Hoang Le Minh

Migration project with Serverless Framework and Amazon Web Services API

Technology and Communication
2020

ACKNOWLEDGMENT

First of all, I would like to show my respect and gratefulness toward Mr. Timo Kankaanpää, my supervisor, teacher, ex-employer and a great supporter to my career path during the time I was studying in VAMK. The strength and motivation I have today were built up from his lessons and encouragement.

I would like to thank my employers for having me in the team and trusting me enough with this big project decision, especially the CTO, he gave me instructions and helped me to organize the thesis work. Also, I want to thank my colleague for helping me with the work.

Finally, to all my classmates at VAMK, family and friends in Vietnam, thank you all for supporting my decisions.

Wish the best to all of you.

Vaasa, 26.04.2020

Hoang Minh Le

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Hoang Le Minh
Title	Migrating a Serverless Framework API with Amazon Web Services
Year	2020
Language	English
Pages	58
Name of Supervisor	Timo Kankaanpää

In the era of the Internet, it is a must for every company, small or big, to have a website for people to visit and connect with them. Therefore, the demand for quality applications and e-services is strongly increasing. As a result, there have been many different technologies to help to build API services different from the traditional way depending on the scenarios and capabilities. The Serverless framework is one in those that have been insights for many developers and companies which are both new and old. This thesis focuses on the main aspects of Serverless technology and why it works well and not well with Amazon Web Services based on the benefits it brought and the problems that occurred while developing the product.

This thesis is built based on a project in a start-up company where I have been working for. The main purpose of it was to give an understanding of the architect of an API using the Serverless framework and Amazon Web Services through solving a scalability problem that occurred during the development process.

Keywords e-commerce, AWS, Serverless, API

CONTENTS

ACKNOWLEDGMENT
1 INTRODUCTION 10
Server & Serverless Application 10
Migration Project 10
2 BACKGROUND OF THE PROJECT 12
2.1 Serverless Framework 12
2.2 Amazon Web Services 14
2.2.1 Amazon Cognito User Pools and Identity Pools 16
2.2.2 Amazon DynamoDB 16
2.2.3 Amazon Lambda 17
2.2.4 Amazon API Gateway and Amazon Route 53 17
2.3 MongoDB 18
2.4 Docker, Jenkins & Automation Testing 18
3 STRUCTURE DESIGN 20
3.1 Main Object 20
3.2 Infrastructure 21
3.3 Package Diagram 22
3.4 Index.js 23
3.5 Execution sequences 25
4 IMPLEMENTATION 26
4.1 Serverless.yml 26
4.2 Lambda function configuration 27
4.3 Authentication 28
4.4 Database queries 30
5 MIGRATION PROJECT 32
5.1 The reason 32
5.2 Solutions 33
5.3 MongoDB Atlas 36
6 MIGRATION AND TESTING 38
6.1 Setup Connection 38

6.1.1 MongoDB Initialization and setup VPC connection 38
6.1.2 Connect/Disconnect function MongoDB 39
6.1.3 Schema definition 40
6.2 Querying 42
6.3 Compatibility 42
6.3.1 Connections management 42
6.3.2 Indexes 43
6.4 Testing and Deploying 43
6.4.1 Unit Testing 44
6.4.2 Integration Tests 44
6.4.3 Automation Testing 46
6.4.4 Deployment 51
7 CONCLUSION 56
REFERENCES 57

LIST OF FIGURES

Figure 1: Differences between software services 13
Figure 2: The diagram indicates which layer FaaS's customers will cover 14
Figure 3: Example Services that Amazon will provide (taken from AWS Console) 15
Figure 4: How AWS Lambda works 17
Figure 5: AWS's architecture 21
Figure 6: Package structure of the project 22
Figure 7: The structure of the Lambda function in JavaScript 23
Figure 8: An example of the Event object in Lambda function 23
Figure 9: Sequences Diagram 25
Figure 10: Example of the Serverless.yml with environment variables 26
Figure 11: Configuration of the API path in Serverless.yml 27
Figure 12: Example of extracting body from a POST request to /notes then create a record in DynamoDB 27
Figure 13: Example of generating App client id for notes-user-pool User Pool.. 29
Figure 14: Login and get authorization token from User Pool 29
Figure 15: Simply getting one note from DynamoDB 30
Figure 16: Grant access to DynamoDB inside Serverless.yml 30
Figure 17: How Item size is calculated on DynamoDB (source: AWS DynamoDB Documentation) 33
Figure 18: MongoDB 36
Figure 19: MongoDB Production URI setup in Serverless.yml as an environment variable 38
Figure 20: Setup a VPC connection to Atlas MongoDB 39
Figure 21: Connect to MongoDB via a URI 40
Figure 22: Example of creating a MongoDB Schema with validations 41
Figure 23: Example of a simple get one item with option query with a model named "Adventure" 42
Figure 24: Deploy to production 43
Figure 25: Example testing indexOf() function 44
Figure 26: Example of sequence tests 45

Figure 27: Jenkins EC2 Instance 47
Figure 28: Docker installed on an EC2 Instance 47
Figure 29: The container is up and running 48
Figure 30: Jenkins home page 48
Figure 31: Add Github account token to Jenkins configuration 49
Figure 32: Pipeline configuration for Git project 49
Figure 33: List of actions in Jenkinsfile 50
Figure 34: An example of a successful build 51
Figure 35: The result after the deployment 52
Figure 36: API Gateway Custom domain names 53
Figure 37: The API Gateway record is connected with the testing API Lambda, which is deployed on the previous section 54
Figure 38: Route53's record set for the API Gateway record API 55

LIST OF TABLES

Table 1: List of required attributes 20
Table 2: Analysis of different solutions.. 35

LIST OF ABBREVIATIONS

API	Application Programming Interface
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
AWS	Amazon Web Services
PaaS	Platform-as-a-Service
FaaS	Function-as-a-Service
YAML	Yet Another Markup Language
	YAML Ain't Markup Language
VPC	Virtual Private Cloud
URI	Uniform Resource Identifier
URL	Uniform Resouce Locator
CPU	Central Processing Unit
RAM	Random-access Memory
SSH	Secure Shell

1 INTRODUCTION

Server & Serverless Application

Traditionally, companies who want to build their websites or applications have to rent, manage and maintain the server. However, hiring a good system development team is costly, not to mention it could still create pressure for both the development team and business aspect when it comes to incidents and accidents happened which are related to the operational aspect. Another factor is that when the team decided to use a new technology/framework but a change in the current system is needed, a transition or migration process would be a difficult decision. These problems are even bigger for small companies such as start-ups because their resources are limited.

With the innovation of cloud services, more opportunities have been offered for developers. They are a variety of choosing trust-worthy providers, infrastructure to even software-based service, giving them the chance to work specifically on their targets without worrying about scalability and reliability.

Serverless is on top of that. It is a "function-as-a-service" architect that enables developers to shift more of the operational responsibilities to cloud providers and focus more on the delivered products. Serverless applications are light-weight applications that only include the core logic without overhead servers or runtimes configurations but still greatly scalable.

Migration Project

The backbone of the project has already been done which is used by a few customers. However, there are lots more features and customizations depends on the strategy and needs of each customer with their products, therefore, the project was still far from the goal and everyone has to understand the concept from scratch. During the development process, we had encountered a scalability problem that could affect the original architect.

Overall, this thesis work is to take responsibility for the migration of the database and make sure it never happens again by making integration and automation testing.

This project is a service API built on the Serverless framework on Node.js runtime environment provided by Amazon Web Service.

2 BACKGROUND OF THE PROJECT

2.1 Serverless Framework

The Project is built on the Serverless framework, one of the latest fast-growing technology.

The word “Serverless” means “the smaller contribution of the Server”. Traditionally, web applications are hosted on web servers. To keep the application up and running, the Operating System of the host machine has to be always updated and patching during that time to up downloading, compiling and connecting all sorts of components. This is time and power-consuming since every website needs a web server and not every second the server gets a request but they have to always be ready to prepare for it. Besides, managing servers is a complicated task since it requires dedicated and experienced engineers.

Lately, when the network and the internet have been improved significantly (speed, latency, etc), together with the invention of virtual machines they have brought the cloud computing industry closer to the users. Therefore, instead of having their hardware to customize the configuration and serve their websites, they can rent a machine somewhere on the internet and just focus on developing the product without worrying about the maintenance. Those who are giving the service are called Infrastructure-as-a-service (IaaS) and Platform-as-a-service (PaaS) operators.

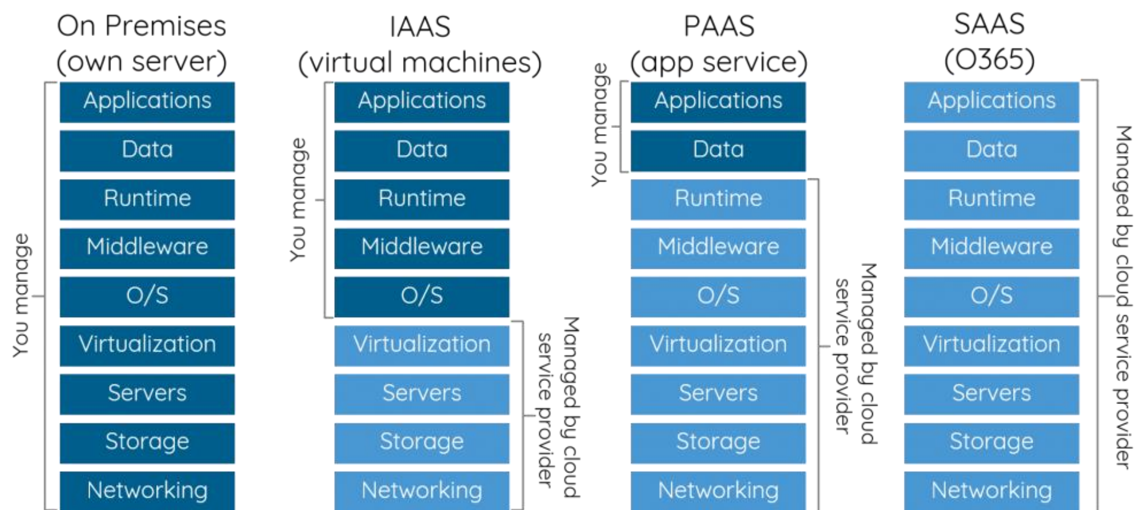

Figure 1: Differences between software services

However, there is still a more powerful and convenient way for developers to manage their projects.

"The Serverless Framework helps you build serverless apps with radically less overhead and cost. It provides a powerful, unified experience to develop, deploy, test, secure and monitor your serverless applications." - **Serverless, Inc. © 2019**

Without having to build things from scratch, or concern about cost and maintenance, Serverless is a Function-as-a-Service that can connect developer's code with existing functional components provided from the cloud, resulting in a loosely coupled, efficient and scalable application quickly but less complexity.

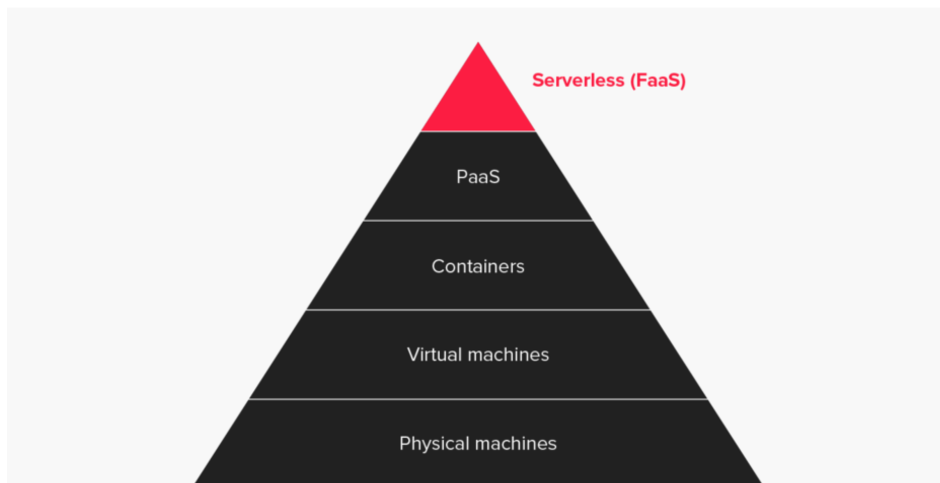

Figure 2: The diagram indicates which layer FaaS's customers will cover

Currently, there are three biggest and most popular providers: Microsoft Azure with Azure Functions, Google Cloud Platform with Google Cloud Functions and Amazon Lambda Functions from Amazon Web Services. There are only some minor differences among them but the values that they share are nearly the same. But for the scope of the project, Amazon is the provider for the application cloud solution because of the huge benefits that Amazon Web Services could bring.

2.2 Amazon Web Services

Amazon Web Services (AWS) are Cloud Computing Services provided by Amazon, one of the biggest technology companies in cloud computing, also in e-commerce and artificial intelligence. They have led the market of cloud computing services a long time ago before the two greatest opponents decided to jump in after witnessing the potential of this era. Even though both Microsoft and Google have proved their names on the market, Amazon is still stabilized as the largest shares by owning almost half of the world's public cloud infrastructure market. Being the biggest and oldest age but remaining strong and powerful, Amazon Web Services ecosystem is the perfect choice for the project.

Amazon Web Services provide all tools and technologies needed for developer's applications, for example, power computing, database storage and content delivery, in a secure container that can be accessed anywhere. Additionally, it is also easy to use any services with powerful AWS SDK provided by Amazon that is available in any language.

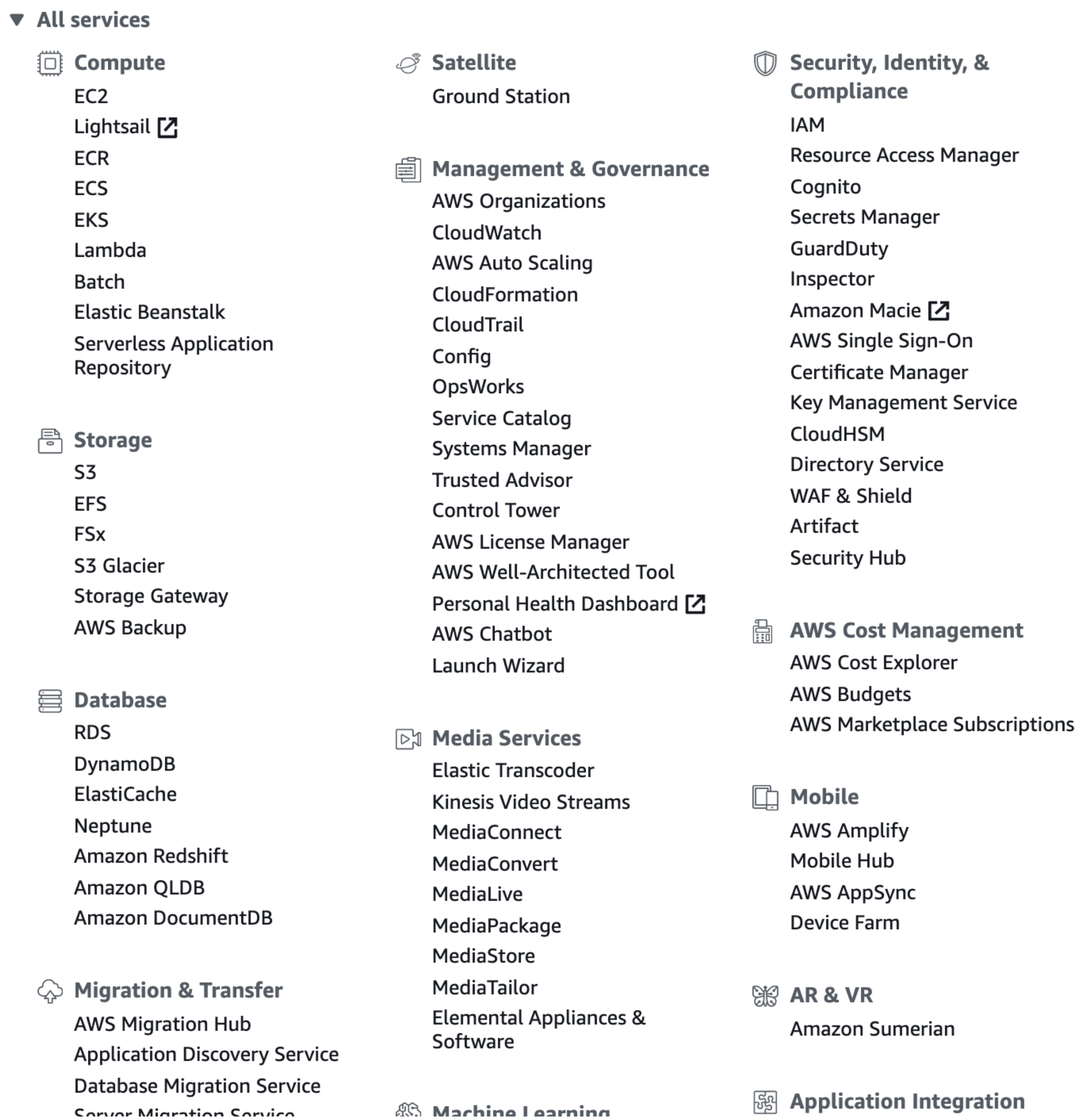

Figure 3: Example Services that Amazon will provide (taken from AWS Console)

It depends on the project that developers can decide which services they want to use for their application. For an application that requires user data, it is a must that storing and authorizing them by users are available and secured, so as a result Amazon Cognito services and Amazon DynamoDB are used in this project.

2.2.1 Amazon Cognito User Pools and Identity Pools

It is common today that applications are authenticated with user name and password. Therefore, Amazon Cognito provides a fully basic functional authentication process for the application with storing users' data, generating sign-up, sign-in, activating an account, forgetting a password, etc. In addition, every successful login will generate a token to authorize every request that called on the behavior of that user. This eased the work since the API does not require any extra features out of this authentication process and implementing it into the project was a simple job.

The other service is called Identity Pools. These services provided AWS credentials for other developers to access the same AWS resources. Since our developers are working as a team, it is convenient that we all can access the console. The API does not require this feature but it optimized the development progress by saving developing time.

2.2.2 Amazon DynamoDB

For the database solution, we decided to save our users' data in Amazon DynamoDB. It is a NoSQL database that supports the key-value structure and document data models. To run high performance, internet-scale applications, DynamoDB overburdens traditional relational databases, enables developers to build fast, modern serverless applications that can start small but are globally scalable.

With the power of handling more than 10 trillion requests per day and 20 million requests per second, it seems to be a good match with the Serverless application. With the fact that data schema is a single document key-value structure, it leaves DynamoDB as a perfect option for the project.

2.2.3 Amazon Lambda

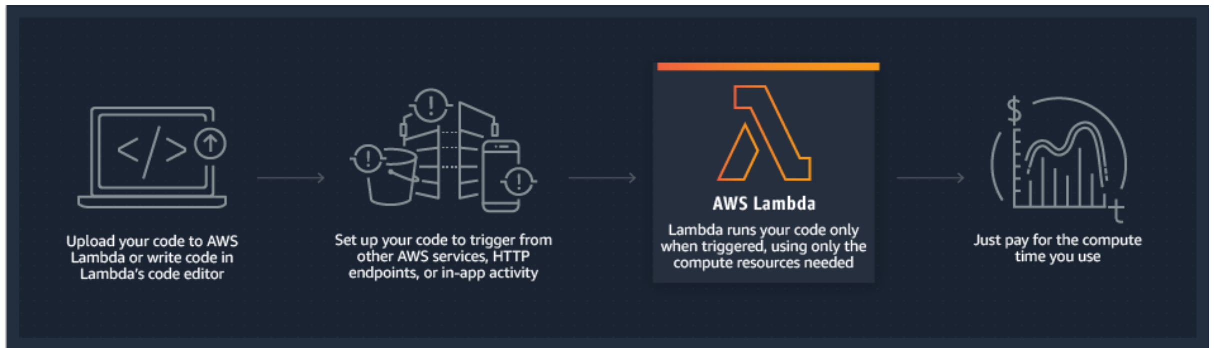

Figure 4: How AWS Lambda works

This is what makes the application alive. Lambda is one in AWS tools to help to build Serverless applications. Lambda lets developers run code without having to manage servers, all they have to do is just to focus on developing their features. Each request to Lambda is triggered by Lambda functions which can be written in any common programming language easily.

By using Lambda for your Serverless applications, developers only have to pay for the functions that they created, which is different from the traditional server hosting method that needs to be paid even the idle time. Therefore, it helps individuals and small development teams cut costs and save resources.

2.2.4 Amazon API Gateway and Amazon Route 53

To protect the source code and data of our APIs from being directly accessed without knowledge, Amazon API Gateway is used to act as a front door to applications to access Amazon services. It manages and handles all the tasks that require accepting and processing API calls for the applications.

Finally, Amazon also provides a Domain Name System (DNS) service called Route 53 that lets the user create, maintain and connect custom domains to their application endpoints.

2.3 MongoDB

Same with DynamoDB, MongoDB is a NoSQL, document-based database that is fast and scalable. The difference is that MongoDB has validated Schema for each table (Collection) it has that can control the data structure while DynamoDB requires users to define their own. MongoDB can be installed anywhere and any place easily with simple setup processes.

For cloud solutions, serverless applications can directly connect to MongoDB Atlas database as a service, available on all three popular cloud providers: Microsoft's Azure, Amazon Web Services, and Google Cloud Platform.

By allowing the maximum document size of 16MB and rich but simple and fast query types, MongoDB provides a large scale of data storing that eases the system design for scalable applications.

2.4 Docker, Jenkins & Automation Testing

In DevOps, automation is the key principle. To make sure the API performs only better after each change, the team had to not only make the new tests for the new features but also had to make sure that all the old tests are passed. Manually testing is an easy task but it is time-consuming and has low-efficiency, not to mention the risks of creating wrong or missing tests are high in practice, which leads to the result of not all tests being covered. To avoid building that kind of a fragile system, an automatic and continuous testing process run after each change seems to be a safe and optimized solution.

Jenkins is an open-source automation server that enables automatically tests running and deploying applications by being triggered by any Commit of any version control system; or by scheduling a run via a cron job, which made it become the standard of automation testing in DevOps. Jenkins can be added with plugins to extend the functionality.

To ease the process of installing Jenkins and to keep Jenkins always running, Docker will be used. Docker creates an environment for developers to develop apps

regardless of languages, frameworks, architectures, and versions of the tools. A Docker container can be created following a subset of configurations for a specific purpose, which is a Docker Image. In the project, Jenkins Docker Image was used to install Jenkins on a Docker container.

3 STRUCTURE DESIGN

3.1 Main Object

With the target of keeping the API fast and simple to implement, data is stored inside the database is non-relationally, it is just a list of items with attributes.

An Item represents an object. Under the scope of this project, a Note will be an Item during this and the upcoming sections.

Name	Type	Description	Example
id	UUID	The unique string indicates a note among others in the databases	8d9f3ca7-4e97-4405-941c-2952117b5a69
title	String	The name of the Item (note)	“New Note”
createdBy	UUID	The unique string indicates the user created the note among others	“3x2f3ca74e972605941c2954337b5b13”
created	Timestamp	The time that the note is created	1542785399846
description	String	The content of the note	“Get 2 eggs instead”
noteHistory	Array	List of note details that had been changed. Each element contains the time and description that were changed during that time:	[{ “time”: 1542785399846, “description”: “Buy an egg”, }]

Table 1: List of required attributes

3.2 Infrastructure

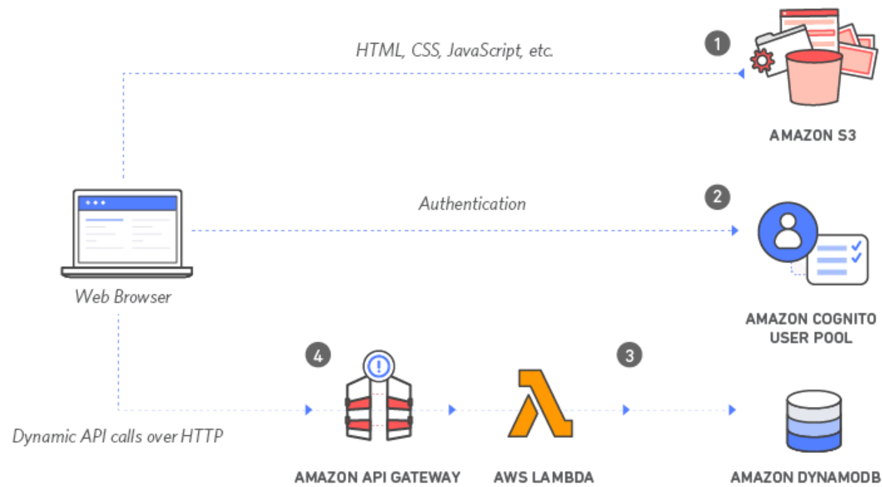

Figure 5: AWS's architecture

Figure 6 describes how Amazon Web Services communicates in the application. Our development client-side files (HTML, JavaScript, and CSS files) are stored in Amazon S3 (1). Users will authenticate through AWS Cognito User Pool services to able to get the token (2). After that, every API request will use that access token, then being sent to Lambda service (3) through Amazon API Gateway (4) to be able to perform suitable queries syntax to DynamoDB.

3.3 Package Diagram

Users will have options to get, create, edit or delete the notes from the API. Authentication is also needed.

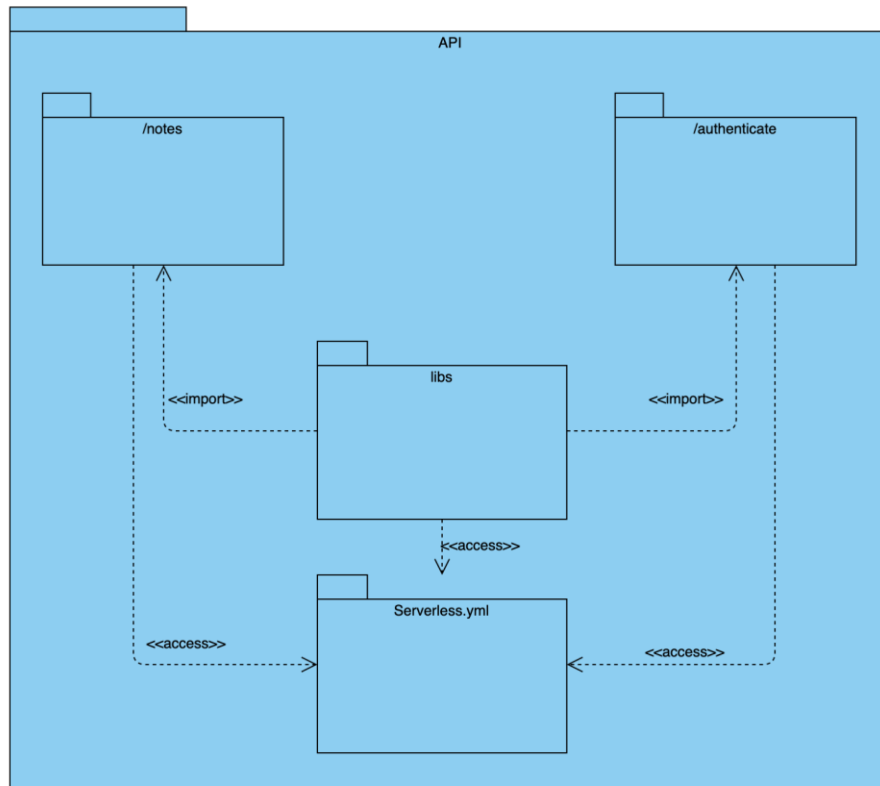

Figure 6: Package structure of the project

The application backbone is built on a single YAML file (.yml) that normally named Serverless.yml. It contains configurations for Lambda service for the API such as provider, engine runtime, environment variables, execution time, etc.

The code for the API will be handled inside *index.js*. This is the main event for everything happens to the API: request body, headers, query-string, path parameters, etc. inside an object called *Event*. Routes are also handled in *index.js* for serving different purposes, such as:

- */authenticate*: for authentication.
- */notes*: handle notes data.

Each route has a different way of extracting data from the Event object, but sometimes they share common functionality. To fulfill the condition, useful functions are stored inside a separated area, which is inside *libs*. *Libs* can also import global variables listed in *Serverless.yml*.

3.4 Index.js

Example index.js File – HTTP Request with Callback

```
const https = require('https')
let url = "https://docs.aws.amazon.com/lambda/latest/dg/welcome.html"

exports.handler =  function(event, context, callback) {
  https.get(url, (res) => {
    callback(null, res.statusCode)
  }).on('error', (e) => {
    callback(Error(e))
  })
}
```

Figure 7: The structure of the Lambda function in JavaScript

A Lambda function requires three parameters by orders to be able to perform the computing:

- *Event* (object): this JSON object contains information about the request that got sent to the API, including the request headers and body. All information we need for the application to work is available in this object.

Example Event from an Application Load Balancer


```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-2:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-2.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": false
}
```


Figure 8: An example of the Event object in Lambda function

From the JSON object above, it is clear that this is an HTTP GET request to https://lambda-alb-123578498.us-east-2.elb.amazonaws.com/lambda route with query string parameter "query" has the value of "1234ABCD". This is a GET request so the request body is empty, which is understandable. In addition, we can also see extra data such as which browser the sender was using or other AWS resource name (ARN).

- *Context* (object): this object gets passed into the handle before the Lambda service executes. It contains both methods and properties to help define the invocation, function and execution environment.
- *Callback (function)*: an asynchronous function acts as an exit point for the Lambda function.

3.5 Execution sequences

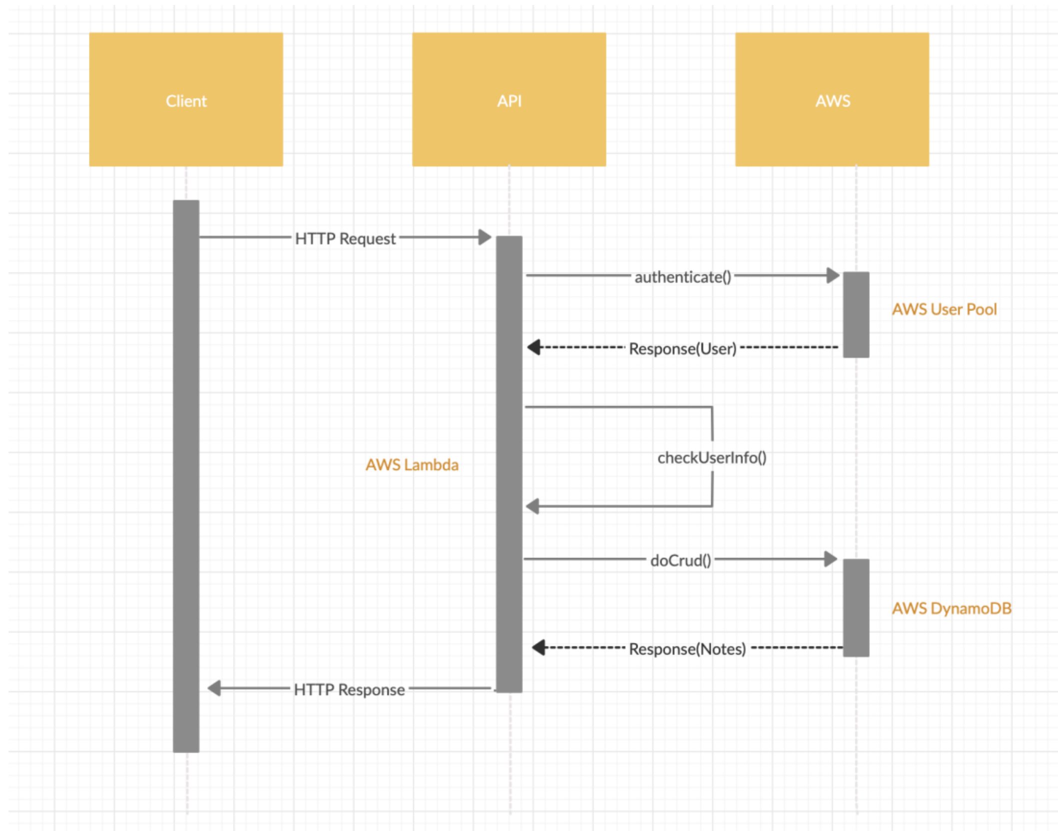

Figure 9: Sequences Diagram

The diagram indicates the process when an API call is performed. Firstly, the request is passed to the API resource through Amazon API Gateway, where it then be validated for the sakes of security and monitoring purposes later on. After reaching the resource, the Lambda function which takes the request data gets triggered. It then connects to the AWS User Pool to retrieve the correct user data according to the request object. When the previous process is completed, it will start to fetch the note data from DynamoDB according to the user information. When the data is settled, it gets saved to the database and sent back to the client inside the response object.

4 IMPLEMENTATION

4.1 Serverless.yml

Serverless.yml is the unique YAML file that acts as the backbone of the API and consists of variables and configurations for the Lambda computing services. As it can be seen from the example, it is clear that developers can understand what is needed for the code execution, such as the version of the Serverless framework, provider, runtime environment, etc. Global variables are defined under provider/environments so they could be imported in project files.

```yaml
# serverless.yml

service:
  name: myService
  awsKmsKeyArn: arn:aws:kms:us-east-1:XXXXXX:key/some-hash # Optional KMS key arn which will be used for encryption for all

frameworkVersion: '>=1.0.0 <2.0.0'

provider:
  name: aws
  runtime: nodejs12.x
  stage: ${opt:stage, 'dev'} # Set the default stage used. Default is dev
  region: ${opt:region, 'us-east-1'} # Overwrite the default region used. Default is us-east-1
  stackName: custom-stack-name # Use a custom name for the CloudFormation stack
  apiName: custom-api-name # Use a custom name for the API Gateway API
  profile: production # The default profile to use with this service
  memorySize: 512 # Overwrite the default memory size. Default is 1024
  reservedConcurrency: 5 # optional, Overwrite the default reserved concurrency limit. By default, AWS uses account concurre
  timeout: 10 # The default is 6 seconds. Note: API Gateway current maximum is 30 seconds
  logRetentionInDays: 14 # Set the default RetentionInDays for a CloudWatch LogGroup

  logs:
    restApi: # Optional configuration which specifies if API Gateway logs are used. This can either be set to `true` to use
      accessLogging: true # Optional configuration which enables or disables access logging. Defaults to true.
      format: 'requestId: $context.requestId' # Optional configuration which specifies the log format to use for access logg
      executionLogging: true # Optional configuration which enables or disables execution logging. Defaults to true.
      level: INFO # Optional configuration which specifies the log level to use for execution logging. May be set to either
      fullExecutionData: true # Optional configuration which specifies whether or not to log full requests/responses for exe
      role: arn:aws:iam::123456:role # Existing IAM role for ApiGateway to use when managing CloudWatch Logs. If 'role' is n
      roleManagedExternally: false # Specifies whether the ApiGateway CloudWatch Logs role setting is not managed by Serverl
    websocket: # Optional configuration which specifies if Websocket logs are used. This can either be set to `true` to use
      level: INFO # Optional configuration which specifies the log level to use for execution logging. May be set to either
    httpApi: # Optional configuration which specifies if HTTP API logs are used. This can either be set to `true` (to use de
      format: '{ "requestId":"$context.requestId", "ip": "$context.identity.sourceIp", "requestTime":"$context.requestTime",

    frameworkLambda: true # Optional, whether to write CloudWatch logs for custom resource lambdas as added by the framework

package: # Optional deployment packaging configuration
  include: # Specify the directories and files which should be included in the deployment package
    - src/**
    - handler.js       You, a minute ago • Uncommitted changes
  exclude: # Specify the directories and files which should be excluded in the deployment package
```

Figure 10: Example of the Serverless.yml with environment variables

Serverless.yml is also used to define paths for the API. Figure 12 indicates the structure of an API path.

```yaml
functions:
  usersCreate: # A Function
    handler: users.create # The file and module for this specific function.
    name: ${self:provider.stage}-lambdaName # optional, Deployed Lambda name
    description: My function # The description of your function.
    memorySize: 512 # memorySize for this specific function.
    events: # The Events that trigger this Function
      - http: # This creates an API Gateway HTTP endpoint which can be used to trigger this function.  Learn more in "events/apigateway"
          path: users/create # Path for this endpoint
          method: get # HTTP method for this endpoint
          cors: true # Turn on CORS for this endpoint, but don't forget to return the right header in your response
          private: true # Requires clients to add API keys values in the `x-api-key` header of their request
          authorizer: # An AWS Cognito User Pool authorizer
            arn: arn:aws:cognito-idp:AWS_REGION-1:ID:userpool/USERPOOL_ID
```

Figure 11: Configuration of the API path in Serverless.yml

The path */users/create* of this API is configured to have the handler is *users.create,* can be connected through API Gateway HTTP endpoint with GET method. This HTTP endpoint of the path also has an *authorization* step that takes the resources from AWS Cognito UserPool ARN (Amazon Resource Name).

4.2 Lambda function configuration

The handler of an API path is a function that will be triggered when a request got sent to the API path, as shown in Figure 13.

```javascript
import uuid from "uuid";
import * as dynamoDbLib from "./../libs/dynamodb-lib";
import { success, failure } from "./../libs/response-lib";

exports.handler = async function (event, context, callback) {
  const data = JSON.parse(event.body);
  const params = {
    TableName: "notes",
    Item: {
      id: data.id ? data.id : uuid(),
      title: data.title,
      description: data.description,
      createdBy: event.requestContext.identity.cognitoIdentityId,
    }
  };

  context.callbackWaitsForEmptyEventLoop = false

  try {
    await dynamoDbLib.call("put", params);
    return callback(null, success(params.Item));
  } catch (e) {
    return callback(null, failure({ status: false }));
  }
}
```

Figure 12: Example of extracting body from a POST request to /notes then create a record in DynamoDB

In the handler, a request to path */notes/create* should create a user on the databases and return the result. It can be seen that from the Event object, we can retrieve request body data (line 6). The data then was used to make database queries (DynamoDB *put* syntax to create a record*)* and the response was sent in the *callback().*

On the other hand, the Context object has the configuration for the Lambda services, as mentioned in section **3.4 index.js** to show the basic structure of a Lambda function. In the project, the most obvious usage of this object is to retrieve the Request ID if the function passes through the Lambda compute service, and one special configuration **context.callbackWaitsForEmptyEventLoop** (line 19), which simply just terminates all the event loops that are left in the Event object and send the response immediately when it is set to *false* (default is *true*).

In the handler, the *callback* is the third argument, which is a callback function that gets called whenever the Lambda function needs to stop executing. It can be that the data is fully retrieved and ready to be sent as a response, or an error has occurred and the process needs to be stopped and sent the error message to the client.

4.3 Authentication

We already have Cognito where user data is stored. But to be able to connect to the resources from the application, a tunnel needs to be created. To do so, a special token will be needed for Cognito service to recognize the origin of the data source. In this case, the *clientId* of the User Pool is used.

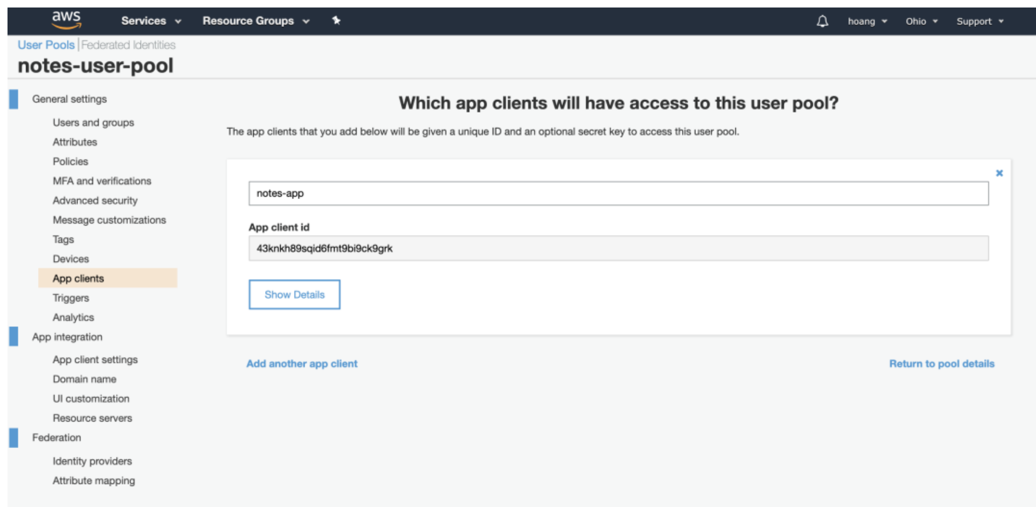

Figure 13: Example of generating App client id for notes-user-pool User Pool

After getting the *clientId* successfully, it is possible to start creating a connection. Amazon has a special development kit for JavaScript developers to achieve the purpose: *amazon-cognito-identity-js*. Simply just adding configuration for both sides will do the job. There are many features but under the scope of the project, log in and sign up have already fulfilled the needs.

```
module.exports = ({ credentials, callback, success, error }) => {
    const auth = new AWSCognito.AuthenticationDetails({
        Username: credentials.email,
        Password: credentials.password
    });

    const user = new AWSCognito.CognitoUser({
        Username: credentials.email,
        Pool: userPool
    });

    user.authenticateUser(auth, {
        onSuccess: result => {
            const token = result.getIdToken().getJwtToken();
            callback(null, success({ token }));
        },
        onFailure: err => {
            callback(null, error({ err }));
        }
    });
}
```

Figure 14: Login and get authorization token from User Pool

4.4 Database queries

```
const params = {
  TableName: "notes",
  // 'KeyConditionExpression' defines the condition for the query
  // - 'userId = :userId': only return items with matching 'userId'
  //   partition key
  // 'ExpressionAttributeValues' defines the value in the condition
  // - ':userId': defines 'userId' to be Identity Pool identity id
  //   of the authenticated user,
  KeyConditionExpression: "createdBy = :createdBy",
  ExpressionAttributeValues: {
    ":createdBy": event.requestContext.identity.cognitoIdentityId
  }
};

try {
  const result = await dynamoDbLib.call("query", params);
  return success([result.Items, stats]);
} catch (e) {
  return failure({ status: false });
}
```

Figure 15: Simply getting one note from DynamoDB

DynamoDB *Query* finds items base on primary key values and in this case, is *createdBy* attribute. The items returned will all contain *createdBy* equals to the *cognitoIdentityId (userId)* provided in the *requestContext* of the *event* object.

After creating the table on the AWS Console, the same idea with Cognito, we need a client app interface and a tunnel to connect to the resource. It can be easily done by *serverless-dynamodb-client* npm package, and the resource is authorized by creating a record of access granting inside the backbone *Serverless.yml.*

```
# Grant access to the Lambda to the DynamoDB table
- Effect: Allow
  Action:
    - dynamodb:Query
    - dynamodb:Scan
    - dynamodb:GetItem
    - dynamodb:PutItem
    - dynamodb:UpdateItem
    - dynamodb:DeleteItem
  Resource: "arn:aws:dynamodb:${opt:region, self:provider.region}
# Grant access to the Lambda to the DynamoDB table index
- Effect: Allow
  Action:
    - dynamodb:Query
  Resource: "arn:aws:dynamodb:${opt:region, self:provider.region}
```

Figure 16: Grant access to DynamoDB inside Serverless.yml

As it is described, the database actions have to be declared for each data querying syntax.

5 MIGRATION PROJECT

5.1 The reason

As the note keeps being edited, the *noteHistory* attribute increases in size (see 3.1). This will happen for several reasons such as being frequently changed or a huge note description per change. At a certain point in time, it will result in an error that stated: “***DynamoDB: Item size has exceeded the maximum allowed size***”.

“An item is the core unit of data in DynamoDB. It is comparable to a row in a relational database, a document in MongoDB, or a simple object in a programming language. Each item is uniquely identifiable by a primary key that is set on a table level. An item is composed of attributes, which are bits of data on the item. This could be the "Name" for a User, or the "Year" for a Car. Attributes have types -- e.g., strings, numbers, lists, sets, etc -- which must be provided when writing and querying Items.”

- dynamodbguide.com/anatomy-of-an-item

An Item contains a Note data which is stored inside the DynamoDB. By following this idea, it is a big chance that the problem was caused by DynamoDB itself. The flow of the data was fulfilled and the error was caught with the message. It was also found out that the Get Items query to DynamoDB was responsible for this error message.

After looking through either on the documentation or Q&A forums, there were similar problems posted, and one quote from the documentation was mentioned more than 70% of those cases:

“The maximum item size in DynamoDB is 400 KB, which includes both attribute name binary length (UTF-8 length) and attribute value lengths (again binary length). The attribute name counts towards the size limit.”

- docs.aws.amazon.com/amazondynamodb

The size of the data was never being one consideration during development. But it surely gave the team a sight of what happened. Data can be stored on DynamoDB at any type because it is schemaless and here is how it is users can estimate the Item size with the formula from DynamoDB Developer's guide.

DynamoDB Item Sizes and Formats

PDF | Kindle | RSS

DynamoDB tables are schemaless, except for the primary key, so the items in a table can all have different attributes, sizes, and data types.

The total size of an item is the sum of the lengths of its attribute names and values. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is *(length of attribute name) + (number of UTF-8-encoded bytes)*.
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed. The size of a number is approximately *(length of attribute name) + (1 byte per two significant digits) + (1 byte)*.
- A binary value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size. The size of a binary attribute is *(length of attribute name) + (number of raw bytes)*.
- The size of a null attribute or a Boolean attribute is *(length of attribute name) + (1 byte)*.
- An attribute of type `List` or `Map` requires 3 bytes of overhead, regardless of its contents. The size of a `List` or `Map` is *(length of attribute name) + sum (size of nested elements) + (3 bytes)* . The size of an empty `List` or `Map` is *(length of attribute name) + (3 bytes)*.

Figure 17: How Item size is calculated on DynamoDB (source: AWS DynamoDB Documentation)

For that calculation, approximately a note with 400000 characters can be created without any further changes or a note with 2000 characters can be changed 200 times. The mission is set out to be able to exceed this boundary for this project.

5.2 Solutions

The problem could be solved if the Item size, or to be more specific, *noteHistory*, could be reduced.

The attribute *noteHistory* stores all the updated note data from the beginning so the customer can have all the information to perform analytics purposes such as undo or check history.

In the real case, having missed one or many histories can make the data-collecting work difficult. For that reason, a solution which can keep all history is necessary.

After gathering information, a table of solutions was formed.

Solution	Description
Using compressed method	The idea would be using compressing JavaScript libraries such as zlib or gz since DynamoDB can store any kind of data. The size then could be reduced more than 2/3, leaving the customers being able to have a longer note history list. However, the drawback is the time and resource-consuming for compressing/decompressing data every request. **Possibility: 6/9** **Practicality: 5/9**
Restrict the input	This is the most simple solution and easiest way to solve the case without any breaking changes by providing a validation that could calculate and reject a request that exceeds the maximum item size. However, this would create a bad user experience for every end-user such as they can not having more than this number of characters or change the note more than a specific time. **Possibility: 8/9** **Practicality: 3/9**
Make a separate table for storing the history in the DynamoDB	Since we are using a table to store all the notes as the items and the problem is the array of note history could exceed the limit, we could just make a different table called NoteHistory to store only the note history data, connected by the Note ID, which makes the item size only around 50B. The drawback of this the fact that the Note Data table can have billions of rows since all the user's note data is also stored there. Moreover, since the NoSQL database is being used so there is no relation between the table that corresponding data has to be fetched manually, which is efficiency wasted and time-consuming, especially when are using AWS Lambda Serverless where the services are paid by the usage. **Possibility: 6/9** **Practicality: 4/9**
Using a different databases solution	If the database is the problem, then it needs to be changed. There are many options nowadays, various from SQL to NoSQL, even real-time Databases could be a possible solution. But the first came to the mind while thinking about migrating to a new database is what breaking changes it could make,

	how much time and efforts it needs to take because that could affect the road map of the whole project, also the budget plan for the migration also needs to be considered. Firstly, SQL databases would not be used since the incompatibility with the old project structure. The data still is a simple key-value object type and each Item is referenced by a user from AWS Cognito service so there is no need for extra tables at the moment. As a result, this leaves the project with options of NoSQL and Real-time Databases such as Redis, MongoDB, DocumentDB, Firebase, etc. Secondly, the maximum item size has to be calculated carefully for the performance and real-world situations coverage. **Possibility: 5** **Practicality: 8**

Table 2: Analysis of different solutions

As can be seen from the results, the last solution (using different databases solution) seems to be the safest and the most efficient way, even though it would require lots of researches and discussions, but it would be worth the effort.

5.3 MongoDB Atlas

Figure 18: MongoDB

After gathering information about some of the NoSQL Databases available such as MongoDB, DocumentDB, Redis, or even Google Firebase, some estimations of the pros and cons of each type of database was made in different paces of migration: implementation, compatibility, differences, and pricing. MongoDB then stood out and became the priority.

"MongoDB is a document database designed for ease of development and scaling. The Manual introduces key concepts in MongoDB, presents the query language, and provides operational and administrative considerations and procedures as well as a comprehensive reference section."

- https://docs.mongodb.com/manual/introduction/

Reasons for the decision:

- It is a NoSQL database with schema definition, which restricts the input before actually putting data inside the database while DynamoDB lacks this feature.
- Both databases share lots of commons in configuration and data manipulating since they are NoSQL databases and needed with support libraries.
- MongoDB is also hosted on AWS server to avoid latency and complicated authentication process when it comes to accessing resources outside of AWS server
- npm package Mongoose.js fully support connecting, querying and casting to MongoDB from Node.js runtime server/service

- It is one of the most popular NoSQL databases with huge supports from the community.
- Same to other NoSQL databases, its speed has been proven for such heavy jobs with simple data.
- Data has version control that can be kept track.
- Max item size support: 16MB. This is one of the main reasons for the decision since it is 40 times more than the limit that DynamoDB allows.
- The maximum query size is 16MB. Compare to 1MB of DynamoDB, it allows the query to retrieve much more data per time.
- More query options. For example, there is currently no update query for DynamoDB, which leads to having to use data over-writing. But all four basic actions create, read, update and delete are available officially for MongoDB.
- MongoDB usage is paid monthly instead of paid by the demands of DynamoDB. By simple estimation from previous months, the tire M10 of MongoDB's Atlas which is enough for the needs only costs 2/3 of what we had spent on DynamoDB (it could be much higher due to incoming demands). Future upgrading is possible without losing data.

6 MIGRATION AND TESTING

The migration process includes:

- Create database's Instance and set up connections and configurations
- Change database queries and response handling
- Testing
- Documentation and Deployment

Because the current version is still being used by many customers, a brand new version of the API should be published along with the current one.

6.1 Setup Connection

6.1.1 MongoDB Initialization and setup VPC connection

```
16  provider:
17    name: aws
18    runtime: nodejs8.10
19    stage: prod
20    region: us-east-2
21    # To load environment variables externally
22    # rename env.example to env.yml and uncomment
23    # the following line. Also, make sure to not
24    # commit your env.yml.
25    #
26    environment:
27      MONGODB_URI: ${self:custom.apiEnvironment.DB.${self:custom.apiStage}}
28
```

Figure 19: MongoDB Production URI setup in Serverless.yml as an environment variable

With this configuration, the team can customize the username and password without providing plain text since they are sensitive information. It is great for security purposes. Also, the URI can be accessed securely within the application with the environment object *process.env.MONGODB_URI.*

Traditionally, the connection to the Atlas's MongoDB via URI from the application is going through HTTP/HTTPS, which means it will go to the Internet. However, since MongoDB was chosen to be the best option because it is also hosted in the

AWS server, a Virtual Private Cloud (VPC) can be set up between services inside the AWS ecosystem to reduce the latency and increase the security.

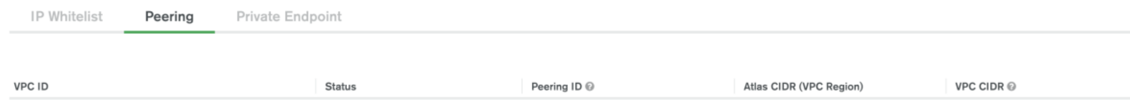

Figure 20: Setup a VPC connection to Atlas MongoDB

VPC can be set up directly on MongoDB Cloud Console in *Network Access/Peering* section. The information needed for this setup is AWS account ID, region information of the application and blocked subnets information.

As a result, the responses received through VPC connection have a minimum 100 to 300ms reduction compared to the traditional way. It means a lot when it comes to budget-saving since Serverless applications are billed with the consuming time. For example, if there are 3 million requests per month and 200ms are cut down for each request, 600000 seconds of useless execution time could be saved every month.

6.1.2 Connect/Disconnect function MongoDB

Mongoose.js is MongoDB fully-supported library with user-friendly syntaxes for Node.js runtime application and it will be used throughout the application to replace the original.

Connections will be opened whenever there are needs for data querying. Connection will be closed when being timeout (by default is 30 seconds) or *mongoose.disconnectMongoDb()* gets called.

```
const mongoose = require('mongoose');

const uri = 'mongodb+srv://username:badpw@cluster0-OMITTED.mongodb.net/' +
  'test?retryWrites=true&w=majority';
// Prints "MongoError: bad auth Authentication failed."
mongoose.connect(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  serverSelectionTimeoutMS: 5000
}).catch(err => console.log(err.reason));
```

Figure 21: Connect to MongoDB via a URI

During the connection initialization, some special parameters need to be provided to serve the purpose of the application:

- *useNewUrlParser* (Boolean): The MongoDB Node.js Driver rewrote the way it parses the connection string and the old way will be removed in the future, so providing true value to this parameter will keep the connection initializing process always up-to-date.
- *poolSize* (Integer): Define how many sockets per connection that the current MongoDB tier can handle according to the application usage. If the sockets are full, it will start a new connection. By default, it is 5 and the maximum connections allowed for free tier are 100.
- *autoIndex* (Boolean): Apply the newly defined Indexes structure to the current Schema. By default it is false.

6.1.3 Schema definition

It is great that MongoDB is a Schema-type structure with validation for all database queries. Before, on DynamoDB, Item validation was manually developed and used whenever a query got executed.

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea'],
    required: function() {
      return this.bacon > 3;
    }
  }
});
var Breakfast = db.model('Breakfast', breakfastSchema);
```

Figure 22: Example of creating a MongoDB Schema with validations

With this setup and required attributes from section **3.1 Main Object**, a Note Item Schema can be initialized and validated whenever a document is being inserted into the database. It is an important upgrade that helps the team with there being no need to implement a custom validator for the input data as in the past, which is less efficient and secure since it is already provided by the MongoDB features.

Beside simple validation, it is possible to define a custom validation for Item attributes. Also, new attributes and validations can be added later to schema without the worry of changing the current data in the database

After finished creating validation, the Item Model can be exported from Item Schema. Indexes can also be added as an optional when a Schema is being defined.

Multiple schemas can also be added as in the example.

6.2 Querying

A database query can be performed using the Model object that had been exported.

```
// include all properties except for `length`
Adventure.findById(id, '-length').exec(function (err, adventure) {});

// passing options (in this case return the raw js objects, not mongoose documents by passing `lean`
Adventure.findById(id, 'name', { lean: true }, function (err, doc) {});

// same as above
Adventure.findById(id, 'name').lean().exec(function (err, doc) {});
```

Figure 23: Example of a simple get one item with option query with a model named "Adventure"

A Model is a primary tool for interacting with MongoDB. In the application, *Item.findById()* is the Model class asynchronous function which returns a single document (an Item) that matched with the ID and conditions provided.

6.3 Compatibility

There is not much of unique concern for this change because of the incomplex data structure. However, during the migration, some manual configurations need to be done in able to fully experience the benefit of MongoDB.

6.3.1 Connections management

As is mentioned in section 6.1.2, MongoDB allows users to control the limit of connections to the database by determining how many sockets that one connection can handle before creating a new one to save up resources and time. With a socket is being reused, the querying speed is much higher because there is no need to re-initialize the configuration.

However, by default, Lambda service clears all the cache whenever a Lambda function is finished, which means even though multiple requests are coming from the same origin, there is no way to get the information from the connection is being used. This results in a new connection having to be initialized and during the testing, as maximum connections errors had occurred so many times.

```
Hoangs-MacBook-Pro-2:aws_api hoang$ sls offline start --stage mongo-production
 --skipCacheInvalidation --poolSize 20
```

Figure 24: Deploy to production

Luckily, Lambda service allows us to use cache data and it is easy to turn on by providing *--skipCacheInvalidation* ***true*** when deploying to Lambda serverless service.

6.3.2 Indexes

To increase the querying speed of the database, besides choosing the correct plan and tier, indexes are also important. For DynamoDB, indexes are defined from the beginning when the database is created or edited on the AWS console.

For MongoDB, there are two ways to define indexes:

- MongoDB Atlas Cloud: This is the place to view data and monitor the state of MongoDB (usage, limit, RAM, CPU e.g). It is also able to create or delete indexes of the selected Collection
- *MongooseSchema.index()*: Indexes can be added to the existing list using Mongoose Schema by setting *autoIndex to* ***true*** in the configuration when a connection is set up.

6.4 Testing and Deploying

Previously, due to the shortage of resources, there were only unit tests available. During this migration process, a better testing environment was requested to be developed, not only to strengthen the currently fragile system but also to make sure the newly migrated database works as expected.

The tests are built with *Mocha.js* – a powerful testing library for JavaScript.

6.4.1 Unit Testing

Assert is a verifying invariants module provided within Node.js modules.

With *Mocha*'s help, test cases can be divided into simple and flexible series, making them easier to report and map the results.

```
var assert = require('assert');
describe('Array', function() {
  describe('#indexOf()', function() {
    it('should return -1 when the value is not present', function() {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```

Figure 25: Example testing indexOf() function

By default, simply run *mocha* inside the terminal will trigger JavaScript files with prefix *.test* to run the jobs, for example, *indexOf.test.js.*

6.4.2 Integration Tests

These tests are made to be sure that the API calls get the correct body data and the Lambda functions return the correct response, which could not be tested within unit testing. The test cases require handling asynchronous API calls and therefore, a special tool is required for the specific needs, *superagent*.

"SuperAgent is light-weight progressive ajax API crafted for flexibility, readability, and a low learning curve after being frustrated with many of the existing request APIs. It also works with Node.js!" - visionmedia.github.io/superagent/

```
const expect = require('chai').expect;
const request = require('superagent');

// Create a session for testing
let user = request.agent()

describe('Example Integration test', () => {
    step('Authenticate', (done) => {
        user.post(`${server}/authenticate`).send(credentials).end((err, res) => {
            if (err) done(err)
            else {
                expect(result.statusCode).to.equals(200)
                console.log('authenticated!')
                return done()
            }
        })
    }).timeout(6000)

    // Go to next step if passed
    step('Test /Get Items', (done) => {
        user.get(`${server}/notes`)
            .set('Authorization', token)
            .end((error, result) => {
                if (error) return done(error);
                expect(result.statusCode).to.equals(200)
                return done();
            });
    });
}
```

Figure 26: Example of sequence tests

With the help of *step*, asynchronous API calls are now synchronous. This *step* will not start if the previous *step* does not get executed successfully. In this particular case, the authentication process is expected to return an error if the client sends a string and it will return token if the request body data is correct.

The *timeout(3000)* indicated that if the step took more than 3000 milliseconds (3 seconds) it would return an error immediately.

With this setup, it is possible to fully create API call test cases.

6.4.3 Automation Testing

With all the tools available, it is easy to do manual testing with the system. But there are updates every day and for every update, it is necessary to test again to make sure that the new test run does not create conflict with the existing one, and therefore, the current method of testing is time-consuming and could slow the development progress.

In this section, an automation testing process for Note API will be built to solve the problem using Jenkins. The idea is that Jenkins will do all the tests that have been scripted every time an update happens (a commit), automatically.

1. **Create an EC2 instance and install Docker**

   By going to EC2 AWS, there will be a list of instances available and their details. Create an instance that is also available with **Launch Instance.**

   This should be a simple free-tier instance since its only purpose is to keep Jenkins up and running. It has been monitored and evaluated the resources needed.

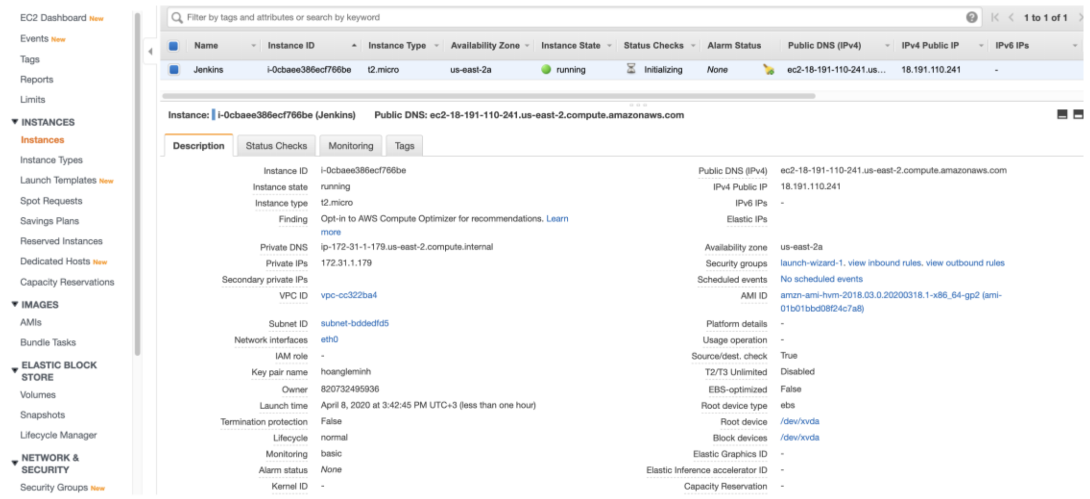

Figure 27: Jenkins EC2 Instance

After creating the instance, it can be connected by setting up an SSH connection with secret key pairs authorization to public IPv4 IP address, or directly on the console with a browser-based SSH connection.

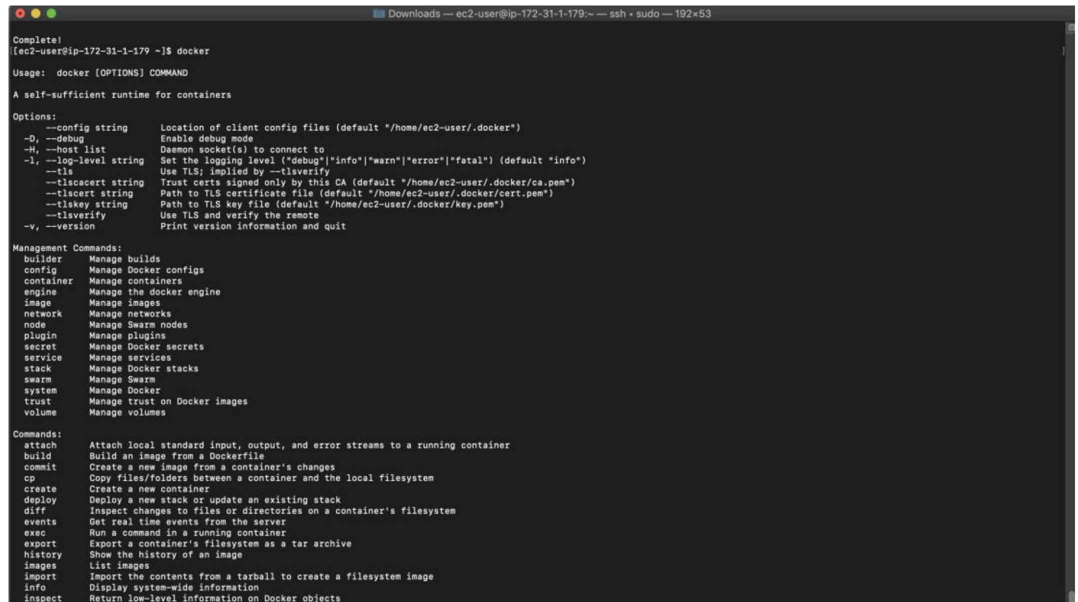

Figure 28: Docker installed on an EC2 Instance

2. Create a Jenkins container from Jenkins Docker Image

jenkinsci/blueocean is An Image contains necessary packages to install Jenkins and *blueocean*, a popular friendly user interface that comes with it.

Jenkins will be hosted on a container that will be created from the Image listed above by using the command

docker create --name jenkins-v1 -p 80:8080 jenkinsci/blueocean

This command means to create a container named "jenkins-v1" from the base Image *jenkinsci/blueocean* which port 8080 will be connected to port 80 on the Instance. Therefore, when a user accesses the default public address of the Instance, the home page of Jenkins will be returned.

Figure 29: The container is up and running

The container can be started with command *docker run*

Figure 30: Jenkins home page

3. Configure Jenkins behaviors

Firstly, to be able to trigger the test actions for every change on the Git project, an authorization process must be added to Jenkins. The credentials token key can be generated from Github profile settings.

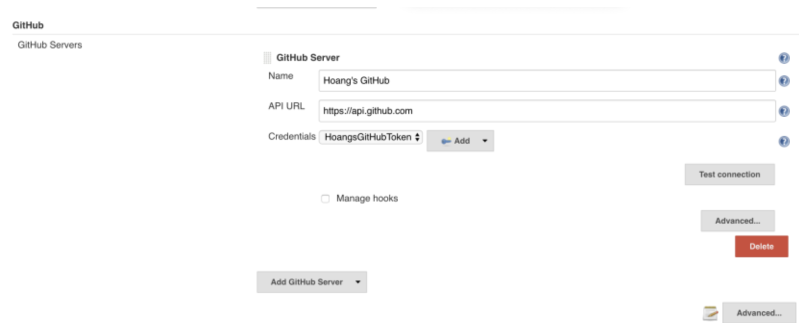

Figure 31: Add Github account token to Jenkins configuration

A multiple pipeline Jenkins project is created for this project. Add the remote branch and configuration to the pipeline settings.

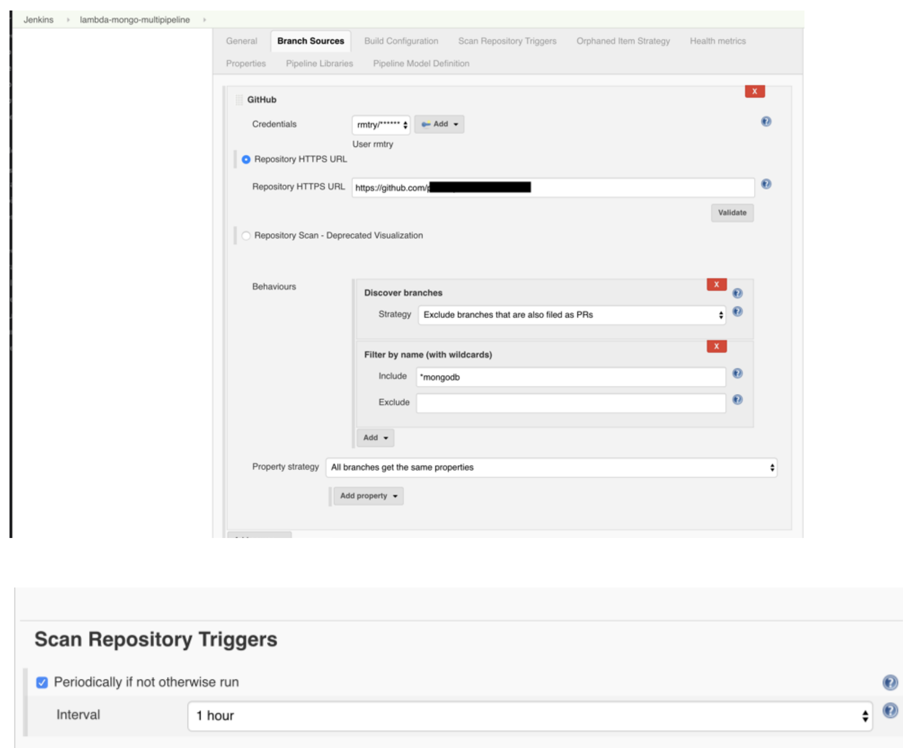

Figure 32: Pipeline configuration for Git project

This indicates that Jenkins will subscribe to the changes of the Git project on the Repository HTTPS URL with the Git credentials added. Additionally, every branch that named *mongodb* that has new commits will be triggered every 1 hour. The reason for the name filtering is that we only want the branch with the migrated MongoDB will be able to perform the tests.

4. **Configure Jenkins actions for the project**

   Since a tunnel has been made and there are test cases, there were not any specific job for Jenkins to do. Therefore, a list of required actions for Jenkins to perform needs to be made. By default, Jenkins will always search for actions Jenkinsfile in the root directory of the Git project.

```
pipeline {
    agent {
        docker {
            image 'node:8'
            args '-p 3000:3000'
        }
    }
    environment {
        CI = 'true'
    }
    stages {
        stage('Build') {
            steps {
                sh 'npm -v'
                sh 'npm install'
                sh 'npm install -g serverless'
                sh 'sls config credentials --provider aws --key AWS_KEY --secret AWS_SECRET'
            }
        }
        stage('Test') {
            steps {
                sh 'npm run test'
            }
        }
        stage('Deploy') {
            steps {
                sh 'sls deploy --stage testing --poolSize 10 -v'
                sh 'echo Run Integration Tests!'
                sh 'npm run test-integration-live'
            }
        }
    }
    post {
        success {
            echo 'Ready to deploy!'
        }
    }
}
```

Figure 33: List of actions in Jenkinsfile

- *agent*: Environment required to perform the tests. Since the project is built with JavaScript with Node.js runtime, a Node V8 was used.
- *stages*: Defined actions in 3 particular stages: Build, Test and Deploy:

    - *Build*: Install necessary packages and dependencies (Node.js modules and Serverless CLI)
    - *Test*: Run the unit tests
    - *Deploy*: Deploy to the testing environment and run the integration tests.
- *post*: Send a message after the build depends on different cases (success, failure, aborted, .eg)

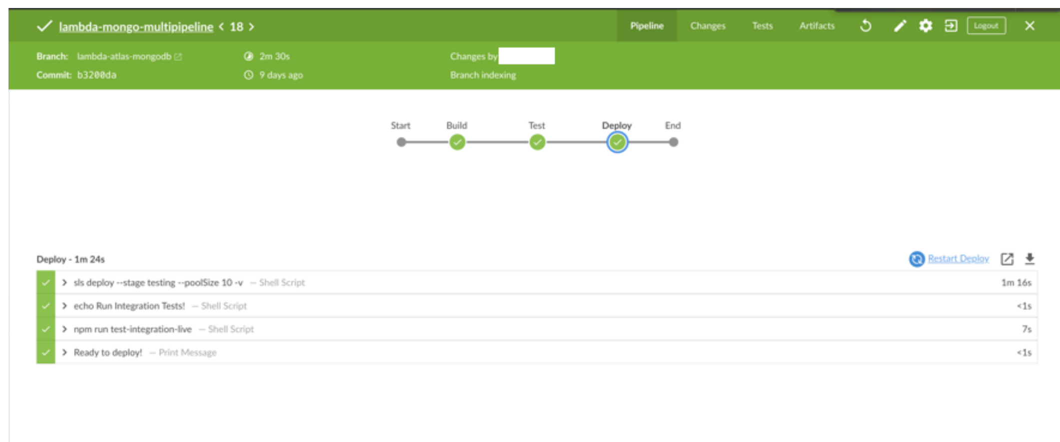

Figure 34: An example of a successful build

6.4.4 Deployment

1. Deploy to Lambda Service

In this section, the deployment to the Testing environment will be described.

The command to deploy to Lambda service:

sls deploy --stage prod --autoIndex false --poolSize 20 -v

While:

- *sls*: Serverless framework CLI (Command Line Interface) which authenticated with AWS credentials
- *deploy*: action to deploy current workspace to Lambda service

- --*stage*: define the current environment (*dev*, *testing* or *prod*). Depends on each environment that some of the environment variables will have different values
- --*autoIndex*: Check for new Database indexes, set to *false* will make the Lambda use only the old existing indexes
- --*poolSize*: Define the number of sockets available in one connection. Production tier Database has more resources than testing tier Database, therefore, it is needed to extend the usage of the production stage database.
- -*v*: An option to log all the details to the command-line tool during the deployment

```
Service Information
service: notes-app-api
stage: prod
region: us-east-1
api keys:
  None
endpoints:
  POST - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes
  GET - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
  GET - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes
  PUT - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
  DELETE - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/notes/{id}
  POST - https://ly55wbovq4.execute-api.us-east-1.amazonaws.com/prod/billing
functions:
  create: notes-app-api-prod-create
  get: notes-app-api-prod-get
  list: notes-app-api-prod-list
  update: notes-app-api-prod-update
  delete: notes-app-api-prod-delete
  billing: notes-app-api-prod-billing
```

Figure 35: The result after the deployment

As can be seen, the deployment is successful and an API path is generated by default. This path is accessible through the Internet.

2. **Export the API**

Now the API is ready, but its name is not in a humanly readable format. I already had a signed domain (****.com) associated with the project and the

only work needs to be done is to connect the domain to the Lambda resources. The current API resource is inside Amazon's ecosystem, therefore, it needs to be redirected to a gateway to be able to connect with the outside world. In this case, it is *Amazon API Gateway.*

During this section, an API path associated with the domain will be created and linked with the Note API in the previous section.

a. Configure the API Gateway

The API path will be called *test-v3.****.com* in this example.

It is necessary that a custom domain is created and it will be the API gateway, letting connections from outside to Amazon's resources through this generated Amazon API Gateway domain (*example.cloudfront.net*).

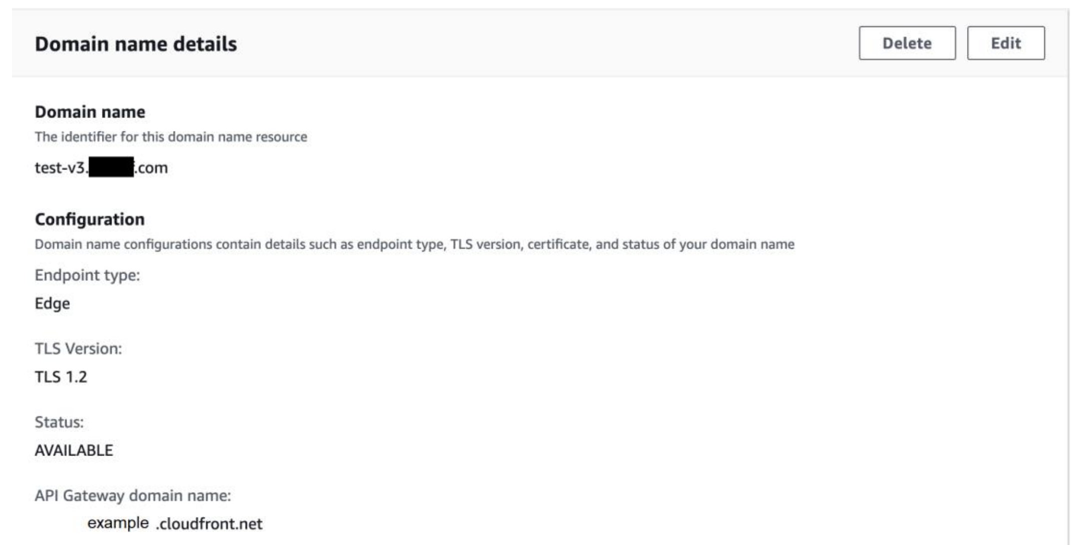

Figure 36: API Gateway Custom domain names

After that, the gateway can be connected to the Lambda resources by adding the API Gateway record to the testing Lambda function trigger on the Lambda service console.

Figure 37: The API Gateway record is connected with the testing API Lambda, which is deployed on the previous section

Now there is an easier reading API name that is available publicly, but that will not be enough for a production API.

b. Routing the API with Route53

With a valid signed domain, Route53 will setup and redirect securely the connections to this domain to the API Gateway record (*example.cloudfront.net*)

Figure 38: Route53's record set for the API Gateway record API

As can be seen, the record *"test-v3.****.com"* is connected to the same Alias Target as the API Gateway record, which means every request will go to this record will be forwarded to the CloudFront domain name and finally trigger the Lambda function. With this setup, the API is now published successfully.

7 CONCLUSION

The migration project is a great example of a challenging practical use-case scenario for developers, especially when having to give a hard decision such as replace a technology that could not be maintained, to meet the requirement of the customers. Even though the coding part is not much, a lot of hours were put to find, to research and to discuss the most optimized methodology.

For an eager developer, not the sufficient coding skills but rather the problem solving and future self visualization are the essential checkpoints in the career-based activity list.

In addition, getting individual horizons familiar with the Amazon Web Services collaboration and Serverless framework API configuration has been significantly enhancing skill throughout the developing workflow. As a result, since AWS has been an up-to-date service, starting to be a connoisseur of utilizing AWS is not only an utmost wholesome experience broadening opportunity but also a promise for a brighter career path in Cloud Technology has opened.

Future work

Even though the most crucial problem has been solved for this project, there are plenty of things that could be done in the future such as ensuring the performance for a huge amount of active items, sockets and connections managing for MongoDB, upgrading the automation test procedure and improving the logging system to be able to get up-to-date with latest events.

REFERENCES

1. Serverless framework documentation.
https://serverless.com/framework/docs/getting-started/

2. Why Serverless.
https://dev.to/yos/why-serverless--3pn7

3. Amazon Web Services
https://aws.amazon.com/

4. AWS Lambda introduction.
https://aws.amazon.com/lambda/

5. AWS DynamoDB introduction.
https://aws.amazon.com/dynamodb/

6. AWS API Gateway introduction.
https://aws.amazon.com/api-gateway/

7. AWS Cognito introduction
https://aws.amazon.com/cognito/

8. AWS EC2 introduction.
https://aws.amazon.com/ec2/

9. AWS Route 53 introduction.
https://aws.amazon.com/route53/

10. Simple Serverless AWS application architect tutorial.
https://aws.amazon.com/getting-started/hands-on/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/

11. JavaScript npm package for AWS Cognito configuration.
https://www.npmjs.com/package/amazon-cognito-identity-js

12. Atlas's MongoDB.
https://www.mongodb.com/cloud/atlas

13. MongoDB Indexes.
https://docs.mongodb.com/manual/indexes/

14. Setup VPC Connection for MongoDB
https://docs.atlas.mongodb.com/security-vpc-peering/

15. Mongoose npm package.
https://mongoosejs.com/

16. Mocha.js, JavaScript testing library.
https://mochajs.org/#getting-started

17. Superagent, npm package for API call testing.
https://visionmedia.github.io/superagent/

18. Jenkins pipeline introduction.
https://www.jenkins.io/doc/book/pipeline/

19. Docker container introduction.
https://www.docker.com/resources/what-container

20. Setup Jenkins on a Docker container.
https://www.jenkins.io/doc/book/blueocean/getting-started/

21. Basic comparison between DynamoDB and MongoDB.
https://www.mongodb.com/compare/mongodb-dynamodb

22. Data compressing solution for network bottlenecks by Ebay.
https://tech.ebayinc.com/engineering/how-ebays-shopping-cart-used-compres-
sion-techniques-to-solve-network-io-bottlenecks/

23. OpenAPI framework.
https://swagger.io/docs/specification/about/