not match [21].

The LDAP is as aforementioned just a protocol that connects to the Active Directory. It may be perceived as a one line of code or a connection string. The problem in our case, is that the monolithic application would be running behind the VPN as well, having access to the Active Directory as well. In our serverless case, this is not possible as further steps need to be taken. Without changing the external infrastructure, we need to be able to connect from within the cloud provider infrastructure to the VPN of FIIT

STU.

We can overcome the VPN problem using the AWS site-to-site VPN [4], with components being the Transit Gateway that acts as a port from AWS, and a client gateway, acting as a port on the client side. The integration is not trivial and access from both infrastructure teams is required.

Javascript Web Token is on the other hand a form of authorization security measurement. The data coming from LDAP is important as it contains information about the user, which can be later utilised by the JWT module. In JWT token, one can find information representing claims to be transferred between two parties. Generally, a JWT token consists of boolean flags declaring the Role-Based-Authorization method. If one can decode the JWT token, change the flag and send an administrator request to the server. Then the appended hash would not match and the request would be denied [13].

AWS has integrated the JWT authorization method already into their components such as in AWS ApiGateway. All there is to do is to create a JWT authorizer and attach it onto the ApiGateway resource. This way we can make sure that no unauthorized requests are coming to the application.

5.2.3 Service Modules

If we extend the layer service modules in figure 6 in section 5.2, we get following modules seen in figure 8. The App module acts as the main module in the application and only imports types, interfaces and other modules. The MongoDB module acts as the connector to the MongoDB database and provides templates and schemas to the application. Therefore, the transformation will be done on Admin and Api modules. Classes in the application are called services and help with the creation of objects in the database.

In transforming the application, we can map each service to a serverless function mentioned in section 3, being the AWS lambda. Since the AWS lambda operates on plain NodeJS environment, the structure of the code will change.

Together we create new Aws Lambda component for each service of the application, to couple objects of the same type together. As an example transformation, not all features will be implemented since most of them use special libraries from the framework itself which are not trivial to implement in plain NodeJS. An AWS Lambda function example of video service can be seen below.

```
resource "aws_lambda_function" "video_service" {
  filename = "lambda_function_video.zip"
  function_name = "VideoService"
  role = aws_iam_role.iam_for_lambda.arn
  handler = "index.test"
  runtime = "nodejs16.x"
}
```

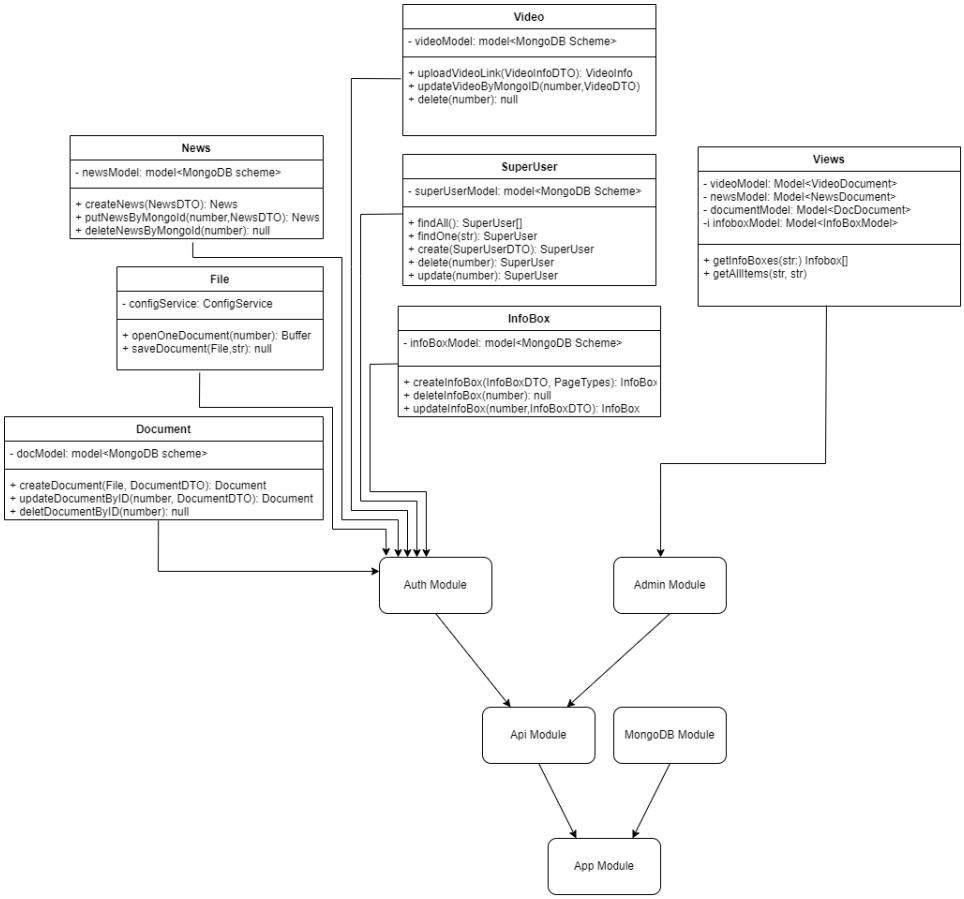

Figure 8: Modules in Nest.Js server application together with services

First obstacle we come to face is the url address of the database. Since Lambda Functions use an uploaded ZIP file which contains, the URL of the mongodb serverless instance is defined once and cannot be changed until a redeploy event. Of course, during the first initialization, the database is setup and passed as an environment variable to the lambda function, which then takes it up and uses it in a connection string to the database. If the database is brought down, or is migrated to a new instance, the URL becomes invalid and our functions cannot perform operations on the database, thus lose the the connection. Keep in mind, that each lambda function that creates and updates data does need to have a connection to the database, thus all these functions are affected.

To overcome this issue, we may include a DNS server in our AWS infrastructure in which the entries will be updated, upon further configuration the lambda functions will only receive the endpoint to the DNS server which does not change on a periodic basis. By this approach, we do not need to

redeploy functions with new environment variables pointing to our database. An example of a lambda function that creates a new video and loads environmental variables can be seen below.

Next problem are the NPM packages, since not everything can be done in pure JavaScript code. When developing an application, we download and install node packages that can be later used in the application. In our case, we are using e.g. the MongoDB npm package that lets us perform connections and operations on the database. Remember that AWS Lambda functions are stateless and so does not persist any data in the environment, so the packages cannot be installed during the execution time and then saved for later usage. We have therefore uploaded not only the main function but a compiled javascript code as well. The layer is a zip file containing all packages and is served to all Lambda functions.

```
const { MongoClient } = require("mongodb");
const conn_uri = process.env.URI
exports.handler = async function (event, context) {

    const client = new MongoClient(conn_uri);
    await client.connect();

    const database = client.db("timak");
    const videos = database.collection("videos");
    const insertResult = await videos.insertOne(event);
    console.log(insertResult);

    // Ensures that the client will close when you finish/error
    await client.close();

};
```

We can also mention the issue with a filesystem storage, as system does not usually store large files inside the database, rather it stores only the metadata in the database and keep the actual file in a filesystem of some sort. In stateless serverless functions, this approach is rather impossible. We can create a Simple Storage Space (AWS S3), that acts as a bucket where the serverless function will store the actual working files [9].

5.2.4 Routing

A crucial component of the Monolithic Application, the request router aids in presenting the monolith and new microservices surrounding it to the outside world as a single unified application. Requests must be sent to either a monolithic application or a microservice, depending on the place where the implementation is located to serve a specific request [22]. Rarely does a monolith and a brand-new microservice coexist side by side without requiring access to each other's data.

Users can expose their business logic running on EC2 instances, AWS Lambda, or any other application via RESTful API using the fully managed

solution known as Amazon API Gateway. [1]

It permits executing different API versions concurrently, which can be used for testing and releasing new versions. Amazon CloudWatch keeps an eye on the entire API. Later on, while investigating performance and latency issues, data collected by Amazon CloudWatch may be used. [1]. Amazon API Gateway allows implementing one separate AWS Lambda as a handler of the user authentication.

An example approach is to gather the API resources of an application and map AWS Lambda Functions from section 5.2.3 to different API objects. Objects that corellate with each other are bundled into single Lambda Function. An example can be seen in figure 9.

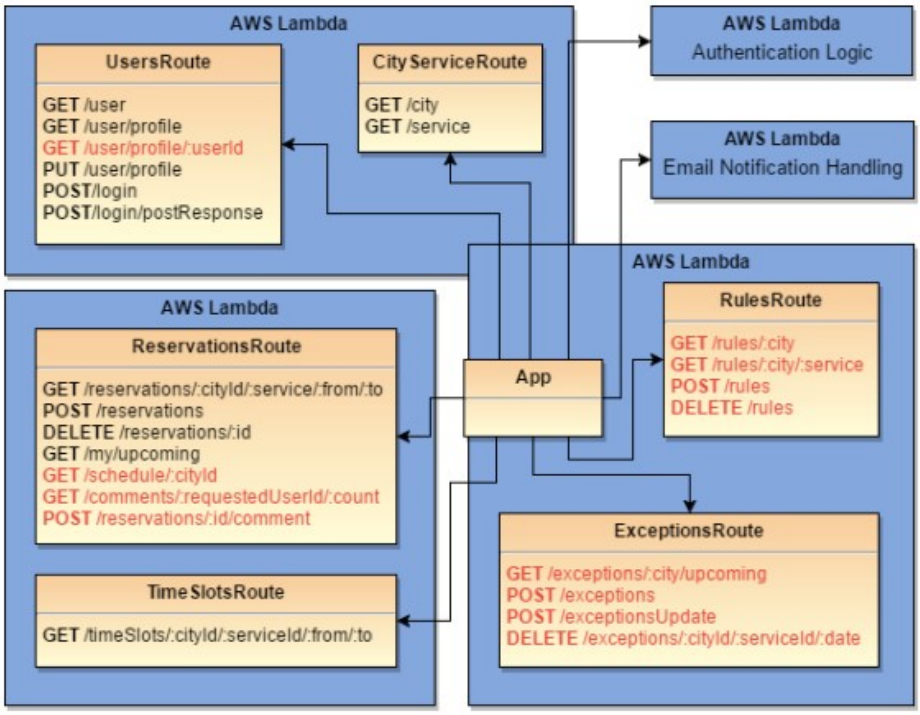

Figure 9: Solteq Wellbeing API moved to AWS Lambda [22]

In our methodology proposition, we perform a total migration, so we can create a new resource of a request router that would not interfere with the old application and acts as a brand new instance. For creating a new serverless app routing option, in our lightweight implementation we have used AWS ApiGateway [1].

6 Related Work

In the work of Kritikos et al. [14], we can see different approaches for serverless transformations. The authors of the document aim to differentiate between different methodologies and showcase the benefits of each transformation methodology.

One particulary interesting solution mentioned is simply called "Serverless Framework" [5]. It is an abstraction framework which abstracts away from the technicalities of two or more serverless platforms. The Serverless Framework was designed to provision AWS Lambda Functions, Events and infrastructure Resource via a simple manifest. An example of this manifest can be seen below.

```
# serverless.yml

provider:
  name: aws
  stage: dev
  region: us-east-1
  profile: production
  stackName: custom-stack-name
  tags:
    foo: bar
    baz: qux
  # Optional CloudFormation tags to apply to the stack
  stackTags:
    key: value
  deploymentMethod: direct
  # List of existing Amazon SNS topics in the same region where notifications about stack events are sent.
  notificationArns:
    - 'arn:aws:sns:us-east-1:XXXXXX:mytopic'
  stackParameters:
    - ParameterKey: 'Keyname'
      ParameterValue: 'Value'
  # Disable automatic rollback by CloudFormation on failure. To be used for non-production environments.
  disableRollback: true
  rollbackConfiguration:
    MonitoringTimeInMinutes: 20
  tracing:
    # Can only be true if API Gateway is inside a stack.
    apiGateway: true
    # Optional, can be true (true equals 'Active'), 'Active' or 'PassThrough'
    lambda: true
```

The Serverless Framework, an infrastructure as a code development stack, converts all of the syntax in this manifest into a single AWS CloudFormation template. A real AWS resource is generated in the cloud provider system for each AWS resource defined in the code. It is by nature a command-line tool that deploys both the code and cloud infrastructure required to create use-cases for serverless applications using simple and understandable YAML syntax. It is a cross framework that works with languages

including Java, Python, Go, Typescript, and Node.js. Additionally, the Framework is totally extendable through the addition of more serverless use-cases and workflows via over 1,000 plugins.

Compared to our solution, it performs all tasks in a more sophisticated way. It is an automatic process and all there is for a developer to do is to create a configuration manifest. It works in multiple paradigms and can make a fully functional serverless platform.

7 Conclusion & Future Work

In the document we have described what the term "serverless" means and what are its uses in cloud infrastructure. We have talked about AWS Lambda and FaaS in general and its differences between a serverless function and a monolithic application. The lightweight implementation of was done in Terraform and is available as an external resource to the document. We have also come across different issues during implementation and answered each issue individually.

The actual implementation of all services and components was done only to a certain degree to show the actual process of transformating a prototype of the application. During the transformation process, a lot of questions arose on the implementation process. These questions such as the connectivity to virtual private network or active directory connection may be implemented in many not trivial different ways.

After the acquisition of knowledge about serverless infrastructure and its benefits, there exists a new solution to our problem called edge networks, which builds on the knowledge of serverless platforms. Edge networks are shards of code deployed on custom delivery networks (CDN) all over the globe. These edge networks aim to battle the cold-boot start of functions, which poses a bad user experience in certain situations.

References

[1] Aws apigateway. `https://aws.amazon.com/api-gateway/`. Accessed: 2022-11-16.

[2] Google trends (n.d.), serverless. `https://trends.google.com/trends/explore?date=2013-10-04%202022-11-04&q=serverless`. Accessed: 2022-11-04.

[3] Mongodb pricing. `https://www.mongodb.com/pricing`. Accessed: 2022-11-06.

[4] Serverless framework. `https://docs.aws.amazon.com/vpn/latest/clientvpn-admin/scenario-onprem.html`. Accessed: 2022-11-19.

[5] Serverless framework. https://www.serverless.com/. Accessed: 2022-11-16.

[6] Serverless vs function-as-a-service (faas): What's the difference? https://www.bmc.com/blogs/serverless-faas/. Accessed: 2022-11-04.

[7] S. T. Ali, J. Long, V. K. Khatri, and M. A. Khuhro. An approach to break down a monolithic app into microservices.

[8] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[9] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264, 2008.

[10] K. Gos and W. Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[11] N. Jean. Migrating monolithic apps to serverless architecture on aws. https://www.sicara.fr/blog-technique/serverless-architecture-migration-aws. Accessed: 2022-11-05.

[12] A. Jindal and M. Gerndt. From devops to noops: is it worth it? In *International Conference on Cloud Computing and Services Science*, pages 178–202. Springer, 2020.

[13] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). Technical report, 2015.

[14] K. Kritikos and P. Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168. IEEE, 2018.

[15] M. Kumar. Serverless architectures review, future trend and the solutions to open problems. *American Journal of Software Engineering*, 6(1):1–10, 2019.

[16] K. Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[17] A. Parasrampuria. Modernizing monolithic applications to serverless architecture on aws. https://impetustech.medium.com/

`modernizing-monolithic-applications-to-serverless-architecture-on-aws-a813c5cad5d0`. Accessed: 2022-11-05.

[18] R. A. P. Rajan. Serverless architecture-a revolution in cloud computing. In *2018 Tenth International Conference on Advanced Computing (ICoAC)*, pages 88–93. IEEE, 2018.

[19] M. Roberts and J. Chapin. *What is Serverless?* O'Reilly Media, Incorporated, 2017.

[20] P. Sbarski and S. Kroonenburg. *Serverless architectures on AWS: with examples using Aws Lambda*. Simon and Schuster, 2017.

[21] M. Wahl, H. Alvestrand, J. Hodges, and R. Morgan. Authentication methods for ldap. Technical report, 2000.

[22] M. Zaymus. Decomposition of monolithic web application to microservices. 2017.