Musication: A location-based music service

Luke Storry
Computer Science
University of Bristol
LS14172@bristol.ac.uk

Hakeem Kushoro
Mathematics and Computer Science
University of Bristol
HK14989@bristol.ac.uk

Abdu Elturki
Computer Science and Electronics
University of Bristol
AE15920@bristol.ac.uk

***Abstract*—In this paper, we present and discuss our cloud-based application, Musication, developed using Amazon Web Services, and AWS aspects such as AWS Amplify, Amazon Cognito, and DynamoDB. Musication is a music application for creating and streaming "MusicMappings" - A playlist allowing people to link certain regions of the world map to an uploaded mp3 file, causing the music file to play whenever the user is in said region.**
The App is available at: https://d312rk7qk4a72n.cloudfront.net (Can use User:example and password:Example.1234, or create an account)
Sourcecode: https://github.com/LukeStorry/musication

I. Introduction

A. Motivation

Background music has been used to enhance entertainment media, such as TV shows and movies, for a very long time. This can be used to make a fight scene more exciting, a romantic scene more lovely or a tragic scene more impactful. We wanted to try and replicate this in real life, so the day-to-day experiences of others could seem more exciting and enjoyable.

B. Description

Musication is a cloud music service that plays music based on the users current location. The user uploads a series of MP3 files, and then uses an interactive web interface to link those files to various locations on a map. These *MusicMappings* would then be saved on the users profile. Then, when playback is requested, the web app runs in the background on the users phone (or from the user's PC), keeping track of the current location, and streams the selected music files.
We define a *MusicMapping* as a pair containing an MP3 file and a location on the map (the location is a longitude and latitude pair which can both be used to find the exact location on the Google Maps API)

Once logged in, a user can easily upload mp3 files to their folder in an S3 bucket. Next, they click on any point on the map provided to associate a location with each song. This MusicMapping is then be saved in the user's section of a DynamoDB storage, through our API Gateway. When the Play button is pressed, the user's current GPS location is requested, and sent to our API, which calculates the closest location among that user's set of MusicMappings, and then returns the song key associated with the calculated MusicMapping. The front-end then queries the S3 bucket to get the URL for the song file, using the logged-in user's authentication, before feeding it through to a HTML DOM Audio Object. The URL expires after the user has finished streaming, so it cannot be exploited by anyone other than the user, hence the contents of the S3 bucket remain secure.
The Audio Object then streams and controls the mp3 by invoking audio.play() when the button is clicked (and audio.pause(), if the song is already playing).

II. Architecture

The architecture of this application can be seen in Figure 1. The architecture is fully serverless - this seperated and more complex system was designed to ensure enhanced scalability of all aspect of the system, and also reduced the costs of the application. As the application is currently small, a serverless approach would be cost effective, as it only charges based on the number of requests. Whereas a non-serverless approach, including methods such as renting *Infrastructure as a Service* (IaaS), would have an hourly cost, which was not needed for the occasional function and API calls..

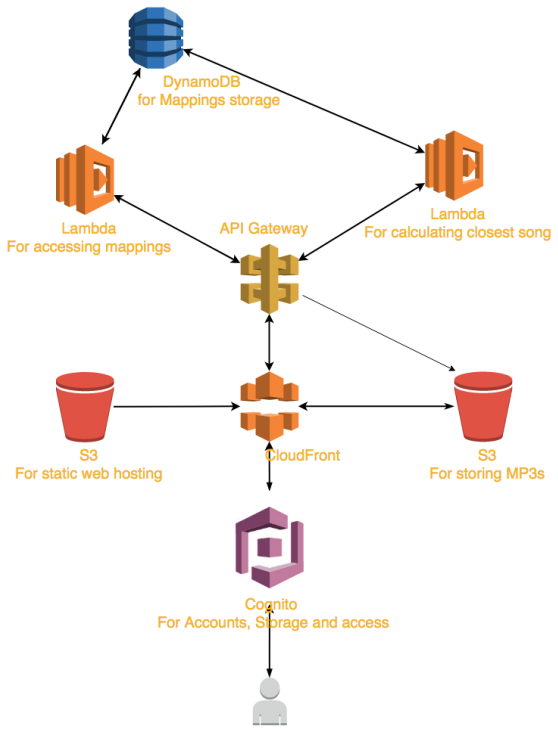

Fig. 1. Musication's Cloud architecture

A. Cloud Provisioning

AWS-Amplify is a new Command-Line-Interface (CLI) tool from Amazon that helps set up cloud services, by automating the creation of CloudFormation files and linking serverless services together securely, along with other useful CLI tools such as publishing and local testing.

To ensure high availability of the service, the architecture is available in both Ireland (AWS eu-west-1) and London (AWS eu-west-2) regions for rare cases when either service goes down. Furthermore, *AWS CloudFront* is used to automatically serve from the closest available region.

B. User Accounts: AWS Cognito

The users of the application of this application would upload MP3 files and select GPS locations, and therefore a method of saving of their content is required - An authentication system where clients would be required to create an account, to save their MusicMappings. This is so the application can distinguish which user is currently using the application

To solve this problem, *AWS Cognito* service was used, which is described in figure 2. To register, Users supply a username, password, phone number and email address (after which a code is sent to the user's email and they are asked to input this to confirm their identity). All the information provided is stored within the *User Pool*.

Whenever someone would want to log in, they would supply their username and password to the User pool, and after successful authentication, they would receive tokens to exchange with the *Identity Pool* for AWS credentials. These credentials would then allow the user through an Amazon API Gateway to access Amazon S3 (to access/upload MP3 files) and DynamoDB (to access/upload GPS locations).

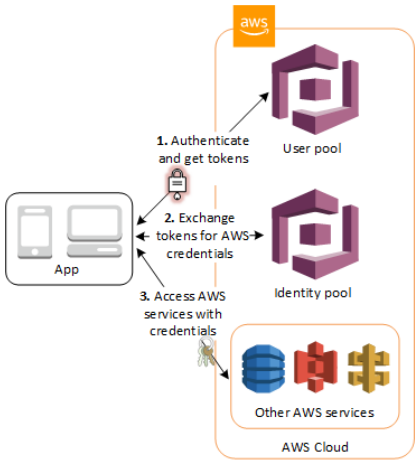

Fig. 2. Amazon Cognito model [3]

For purpose of demostration, an account is provided to try the application. The login information is as follows, **Username:** example and **password:** Example.1234

C. MP3 Upload and Storage: Amazon S3

All the MP3 files uploaded by the user is uploaded and stored to *Amazon Simple Storage Service* (Amazon S3). The content in the bucket is not shared with all users - In other words, if a user uploads their own MP3 file, it will not accessible to other accounts. The MP3 files are placed into buckets and identified within each bucket by a unique, user-assigned key.

Amazon S3 was chosen as it is very simple to operate, and provides automatic scalability and data availability. As for the costs, the free AWS tier provides 5GB of S3 storage, 20,000 GET requests, 2,000 PUT requests, and 15GB of data transfer each month for a year. After that, it costs $0.024 per GB for the first 50 TB / Month.

D. Mappings Storage: DynamoDB, Lambda, and API-Gateway

The MusicMappings are stored using *DynamoDB* and accessed via *AWS Lambda* and *Amazon API-Gateway*. *DynamoDB* stores the usernames as a key and a list of GPS and MP3 name tuples as the corresponding value. These values can be illustrated as in figure 5.

DynamoDB is used since it is an NoSQL database, which provides flexible scalability. It automatically scales tables up and down to adjust for capacity and maintain performance. As for the costs, DynamoDB provides pricing for on-demand capacity mode and provisioned capacity mode.

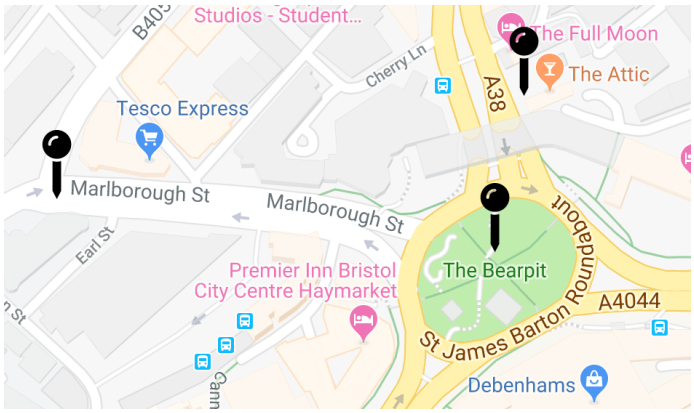

Fig. 3. Example of mappings

We wrote a series of *AWS Lambda* functions to access mappings from *DynamoDB*, allowing a user to retrieve a previous mapping that they created, request the closest song to a given GPS location, or create a new mapping. This *AWS Lambda* is accessed through REST API using *Amazon API-Gateway*, and an example of the API used to read mappings shown below:

```
{
"mapping":[["51.466,-2.5832","Building.mp3"],
           ["51.458,-2.585","Long_Stream.mp3"]],
"user":"luke2"
}
```

E. Closest-Song Calculation: DynamoDB, Lambda, and API-Gateway

Very similar to accessing mappings, an *AWS Lambda* function (accessed through using Amazon API-Gateway), calculates the MusicMapping with the closest map location: When given a latitude and longitude (the user's current location), it will get the list of that user's mappings from the DynamoDB, calculate the distances, and return the song assigned with the closest MusicMapping. For example, using latitude and longitude of "51.45755, -2.59768" the API-Gateway would provide:

```
{
    "song":"Long_Stream.mp3",
    "distance":0.00016109705373844029
}
```

F. Front-End: JS React/Google Maps API

The Front-End was designed using a mixture of Javascript React and Google Maps API (We imported from the **google-maps-react** module).

The main site is hosted in a static S3 bucket, served by CloudFront. Once Authentication has completed and the user's identity has been verified, CloudFront retrieves the site's code from the static S3 bucket and displays it to the user.

To use the site, the user must first upload the music file(s) they wish to have be played by Musication. After uploading all of this, they then click on the Map (provided by the Google Maps API) where they want their music file to play for the first song they upload, where they choose the file first and then press upload like in figure 4.

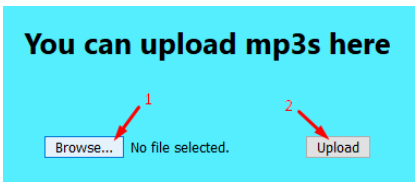

Fig. 4. Musication's Upload interface

Then, they click where they want the second song to play. Then, again for the third. And, so on until all songs have been assigned a location. This can be seen in figure 5.

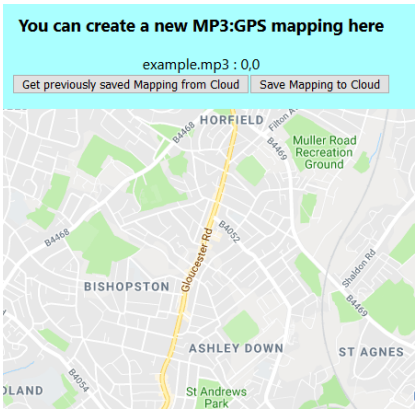

Fig. 5. Musication's Mapping interface

After the user has assigned every song to a location, they click the 'Save to Cloud' button in figure 6 and their MusicMappings shall be saved (to their key in the DynamoDB database).

This will then be accessible by the user (and only the

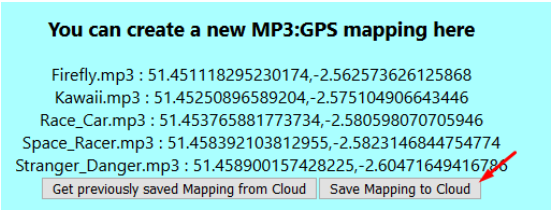

Fig. 6. Musication's Mapping save

user) when they log in to the app on their phone (or from their PC). Pressing the 'Get previously saved Mapping from Cloud' button will show their MusicMappings. When they press the play button on their phone/PC, Musication shall play whichever song is linked to the closest MusicMapping[1].

III. Scalability

This section discuss the scalability of this infrastructure and load testing that was done.

A. Serverless infrastructure

As discussed in previous sections, the serverless approach was chosen to enhance the scalability of the application, with a more modern approach that shifts operational responsibility onto AWS, making the most of their automated high-availability and super-flexible auto-scaling.

By separating out the various aspects of the architecture, the scaling only takes place in the areas that require it: If many users are using the application to stream files, the Lambda Functions and that serve that can be run continuously, but if many users are adding files, or many users are adding mappings, or signing up and doing nothing, each of those sections are individually scaled up or down automatically.

[1]the MusicMapping with the closest longitude/latitude to the user's current location

The *AWS Lambda* for both accessing mappings and calculating the closest MusicMapping each have a limit of 1000 concurrent executions. Furthermore, *AWS Lambda* would create an instance for each of those requests. A typical transaction in *AWS Lambda* takes about 100ms, so if there were exactly 82 requests per second, continuously, for the entire day, an EC2 instance with 2 vCPU and 8 GB RAM would be more cost effective [4]. However, this is an unlikely scenario for this kind of application, and taking advantage of *AWS Lambda*'s auto-scaling is much more cost-effective, and quicker to react to drastically increased load, with the downside of slow cold-starts being more common.

DynamoDB and *S3 bucket* are both of the type *Platform as a Service* (PaaS) - they provide flexible scaling as they scale capacity based on usage. Finding alternatives of this (such as having IaaS running own developed platform) would be time consuming and more difficult to maintain.

The major benefit of this infrastructure when it comes to scalability, is that it is dealt automatically by AWS. This succeeds in reducing maintenance time for this application, and provides more time for improvement with agile development.

B. Load Testing

Load testing was performed on the applications front-end, with an online service called **BlazeMeter**, as it performs load testing using **Apache JMeter**. The testing was done with 50 virtual users (VUs) as it is the maximum amount allowed by **Blazemeter**'s free-tier. During the test, the number of VUs increased linearly, over time, until it reached 50. The average throughput for this test was 71.4 Hits/s, a graph of this result can be seen in figure 7.

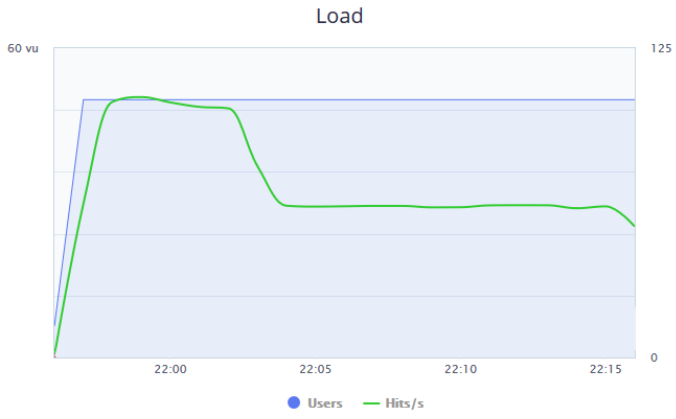

Fig. 7. Load Testing: Throughput

As for response time, in this test the average response time was 497.02ms. However as it can been seen in figure 8, the response time was mostly in range of 780-800ms during the stress test.

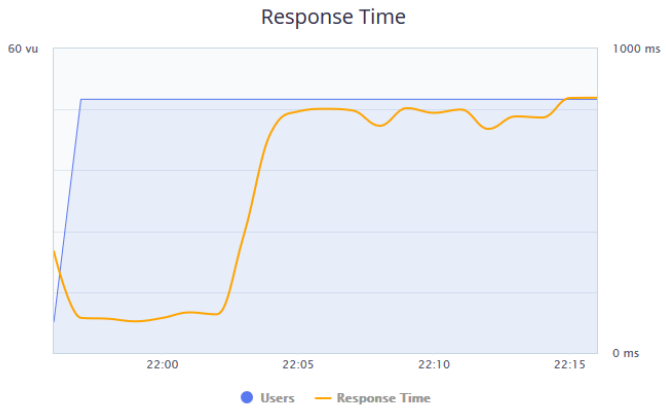

Fig. 8. Load Testing: Response Time

Despite the intensity of the stress testing, the application was still usable without any noticeable delay.

IV. Potential uses outside of personal-use/entertainment

A. Assisting Blind People

This app has the potential to assist the blind community. Since this app utilises the hearing sense (and doesn't necessarily has a requirement for sight), and the way it utilises this sense depends on the GPS-position of the user, this can convey information to blind people through the sense they rely on most, more specifically, where they are.

B. Guided tours

By uploading MusicMappings with audio recordings of speeches to relevant locations, guided tours could be made using this app. This could include Museums giving spoken guided tours of their museum, councils offering a guided tour of certain landmarks within their city, etc.

V. Potential alternatives

A. Firebase

Firebase is possibly the best alternative for this application, as all the services it provides are designed to have backend that is more focused on app development instead of different business models. The costs of both AWS and Firebase are very similar, but Firebase suffers more from **vendor lock-in** than AWS. Creating an application on Firebase would be difficult to transfer to different services, in future. This is because if an application is developed for Android and iOS, they can only be developed with the Firebase SDK.

AWS Amplify, described in section II-A, provides this simplicity for the backend without having the application locked in to AWS.

B. Google Cloud Platform and Microsoft Azure

Google Cloud Platform (GCP) and Azure are direct competitors to AWS. The prices for the two are very similar to AWS and they are converging towards a similar set of cloud services. GCP's weakness is being late entrants to the IaaS market and containing fewer features and services, which is not desirable if the application plans to use same provider for long time. As for Azure, while it is the second largest provider, it lacks documentation and management tooling. Azure would probably be the best option if the application was tightly integrated with other Microsoft applications and services. And thus, this makes AWS the optimal option for this application, since it provides strong documentation and tools and does not lack features.

VI. Potential future updates

If we had more time, there are a number of elements we could have added or improved. These include:

A. Moving Away from Serverless Approach

While currently the application being serverless is more easily auto-scalable and cost effective, it could perhaps be sustainable to move towards renting an IaaS and having multiple containers instead, if the number of users is consistently higher. As number of requests increases and usage is steady, IaaS would become cheaper to use. Additionally, having an application that does not rely on serverless tends to make the application faster to run. This is due to functions in AWS Lambda suffering from cold start.

B. Sharing MusicMappings

To encourage the social element of the app, a potential improvement would have been to provide users the ability to share their mappings. A platform dedicated to this could enrich the community and allow people to listen to the MusicMappings of others.
However, there is a downfall depending on which way we go implementing this: If we implement this by copying music files and geometric locations in our DynamoDB, this will take up a lot more memory and resources, increasing storage requirements for each time someone shares their MusicMappings - This will decrease the quality of our scalability. The alternative method is to give people direct access to the sharing user's music files and MusicMappings in their database, however, this has the potential to cause security issues, as we are breaking authentication protocol and giving users access to content that should only be accessible to the authenticated user who uploaded the music files. This opens up the possibility of potential attackers exploiting this granted access and potentially even getting access to unauthorised memory.

C. Calculating the nearest song periodically

To enable the song playing, the user has to press the 'Play' button. An improvement we would have implemented, had we had more time, would be for the app to calculate the closest MusicMapping automatically every set amount of time (i.e. every 30 seconds) and if the closest MusicMapping had changed from the last one, stop playing the current song and play the song associated with the new closest MusicMapping. However, periodically checking and calculating our closest MusicMapping invokes our Serverless functions more times, which consumes more resources, so we reduced this after our stress-testing caused us to hit the limits of AWS free tier.

VII. Conclusion

In this paper, we have presented Musication - Our location-based music streaming service. Users with an account can link music to locations in the world, enriching their experience outdoors (or, indeed, indoors).
Our app has made use of the latest AWS technologies and has fully taken advantage of serverless capabilities to increase scalability.

References

[1] Amazon, *AWS Documentation*, https://aws.amazon.com/, Accessed November 2018.

[2] Amazon Web Services, *Serverless Applications Lens*, AWS Well-Architected Framework, 2017.

[3] Amazon Cognito, *Amazon Cognito Documentation* Accessed January 2019 https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito

[4] Warzon A., *AWS Lambda Pricing in Context - A Comparison to EC2* Accessed January 2019 https://www.trek10.com/blog/lambda-cost/

[5] Blazemeter, *Blazemeter Documentation* Accessed January 2019 https://guide.blazemeter.com/hc/en-us

[6] Apache JMeter, *Apache JMeter Documentation* Accessed January 2019 https://jmeter.apache.org/usermanual/