



**University of
Zurich^{UZH}**

A Hands-on Study on Serverless Computing

*Cepilov Ile
Cugnasco-Gerra, Ticino, Switzerland
Student ID: 13-929-740*

Supervisor: Bruno Rodrigues, Eder Scheid, Dr. Thomas Bocek,
Prof.Dr. Burkhard Stiller
Date of Submission: May 10, 2020

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Description of Work	1
2	Related Work and Background	3
3	Introduction to Serverless	5
3.1	Fundamentals	5
3.2	Motivation	6
4	Serverless Code Patterns	7
4.1	Microservices Pattern	7
4.2	Services Pattern	8
4.3	Monolithic Pattern	9
5	Serverless API Endpoint	11
5.1	Architecture	11
5.2	Deployment of a Serverless API Endpoint	12
5.2.1	Prerequisites	12
5.2.2	Identity and Access Management	12
5.2.3	AWS Cognito	14
5.2.4	API Gateway	16
5.2.5	Lambda Function	19

5.2.6	DynamoDB as Database storage	23
5.3	Nested Stacks with CloudFormation Templates	26
5.3.1	Reference resources from Root to Child stack	27
5.3.2	Cross stack referencing	28
5.3.3	Deployment script	29
6	Discussion	31
7	Summary and Conclusions	33
	Bibliography	35
	Abbreviations	37
	List of Figures	37
	List of Tables	39
	Listings	41

Chapter 1

Introduction

1.1 Motivation

Cloud computing has gained more popularity in business in the last few years due to its capacity of processing large amounts of data. As stated by IDC (International Data Corporation) industry analysts, public IT (Information Technology) cloud services revenue reached \$100 billion in 2016 and a prediction that the market will grow over \$200 billion in 2020 has been done [5].

Nowadays, analysts and companies have acknowledged the cloud as a major force which is slowly changing the entire IT landscape. Such transformation process involves not only how IT companies are going to architect and develop their solutions, but also how data centers are going to be built, and how software is deployed, modifying how the whole lifecycle of a product is going to be handled.

Since IT plays a huge role in today's business environment, also the operational plane of companies has to change. In this regard, it does not matter the size of the company, or which industry it base its operation. Cloud-based solutions, platforms and even infrastructure can help gain better visibility, improve customer experience and allow a quick time-to-market. With cloud platforms, also serverless applications and architecture are gaining momentum and are getting always more popular between companies of all sizes. Fault tolerance, centralized logging, horizontal scalability and deployments are usual problems that developers have to face when dealing with systems and servers. Serverless software takes care of it while still having the freedom to customize the code without hindering functionality.

1.2 Description of Work

The goal of this work is to determine best practices in the serverless eco-system to achieve the aforementioned attributes and produce a manual, which can further be used to guide similar cloud deployments. Also, the goal of this work is to obtain insight over concepts

related to cloud providers. In doing so, a detailed pipeline of how to setup a project with continuous integration best practices and an easy management between environments (local/cloud and development/production) is going to be provided. Since cloud platforms are constantly evolving, the whole pipeline will be enough flexible to adapt to new requirements and technologies.

Chapter 2

Related Work and Background

This manual evolved from the following two books [1] and [2]. The first one includes some practical example which help in comprehending and implementing the popular design patterns mostly being used with serverless architectures. It also shows how to build other important pattern in the big data context, like MapReduce and the Extract, Load, Transform (ETL) which is used to copy data from one source into a destination where the data is represented differently. Moreover, it shows some deployment and CI/CD (Continuous Deployment/Continuous Integration) patterns. The only drawback is that its examples are written in Python and this manual is based on NodeJS. But, patterns still apply independently of the programming language used.

The second book explains how to best utilize technologies such as the cloud, virtualization and configuration automation to manage IT infrastructures. It shows how to manage infrastructure in a dynamic way so that provisioning and service delivery is streamlined and thus, once a pipeline has been implemented, it is fairly simple to bring up a functioning infrastructure. Unfortunately, while it really helped in grasping some conceptual aspects of managing infrastructure, it does not provide any tech recipes scripts or template to get started with.

Chapter 3

Introduction to Serverless

3.1 Fundamentals

There are several ways to define serverless computing (or serverless architecture), like *"...serverless computing is a method of providing backend services on an as-used basis. A Serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure...[3]"* or *"...serverless allows you to build and run applications and services without thinking about servers. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning...[4]"*.

From various definitions found, a few common key attributes used to describe it are:

- Lack of Operating Systems (OS) to configure or manage
- Ability Scale automatically with the usage
- Built-in availability and fault tolerance
- Billing model based on a pay-per-invocation basis (in most cases)

As the term *serverless* suggests, there are no servers to be managed. Recently, developers, in general teams, had to purchase hardware/servers and configure them according to the required application, as well as be concerned with their maintenance in an appropriate location. This process could have taken days and increase costs before developers could deploy their software. With the increasing popularity of cloud computing, the aforementioned process has been improved.

As of today, with only few clicks or API calls, developers are able to get their own **Virtual Private Servers (VPS)** in minutes rather than weeks or months. The tasks that a developer does not have to longer worry about include updating the OS and realizing a number of configurations to deploy the development environment.

Behind a serverless architecture there is no magic, the effort of setting up an environment is not vanished, but rather, it is delegated to specific teams behind platforms that offer

such services, like Amazon Web Services (AWS) or Google Cloud Platform (GCP).

With serverless systems, a new pay-per-use model was introduced. This billing model is based on the time spent while the code is being executed, *i.e.*, total invocation time. It is designed for systems that are stateless and have a finite, relatively short duration.

It is important to highlight that there is no idle time in a serverless deployment at the cloud. On the one hand, when functions are not in use, they are paused thus releasing and making more resource available. Therefore, such a system inherits the cloud flexibility being able to scale down resources based on its usage.

On the other hand, when the number of invocations spikes, the underlying system automatically scales up, providing concurrent invocations. Most cloud providers provide the ability to scale up or down based on different metrics and therefore, reducing costs when the system is not being used outside rush hours.

Failures on servers or hosts is also another important aspect, which developers do not have to worry about. It is guaranteed that when the application code should be invoked, then it will be.

3.2 Motivation

It is difficult not to use serverless nowadays due to its speed, flexibility and adaptability, but it is also important to know in which circumstances it makes more sense.

With serverless, developers usually have to write a single function that will be in charge of computing tasks very well define, *i.e.*, retrieve a user information from the database or update its informations. Therefore, the whole system or application has to be computationally efficient and also predictable (their lifetime is usually known).

Applications can have a state and functions can store their state using a database or store, but the latter do not retain state between invocations in a stateless context.

Moreover, cloud providers can impose an upper limit in memory and in duration of these functions, which can vary. Usually, it goes from up to 900 seconds of invocation time and 3 GB for memory usage [11]. Therefore, long-running processes that need to share state and thus require a great amount of memory usage, are not a good fit for the serverless world, *e.g.* a WebSocket server that needs to retain state and its connection can remain open for hours or even days.

Chapter 4

Serverless Code Patterns

The introduction of serverless architectures also requires a change of paradigm concerning how web applications and partition logic are conceived. An approach toward efficient deployment and operation of serverless applications is described as an example. Further, patterns are presented based on intuitive use cases.

4.1 Microservices Pattern

In an architecture based on a microservice pattern, its applications are typically arranged as a collection of loosely coupled functions. Moreover, those functions are independently deployable, allowing the user to develop in different programming languages and use different databases. This provides a higher degree of flexibility and customization when adopting this pattern.

```
1 service: serverless-books
2 provider: aws
3 functions:
4   booksCreate:
5     handler: handlers.booksCreate
6     events:
7       - http: post books/create
8   booksGet:
9     handler: handlers.booksGet
10    events:
11      - http: get books/get
12   booksDelete:
13     handler: handlers.booksDelete
14     events:
15       - http: delete books/delete
16   messagesCreate:
17     handler: handlers.messagesCreate
18     events:
19       - http: post messages/create
20   messagesGet:
21     handler: handlers.messagesGet
22     events:
23       - http: get messages/get
```

Listing 4.1: Microservice pattern example

As shown in listing 4.1, thanks to the microservice pattern there is a total separation of concerns. Each endpoint represents a single function and each of them addresses a separate concern. If one of them crashes, the environment of others will not be affected and thus, they will keep functioning as supposed.

Moreover, each lambda function is in charge of handling a single event making the debugging a straightforward task. Therefore, even adding new functionality is easily achieved by simply pushing a new unit of deployment.

An important drawback that should be considered when adapting this pattern is the overhead on programming. The more functionality are added into the system, the harder it will get to be managed and often code is duplicated for every function since it cannot be shared. Having an overview of the system can be cumbersome.

Moreover, performance-wise the response time can be slower due to cold starts. In a serverless environment, lambda functions execute inside a container and when they are not invoked for a certain period of time, the whole instance of a container is shutdown in order to release resources. Cold starts happen when a new instance of a container has to be brought up because there are no warm containers available. This process adds a delay in the time taken for the request which vary depending on cloud providers and programming languages.

Lastly, the file size limit of the template defined for specifying and launching resources can be reached in a straightforward manner. As result, for adding new resources certain workarounds have to be applied, which will be further described in this study.

4.2 Services Pattern

The main characteristic of a service patterns is that a single Lambda function handles multiple functionalities, which are usually related by a common data model, like the entity **books**.

```
1 service: serverless-books
2 provider: aws
3 functions:
4   books:
5     handler: handler.books
6     events:
7       - http: post books
8       - http: put books
9       - http: get books
10      - http: delete books
11   messages:
12     handler: handler.messages
13     events:
14       - http: post messages
15       - http: put messages
16       - http: get messages
17       - http: delete messages
```

Listing 4.2: Service pattern example

In this example, all basic CRUD operations (Create, Read, Update and Delete) are performed in a single Lambda function but for each CRUD operation an HTTP endpoint is created.

At the beginning of the Lambda function, a routing method can be implemented in order to parse the *event* body of the request and extract the corresponding HTTP request. By doing so, the correct operation can be performed.

As a result of this pattern, less Lambda functions are required to be implemented and, thus, managing them will be easier. There is a separation of concerns, since a function is going to be responsible for a specific data model.

Moreover, performance can be optimized. When a Lambda function is responsible of managing different operations as observed in the snippet 4.2, it will be invoked more regularly. As a consequence, the container where it is running will stay warm meaning that the end-consumer will run less into cold-starts.

Unluckily, debugging the code can get complicated because Lambda functions are now responsible for multiple jobs. Moreover, a simple router functionality in order to parse and apply the correct logic based on the request or endpoint is required.

4.3 Monolithic Pattern

In this pattern, the entire application logic is written into a single lambda function. It can be easier to manage and comprehend at the beginning, but, as the complexity of the application grows, operations can increase in complexity and become difficult to be maintained. It is more of a traditional setup, where it is possible to deploy a single function in charge of starting an application, e.g. the main function in Java.

Chances of running into cold-starts are then reduced and, theoretically, overall performance shall be slightly increased. However, since different operations are going to be executed, it is hard to understand how it really performs. Each execution of the single Lambda function may have a completely different duration. Moreover, a simple router used in the previous pattern, starts getting nested and thus ensuring that it always directs to the appropriate logic can be hard to be guaranteed. Lastly, the size limit of the function can be easily hit in real world applications.

```
1 service: serverless-books
2 provider: aws
3 functions:
4   books:
5     handler: handler.books
6     events:
7       - http: post books
8       - http: put books
9       - http: get books
10      - http: delete books
11      - http: post messages
12      - http: put messages
13      - http: get messages
14      - http: delete messages
```

Listing 4.3: Monolithic pattern example

Chapter 5

Serverless API Endpoint

In this chapter, an overview of a common API Endpoint architecture based on AWS is given. The most used services in such architecture are briefly introduced and described, *e.g.*, AWS Cognito, IAM, CloudFormation. Afterwards, the process of setting up a serverless API Endpoint is described by means of examples. A small application is implemented and deployed to demonstrate some fundamental parts of the manual. A bash script used for automating the process of packaging and deploying multiple CloudFormation templates is provided at the end as well.

5.1 Architecture

Figure 5.1 represents a common and simplified architecture, which is followed throughout this tutorial. On the left-hand side **Users** are going to utilize the API. Hence, users have to be authenticated. **Amazon Cognito** is the service responsible for handling authentication and granting users access to the corresponding resources thanks to an *Access Token*.

Once authenticated, users can send HTTP requests to the **API Gateway**. The latter handles all tasks related to accepting and processing API calls, including traffic management, CORS support, authorization and access control. It acts as the “front door“ for applications in order to access data and functionality from backend services [6].

The API Gateway forwards **AWS Lambda** enables developers to run code without managing servers or provisioning them. It runs code when triggered, in this case from the API Gateway, and it uses only the necessary resources specified [7].

Different scenarios are possible depending on the called endpoint. A common use case is when a AWS Lambda function has to write or retrieve data from the storage and return the result to the user. In this case, a relational database is used to persist information about users. **DynamoDB** service is going to be adopted since it simplifies setting up, operating and scaling a key-value and document database in the cloud.

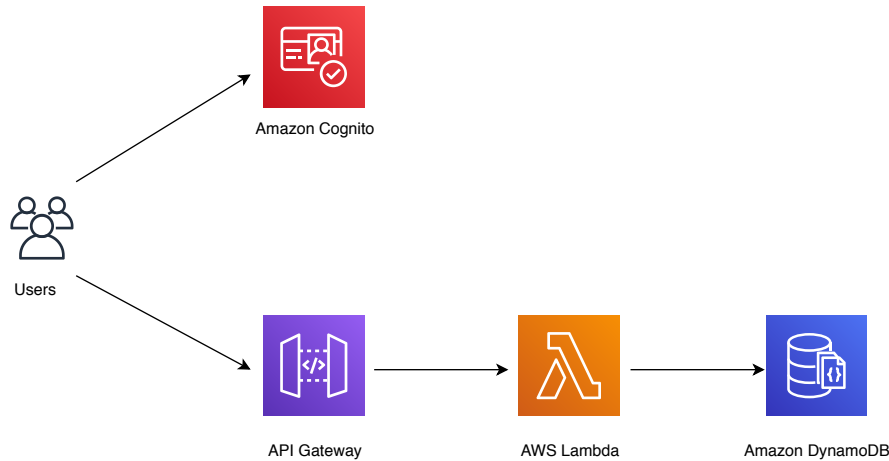


Figure 5.1: Serverless API Endpoint Architecture

5.2 Deployment of a Serverless API Endpoint

In this section, a hands-on guide on how to deploy a serverless API endpoint starting from the beginning is provided. The platform used herein is the AWS cloud. A valid alternative is *Google Cloud Platform* (GCP). The decision for AWS was taken after an analysis of which platform was the most popular platform used. As a metric shown in [9], there are more questions on Stack Overflow regarding the AWS ecosystem than GCP. Furthermore, statistics on web searches show that AWS is more popular.

First, every step taken to setup the Identity and Access Management is explained through the AWS online console. This allows the user to get used with the AWS services. Then, the following steps are going to be shown my means of CloudFormation template. **CloudFormation** is an AWS service assisting the modeling and setting of all AWS resources needed. By providing a template with all the required resources, switching deployment region or environment can be easily achieved in a reliable and quick way. Stacks defined by templates can be updated or replicated as needed.

5.2.1 Prerequisites

In order to follow this guideline, some requisites are specified. Firstly, an AWS account has to be created, this step is not covered in this tutorial. Moreover, the AWS Command Line Interface has to be installed and configured with your account. In order to do so, the following official guidelines can be used [10].

5.2.2 Identity and Access Management

AWS Identity and Access Management (IAM) is a service enabling to manage AWS services and resources. Users and groups with permissions can be specified to enable users

to access some AWS resources. Otherwise, access to these services is denied. To deploy resource and the CloudFormation template, a user with the right permissions has to be created. First, it is necessary to log in into the **AWS Management Console** and look for the service named **IAM** using the search bar. Once selected, on the left menu, select the item **Groups**. There, it is possible to create a group with all permissions. For the sake of simplicity we will just create a group in which members are able to execute every action. Start by creating a group like shown in the screenshot 5.2, then move to the next step and assign the policy *AdministratorAccess*, like shown in 5.3 After creating the group, a

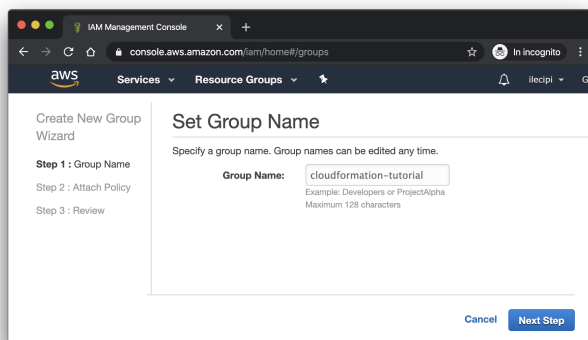


Figure 5.2: Create a new group

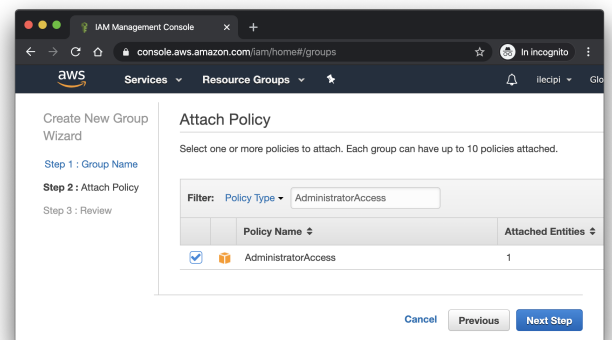


Figure 5.3: Attach a policy to a group

new user has to be created and assigned to that group as well. In order to do so, on the left side select the option **Users**, then click **Add user**. When adding a user, make sure to select the correct **Access Type** like in 5.4, i.e. **Programmatic Access**. After that assign it to the group created before, in 5.5 to the group *cloudformation-tutorial*.

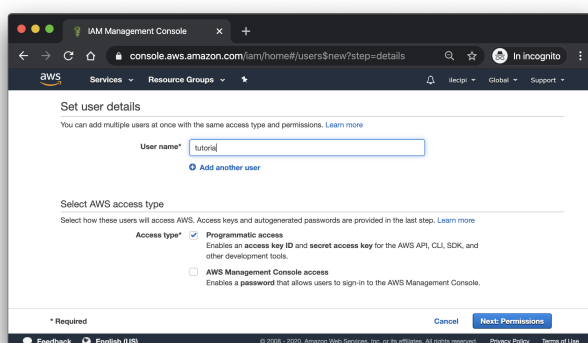


Figure 5.4: Set user details

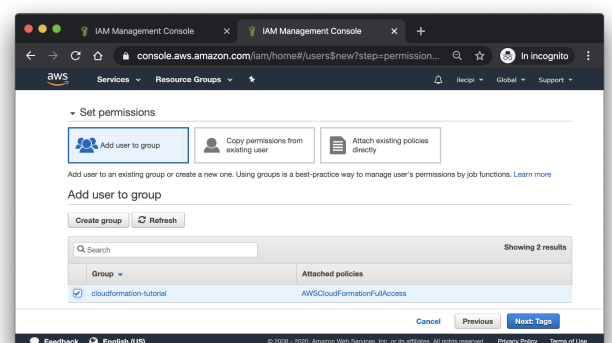


Figure 5.5: Add user to a group

After creating a user, make sure to store the **Access key ID** and the **Secret access key**, shown in 5.12. They are needed in order to be able to deploy from the AWS CLI. This step is not going to be covered since it is a prerequisite.

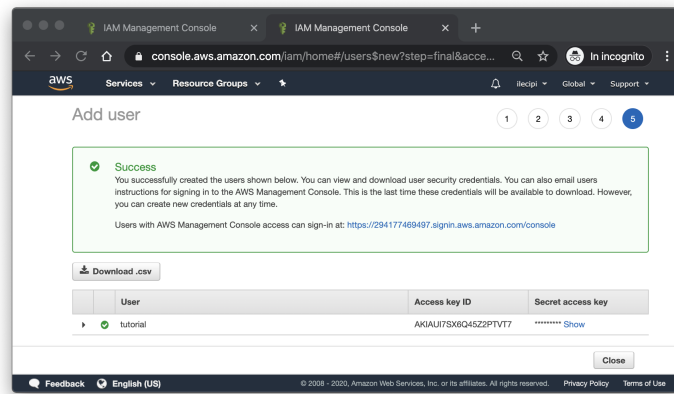


Figure 5.6: User Access Key and Secret access key

5.2.3 AWS Cognito

Now that everything is set up, the architecture shown in 5.1 can be translated into a CloudFormation template.

```

1  Description:
2    Serverless API Endpoint tutorial
3
4  Parameters:
5    VariableTest:
6      Type: String
7      Default: The name value of a variable that can be used.
8    Environment:
9      Type: String
10     Default: dev
11
12  Resources:
13    UserPool:
14      Type: "AWS::Cognito::UserPool"
15      Properties:
16        UserPoolName: MBM-Tutorial
17        AliasAttributes:
18          - email
19        AutoVerifiedAttributes:
20          - email
21
22    UserPoolClient:
23      Type: "AWS::Cognito::UserPoolClient"
24      Properties:
25        ClientName: MBM-tutorial-Web
26        UserPoolId: !Ref UserPool
27        GenerateSecret: false

```

Listing 5.1: Amazon Cognito User Pool

The YAML file starts with a field **Description**, which can be arbitrarily defined. Underneath it, environmental parameters can be defined, they can be useful to store values across multiple Lambda functions and avoid to repeat them. The **Resources** section is fundamental for this guide. It declares the AWS resources that will be used in the stack, such as Cognito Pools, S3 buckets, EC2 instances and so on. Resource are specified as follows:

```

1  Resources:

```

```
2 Logical ID:
3   Type: Resource type
4   Properties:
5     Set of properties
```

Listing 5.2: Definition of generic AWS Resource

In the resource section a *UserPool* is defined by defining the **Type** as *AWS::Cognito::UserPool*. Only the email is set as an attributes for an user. Right after it, a **UserPoolClient** has to be defined in order to be able to register an user for the first time without authentication. Notice that the **UserPool** created before is now dynamically referenced with:

```
UserPoolId: !Ref UserPool
```

where **UserPool** is the shown Logical ID in 5.2. Now that everything is configured, the YAML file can be packaged and deployed so that a stack is created. Open the console where the YAML file is located and type the following command in order to package it:

```
aws -profile mbm cloudformation package -template-file serverless.yaml
-output-template-file outputtemplate.yaml -s3-bucket tutorial-backend
```

where **-profile** indicates the corresponding profile configured in the AWS CLI, **-template-file** indicates the just created yaml file that needs to be uploaded, **-output-template-file** indicates the generated file which will create the stack and **-s3-bucket** indicates the location in which the YAML file is going to be uploaded.

If the file gets successfully packaged, it is ready to be deployed. Type the following command:

```
aws -profile mbm cloudformation deploy -template-file outputtem-
plate.yaml -stack-name tutorial-mbm
```

where **stack-name** indicates the stack which is going to be created or updated when deploying it. Usually, when multiple developers are working on a project, each of them will have its own stack so that if something wrong gets deployed, it won't affect other stacks and thus everybody else will be able to develop without any issue. If everything went successfully, a new stack is deployed. In the AWS Console under the **CloudFormation** service, the new create stack can be seen in figure 5.7 and once selected, in the resources, both **UserPool** and **UserPoolClient** are created as well, like shown in 5.8. Now that Amazon Cognito is configured and a UserPool is created, the next step is to define an API Gateway.

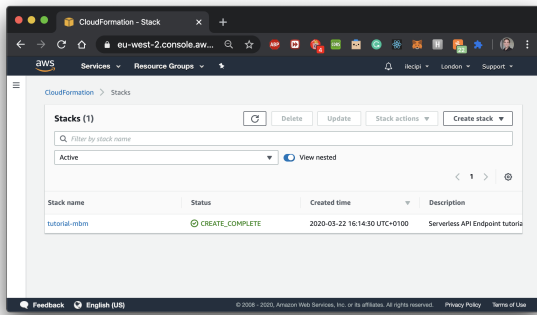


Figure 5.7: New created stack

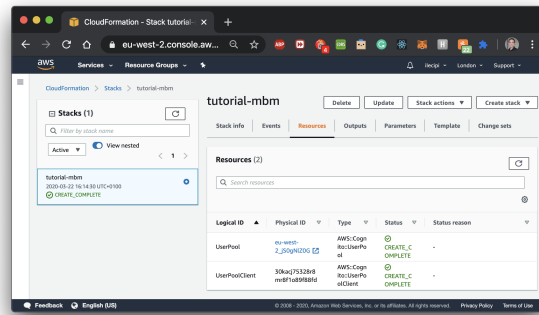


Figure 5.8: New resources for a stack

5.2.4 API Gateway

The API Gateway is a necessary service in order to access Lambda functions. For an app to call or use the Lambda functions, they need to be exposed through API methods in the API Gateway. In this section, an API Gateway is deployed and with it a Lambda function. It is not possible to deploy only a API Gateway without having at least one Lambda function, an error gets thrown otherwise.

In snippet 5.3, the API resource is declared. The `StageName`, even though not required, is an important field since it can be used to differentiate between development and production API Gateways. Depending on the use case, `CORS` (Cross-Origin Resource Sharing) might be desired. `CORS` allows methods in API Gateway to request resources from different domains, *e.g.* a Lambda function that has to call an external API to gather weather information or location informations (from Google Maps). Methods, headers and origin can be generically specified like shown in 5.3.

Instead of using IAM roles and policies to access endpoints exposed by the API Gateway, it is possible to use an Amazon Cognito user pool that has been created before. The existing user pool can be simply referenced by writing:

```
UserPoolArn: !GetAtt UserPool.Arn
```

Users must first sign in to the user pool, obtain an access token and then include the latter in the request. The API call succeeds only if the token is present and still valid.

As said before, it is not possible to deploy only an API Gateway without providing a Lambda function. Therefore, a simple Lambda function is explained in the following section. In 5.3 at line 15 a `GetUser` service is specified. The type is of `AWS::Serverless::Function`, which is used to declare Lambda Functions. Since it is only a small and brief example, the code of the function is directly written into the template instead of referencing the file. This is possible by specifying the `InlineCode` property. The `Handler` indicates the name of the method within the code that the Lambda function calls in order to execute the function. In the example, the method is exported as `lambdaHandler` and therefore the property is `index.lambdaHandler`. The `Runtime` identifies the function's runtime, in this case NodeJS.

An important property is **Events**. With this property it is possible to specify with which events the Lambda function should be invoked. Since it has to be executed from the API Gateway, an event of type **API** is declared. **Path** determines the path of the endpoint and **Method** indicates the HTTP method required. In the example, the endpoint is reachable when calling the following path as a **GET** request: **API_endpoint/auth/users**. Since it is possible to have multiple API Gateway, it is necessary to connect it to the gateway deployed before. It can be done by referencing it in the **RestApiId** as follows:

RestApiId: !Ref ApiGateway

```

1  ApiGateway:
2    Type: "AWS::Serverless::Api"
3    Properties:
4      StageName: !Ref Environment
5      Cors:
6        AllowMethods: "'OPTIONS,GET,POST,PUT,DELETE'"
7        AllowHeaders: "'*'"
8        AllowOrigin: "'*'"
9      Auth:
10       Authorizers:
11         MyCognitoAuthorizer:
12           UserPoolArn: !GetAtt UserPool.Arn
13           Header: "Authorization"
14
15  GetUser:
16    Type: AWS::Serverless::Function
17    Properties:
18      InlineCode: |
19        'use strict';
20
21        exports.lambdaHandler = async event => {
22          return {
23            statusCode: 200,
24            body: JSON.stringify(
25              {
26                message: '*MBM-Tutorial* Successfully executed your lambda function!',
27              },
28              null,
29              2
30            ),
31          };
32        };
33
34      Handler: index.lambdaHandler
35      Runtime: nodejs12.x
36      Events:
37        GetUser:
38          Type: Api
39          Properties:
40            Path: /auth/users
41            Method: get
42            RestApiId: !Ref ApiGateway
43            # Auth:
44            #   Authorizer: MyCognitoAuthorizer

```

Listing 5.3: Definition of an API Gateway

If the deployment of the template from the AWS CLI was successful, the following result should be achieved:

```

ilecipi@MacBook-Pro-di-Ile-390: ~/resilio/Dropbox/UZH/Master/MBM/tutorial
➜ ~/resilio/Dropbox/UZH/Master/MBM/tutorial aws --profile mbm cloudformation package --template-file serverless.yaml --output-template-file outputtemplate.yaml --s3-bucket cf-templates-1skjxe1n6jsw-eu-west-2 && aws --profile mbm cloudformation deploy --template-file outputtemplate.yaml --stack-name tutorial-mbm --capabilities CAPABILITY_IAM
Uploading to 86655279f75cf266317f50a55e5446b1 5539 / 5539.0 (100.00%)
Successfully packaged artifacts and wrote output template to file outputtemplate.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file /Users/ilecipi/resilio/Dropbox/UZH/Master/MBM/tutorial/outputtemplate.yaml --stack-name <YOUR STACK NAME>

Waiting for changeset to be created..
Waiting for stack create/update to complete
Successfully created/updated stack - tutorial-mbm
➜ ~/resilio/Dropbox/UZH/Master/MBM/tutorial

```

Figure 5.9: Result of a successful deployment

In the AWS Lambda service, under applications, there should be a new entry, namely our newly deployed application like shown below.

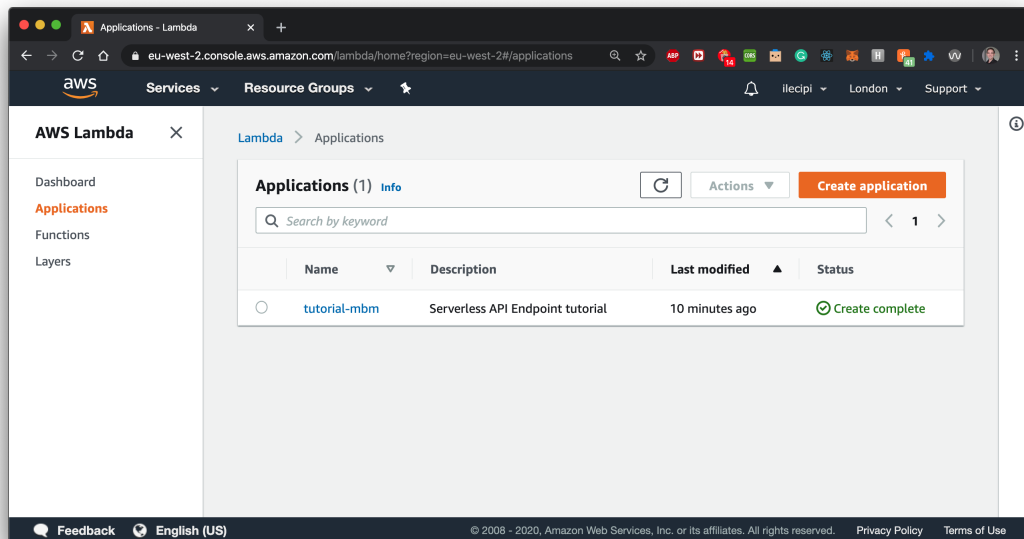


Figure 5.10: Applications in AWS Lambda

Once opened, the endpoint, under which the application is reachable, is displayed. Moreover, under **Resources**, the API Gateway and the Lambda function are available as well.

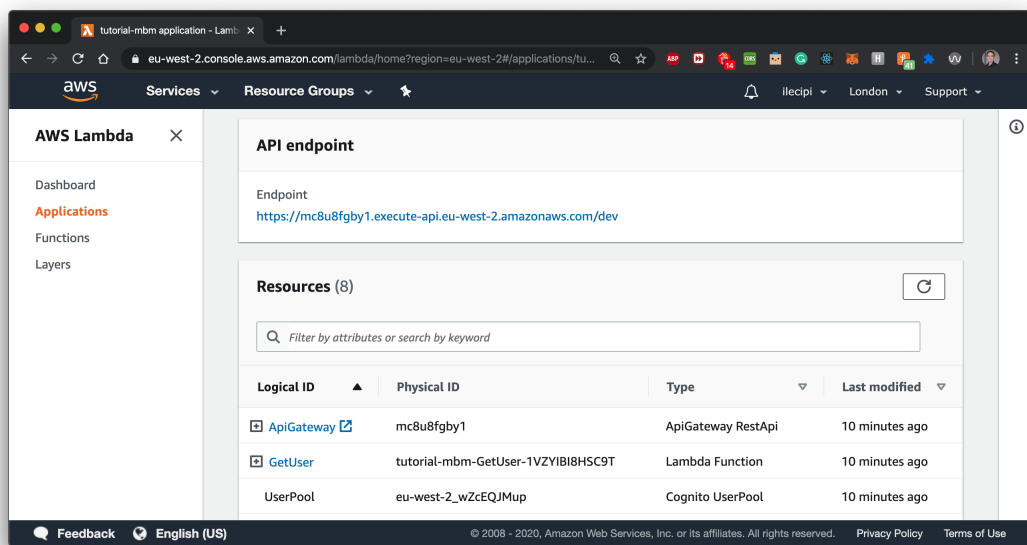


Figure 5.11: API Endpoint and Resources of the newly created application

Now, if the API Endpoint url is visited and the path of the lambda function is added as well (/auth/users), the result of our function is retrieved as shown:

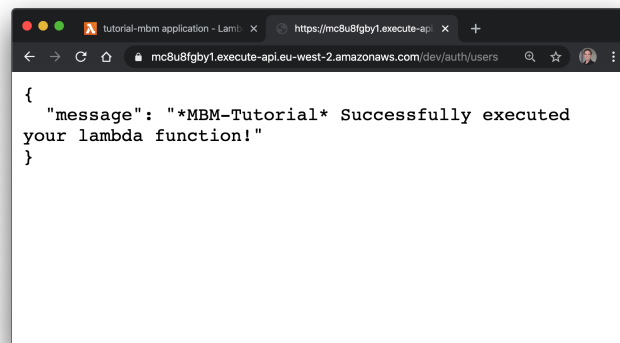


Figure 5.12: Result of successfully calling the endpoint

So far only a simple Lambda function has been deployed, this because the focus of this section was actually deploying an API Gateway and not a Lambda function. In the next section, another approach to how to properly deploy a Lambda function is given.

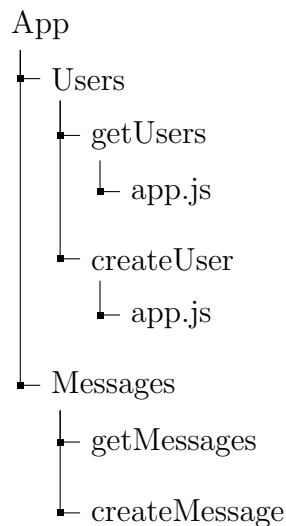
5.2.5 Lambda Function

In this section two different endpoints are going to be explained and deployed. The first one is going to be in charge of creating a new user and the second one of retrieving a list

of users. Firstly, the folder structure will be explained so that the reader can understand the microservice pattern better.

Serverless Endpoint Folder Structure

In this section, the folder structure of a Microservice based pattern is given. As said in a previous chapter, in this type of architecture, applications are arranged as a collection of loosely couple functions and are independent of each other. This behaviour is reflected below where the entity **User** is separated from the entity **Messages**. Moreover, each operation done on an entity is separated in a different folder as well which usually contains a single entry point, namely the file `app.js`.



In doing so, it is possible to change the logic of a method, *i.e.* `createUser`, and be sure that the other endpoint are not going to be broken after the deployment. In addition, other endpoints do not need to be redeployed since they haven't changed.

CloudTemplate example

In 5.9 the YAML code snippet for the CloudTemplate is shown. Two endpoints are now defined, `GetUsers` and `CreatUser`. Their definition is almost identical, differences are to be seen in the property `CodeUri` and in the `Method` type. As explained above, the respective code to be executed is located in different folders, therefore one endpoint will point to `Users/getUsers` and the second one to `Users/createUser`. The method type is based on endpoint intent. The `GET` method is usually used to read or retrieve a representation of resources, whereas the `POST` method is mostly utilized to create new resources.

```
1  GetUsers:
2    Type: AWS::Serverless::Function
3    Properties:
4      CodeUri: Users/getUsers
5      Handler: app.lambdaHandler
6      Runtime: nodejs12.x
7      Events:
8        GetUser:
9          Type: Api
10         Properties:
11           Path: /auth/users
12           Method: get
13           RestApiId: !Ref ApiGateway
14
15  CreateUser:
16    Type: AWS::Serverless::Function
17    Properties:
18      CodeUri: Users/createUser
19      Handler: app.lambdaHandler
20      Runtime: nodejs12.x
21      Events:
22        GetUser:
23          Type: Api
24          Properties:
25            Path: /auth/users
26            Method: post
27            RestApiId: !Ref ApiGateway
```

Listing 5.4: Declaration of Users' endpoints

In 5.9 the javascript code to run the lambda function is shown. It is quite simple, it has a single exported method `lambdaHandler`, which is the one referenced above in the CloudFormation template and it gives back a list of predefined users. Note that at the moment no databases is created and connected. Therefore, users are hardcoded and always the same.

```
1  'use strict';
2
3  exports.lambdaHandler = async event => {
4    const users = [
5      {
6        firstName: "Victor",
7        secondName: "Avery"
8      },
9      {
10       firstName: "Christopher",
11       secondName: "Sanderson"
12     }
13   ];
14   return {
15     statusCode: 200,
16     body: JSON.stringify(users)
17   };
18 };
```

Listing 5.5: Get a list of users in NodeJS

The next code snippet in 5.10 shows the `createUser` endpoint. Outside of the exported method, there are two variables, *users* and *invocationCounter*. Those variables are used inside the main method to add a new user each time the Lambda Function is invoked. An interesting point here is that between invocation the values inside those two variables are going to be persisted as long as the container running the lambda function is running. This means that the *invocationCounter* will keep increasing if we call this function within

an average of 15 minutes. Otherwise, the container will be shutdown and the Lambda function loses its state.

```

1  'use strict';
2  const users = [];
3  let invocationCounter = 0;
4
5  exports.lambdaHandler = async event => {
6      const newUser = {
7          firstName: "Doe",
8          lastName: "John",
9          invocation: invocationCounter++
10     };
11
12     users.push(newUser)
13     return {
14         statusCode: 200,
15         body: JSON.stringify(
16             {
17                 message: 'New user successfully created',
18                 invocationCounter,
19                 data: users,
20             }
21         ),
22     };
23 };

```

Listing 5.6: Create a new user in NodeJS

This behavior is exposed in 5.13 where the endpoint is actually invoked. It has been invoked three times and therefore the invocationCounter is equals to 3. At each execution, a corresponding user with the respective invocationCounter is added and therefore we see that the variable `invocation` is increased by one as users are added.

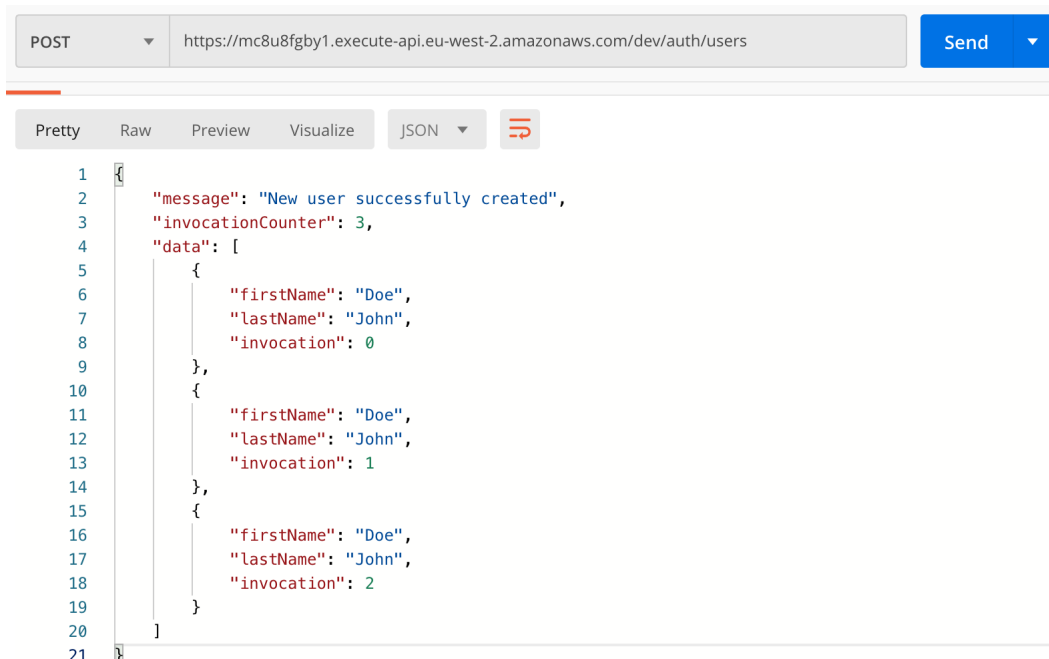


Figure 5.13: CreateUser response

This scenario is brought here only for demonstration purposes. A real use case for this would be when a Lambda function is in charge of managing the connection to a database.

Since as few as possible connections should be done to a database, there could be a single connection or a pool of connection defined outside the exported method. In doing so, when the function is executed it can utilize the already available connection and once the execution is done, it will be released. This would improve performance since a new connection does not have to be opened every time and it also reduces the total amount of new connections because they can be reused.

So far, there was no persistent storage in our application. Users were not saved outside their lambda functions and not retrieved as well. In the coming section it is explained how to setup a database and establish a connection from within our Lambda functions so that they can communicate with each other.

5.2.6 DynamoDB as Database storage

Amazon DynamoDB is a NoSQL database that delivers single-digit millisecond performance at any scale and at the same time it requires a minimal amount of setup and maintenance from the developer's side. It supports key-value and document data structure, like JSON objects.

DynamoDB is a very popular choice for serverless applications running in AWS since it is aligned with all the values of a serverless ecosystem, like automatic scaling, pay-per-invocation and it has no servers to manage.

In 5.7 a simple way to manage a DynamoDB table in the YAML file is shown.

```
1  UsersTable:
2    Type: AWS::DynamoDB::Table
3    Properties:
4      TableName: users
5      AttributeDefinitions:
6        - AttributeName: id
7          AttributeType: S
8        - AttributeName: username
9          AttributeType: S
10     BillingMode: PROVISIONED
11     KeySchema:
12       - AttributeName: id
13         KeyType: HASH
14       - AttributeName: username
15         KeyType: RANGE
16     ProvisionedThroughput:
17       ReadCapacityUnits: 1
18       WriteCapacityUnits: 1
```

Listing 5.7: Create a table in DynamoDB

In this example a **users** table is created when deploying the application with an **id** as partition key and **username** as a sort key. Multiple aspects can be managed like attributes, partition keys and provision properties. In this section another important aspects of the AWS CloudFormation template is introduced. Resources in AWS templates tend to have shared configuration, such as Environment, Runtime and so on. Instead of duplicating those values in every resource, it is possible to simplify the management of environment variables and avoid repeating any hard-coded values by writing them once in the **Globals** section and let the resources inherit those values. Listing 5.8 shows how to do that by

referencing the name of the dynamodb tables just created. This allows every lambda function to gather the correct name of the table from the environment variables.

```

1  Globals:
2    Function:
3      Environment:
4        Variables:
5          USERS_TABLE: !Ref UsersTable

```

Listing 5.8: Globals Section in Template

Listing 5.9 shows a simplified version of how a user or a list of users can be returned from a table in DynamoDB. At line 2 & 3 the AWS SDK is declared and from it the `AWS.DynamoDB.DocumentClient` class is exposed, which will allow to directly execute some methods to gather data. The variable `USERS_TABLE` contains the name of the table where users are saved, this is taken from the environmental variables and will be then passed as parameter in the method used for retrieving data. Based on the query parameters in the request, either a single user is going to be returned from the database based on its id or a list of existing users.

```

1  'use strict';
2  const AWS = require("aws-sdk");
3  const dynamodb = new AWS.DynamoDB.DocumentClient();
4  const {USERS_TABLE} = process.env
5
6  exports.lambdaHandler = async event => {
7    const id = event.queryStringParameters ? event.queryStringParameters.id : undefined
8
9    if(id){
10     return await getUser(id)
11   } else {
12     return await getAllUsers();
13   }
14 };
15
16 const getAllUsers = () => {
17   return dynamodb
18     .scan({
19       TableName: USERS_TABLE,
20     })
21     .promise()
22     .then(response => {
23       return responseHandler(true, response.Items, 201);
24     })
25     .catch(err => {
26       return responseHandler(false, err, err.statusCode);
27     });
28 };
29
30 const getUser = id => {
31   return dynamodb.query ({
32     TableName: USERS_TABLE,
33     KeyConditionExpression: 'id = :id',
34     ExpressionAttributeValues: {
35       ':id': id
36     }
37   }).promise()
38     .then(response => {
39       return responseHandler(true, response.Items, 201);
40     })
41     .catch(err => {
42       return responseHandler(false, err, err.statusCode);
43     })
44 };
45 const headers = {

```

```

46  "X-Requested-With": "*",
47  "Access-Control-Allow-Headers":
48  "Content-Type,X-Amz-Date,Authorization,X-API-Key,x-requested-with",
49  "Access-Control-Allow-Origin": "*",
50  "Access-Control-Allow-Methods": "POST,GET,OPTIONS,PUT,DELETE"
51  };
52  const responseHandler = (success, data, code) => {
53    let body;
54    if (success) body = { success: success, data: data };
55    else {
56      // make sure error object has a message field
57      if (data && data.message) body = { success: success, error: data };
58      else body = { success: success, error: { message: data } };
59    }
60    return {
61      statusCode: code || (success ? 200 : 401),
62      headers,
63      body: JSON.stringify(body)
64    };
65  };

```

Listing 5.9: Get a single user or a list of users from DynamoDB

The snippet in 5.10 on the other hand, shows how to insert a new user in the database. The new user details are taken from the request's body, like shown at line 10. Those two code snippets are here only to help the reader understand better the whole tutorial. Explaining what happens in it is out of scope and therefore not explained in details.

```

1  'use strict';
2  const AWS = require("aws-sdk");
3  const dynamodb = new AWS.DynamoDB.DocumentClient();
4  const {uuid} = require("uuidv4");
5  const {USERS_TABLE} = process.env
6
7
8
9  exports.lambdaHandler = async event => {
10   const body = JSON.parse(event.body);
11   const user = {
12     id: uuid(),
13     ...body
14   }
15   console.log(user)
16   return await createDynamoDBUser(user)
17 };
18
19 const createDynamoDBUser = user => {
20   console.log("create dynamo db user");
21   return dynamodb
22     .put({
23       TableName: USERS_TABLE,
24       Item: user
25     })
26     .promise()
27     .then(response => {
28       return responseHandler(true, user, 201);
29     })
30     .catch(err => {
31       return responseHandler(false, err, err.statusCode);
32     });
33 };
34
35 const headers = {
36   "X-Requested-With": "*",
37   "Access-Control-Allow-Headers":
38   "Content-Type,X-Amz-Date,Authorization,X-API-Key,x-requested-with",
39   "Access-Control-Allow-Origin": "*",
40   "Access-Control-Allow-Methods": "POST,GET,OPTIONS,PUT,DELETE"

```

```
41 };
42 const responseHandler = (success, data, code) => {
43   let body;
44   if (success) body = { success: success, data: data };
45   else {
46     // make sure error object has a message field
47     if (data && data.message) body = { success: success, error: data };
48     else body = { success: success, error: { message: data } };
49   }
50   return {
51     statusCode: code || (success ? 200 : 401),
52     headers,
53     body: JSON.stringify(body)
54   };
55 };
```

Listing 5.10: Create/Insert a user in DynamoDB

At this point, every step from configuring an incognito pool for managing users, to an API gateway to manage connections and a database to store the information was explained. All those resource are defined in the AWS CloudFormation template which simplifies the provisioning and management of AWS services. In the next section, it will be explained how to manage different stacks and templates when a single template reaches its maximal number of resources. Moreover, a guideline to how to distinguish between development and production environment and other important aspects is going to be explained.

5.3 Nested Stacks with CloudFormation Templates

Nested stacks are AWS CloudFormation stacks inside other stacks. Those stacks are called child stacks and are considered as resources of the parent stack. As the infrastructure and size of the application grows, parts of it might be repeated multiple times and overlapping. The size of the template itself keeps uselessly growing as well. This not only obstructs a clear overview of the system and its usability but it can also lead to encounter the AWS CloudFormation limits. Indeed, the template body size may have a maximum size of 51,200 bytes and the maximum number of resources declared can be 200 [12]. The turning strategy suggested from AWS is to separate the template into multiple templates by using nested stacks. Therefore, in this section this problem will be tackled by proposing a simple solution. Moreover, a solution for staging environments and delegating the whole automation for deployment to a bash script will be proposed.

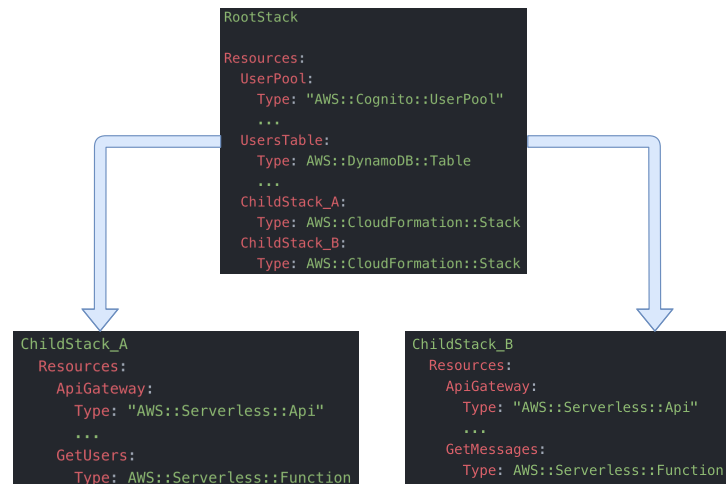


Figure 5.14: Nested Stacks in AWS CloudFormation

Figure 5.14 shows a Root stack, which is the entry point for the CloudFormation template and two child stacks which are referenced in the Root stack. Each stack has to declare an API Gateway if it needs to expose its endpoints. In the root stack are usually defined resources that need to be shared by multiple stacks and referencing them from the Parent stack to the child stack is straightforward. There might be multiple ways of structuring your nested templates, like separating them by most used resources so that the template can be reused one-to-one somewhere else. Since the examples proposed in this manual are mostly based on a microservice pattern, nested template can also be structured based on the separation of concepts design pattern. Each template can be in charge of a well-defined entity, like users or messages.

5.3.1 Reference resources from Root to Child stack

Resource defined in the Root stack can be passed as a reference to children stacks. This allows to share common resources among nested stacks and avoid duplicating resources. In the following snippet, the UserPool and the UsersTable are passed down to the child stack. This is possible thanks to the functions `!Ref`, which returns the value of the specified parameter or resource and `!GetAtt`, which returns the value of an attribute from a resource in the template. For being able to use the UserPool in a child stack, it has to be referenced by its Amazon Resource Name (ARN), which uniquely identifies AWS resources. Important to mention is that if a variable needs to be passed from the Root to the Child stack, then it has to be declared in the `Parameters` section of the template, like shown in 5.11.

```

1 Description: RootStack - Serverless API Endpoint tutorial
2
3 Resources:
4   UserPool:
5     Type: "AWS::Cognito::UserPool"
6     ...
7   UsersTable:
8     Type: AWS::DynamoDB::Table
9     ...
  
```

```

10 ChildStack_A:
11   Type: AWS::CloudFormation::Stack
12   Properties:
13     Parameters:
14       Environment: !Ref Environment
15       UserPoolArn: !GetAtt UserPool.Arn
16       UsersTable: !Ref UsersTable
17       TemplateURL: !Sub "https://s3.amazonaws.com/${TemplatePath}/child.yaml"
18
19   ----- END OF ROOT STACK DEFINITION -----
20   ----- CHILDSTACK_A DEFINITION -----
21
22   Description: ChildStack_A Serverless API Endpoint tutorial
23
24   Parameters:
25     UserPoolArn:
26       Type: String
27
28   Resources:
29     ApiGateway:
30       Type: "AWS::Serverless::Api"
31       Properties:
32         ...
33       Auth:
34         Authorizers:
35           MyCognitoAuthorizer:
36             UserPoolArn: !Ref UserPoolArn
37             Header: "Authorization"

```

Listing 5.11: Reference resources as environment variables

5.3.2 Cross stack referencing

A cross-stack reference is a way for one CloudFormation template to refer to the resources in another CloudFormation template. In the precedent section it was explained how to reference resources from a Root stack down to a Child stack. Now, it is shown how to reference resources between children on the same level. This can be useful when a Lambda function needs to be invoked in another Lambda function defined in another template.

In order to achieve this, the **Outputs** section of a template has to be defined with the corresponding values that need to be exported. Let's assume that a Lambda Function denominated **GetMessages** is defined and has to be invoke from another Lambda Function in another template/stack. It will be exported as follows in its own template (one of the children):

```

1  Outputs:
2    GetMessagesArn:
3      Description: the GetMessages Function Arn
4      Value: !GetAtt GetMessages.Arn
5
6  Resources:
7    GetMessages:
8      Type: AWS::Serverless::Function
9      Properties:
10     ...

```

Listing 5.12: Export a Function's ARN in the Outputs section

In order to be imported by another child, it has to be declared in the parent/root stack. Now, let's assume that the function that has to be exported is defined in the stack called

ChildB and it has to be invoked by another lambda function in the stack ChildA, the parent template will look as follows:

```

1  Resources:
2    ChildA:
3      Type: AWS::CloudFormation::Stack
4      Properties:
5        Parameters:
6          GetMessagesFunctionArn:
7            Fn::GetAtt:
8              - ChildB
9              - Outputs.GetMessagesArn
10     TemplateURL: !Sub "https://s3.amazonaws.com/${TemplatePath}/child_a.yaml"
11
12    ChildB:
13      Type: AWS::CloudFormation::Stack
14      Properties:
15        TemplateURL: !Sub "https://s3.amazonaws.com/${TemplatePath}/child_b.yaml"

```

Listing 5.13: Cross stack referencing in the parent template

At this point, the function's ARN is available in the other stack and thus it can be invoked.

5.3.3 Deployment script

Dividing a big CloudFormation template may bring many benefits, but this means that multiple templates have to be deployed at the same time. If the whole architecture is split into many sub templates, then it might be difficult to manage all of them when deploying an application.

For this, a small script is provided which helps in packaging all defined templates into an S3 bucket and then deploy it.

```

1  source ./env.$1
2
3  CHILD_TEMPLATES=(child_a.yaml child_b.yaml)
4  STACK_NAME=$CLOUDFORMATION_NAME-$ENVIRONMENT
5
6  echo "Copying child stack templates to S3"
7  for child_template in ${CHILD_TEMPLATES[*]}; do
8    echo "Packaging template file $child_template"
9    aws --profile $AWS_PROFILE cloudformation package --template-file $child_template --output-
10     template-file outputtemplate_dev.yaml --s3-bucket $ARTIFACT_STORE
11    echo "Uploading template file $child_template"
12    aws --profile $AWS_PROFILE s3 cp outputtemplate_dev.yaml "s3://$ARTIFACT_STORE/templates/
13     $child_template"
14  done
15
16  echo "Packaging master stack"
17  aws --profile $AWS_PROFILE cloudformation package \
18    --template-file root_template.yaml \
19    --output-template-file cloudformation-outputtemplate_dev.yaml \
20    --s3-bucket $ARTIFACT_STORE
21
22  echo "Deploying master stack to $STACK_NAME"
23  aws --profile $AWS_PROFILE cloudformation deploy \
24    --template-file cloudformation-outputtemplate_dev.yaml \
25    --stack-name $STACK_NAME \
26    --capabilities CAPABILITY_IAM CAPABILITY_AUTO_EXPAND \
27    --parameter-overrides \
28    Environment=$ENVIRONMENT \
29    TemplatePath=$ARTIFACT_STORE/templates \
30    AwsRegion=$REGION

```

Listing 5.14: Script for packaging and deploying nested stacks

Moreover, the script can be executed with command line input arguments. As depicted at line 1, the corresponding environment file containing the variables is loaded. If the script is executed with an argument dev, then the content of the file .env.dev will be

parsed. An environment file can look as follows, where arbitrary variables can be added if required:

```
1 export ARTIFACT_STORE=backend-mbm-tutorial
2 export CLOUDFORMATION_NAME=mbm-tutorial
3 export ENVIRONMENT=dev
4 export AWS_PROFILE=mbm
5 export REGION=eu-west-2
```

Listing 5.15: File for environment variables

The script once has loaded the file, specifies the child stacks, in this case `child_a.yaml` and `child_b.yaml`. After doing that, it packages the child stacks in the for loop and it copies them to the corresponding s3 bucket. Once completed, it does the same but for the root package, `root_template.yaml`. The root stack has to be deployed as the last one in order to avoid any problem of missing or not updated resources.

This small script, not only avoids making mistakes when packaging and deploying the AWS stack, but it also facilitates switching environment by just specifying another variable in the command line.

Chapter 6

Discussion

In chapter 3, fundamentals of serverless computing were discussed and a general description of how serverless computing is perceived was explained. Along with it, the main benefits of serverless computing were also detailed. Serverless computing is based on a novel pay-per-use model, in which costs are not accounted when the user application is not running. Thus, developers would only pay for the time which the application executes user's functions. Moreover, a serverless architecture has the ability to scale up and down according to the required workload, which typically maximizes the cost/benefit relation for most applications. Furthermore, once there is no Operating System to be managed, development costs are reduced as well, there is no need to handle updates or infrastructure maintenance.

Despite the benefits, applying such a new technology as serverless computing brings certain challenges. The key ability is to be able to scale down to zero, so that development costs are reduced when the system is idle. But as we have seen in chapter 4, cold starts can be a challenging problem which can lead to increased invocation latency. Also, limits on memory and function execution time can be challenging as well. Before opting for serverless, the scope of the application has to be known. Hence, applying serverless with Big Data use cases can, for example, bring even more complications because of the stateless nature of the serverless architecture.

There are many cloud providers which offer a lot of services. At the beginning, it might be difficult to know where and how to start (*i.e.*, how to deploy an application based on serverless computing). Therefore, one goal of the manual produced in this document is to assist developers to move and make some first steps towards serverless computing. Therefore, in chapter 5 a common architecture based on the services offered by AWS was introduced. A detailed explanation of how to deploy a serverless API endpoint followed and at each step, it was explained which AWS service was used and what it does.

Since another goal of this manual is to find a proper solution for automating a pipeline deployment, for each service, it was shown how to integrate all major steps into a template which can be reused on different cloud platforms while still following general best practices. Some challenges were brought as well, like referencing resources when dealing

with multiple templates and how to deploy different templates with just one command based on a script instead of manually upload them.

The manual itself covers important and fundamental aspects of cloud deployment. Since the intention was to write a manual, the scope of actions described in the manual was restricted to selected introductory cases, not being possible to cover all edge cases when working with other cloud providers besides AWS. Therefore, covering all details was not possible as well since the limited duration of this work.

Chapter 7

Summary and Conclusions

This work had a twofold objective. First, it analyzed what constitutes and characterizes Serverless Architectures. Second, presented a practical introduction toward the deployment of a service based on a Serverless Architecture. In this regard, it was researched which cloud providers are mostly commonly used and opted for AWS. Then, some of the main pattern, *i.e.*, microservice, services and monolithic have been explained with their advantages and disadvantages. Since there is a trend to move to microservices, the practical part was developed based on such pattern. A general architecture has been explained and it has been implemented step-wise following best practices researched in the academic and industry communities. In addition, once applications can grow exponentially, it was further detailed how to organize a CloudFormation template into multiple ones in order to support scaling applications. As observed, this is required when many resources have to be deployed. Handling different templates can be quite cumbersome and therefore a script to handle all of them, including the correct environment, has been implemented and explained.

Even though the produced manual is based on the AWS cloud provider, the best practices can all be applied to other providers as well with minor changes. The CloudFormation template might have to be adapted as well as the deployment script, but in general the main logic can be followed.

Bibliography

- [1] Zambrano, Brian: *Serverless Design Patterns and Best Practices: Build, secure, and deploy enterprise ready serverless applications with AWS to improve developer productivity*
- [2] Kief Morris: *Infrastructure as Code: Managing Servers in the Cloud*
- [3] What Is Serverless Computing?, <https://www.cloudflare.com/learning/serverless/what-is-serverless/>, accessed March 2020
- [4] AWS Serverless https://aws.amazon.com/serverless/?nc1=h_ls, accessed March 2020
- [5] The rise and rise of Cloud Computing https://www.ey.com/en_ie/technology/the-rise-and-rise-of-cloud-computing, accessed April 2020
- [6] Amazon API Gateway https://aws.amazon.com/api-gateway/?nc1=h_ls, accessed March 2020
- [7] AWS Lambda https://aws.amazon.com/lambda/?nc1=h_ls, accessed March 2020
- [8] Amazon RDS https://aws.amazon.com/rds/?nc1=h_ls, accessed March 2020
- [9] Amazon Web Services vs. Google Cloud Platform <https://www.guru99.com/google-cloud-vs-aws.html>, accessed March 2020
- [10] Installing the AWS CLI <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html>, accessed March 2020
- [11] AWS Lambda Limits <https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html>, accessed April 2020
- [12] AWS CloudFormation Limits <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cloudformation-limits.html>, accessed April 2020

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CRUD	Create, Read, Update and Delete
GCP	Google Cloud Platform
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
IT	Information Technology
RDS	Relation Database Service
VPS	Virtual Private Servers

List of Figures

5.1	Serverless API Endpoint Architecture	12
5.2	Create a new group	13
5.3	Attach a policy to a group	13
5.4	Set user details	13
5.5	Add user to a group	13
5.6	User Access Key and Secret access key	14
5.7	New created stack	16
5.8	New resources for a stack	16
5.9	Result of a successful deployment	18
5.10	Applications in AWS Lambda	18
5.11	API Endpoint and Resources of the newly created application	19
5.12	Result of successfully calling the endpoint	19
5.13	CreateUser response	22
5.14	Nested Stacks in AWS CloudFormation	27

List of Tables

Listings

4.1	Microservice pattern example	7
4.2	Service pattern example	8
4.3	Monolithic pattern example	9
5.1	Amazon Cognito User Pool	14
5.2	Definition of generic AWS Resource	14
5.3	Definition of an API Gateway	17
5.4	Declaration of Users' endpoints	21
5.5	Get a list of users in NodeJS	21
5.6	Create a new user in NodeJS	22
5.7	Create a table in DynamoDB	23
5.8	Globals Section in Template	24
5.9	Get a single user or a list of users from DynamoDB	24
5.10	Create/Insert a user in DynamoDB	25
5.11	Reference resources as environment variables	27
5.12	Export a Function's ARN in the Outputs section	28
5.13	Cross stack referencing in the parent template	29
5.14	Script for packaging and deploying nested stacks	29
5.15	File for environment variables	30