Raymond Bergholm

Development of a Facebook Messenger chatbot application for social media event discovery

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

9 April 2018

Abstract

Author Title Number of Pages Date	Raymond Bergholm Development of a Facebook Messenger chatbot application for social media event discovery 42 pages 9 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Antti Piironen, Principal Lecturer; Olli Alm, Senior Lecturer

This thesis documents the development of a chatbot application integrated into the Facebook Messenger platform. The development process was undertaken in affiliation with I Dance Helsinki, a community organiser for Afro-Latin dance events and teaching in Helsinki, in support for an overlapping website-based solution for event organisation and community news. To support I Dance Helsinki's goal of improving the spread of news and notifications of new events in the Afro-Latin dance scene in Helsinki, a chatbot application was presented as solution which leverages the existing social community and connections on Facebook.

This thesis introduces the concept of chatbots, their characteristics as a communication channel and how they differ from other methods of connecting to end-users. As a part of this project, the technological landscape was evaluated, with a focus on the tools available for integration to Facebook and Amazon Web Services for serverless cloud hosting.

This project resulted in the implementation of a chatbot application hosted on the AWS cloud with integrations to Facebook Messenger via the Facebook Graph API and the Facebook Messenger Platform API. The chatbot allows users to query for upcoming events posted to Facebook using natural language, scoped to English within this project.

Keywords	Chatbot, social media, Facebook Messenger, Amazon Web Services, serverless, Node.js, JavaScript ES6, cloud architecture

Contents

List of Abbreviations

1 Introduction 1

2 Background 2

2.1 Characteristics of a chatbot 2

2.2 Natural Language Processing 3

3 Technological landscape 5

3.1 Facebook 5

3.1.1 Facebook Terms of Service and Policies 5

3.1.2 Facebook Graph API 6

3.1.3 Facebook Messenger Platform 6

3.2 Wit.ai 7

3.3 Amazon Web Services 7

3.3.1 AWS Lambda 8

3.3.2 Amazon DynamoDB 8

3.3.3 Amazon S3 8

3.3.4 Amazon API Gateway 9

3.3.5 Amazon Cloudwatch 9

4 Application Design 9

4.1 Data staging 11

5 Facebook Chatbot application 12

5.1 Overall architecture 12

5.2 Application lifecycles 13

5.2.1 Facebook challenge-response 13

5.2.2 Automatic event data query and staging 15

5.2.3 User message processing and response 19

5.3 Facebook application page 22

5.4 Facebook Messenger 23

5.5 AWS API Gateway 23

5.6 AWS persistent storage 25

5.7 AWS Lambda 28

5.7.1 handleFacebookChallenge 30

5.7.2 stageEventData 30

5.7.3 processMessages 33

5.8 AWS Cloudwatch 35

6 Discussion 35

6.1 Review 35

6.2 Challenges and solutions 36

6.3 Future Development 37

6.3.1 Data staging improvement: changing the persistent storage to cope with scaling 37

6.3.2 Improved NLP: incorporating Wit.ai 38

6.3.3 User customisation: implementing a session memory 40

7 Conclusion 40

References 42

List of Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
GDPR	General Data Protection Regulation
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IT	Information Technology
JSON	JavaScript Object Notation
MVP	Minimum Viable Product
NLP	Natural Language Processing
NLU	Natural Language Understanding
RDBMS	Relational Database Management System
REST	Representational State Transfer
UI	User Interface

1 Introduction

The organisation and advertisement of events in the Afro-Latin social dance community in the Helsinki capital region at the time of writing is primarily driven by a combination of individual dance studios, affiliated organisations, event locales and private groups on an ad-hoc basis. Communication is done over Facebook using its groups, events and calendar tools, however the ad-hoc organic nature of event organisation in Helsinki has led to a large number of Facebook groups with overlapping scopes. For a typical event, an organiser can expect to have to advertise to tens of different groups, which can lead to a large amount of spam for subscribed end-users. Additionally, this organisational structure has some notable gaps in that one needs to know the correct groups to subscribe to, or alternatively have enough friends with the same interests going to the same events for an event to appear in an end-user's Facebook feed through Facebook’s standard algorithms.

The goal of this project is to propose and implement a chatbot to improve the organisation of events and facilitate end-users in event discovery. This project will be made in collaboration with I Dance Helsinki, a dance community organisation based in Helsinki, as well as the main dance studios and affiliated event organisers in the capital region.

The chatbot will use Facebook's Graph API to access events as they are created on Facebook. Organisers can benefit from this tool by reducing the number of groups in which they need to advertise: most studios in Helsinki advertise their events and courses manually, so this can save time. The only requirement for an organiser is to allow the Facebook app to access their event data. End-users can benefit from this tool through a unified portal for information: the aggregated event data can be accessed by an end-user within Facebook using its Facebook Messenger chatbot features.

This project will create a publicly visible Facebook page with a Facebook Messenger chatbot plugin to respond to basic user queries by analysing user text input for key phrases and keywords. The end result of this project is to build a minimum viable product (MVP) version of a functional Facebook Messenger chatbot which responds to user queries on relevant dance events related to the Afro-Latin dance scene in Helsinki. The chatbot is required to accept simple user queries containing data on time, date or event types, process this query and respond with a list of corresponding events. In brief, the main purpose of the chatbot is to perform a search for relevant events on the user's behalf.

The technologies used in this thesis are Facebook Graph API, Facebook Messenger API, with cloud-based hosting in AWS utilising Lambdas, DynamoDB and S3.

2 Background

2.1 Characteristics of a chatbot

A chatbot can be classified as a service that is powered by rules which interacts with its users through a conversational interface. Artificial intelligence is a related topic, but it is not a mandatory component. [1] This use of a conversational interface an aspect which distinguishes a chatbot from other forms of human-machine interactions, as conversational interfaces are conversation-driven rather than menu-driven like other forms of interactions.

A conversational interface can be described as a hybrid user interface which interacts with users through the use of natural language, such as written text or voice, as well as graphical UI elements such as buttons, menus or images [2].

Unlike menu-driven interfaces, a chatbot expects human input in the form of natural language. This requires additional steps for the chatbot to convert the input into a context which can be understood: firstly, input recognizers and decoders are used to analyse the input and convert it to text, then natural language processing is applied to the text to analyse and extract semantic information. To handle the application logic, an extra layer above this is the concept of a dialog manager, which is used to handle the flow of information from and to the user. The dialog manager also assigns questions or issues to the relevant task manager which handles one specific issue which requires input from the user. After all of the required input has been received from the user, it is processed and an output is formulated using a natural language generator. [3]

Due to the fact that chatbots communicate to a user through a conversational interface, there are some specific aspects which affect usability. A conversation consists of elements which are short-lived and time-critical: messages which are newer are more relevant to the current stage of the conversation, while older messages reflect the state of the conversation at the time of writing and thus is useful for tracking the progress of the conversation but may not necessarily be up to date. Additionally, the amount of text which can be displayed at one time is limited, with a hard limit imposed by the display medium and a separate soft limit in the form of the amount of words which can be comprehended by a user at any point in time [4].

Another aspect of a chatbot which can affect its features is its apparent personality. The chatbot's personality is determined by the choice of words, tone and reactions to user input. The personality of a bot is distinct from its core functionality: the medium is distinct from the content. This has ramifications for the way a chatbot can reply to users: while it is good usability to keep the content of a message minimal to keep it clear, the medium does not necessarily have to be minimal [4]. This is analogous to a human conversation: there are numerous methods of communicating a concept in natural speech, however each method evokes a different tone.

A final aspect of chatbots to consider is that a conversation can contain messages which do not provide any actionable input and could be unrelated to the core purpose of the chatbot, for example greetings or general frivolity. How the chatbot responds to this can vary greatly depending on the chatbot's role and purpose, for example a sales bot could be expected to act in a friendly and casual manner analogous to a human salesperson, while a bot for a law firm could be expected to act in a formal, professional manner as expected of a human lawyer. [4]

2.2 Natural Language Processing

Natural Language Processing is domain of research focused on the understanding and expressing of natural language by computers [5]. A specific subdomain of interest to this project is Natural Language Understanding (NLU), which is focused on classifying intents and extracting entities from user input.

In the context of NLU, the intent of a sentence is its general purpose and meaning. In natural language, an intent can be expressed in multiple ways using different tones and registers, for example the sentences “Could you help me?”, “help!”, “what?”, “help meeee”, “hlp plz” or “I am really sorry, but would you kindly help me?” all express the same intent of “need help” despite a wide variance in tone, context, usage of informal language or slang, deliberate misspellings or register. The classification of a sentence’s intent is not to completely understand the user, an intent can be understood as the high-level summary of a sentence’s meaning. As a result of this, an intent is intentionally generic and independent of the exact details of a sentence. In the context of a chatbot an intent could also be classified as the type of request to send to the backend.

Entities are related to intents in that they also influence the meaning of a sentence. Entities by themselves do not constitute enough information to create a viable backend request, however when paired with an intent, entities give context to the resulting request. Entities can thus be considered variables of an intent which aid in adding additional details to the backend request. For example, in a sentence “I want to buy the red shoes”, the intent of the sentence is a request to purchase something, while the entities would be “red” and “shoes”. Even if the words were swapped to be “blue” and “shirt”, the intent will still remain the same, even though the details are different. Entities can be classified into types which can be used in the context of the intent: in the previous example red and blue are types of colours, while shoes and shirt are types of clothing. The relevance of entity types depends on the intent: colour and clothing types are relevant to an intent to buy, while they are less likely to be relevant to the prior example of a help request [5].

Both intents and entities are elements which are extracted from user input. There are multiple methods of extraction, a simple example would be the use of heuristics with pattern recognition or rule-based expression matching. While this is sufficient for correctly formed user queries, it may fail for any user input that does not match the expected structure, or due to grammatical or spelling mistakes [6]. Modern chatbots usually implement more sophisticated algorithms to extract intents and entities such as neural networks and AI which offer more flexibility than static pattern matching. A common characteristic of these algorithms is that they need to be trained through the use of training examples. These are sentences which have already been analysed with the intent and entities already classified and labelled for the algorithm. Through these examples, the algorithm can utilise machine learning to apply the examples to user input, to identify the likely candidates for a sentence’s intent and entities [5]. Creating and training a classification algorithm is outside the scope of this project, however there are publicly available NLU algorithms which can be used to decrease the required workload.

3 Technological landscape

3.1 Facebook

Facebook is a social media platform which was first launched to the public in 2006, with a stated worldwide user count of over a billion users in 2012 [7]. It is relevant to this project as the majority of information spreading in the Afro-Latin dance community in Helsinki occurs on Facebook due to its prominence as the main social media platform of the current age. With a user base consisting of an estimate of 2.5 million people in Finland [8], Facebook is the primary method of alerting the majority of people in the dance scene about upcoming events. The share feature is particularly well known to event organisers, who typically start advertising upcoming events up to a week in advance, and occasionally many months in advance for larger events such as festivals. These event shares are a workaround for the current limitation where Facebook only allows up to 50 direct invites to an event: as the dance scene overall is a few orders of magnitudes larger than this, the habit of event sharing was started to attempt to reach the largest percentage of the intended audience possible.

3.1.1 Facebook Terms of Service and Policies

The Facebook Terms of Service affect the scope of the chatbot application, as this application must conform to the responsibilities listed in the terms of service. The application only uses only data which has been published publically by event organisers, thus the most applicable part of the standard end-user terms of service is section 2.4, which states that “when you publish content or information using the Public setting, it means that you are allowing everyone, including people not on Facebook, to access and use that information, and to associate it with you (i.e. your name and profile picture)” [9] where “you” in this sentence refers to a Facebook end-user. In the case of this project, this can refer to event organisers or the chatbot’s own end-users. This section of the end-user terms of service is a disclaimer on the publishing of data for public consumption and thus, public data is fair use for this project.

In addition to the end-user terms of service, Facebook also mandates a Facebook Platform Policy for developers or application operators. The core part of the policy states that the developer should build a quality product, to give people control, protect data, encourage proper use and to follow the law. In addition, the section of the policy focused on the Messenger platform adds additional requirements related to chatbot logic, including a policy of clear user authentication, respecting user opt-out and to only contact users who give consent to be contacted [10]. This project will avoid the use of personal data, which allows the chatbot to circumvent the need to ask for user permissions. This is considered a core requirement to respect the end-user’s privacy and to minimise any hurdles to user adoption of this application.

3.1.2 Facebook Graph API

The Facebook Graph API is a HTTP-based API published by Facebook which allows applications access to the data available on the Facebook social network platform. Through the API, applications are able to query data as well as perform write actions such as posting stories or uploading photos [11]. The Graph API is a central part of the programmatic access to the data found on Facebook, allowing a program to perform the same actions as a standard end-user.

As suggested by the name, the Graph API organises data in a social graph format, with nodes representing objects, fields representing attributes owned by an object, and edges representing connections between objects [11].

3.1.3 Facebook Messenger Platform

The Messenger Platform is an API interface published by Facebook which allows applications to integrate to Facebook Messenger. It is intended for the integration of chatbots and contains features supporting this arrangement, including a built-in Natural Language Processing (NLP) feature using an NLP engine developed by Wit.ai. Chatbot integration to the Messenger platform is handled using a webhook, which is a HTTPS endpoint set up to accept incoming requests. These requests usually correspond to incoming chat messages, although in addition the Messenger integration can also be configured to send metadata events such as read and delivery receipts to check if the message was correctly sent, received and read [12].

3.2 Wit.ai

Wit.ai is a service with affiliations to Facebook offering a NLP engine for open source applications. The Facebook Messenger platform has a built-in integration implementing the Wit.ai NLP engine, which contains default models for selected languages. This integration allows payloads arriving from Messenger to come with an NLP analysis of the message which can be used by the chatbot logic, avoiding the need to implement NLP. Wit.ai also exists as its own service separate from Facebook, unlike the built-in Messenger NLP, Wit.ai allows a developer to create a custom model and modify it to suit the application's requirements. This has been identified as a need in the chatbot logic, as the chatbot will be required to understand specific dance vocabulary and terminology.

3.3 Amazon Web Services

Amazon Web Services, commonly abbreviated to AWS, is a cloud hosting and computing service offered by Amazon. As of 2017, AWS controls 35% of the cloud computing market with major competitors including Microsoft, Google and IBM [13].

AWS is a collection of tools and services which can be combined to create cloud-based applications with an empathise on scalability and availability. AWS have numerous data centres across the globe on all continents, with the European locations of Ireland, London, Paris and Frankfurt of particular interest to this project due to their physical proximity to Finland.

This section gives a brief overview of the AWS components used by this project, describing the functionality offered by the service and their relevance to the chatbot's operation.

3.3.1 AWS Lambda

Lambdas in AWS are standalone functions which can be executed in response to incoming events. These events can be triggered by other AWS resources and thus are suitable for the project’s overall requirements where there is a need to handle incoming HTTPS requests as well as performing batch jobs. Lambdas are stateless and thus are suitable for scaling up to handle as many calls are required [14], and as they do not share states each Lambda instance can handle their respective call independently without affecting other instances.

3.3.2 Amazon DynamoDB

Amazon DynamoDB is a NoSQL database available on AWS. NoSQL databases are non-relational and entries in a NoSQL database typically operate on a key-value basis, as opposed to a more traditional relational database which contain defined tables and schemas. Requests for data from a relational database are typically queries which conform to a version of Structured Query Language (SQL), which is parsed and executed by the Relational Database Management System (RDBMS). In contrast, NoSQL operate on an object-based level with a flexible data model where items may have different structures [15].

Entries placed in DynamoDB contain at minimum a primary key with a value which is used to identify the entry and for data partitioning purposes. The entry may contain any number of key-value pairs and secondary keys may be nominated to improve the search process.

3.3.3 Amazon S3

Amazon S3, an abbreviation of “Simple Storage Service”, is an object storage solution available in AWS with a focus on high scalability, reliability and low-latency data storage [16]. Storage in S3 is handled using buckets, which are containers for storing objects with a common function; settings such as access permissions and object encryption are handled on a bucket level. Some common use cases for S3 are private data storage areas for non-volatile objects, application log storage, or publicly accessible buckets can be used as a method of serving content for static websites. All common use cases of S3 involve objects which are expected to be read often, this is in contrast to long-term storage options such as Amazon Glacier which is more suitable for archived data objects.

3.3.4 Amazon API Gateway

Amazon API Gateway is an AWS component which allows for management and publishing of APIs. It is suitable for creating REST APIs for public consumption as API Gateway has the benefit of leveraging the AWS platform to handle network security, authorisation and traffic management of concurrent requests [17], simplify the development of a public-facing network interface. For this project, API Gateway is expected to handle all network traffic between the AWS-hosted components and Facebook.

3.3.5 Amazon Cloudwatch

Amazon Cloudwatch is a monitoring solution well suited to monitoring the activities occurring within other AWS components, which offers the possibility to collect and track metrics to measure the performance and resources used by AWS. [18] As it is a part of the AWS environment, it is easy to integrate Cloudwatch with the other AWS components used by the project.

Additonally, Cloudwatch allows developers to create Cloudwatch events which can be used to create scheduled events. As this project is expected to require scheduled batch jobs, this scheduled event feature along with the monitoring and debugging features makes Cloudwatch an invaluable part of this project’s infrastructure.

4 Application Design

The dance community in Helsinki is a broad demographic with a wide variance in age, professions and IT literacy. As a result of this, a key requirement for the overall user experience is ease of use, requiring as little user input, IT literacy or specialist knowledge as possible. It should not be assumed that the end-user of the chatbot is highly IT literate, and the chatbot user experience should remain simple and support the end-user’s help requests.

Due to the event data and connections requiring integration to Facebook, the implementation of the project requires the use of the Facebook Graph API for data queries and a functioning server with a publicly available URL for integration as a Messenger webhook.

To facilitate a smooth user experience, the chatbot will be scoped to use the data from the Facebook Graph API which is publicly available, that is, any data which does not require any user permissions or authorisations. This therefore rules out using the user's personal profile information or their current location. While this limits the options available to personalise the chatbot based on the user, e.g. using the end-user's profile data to determine more appropriate message responses or tailor the events returned based on the user's affiliations to different dances or dance schools, this restriction does not prevent the application's main function of event discovery as nearly all organisers in Helsinki share their events on Facebook from public pages. This setup allows the chatbot to access most of the relevant data in an event. One minor exception is that a small minority of event organisers post their events from their personal user account, which prevents any direct query via the Graph API without explicit permissions due to the fact that the data from public pages and private users are handled differently. There are two workarounds for this issue: one is to contact the organiser and ask them to give permission to the app (this also necessitate a user interface for these organisers, for example the chatbot itself or a separate website). The other option is to scan relevant public groups for event advertisements, as the events themselves are still public even if the organiser is a private user. As there is already a need to scan public groups feeds anyway to collect events from smaller decentralised organisers, especially the nightclub venues, the second option will be sufficient for the initial application alpha.

As this implementation is a Facebook Messenger chatbot, the user interface is the Facebook Messenger platform itself. This simplifies the scope of the project as no user interface development is required: all messages go through the Messenger chat interface and the only requirement is ensure that all messages are formatted correctly for the Messenger API.

The backend part of this project will be a cloud-based backend hosted on AWS. This was chosen due to the highly scalable environment which is suitable for the application's entire development cycle, from the initial development stage to the eventual live deployment. AWS allows the issues of scalability, performance and network security to be abstracted out and handled by the components available on the AWS platform. AWS contains all the components necessary for handling the chatbot’s requirements for message processing and persistent storage, in addition it contains other tools which can be leveraged if the chatbot’s scope expands in the future during further development. Additionally, the free tier plan allows access to all of the required services detailed in this project for free during the first 12 months, with usage limits which are unlikely to be reached during development. After the free tier plan ends, the majority of the tools used by the project will remain free, and the rest are sufficiently cost-effective [19] for the project’s projected user base which is expected to remain within a few thousand unique users.

The specific AWS tools chosen for this project are Lambdas for message processing and script execution, DynamoDB and S3 as persistent storage solutions, API Gateway for connecting to the Facebook platform, and Cloudwatch for logging.

4.1 Data staging

For this application, an important feature is to always have to most up-to-date information. In an ideal scenario, every data request could be polled directly from Facebook Graph API and no data has to be staged.

In practice, this live polling would become computationally expensive as the number of users scale upwards, as it would require several HTTPS requests to fetch and collate the data. Due to the nature of a chatbot, it is preferable that replies are returned within a reasonable time window. Additionally, users may send multiple messages with data requests, and multiple users may make similar or the same requests that lead to the same results. As a result of this scenario, in a live polling architecture, there would be a large additional overhead as the same event data will be subject to identical computational calculations and event data analysis every time a user makes a request, which creates a scaling issue.

To solve this problem, a data staging area should be created to keep a local copy of the event data on AWS. Rather than polling the data directly with Facebook Graph API request, the chatbot can poll the data staging area instead, which already contains the relevant event data and all chatbot-specific metadata is pre-calculated ahead of time. This setup allows the application to minimise redundant calculations: the event data can be fetched at set times as a batch job, which collects all of the event data within the relevant time period, then performs keyword analysis to add metadata tags. This analysis step is only necessary for each event once after every initial fetch or update, so by performing the analysis ahead of time, there is no need to repeat it during end-user message processing.

5 Facebook Chatbot application

5.1 Overall architecture

The overall architecture of the application exists within an account in the AWS cloud and has specific external endpoints situated in a corresponding Facebook platform application, which leverages Facebook Graph API and the Facebook Messenger platform.

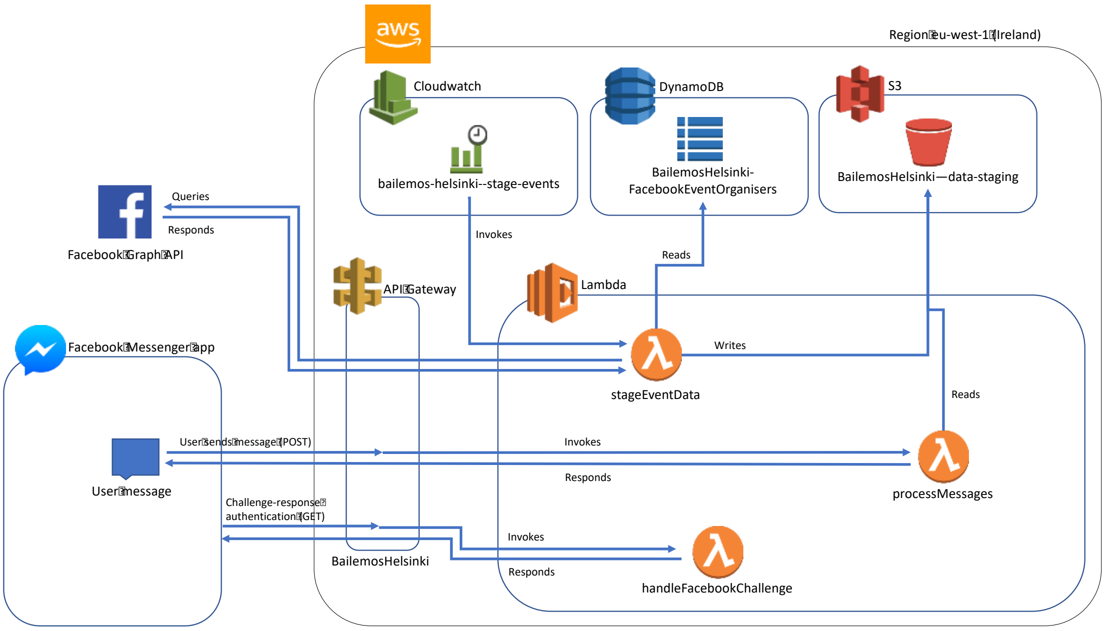


Figure 1. Architecture and component relationships of the chatbot application

The Facebook application is aware of the AWS URL which has been configured to allow HTTPS access from external sources. Both the challenge-response and user message requests are sent to the specific endpoint exposed by the API Gateway. Conversely, the lambdas in this AWS setup has the Facebook application's access token saved to the environmental variables. This access token allows requests originating from these lambdas to be verified as belonging to the application and as a result inherits the application's access rights. These access rights allow the lambda to make requests to the Graph API to fetch event data, as well as granting access to the application page's message inbox.

Figure 1 shows the architecture and relationships between the components implemented in this application. Each component has a specific purpose and task which corresponds to its role in one of the application's lifecycles. As illustrated in the figure, the application's components are hosted on the eu-west-1 data centre which is situated in Ireland. This choice of location was made as a trade-off between cost and proximity: the nearest data centres to Finland are situated in Ireland and Frankfurt, however during the project's implementation Ireland offers a lower overall cost on the services used by the chatbot.

5.2 Application lifecycles

The chatbot application implements three main lifecycles which allow the chatbot functionality to function. These lifecycles are triggered by different processes and run independently of each other, however they function interdependently to create the application functionality.

5.2.1 Facebook challenge-response

This process is executed during the chatbot's integration with the Facebook Messenger Platform. As detailed in the Facebook Messenger Platform guide, when a URL is registered as a webhook, the Messenger Platform will send a challenge-response authentication request to the URL, and the URL is accepted only if it responds correctly.

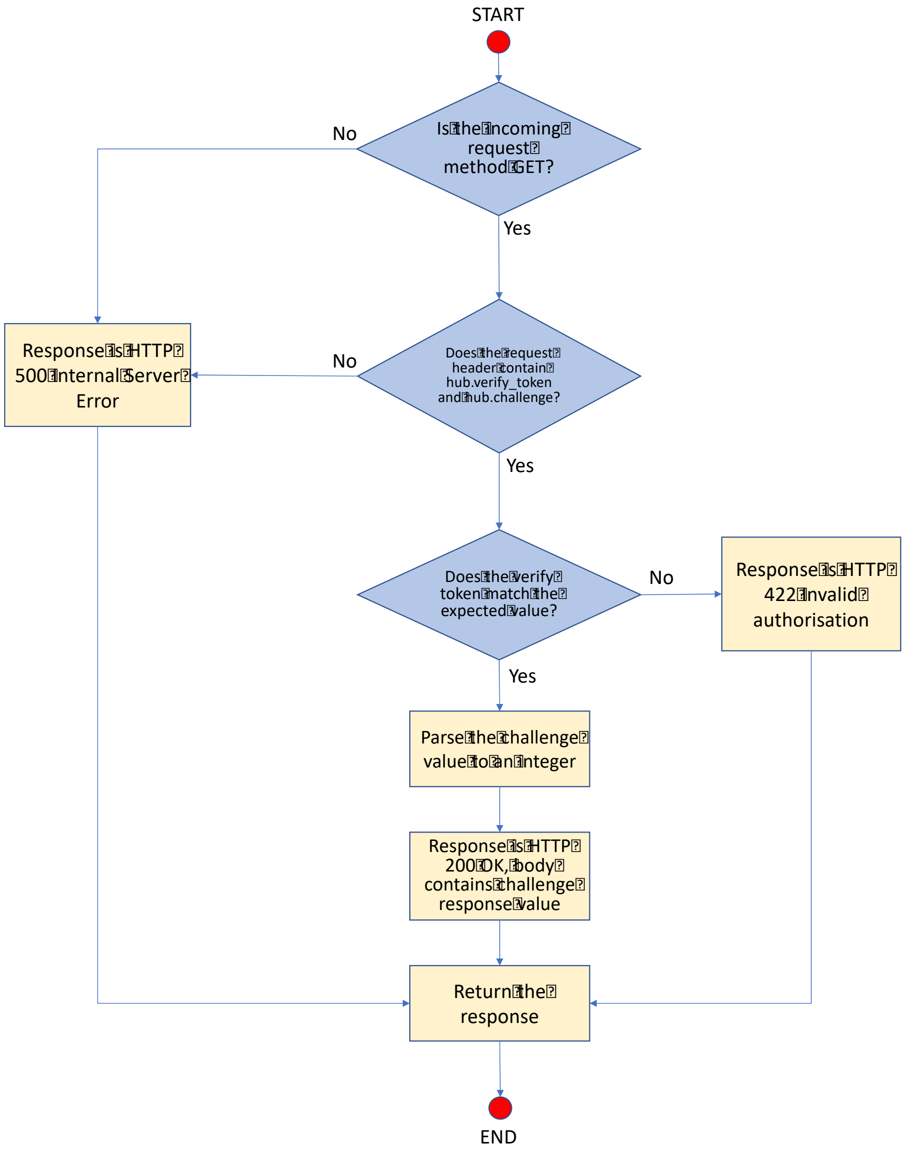

START
Is the incoming request method GET?
No
Yes
Does the request header contain hub.verify_token and hub.challenge?
No
Yes
Response is HTTP 500 Internal Server Error
Does the verify token match the expected value?
No
Yes
Response is HTTP 422 Invalid authorisation
Parse the challenge value to an integer
Response is HTTP 200 OK, body contains challenge response value
Return the response
END

Figure 2. Flow diagram of the Facebook challenge-response process

In the specific case of the Messenger Platform, this authentication is performed by sending a GET request to the URL with two parameters: `hub.verify_token` and `hub.challenge`. The parameter `hub.verify_token` is a plaintext string token which is defined in the Facebook webhook settings and is used as a basic security layer. This token remains as an unchanging constant which is sent with the challenge request, the chatbot's backend can check the value of this token to verify that the request is coming from Facebook. Conversely, the 'hub.challenge' parameter is used by Facebook to verify that this webhook is active and has been correctly configured. Each time the challenge request is sent out from Facebook, it contains a random integer in the ‘hub.challenge’ parameter. To complete the integration to the Messenger Platform, the lambda needs to verify that the verification tokens match, then respond to Facebook with a HTTP status 200 response with the challenge number in the response body. If successful, both sides of the connection can verify each other’s identity user messages will be forwarded from Facebook Messenger to this webhook. Figure 2 shows the flow of the process.

As this process only occurs on registration, this is only required during the initial registration to link the AWS-based solution to the Facebook app, or at any point when the webhook URL changes.

5.2.2 Automatic event data query and staging

The event data found on Facebook is central to this application’s functionality, however the standard fetch requests from Facebook Graph API is not sufficient for the chatbot’s requirements. While it is possible to develop a naïve implementation where every user message also requests event data from Facebook and perform the relevant analysis, this creates additional redundant HTTP requests, calculations and processing which can greatly impact performance and lambda resource requirements.

To solve this redundancy risk, the application automatically populates event data from Facebook and stores the data in the chatbot’s backend as a data staging area. This additional layer results in duplicated data and may occasionally be out of sync with the main data store on Facebook’s servers, however this is a valid trade-off for this application as the data staging area allows the application to modify or enrich the data as required by the application’s needs.

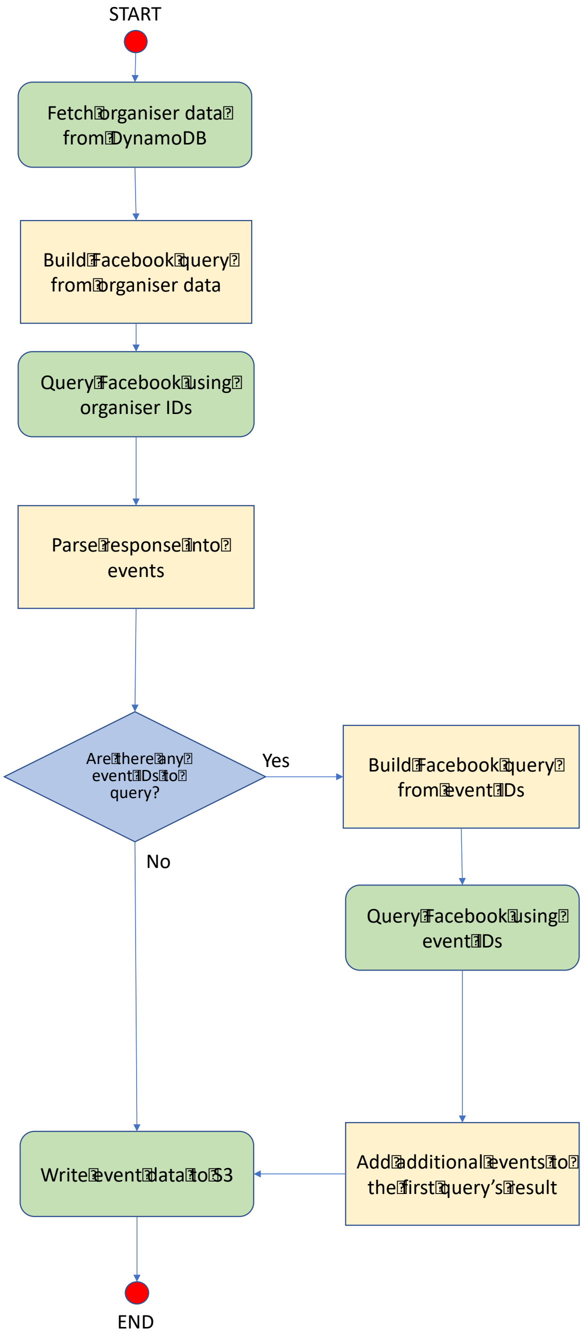

START
Fetch organiser data from DynamoDB
Build Facebook query from organiser data
Query Facebook using organiser IDs
Parse response into events
Are there any event IDs to query?
Yes
No
Build Facebook query from event IDs
Query Facebook using event IDs
Add additional events to the first query's result
Write event data to S3
END

Figure 3. Flow diagram of the event data query and staging process

This lifecycle starts from an automatic batch job which runs twice a day: once during the night, and one during the late afternoon. This setup is a trade-off between data validity and resource costs: based on observations, events are typically created during the day time or evening, while existing events may be updated prior to the event’s start time. Running the batch job twice a day is expected to keep the data synced to an acceptable degree without consuming an excessive amount of lambda runtime or DynamoDB/S3 read/write accesses, all of which will incur additional costs.

The lifecycle starts by querying a set table in DynamoDB which contains page and group IDs, which are collated into batch requests to the Facebook Graph API. For pages, events can be found directly linked to the page so these calls are of the format <page_id>/events. In contrast, groups typically do not have any events of their own as groups are usually used by organisers to advertise events or by users to share info on events or discussion. Thus, groups require calls of the format <group_id>/feed, which will collect the posts found in the feed.

These calls are sent to the Graph API, which then responds with the requested data, for each entry returned the lambda will perform one of two actions depending on whether it is from a page or a group:

- Responses from page nodes have the relevant events directly in the response, so these results are added to the aggregated events array

- Response from group nodes have a list of recent posts, so each post content needs to be read to check if it is advertising an event. Facebook’s built-in event sharing feature follows a static layout which is distinct from a standard user post, so each post can be compared to this template to determine if it is an advertisement. Posts which match the template have the event linked as a URL with the ID visible, as a result the event ID can be inferred from the advertisement. All events IDs collected this way is sent to a secondary query which fetches data from the event nodes directly

All events fetched from either of these options are then processed further using keyword analysis. The event data and description are analysed using a series of regular expressions to find matching words to add the following metadata tags:

- Event type: the analysis attempts to categorise this event as a course, party, workshop or festival. Facebook events do not natively handle the distinctions between these types of events as it is up to the user to distinguish between them. The event category is determined by scanning the title and description for keywords related to festivals, courses and parties. Each type of event typically uses different vocabulary, thus by checking the frequency of different terms and jargon and assigning weights to specific terms, the probability of this event being one of the categories can be calculated. This information is used by user queries to filter searches to a specific category.

- Styles: a simple regular expression keyword analysis of keywords related to dance styles. For this project's scope, the dance styles supported are salsa, bachata, kizomba and zouk, which are the dance styles most closely connected to the IDance Helsinki organisation. Events in Helsinki are frequently mixed and may consist of multiple rooms dedicated to a specific dance style or have a combined dance floor rotating across dance styles. To reflect this setup, the results of the regular expressions are collected into an array and an event may be tagged with multiple dance styles.

- Timezone offset: this metadata tag is collected to solve a technical issue where the timestamp's timezone information is lost whenever a timestamp string is converted to a JavaScript Date object. This is due to the server environment's internal server clock always being set to GMT, and when when a timestamp string is converted to a Date object it is automatically converted to the GMT time equivalent and the corresponding timezone information is lost. This tag is a simple conversion of the end of the timestamp string, extracting the hours and minutes from the "+HHMM" format at the end of the string.

After keyword analysis and metadata tagging, the events are collected into a single JSON structure and then saved to an internal private S3 bucket as a single JSON binary file. This is the only point in the application when write access to the JSON is necessary, during user processing this JSON file is loaded for read-only purposes. Figure 3 illustrates the entire process in the format of a flow diagram.

This lambda is configured to automatically run as a batch job. For this project, it is set to run every day at 04:00 GMT to collect updates while Finnish users are normally asleep, and 15:00 GMT to catch any last-minute updates at a point in time before most events start in Helsinki.

5.2.3 User message processing and response

This process is executed in reaction to an incoming user message sent from Facebook and is the main process of the chatbot. As it is expected to be called frequently, resource and time efficiency is important for this process as this is the main cause of the bulk of all operational costs.

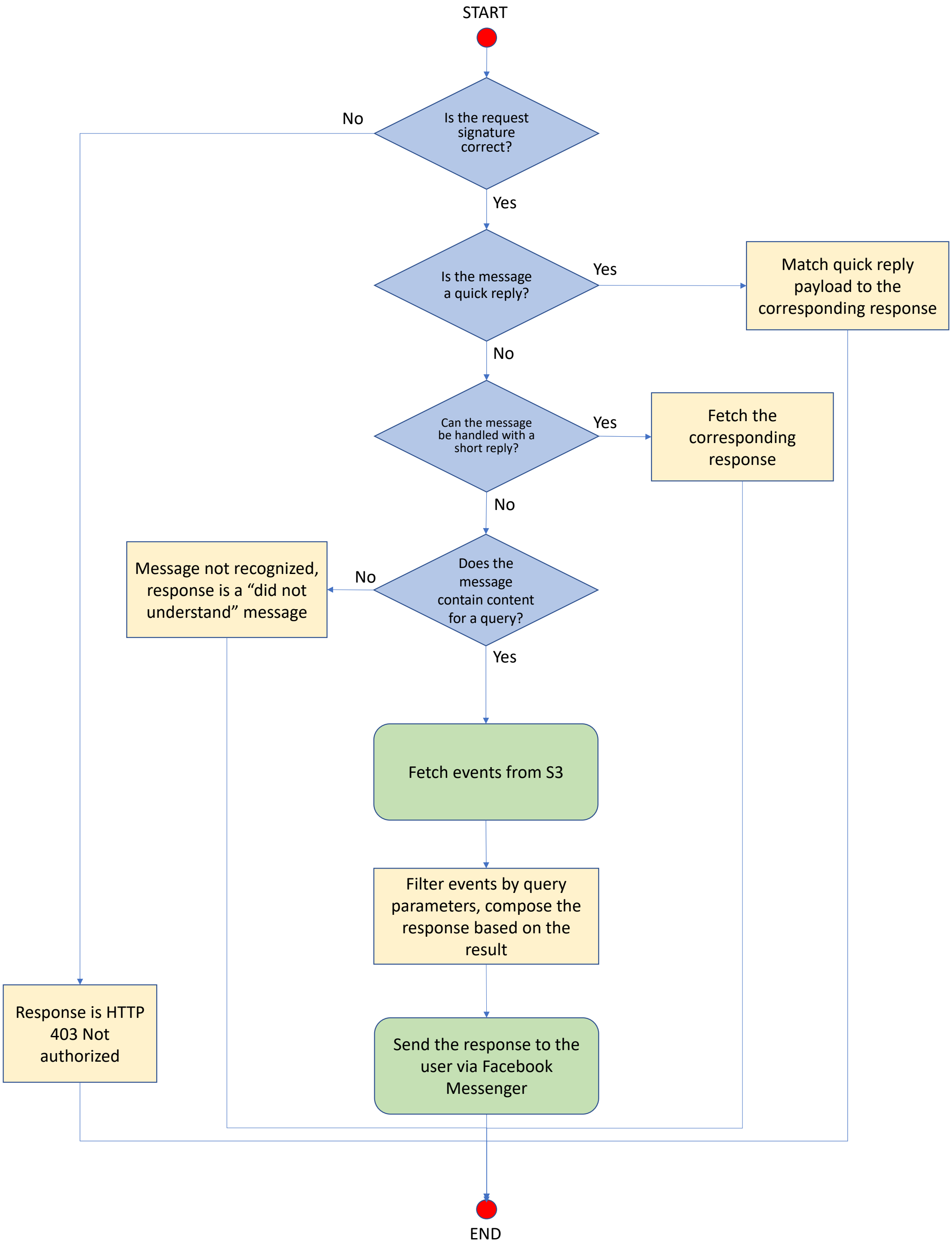

START
Is the request signature correct?
No
Yes
Is the message a quick reply?
Yes
Match quick reply payload to the corresponding response
No
Can the message be handled with a short reply?
Yes
Fetch the corresponding response
No
Does the message contain content for a query?
No
Message not recognized, response is a “did not understand” message
Yes
Fetch events from S3
Filter events by query parameters, compose the response based on the result
Send the response to the user via Facebook Messenger
Response is HTTP 403 Not authorized
END

Figure 4. Flow diagram of the user message processing and response process

This process handles incoming user messages with a series of sequential steps based on how much processing is expected:

- Firstly, the incoming message is verified. Incoming Facebook requests contain a message signature in the payload, which is the entire message body payload which has been processed through a SHA1 hash algorithm with a shared key. The chatbot backend can verify that the incoming message is from Facebook by recreating the signature using the same SHA1 hash process with the message body, as the key is never transmitted, only Facebook's servers and the chatbot backend should know the key to correctly hash the signature. Mismatching signatures causes the message to be discard without further processing, and a HTTP 403 error is returned.

- The next step is to check the message validity. Facebook Messenger supports multiple options which are standard for messaging platforms including delivery receipts, read receipts and postbacks. For this project, only user messages with text context and application-defined quick replies are in scope, all other forms of messages and receipts are discarded by message processing.

- After the message has been verified and validated, the message content is matched using a series of regular expressions to check if the message contains intent which does not require a read request to the persistent storage. This stage handles messages which are conversational in nature and do not require a response with content, for example greetings, giving/receiving thanks and affirmative sentiments. These cues trigger the chatbot to respond with a reply corresponding to the cue and is primarily intended to give the chatbot more of a conversational feeling. Additionally, this section also scans for help request and quick reply payloads which are more functional in nature as they are used to either display the user guide or to allow a predefined conversation chain to progress. While these have a use in guiding users through the chatbot's functionality, it does not require any persistent storage access.

- If the message completes the previous step without matching any quick reply criteria, there is the possibility that this a message with a valid user query. The message content is processed using basic customised NLP algorithms using regular expressions to extract any keywords pertaining to event types or dance styles. Date and time information is already available attached to the message payload from Facebook's built-in NLP feature, thus this lambda is not required to check the message content for date and time. If the message matches any of the regular expressions or contains date and time information from the built-in NLP, this message contains a valid query and will trigger a read request to the persistent storage to read the staged event data. The lambda will then use the query clauses to fetch and filter the event data to find the relevant events, which is returned to the end-user.

- If the message is not identified as a user query, the chatbot will respond with a generic "message not understood" reply.

This process is visualised in Figure 4 in the form of a flow diagram. For the scope of the MVP, each message in the conversation is strictly atomic, thus the end-user is expected to supply all of the arguments required for a search in one message. As the current search filters are time, dance style and event type with time the highest priority filter, this was considered sufficient for the MVP stage of the chatbot.

5.3 Facebook application page

As a part of this project, a page was created on Facebook for the chatbot application. Users access the application through this page, which uses the same medium as normal Facebook public pages. The chatbot is connected to the page's messaging functionality and is set up to respond to any messages sent from users.

This automatic reply functionality is available to any public page on Facebook with a suitable webhook set up to listen for messages, and also supports handover protocol for a page admin to take over the conversation, temporarily suspending the chatbot. As this application should run as a fully automated chatbot, the handover protocol was deemed unnecessary for the current stage of the project. To improve the user experience, the page was given a basic profile with a profile picture, cover picture and an about section detailing the chatbot functionality.

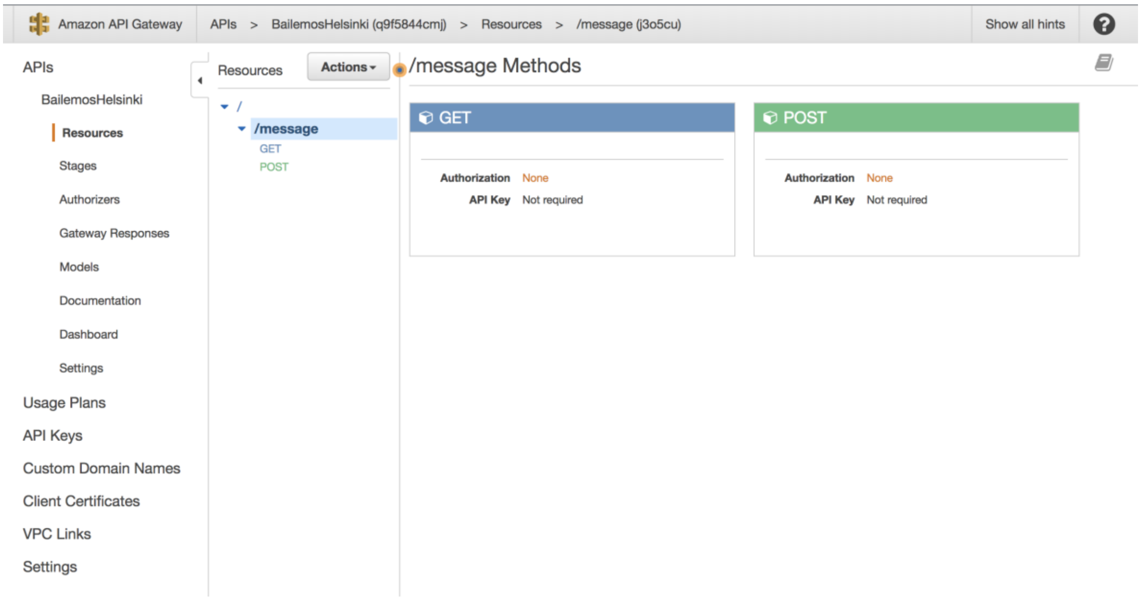
The page will be publicly accessible from open beta onwards from the URL https://www.facebook.com/BailemosHelsinki-124548094901470/.

5.4 Facebook Messenger

The only user-facing section of this application is found in its integration to the Messenger platform. The chatbot application does not need to handle any of the frontend UI as it is handled entirely by the Messenger platform, instead the chatbot only needs to handle receiving, processing and posting messages. This introduces limitations on what can be implemented, as all communication with the user must be performed via the Messenger API and what it offers. All messages sent to and received from the Messenger API are in JSON format, with fields corresponding to the recipient and message payload. The structure of the message payload determines the message type: standard messages have one type of structure, while quick replies have a different payload structure and message templates have a third. For this project, the scope was limited to handling text only: attachments in the form of audio, video, images or files are ignored.

5.5 AWS API Gateway

Once a user leaves a message on the chatbot's Facebook page, the Facebook servers forward the message content to an external webhook configured for the page. From Facebook's perspective, this webhook can be any valid URL which has successfully completed the challenge-response step, Facebook does not need any further knowledge of how the external webhook functions. As the chatbot application uses AWS Lambdas, AWS API Gateway is required as a layer to allow the Lambdas to interface with the incoming payload. Through the use of API Gateway, a REST API was configured for the chatbot. This REST API is designed to be simple, with a single endpoint configured for all integration with Facebook.



/message Methods

GET

Authorization None

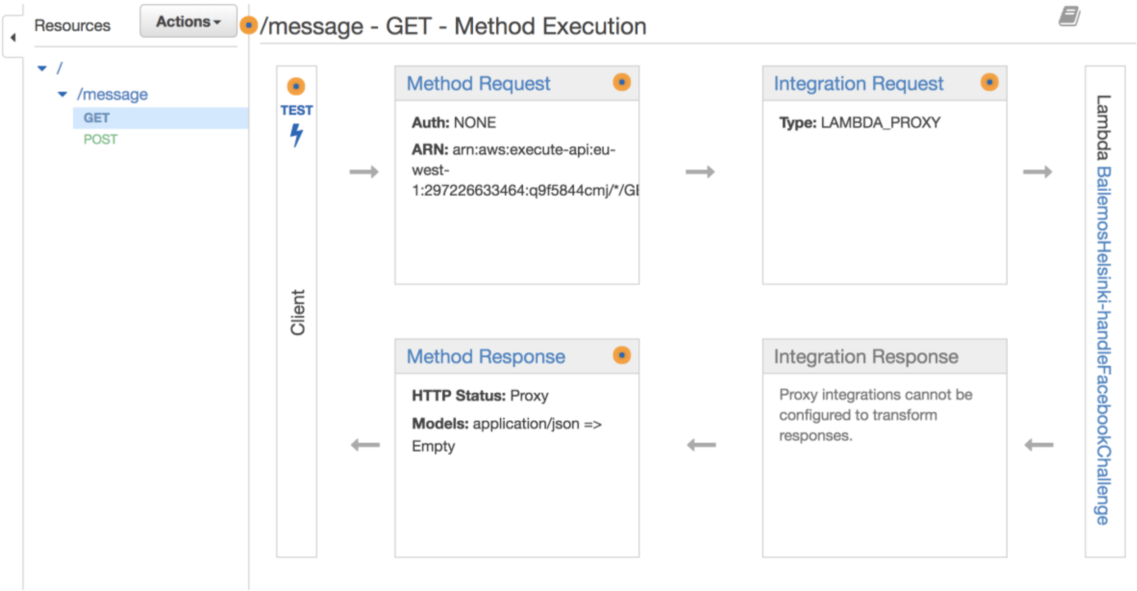
API Key Not required

POST

Authorization None

API Key Not required

Figure 5. AWS API Gateway configuration



/message - GET - Method Execution

Method Request

Auth: NONE

ARN: arn:aws:execute-api:eu-west-1:297226633464:q9f5844cmj/*/GI

Integration Request

Type: LAMBDA_PROXY

Method Response

HTTP Status: Proxy

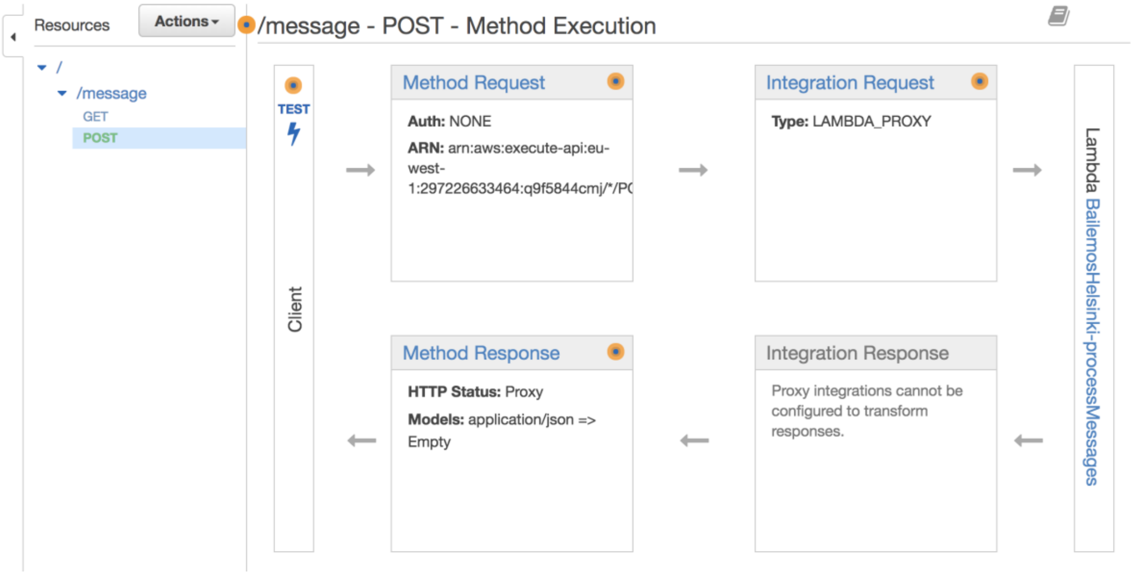
Models: application/json => Empty

Integration Response

Proxy integrations cannot be configured to transform responses.

Lambda BailemosHelsinki-handleFacebookChallenge

Figure 6. AWS API Gateway configuration, /message endpoint GET method



/message - POST - Method Execution

Auth: NONE

ARN: arn:aws:execute-api:eu-west-1:297226633464:q9f5844cmj/*/PC

Type: LAMBDA_PROXY

HTTP Status: Proxy

Models: application/json => Empty

Proxy integrations cannot be configured to transform responses.

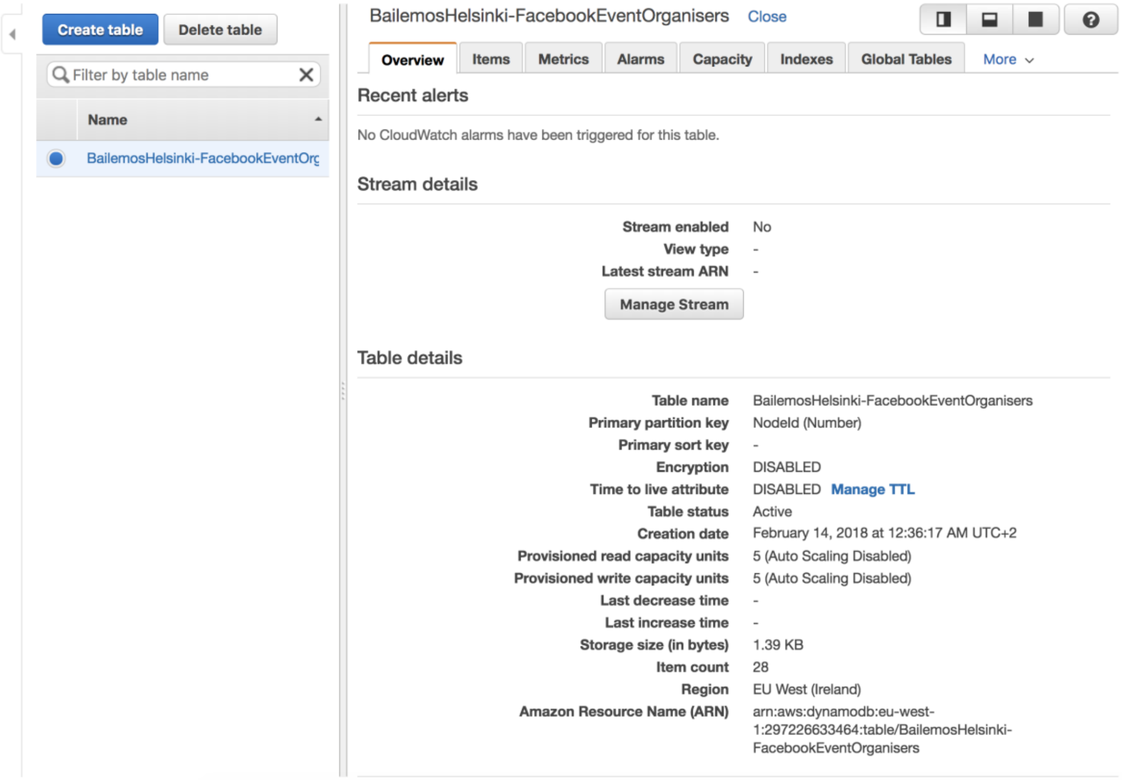
Figure 7. AWS API Gateway configuration, /message endpoint POST method

This setup is due to the way Facebook integration functions: Facebook only allows for a single webhook URL, the challenge-response protocol is sent to the URL as a GET http request, while user messages sent to the page is forwarded to the URL as a POST http request.

Figure 5 shows the setup of the REST API endpoint, with a single /message endpoint hosting the Facebook webhook. Only GET and POST methods are supported, no other method is expected to be required for this endpoint. API Gateway connects the GET and POST requests to this endpoint to the handleFacebookChallenge and processMessages lambdas respectively. Figures 6 and 7 displays the configuration of the GET and POST methods respectively.

5.6 AWS persistent storage

The chatbot application utilises DynamoDB and S3 for its persistent storage requirements. DynamoDB was chosen due to its, while S3 was chosen as a medium which allows for quick implementation.



Recent alerts

No CloudWatch alarms have been triggered for this table.

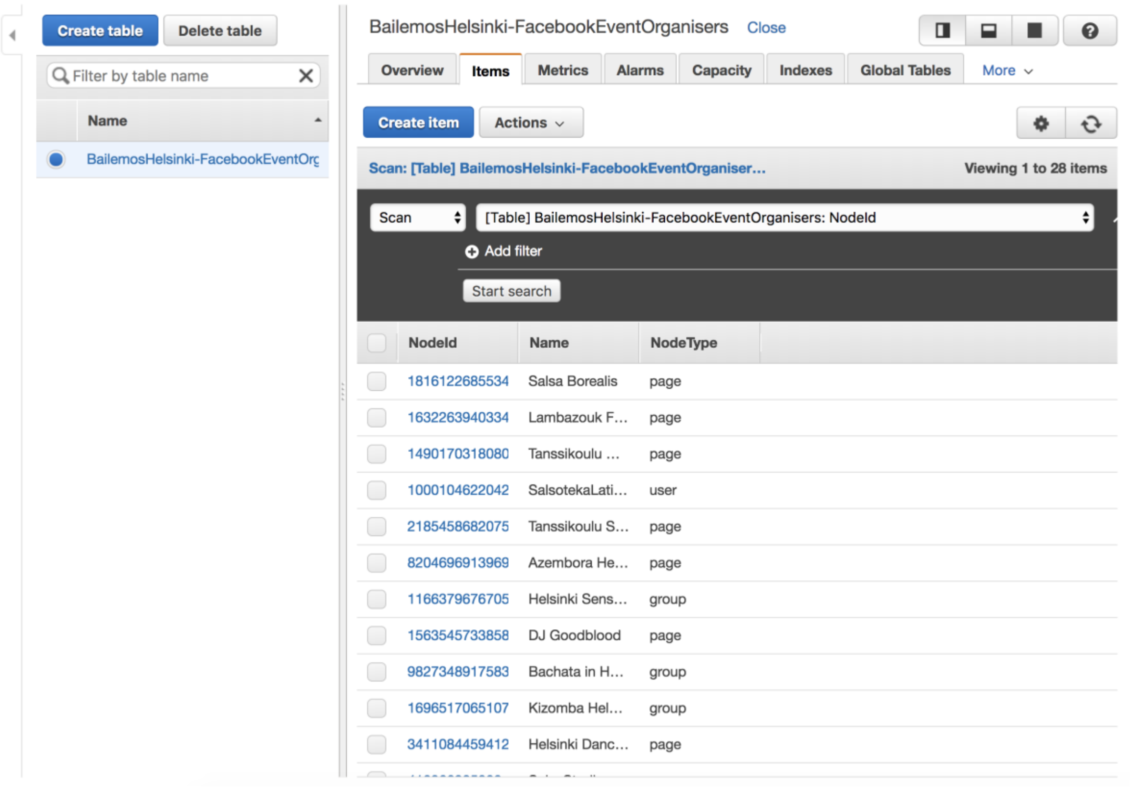
Stream details

Stream enabled	No
View type	-
Latest stream ARN	-

Table details

Table name	BailemosHelsinki-FacebookEventOrganisers
Primary partition key	NodeId (Number)
Primary sort key	-
Encryption	DISABLED
Time to live attribute	DISABLED Manage TTL
Table status	Active
Creation date	February 14, 2018 at 12:36:17 AM UTC+2
Provisioned read capacity units	5 (Auto Scaling Disabled)
Provisioned write capacity units	5 (Auto Scaling Disabled)
Last decrease time	-
Last increase time	-
Storage size (in bytes)	1.39 KB
Item count	28
Region	EU West (Ireland)
Amazon Resource Name (ARN)	arn:aws:dynamodb:eu-west-1:297226633464:table/BailemosHelsinki-FacebookEventOrganisers

Figure 8. AWS DynamoDB BailemosHelsinki-FacebookEventOrganisers table configuration



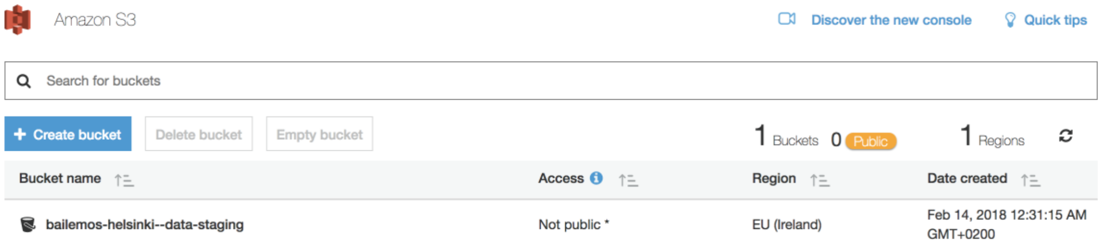
Scan: [Table] BailemosHelsinki-FacebookEventOrganiser... Viewing 1 to 28 items

Scan [Table] BailemosHelsinki-FacebookEventOrganisers: NodeId

NodeId	Name	NodeType
1816122685534	Salsa Borealis	page
1632263940334	Lambazouk F...	page
1490170318080	Tanssikoulu ...	page
1000104622042	SalsotekaLati...	user
2185458682075	Tanssikoulu S...	page
8204696913969	Azembora He...	page
1166379676705	Helsinki Sens...	group
1563545733858	DJ Goodblood	page
9827348917583	Bachata in H...	group
1696517065107	Kizomba Hel...	group
3411084459412	Helsinki Danc...	page

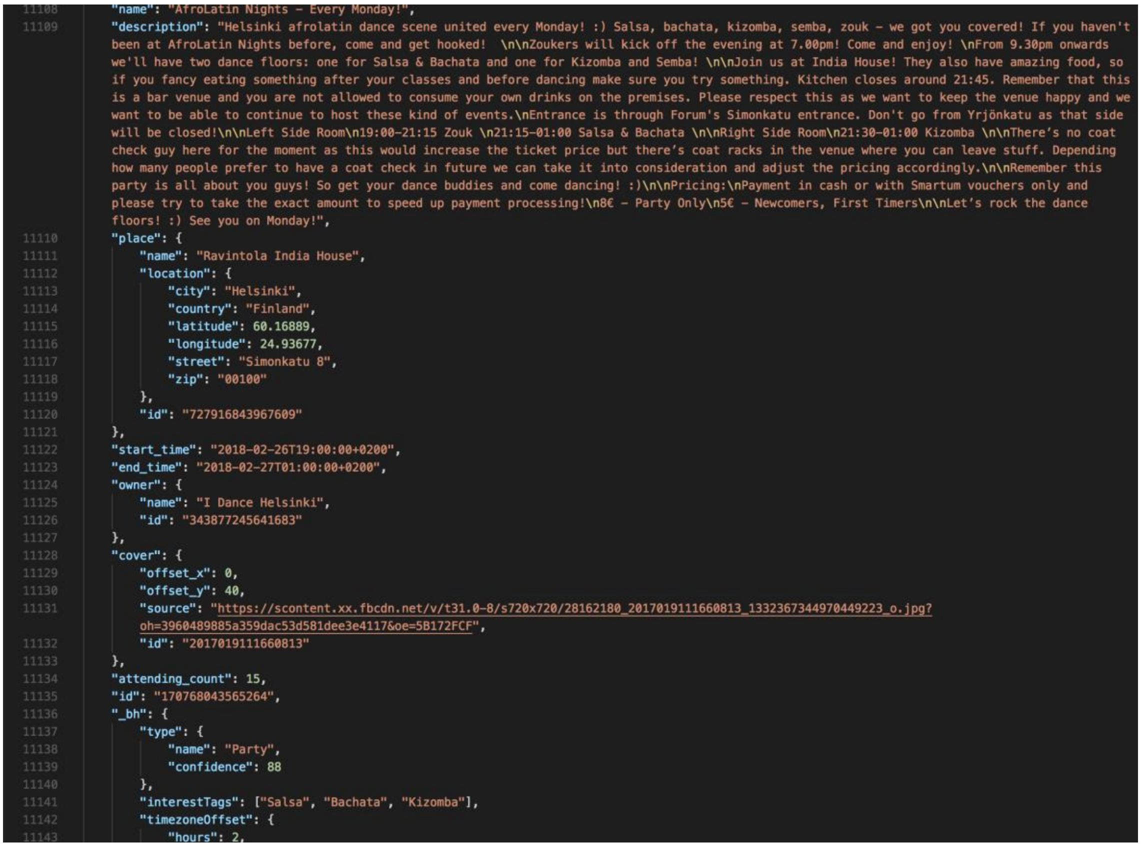
Figure 9. AWS DynamoDB BailemosHelsinki-FacebookEventOrganisers table item contents

DynamoDB contains a single table named BailemosHelsinki-FacebookEventOrganisers which is maintained manually. This table contains the Facebook IDs of known event organisers and dance schools affiliated with the Afro-Latin dance community in the Helsinki region, which is used to query the Facebook Graph API for data staging. Read access to this table is only required during the execute of the stageEventData lambda, no other lambda requires this table. Figure 8 illustrates the DynamoDB table configuration, and Figure 9 displays the format of the items found within.



Bucket name	Access	Region	Date created
bailemos-helsinki--data-staging	Not public *	EU (Ireland)	Feb 14, 2018 12:31:15 AM GMT+0200

Figure 10. Configuration of the AWS S3 bucket bailemos-helsinki—data-staging



```
    "name": "AfroLatin Nights - Every Monday!",
    "description": "Helsinki afrolatin dance scene united every Monday! :) Salsa, bachata, kizomba, semba, zouk - we got you covered! If you haven't
    been at AfroLatin Nights before, come and get hooked!  \n\nZoukers will kick off the evening at 7.00pm! Come and enjoy! \nFrom 9.30pm onwards
    we'll have two dance floors: one for Salsa & Bachata and one for Kizomba and Semba! \n\nJoin us at India House! They also have amazing food, so
    if you fancy eating something after your classes and before dancing make sure you try something. Kitchen closes around 21:45. Remember that this
    is a bar venue and you are not allowed to consume your own drinks on the premises. Please respect this as we want to keep the venue happy and we
    want to be able to continue to host these kind of events.\nEntrance is through Forum's Simonkatu entrance. Don't go from Yrjönkatu as that side
    will be closed!\n\nLeft Side Room\n19:00-21:15 Zouk \n21:15-01:00 Salsa & Bachata \n\nRight Side Room\n21:30-01:00 Kizomba \n\nThere’s no coat
    check guy here for the moment as this would increase the ticket price but there’s coat racks in the venue where you can leave stuff. Depending
    how many people prefer to have a coat check in future we can take it into consideration and adjust the pricing accordingly.\n\nRemember this
    party is all about you guys! So get your dance buddies and come dancing! :)\n\nPricing:\nPayment in cash or with Smartum vouchers only and
    please try to take the exact amount to speed up payment processing!\n8€ - Party Only\n5€ - Newcomers, First Timers\n\nLet’s rock the dance
    floors! :) See you on Monday!",
    "place": {
        "name": "Ravintola India House",
        "location": {
            "city": "Helsinki",
            "country": "Finland",
            "latitude": 60.16889,
            "longitude": 24.93677,
            "street": "Simonkatu 8",
            "zip": "00100"
        },
        "id": "727916843967609"
    },
    "start_time": "2018-02-26T19:00:00+0200",
    "end_time": "2018-02-27T01:00:00+0200",
    "owner": {
        "name": "I Dance Helsinki",
        "id": "343877245641683"
    },
    "cover": {
        "offset_x": 0,
        "offset_y": 40,
        "source": "https://scontent.xx.fbcdn.net/v/t31.0-8/s720x720/28162180_2017019111660813_1332367344970449223_o.jpg?
        oh=3960489885a359dac53d581dee3e4117&oe=5B172FCF",
        "id": "2017019111660813"
    },
    "attending_count": 15,
    "id": "170768043565264",
    "_bh": {
        "type": {
            "name": "Party",
            "confidence": 88
        },
        "interestTags": ["Salsa", "Bachata", "Kizomba"],
        "timezoneOffset": {
            "hours": 2,
```

Figure 11. Example content of the JSON file containing the staged event data, displayed in VSCode for Mac

S3 is used to store the event data after querying the Facebook Graph API. The response is analysed and then aggregated to a single JSON file. Figure 10 shows the S3 bucket bailemos-helsinki--data-staging which contains the event data used by this application stored under events.json. An example of the content of events.json is shown in Figure 11. This bucket is capable of storing more than this single file, so it is scalable for future development: if additional data requires staging, it is expected to be placed in this bucket.

5.7 AWS Lambda

The chatbot application consists of three distinct lifecycles which are implemented as separate lambdas: application registration, data update and message handling. The data update process is designed to execute automatically at set times and cannot be triggered by public actions.

The application registration and message handling can be accessed from a public URL which is connected to the corresponding Facebook app. Due to the constraints of the Facebook webhook API, both processes are registered to the same URL, which is connected to an API Gateway endpoint which forwards the HTTPS GET requests to the application registration process, while HTTPS POST requests are routed to the message handler process.

All lambdas created for this project utilises a Node.js environment running version 6.10, which is the most recent version available on the AWS Lambda platform. Due to this environment, all lambda code is written in JavaScript with partial implementation of ES6 features.

Additionally, all of the lambda source code follow a unified template with the following points:

- Based on suggested lambda best practices, the handler method which is used as an entry hook only handles the connection between the Node.js environment and the application logic. Application logic is handled in a separate function. ???

- All of the entry hook files are named main.js to make it clear that it is the main entry file.

- Source code logic is split into separate modules where appropriate. Following Clean Code best practices, all of the modules are focused on having one general type of functionality [?]. All logic related to Facebook interfacing, including generating appropriate HTTPS requests and message payload structures, are placed in the folder named facebook and all modules are named with the template facebook*.js where * signified the purpose of the module. Similarly, chatbot-specific logic related to message processing, analysis and response generator are stored grouped together in the folder botty and the modules are named botty*.js where * signifies the functionality of the module. Generic reusable components are stored in the utils folder with a template of *utils.js where * displays the domain of the utility file.

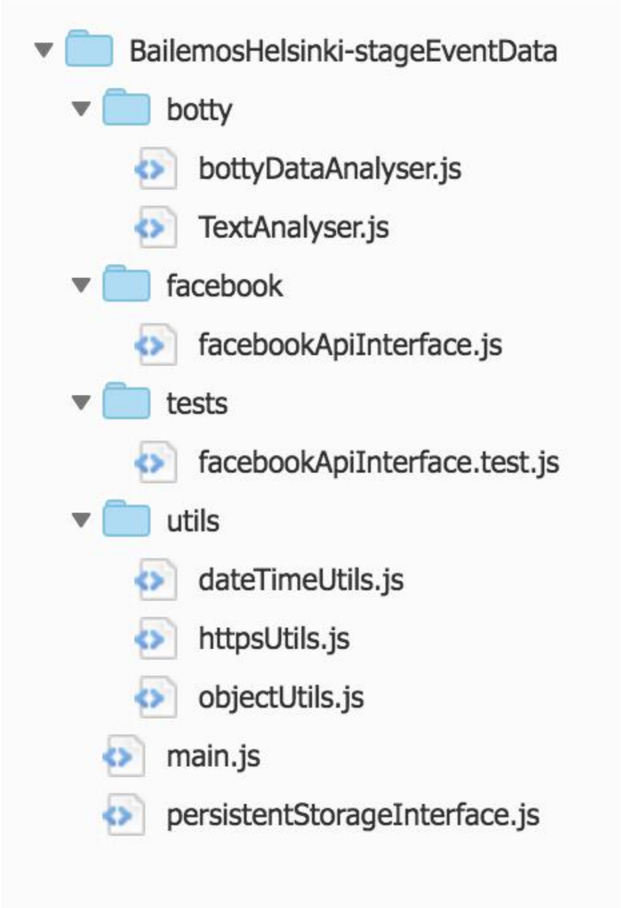
5.7.1 handleFacebookChallenge

This lambda is used to handle the challenge-response authentication process when connecting the webhook to Facebook. As this lambda corresponds to a simple and straightforward process, the lambda code consists of a single file and does not require splitting into modules as the lambda consists of only 50 lines of code.

The lambda logic for handleFacebookChallenge is a straightforward process of extracting the hub.verify_token and hub.challenge parameters from the incoming GET request. The incoming verification token is compared to the one defined locally in the lambda environmental variables, if they match the lambda returns the challenge value in a response with a 200 status code. If the verification token does not match, the lambda returns an error response with a status code of 422, signifying a malformed or unprocessable request, and if the correct parameters could not be found in the request this lambda responses with a status code of 500 to signal that a generic error occurred.

5.7.2 stageEventData

This lambda corresponds to the data staging process for event data fetched from Facebook.

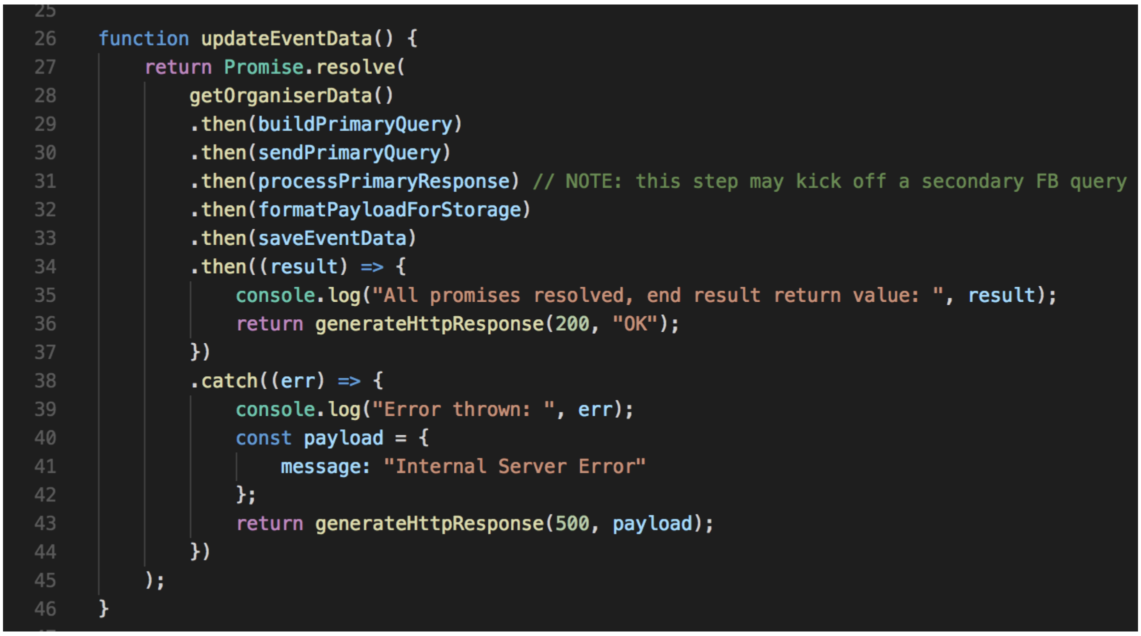


```
▼ BailemosHelsinki-stageEventData
    ▼ botty
        bottyDataAnalyser.js
        TextAnalyser.js
    ▼ facebook
        facebookApiInterface.js
    ▼ tests
        facebookApiInterface.test.js
    ▼ utils
        dateTimeUtils.js
        httpsUtils.js
        objectUtils.js
    main.js
    persistentStorageInterface.js
```

Figure 12. Source code structure of the stageEventData lambda

This lambda has multiple responsibilities and functionality, which necessitates the structuring and organising of the code into separate modules. Figure 12 shows the structure of the lambda code, which has been split into folders and code modules based on responsibilities. The folders illustrates the general categories, and the module names reflect their responsibilities: bottyDataAnalyser.js and textAnalyser.js analyses the event data, facebookApiInterface.js is responsible for formatting and sending the requests to the Facebook Graph API, persistentStorageInterface.js is used to make calls to DynamoDB and S3, and the main.js module is used to coordinate the modules. Finally, the utils folder contains modules with general utilities functions used by the other modules.

This process logic uses both synchronous and asynchronous method calls which are always processed in a sequential and linear fashion. Due to this pattern the code makes heavy use of JavaScript Promises which allows each method to executed in a sequential order regardless of whether it is a synchronous or asynchronous method.



```
function updateEventData() {
    return Promise.resolve(
        getOrganiserData()
        .then(buildPrimaryQuery)
        .then(sendPrimaryQuery)
        .then(processPrimaryResponse) // NOTE: this step may kick off a secondary FB query
        .then(formatPayloadForStorage)
        .then(saveEventData)
        .then((result) => {
            console.log("All promises resolved, end result return value: ", result);
            return generateHttpResponse(200, "OK");
        })
        .catch((err) => {
            console.log("Error thrown: ", err);
            const payload = {
                message: "Internal Server Error"
            };
            return generateHttpResponse(500, payload);
        })
    );
}
```

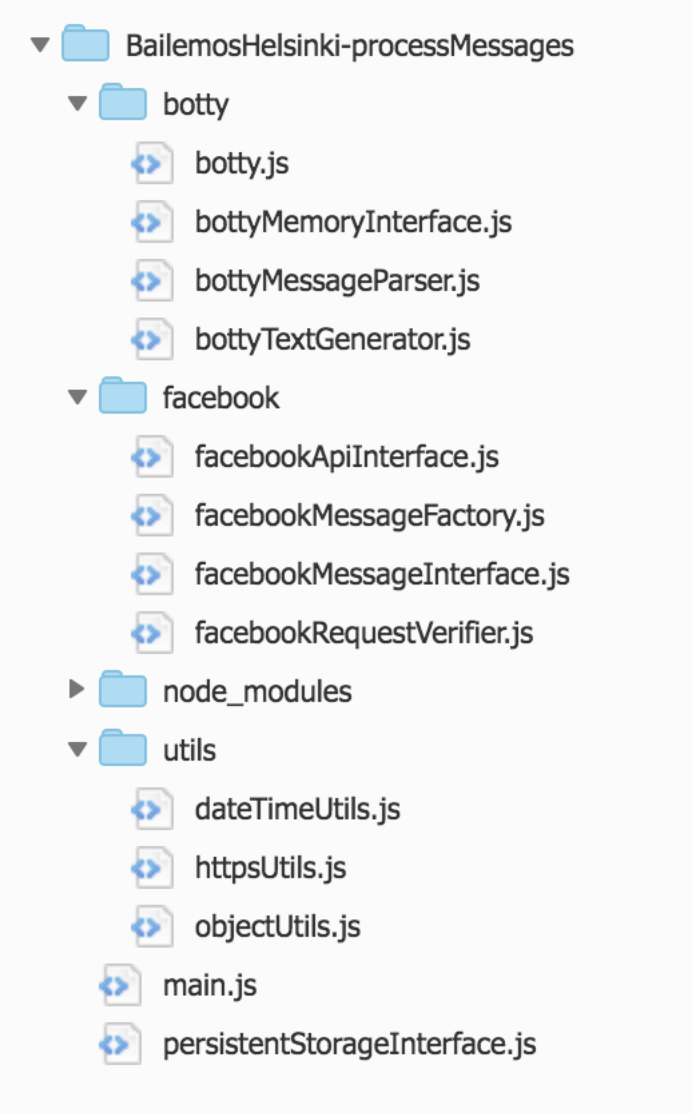
Figure 13. Source code extract from stageEventData, example of promise chaining

As detailed in the MDN, promises represent the eventual completion or failure of an asynchronous operation [20]. Additionally, promise resolution occurs after the current event loop has completed, thus even for synchronous functions promise resolution calls can be trusted to only fire after the synchronous function code has completed. These concepts allow promises to fit the requirements of sequential code which execute in a defined order even if the code depends on asynchronous calls.

Figure 13 illustrates the promise chain used in stageEventData. Each function in this chain returns a promise, which allows each methods to be called in sequence through the use of the .then() method which is only executed once the preceding promise has been completed. Synchronous functions can return a promise through the use of the static Promise.resolve and Promise.reject methods which represent a promise in resolve or reject states, returning these allow synchronous functions to behave in the same manner as asynchronous functions and allows the promise chaining displayed in figure X to mix lambda processing, AWS component calls and external HTTPS request calls.

5.7.3 processMessages

This lambda corresponds to the user message processing and response process. The functionality in this lambda consists of both synchronous and asynchronous AWS cross-component calls and requests to Facebook Messenger, and as a result of this the code structure makes heavy use of promise chaining to connect these elements together.



```
▼ BailemosHelsinki-processMessages
  ▼ botty
      botty.js
      bottyMemoryInterface.js
      bottyMessageParser.js
      bottyTextGenerator.js
  ▼ facebook
      facebookApiInterface.js
      facebookMessageFactory.js
      facebookMessageInterface.js
      facebookRequestVerifier.js
  ▶ node_modules
  ▼ utils
      dateTimeUtils.js
      httpsUtils.js
      objectUtils.js
    main.js
    persistentStorageInterface.js
```

Figure 14. source code structure of the processMessages lambda

As this lambda handles all user input which can require different handlers in addition to interfacing with S3 and Facebook Messenger, this corresponds to a higher level of complexity. As a result of this, the source code of this lambda is split across multiple modules each corresponding to a single purpose. Figure 14 illustrates the source code structure. The core user message processing is performed in botty.js, bottyMessageParser.js and bottyTextGenerator.js which performs message input/output, parsing and generating appropriate responses respectively. Access to S3 is handled by persistentStorageInterface.js, which is a stripped-down version of the module implemented in the stageEventData lambda. Facebook interfacing is handled in the modules facebookApiInterface.js, facebookMessageFactory.js, facebookMessageInterface.js and facebookRequestVerifier.js which are used to interface with the Facebook API, create relevant message data structures and verify the incoming request as a valid request coming from Facebook. These modules are coordinated by the main.js and botty.js modules, which receive incoming requests and handles calls between the modules implemented in this lambda.

One functionality challenge which had to be solved in this lambda was the mapping of user content specifically relating to dance vocabulary: dance styles and dance event types are not given any relevance in Facebook’s built-in NLP model. As the MVP does not include training a customised Wit.ai bot, this challenge was solved through regular expressions and pattern matching as this was considered a simple implementation which would suffice as a starting point. Additionally, for the majority of expected use cases the queries are expected to be simple with a mandatory date range, while dance styles and event types are optional.

An additional customised NLP layer was implemented for detecting dates. This is partially handled by Facebook’s built-in NLP layer, however further post-processing is required to fully translate the NLP result to a date instance. A common use case handled by this lambda is where user query input is in the form of relative date references or ranges, such as “this Friday” or “next week”: the NLP layer can understand this content and will render the result as a single timestamp with a granularity level, however this result requires additional post-processing to turn it into a date range usable by the chatbot’s processing logic.

To handle these use cases this lambda uses the Moment.js library to translate user date input into JavaScript date instances. The Moment.js library implements string parsing methods which can convert imprecise user input into precise date instances, and additionally contains date manipulation methods which allows the application to convert the NLP results to date ranges which can be used in date filtering. This lambda makes heavy use of the startOf and endOf methods, which are used to delimit the start and end of a unit of time, usually a day, week or month.

5.8 AWS Cloudwatch

All components implemented on the AWS platform for this project are integrated to AWS Cloudwatch, which is used to log all actions which require the AWS components implemented in this project. The primary focus of the logging is lambda execution and results. For the handleFacebookChallenge lambda, only the incoming verification token and incoming URL are logged. For the stageEventData lambda, the success or failure of the lambda is logged, the event data is not required in the logs as the results can be checked in S3.

For the processMessages lambda, the inbound and outbound message payloads are logged. Considering the Europe-wide GDPR Act and the decision to avoid using any Facebook feature which requires explicit user permissions, it was decided that, where possible, logging should also avoid saving any personally identifiable data. As a result of this, no user data is requested, and user IDs are not stored to logs. This does not prevent private information from ending up in the logs if they are communicated within the message text content, however this is ultimately up to the user’s discretion. Additionally, the logs for processMessages are only stored for two weeks, any logs older than this threshold is automatically removed.

6 Discussion

6.1 Review

The chatbot application has finished the main alpha development stage and is ready to progress to the closed beta stage, with access given to volunteer users involved in the Helsinki Afro-Latin dance community in Helsinki. The goal of the closed beta stage will be to test the Lambda, DynamoDB and S3 load generated by a small number of users, receive user queries to collect data on likely NLP modifications and to get feedback on the user experience and viability of the application.

Overall, the chatbot’s progress has been promising: the development of the minimum viable product spanned approximately three months and the feedback from the customer has been positive throughout the project. Although the chatbot concept itself is relatively new, the technologies and tools used by this project are mature with proven track records. Facebook’s developer platform has been in existence for many years with in-depth documentation and user guides. Similarly, as of 2017 AWS controls 35% of the cloud computing market [13] in a highly competitive market filled with numerous known companies including Microsoft, Google and IBM, which creates an incentive to provide quality documentation and guides. These technology choices aided development by allowing this project to avoid extra work setting up dedicated servers, and the choice of AWS is ideal for handling scaling issues in the future.

6.2 Challenges and solutions

Although the original goal of this project was met, it was not without some challenges. One particular major impediment to the project was the deployment of version 2.12 of the Facebook Graph API in January 2018. This version brought significant changes to the data returned by events and groups: previously the majority of the fields in event and group nodes were considered public and did not require additional authentication from other parties, however from version 2.12 onwards a large number of fields were changed to be private, even for public groups and events. The chatbot’s functionality was affected and required a partial redesign and rewrite to accommodate the smaller scope available from purely public data. The API version change highlighted the possibility that in the future, if the scope of data public data available shrinks further, the chatbot’s affiliation model may need to change from the current “opt-out” style where event organisers may be passively collated to the organiser list without requiring their express permission, to an “opt-in” model where event organisers must first be aware of this chatbot application and ask to be included and grant the application full data visibility. Regardless, it is still preferred if end-users are always free to use this service without needing to be authenticated or allow application permissions as requiring user permissions is expected to be a barrier to entry for this app.

An additional challenge to this project was the events of the Cambridge Analytica incident of March 2018. Due to this incident, Facebook temporarily disabled the Graph API for applications which have not yet been published, thus delaying the closed beta phase further. This sudden disruption to the data sources bring into question the viability of Facebook itself as a platform for the dance community and the challenges which may lie ahead: the main strength of Facebook is its popularity amongst the user base which makes it the best option available for outreach and advertisement. If this were to change, then alternative solutions may be required, which would also affect the future of this project.

6.3 Future Development

There are three points of improvements identified at the end of the alpha stage as possible points of improvement which will be dependent on the findings of the closed beta stage.

6.3.1 Data staging improvement: changing the persistent storage to cope with scaling

The state of the chatbot during the project was capable of handling the user loads during the closed beta period, and is projected to be sufficient for small user loads of people in Helsinki. However, the use of a single JSON file stored in S3 is a design choice which has been identified as a likely future weak point. While this choice of persistent storage medium is sufficient for use during the project, it will become relatively inefficient as the scope and scale of the application increases, as the current design requires the entire data collection to be fetched for every request. This will become an issue as the size of the JSON file increases, which can occur as more dance schools and organisers contribute to the data, or as the timescale of stored data increases. Additionally, an increasing volume of requests from a higher user count can also contribute to the load.

To futureproof the chatbot from these risks, there are three proposed solutions which can be implemented for this project. The first is to stay with the S3 storage, but partition the data by timestamp to different JSON files. This is the simplest to implement, however from the S3 perspective every batch run will still require a complete reloading of every data JSON, which may remain suboptimal for larger data volumes. The second option is to switch the data storage to a relational database via AWS RDS, which offers multiple RDSMS options including Amazon Aurora, MySQL, MariaDB, PostgreSQL and others.

This offers the flexibility of a relational database and allows data to be queried on multiple columns, which will allow the application to support more complex data queries. However, as of the time writing this is also the most expensive option per GB of storage. The last option is to move the storage to DynamoDB, using multiple tables to segregate the data by timestamp to solve the issue of data read/write volume and access frequency. This option strikes a balance between cost and technical flexibility, but the implementation is the most challenging of the options as DynamoDB is most suited for supporting random data access. This is projected to not be the case, as the closed beta analytics already shows that user queries tend to be more closely clustered towards events occurring in the next two weeks, with the upcoming weekend the most popular query.

All of these options present solutions to the scaling issue, however each solution offers different strengths and weaknesses in technical implementation and cost. It is expected that one of these options will be chosen based on the projected user loads and use cases based on open beta data.

6.3.2 Improved NLP: incorporating Wit.ai

As the user base is primarily situated in Helsinki, it is expected that the majority of users will have Finnish as their native language. While it is expected that users are also fluent in English, it is also desirable to be able to localise the chatbot.

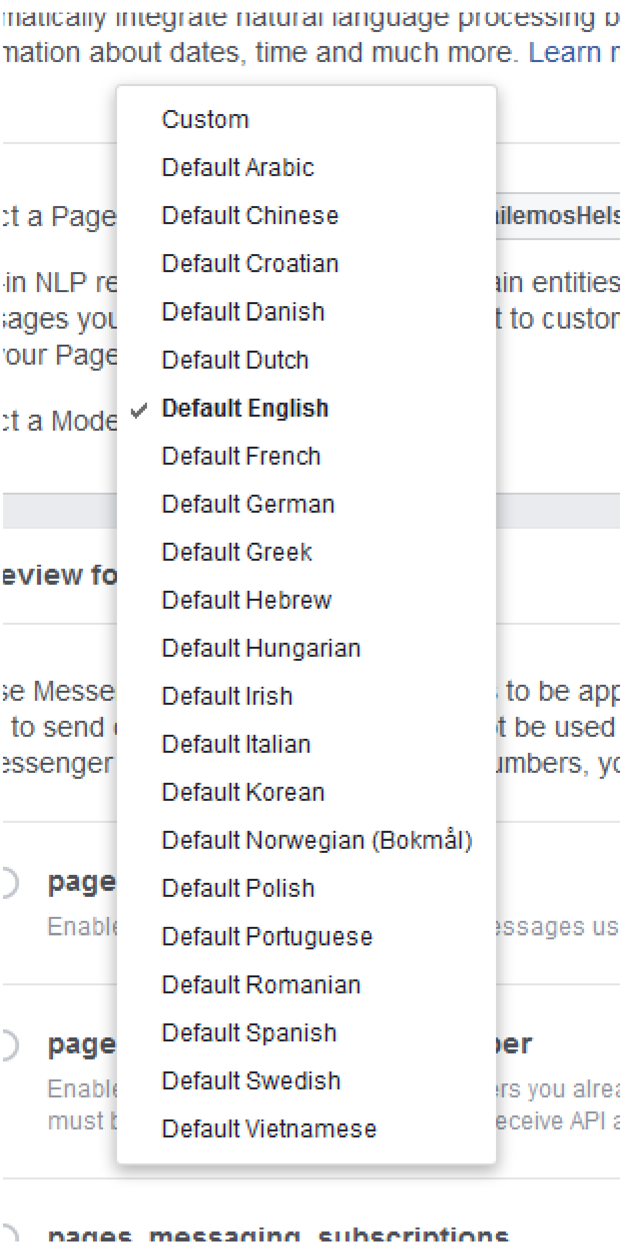


Figure 15. Language options available in the Facebook built-in NLP.

During the project’s development, Facebook’s default NLP solution was implemented as it allowed a part of the text analysis to be handled by viable 3rd party apps. Figure 15 shows the option screen of the Facebook built-in NLP: this screenshot displays two weaknesses of this solution: it only supports one language at a time, and Finnish is not available as an option.

To support the need to localise this app, Wit.ai will be explored as a solution. The Messenger platform offers Wit.ai integration by default, which can be used to fulfil the need for multi-language support including Finnish (and possibly Swedish). Additionally, Wit.ai can be used to handle specific nuances which deviant from standard (US) English which are not available from Facebook's default English settings, including the Scandinavian concept of "next day" such as "next Friday" which often refers to the Friday of next week. Additionally, analysis specific dance community jargon can be offloaded from the lambda logic into Wit.ai, reducing the resource requirements of the processMessages lambda.

6.3.3 User customisation: implementing a session memory

The current state of a conversation with the chatbot is limited by the fact that each message is processed in an atomic manner, without any connection with previous messages by the same user. While this is sufficient for most simple queries, this limitation means that users must specify an entire query in one message and does not support the possibility for spreading query conditions across multiple messages, which was noticed during the closed beta test. Additionally, the lack of memory of previous messages leads to a restriction in the results returned by a query: as Messenger generic message templates have a built-in limit of 10 templates per message, this requires that all queries must result in 10 of fewer entries, otherwise some results are lost.

Both of these issues can be solved by implementing a session memory for the chatbot: message chains can be stored in persistent memory to allow conversations and queries to be staggered across multiple messages, while queries which return more than 10 results can have the additional results cached in persistent memory, with the chatbot returning an additional quick reply payload for the user to access the next set of results.

7 Conclusion

The goal of this project was to create a minimum viable product level of a Facebook Messenger chatbot, with the intention of eventually releasing it for live use. The goal was met, over the course of this project a functioning chatbot application fulfilling the customer's specifications and acceptance criteria was developed.

The chatbot is expected to move to its next release phase, a closed beta release. For this phase selected volunteers from the intended audience, namely active dancers in the

Helsinki Afro-Latin dance scene, will be invited to test the chatbot’s features and give feedback on the user experience and identify any further development which should be considered before progressing to an open beta phase.

Overall, the project has shown promise and it is expected to be a helpful tool for event organisers and dancers alike. A major strength of this project’s approach is that as a Facebook chatbot, it is already integrated on a platform which users are already familiar with and this chatbot offers a context-specific way for users to search for dance events that interest them. A common theme found in comments from dancers have been how difficult it is to find or track information on events due to the sheer volume of events of all kinds on Facebook, not just specifically dance or specifically the exact dance styles the user is interested in. By creating a tool to help dancers discover what is happening around them, the hope is that dancers will find it easier to join events, which increases the number of people at events and generates more revenue for the organisers to further improve their services. Additionally, in the longer term the increased traffic will hopefully help the community grow in number and improve in skill, as the overall health of the dance community is heavily dependent on keeping a critical mass of active members.

References

1 Matt Schlicht. The Complete Beginner's Guide to Chatbots – Chatbots Magazine [online]. URL: https://chatbotsmagazine.com/the-complete-beginner-s-guide-to-chatbots-8280b7b906ca. Accessed 26 March 2018.

2 Jiaqi Pan. Conversational Interfaces: The Future of Chatbots – Chatbots Magazine [online]. URL : https://chatbotsmagazine.com/conversational-interfaces-the-future-of-chatbots-18975a91fe5a. Accessed 30. March 2018.

3 Henk Pelk. Chatbots: a bright future in IoT? [online]. URL: https://itnext.io/chatbots-a-bright-future-in-iot-f9f6cc8e12a9. Accessed 26 March 2018.

4 Kevin Scott. Usability Heuristics for Bots – Chatbots Magazine [online]. URL: https://chatbotsmagazine.com/usability-heuristics-for-bots-7075132d2c92. Accessed 26 March 2018.

5 Timmy McCross. Natural Language Processing – Chatbot News Daily [online]. URL: https://chatbotnewsdaily.com/natural-language-processing-328aa760f8ce. Accessed 3 April 2018.

6 Henk Pelk. Natural Language Processing and Machine Learning [online]. 10 February 2017. URL: https://chatbotsmagazine.com/natural-language-processing-and-machine-learning-the-core-of-the-modern-smart-chatbot-8755c6343fa5. Accessed 26 March 2018.

7 Facebook. Facebook - About [online]. 2018. URL: https://www.facebook.com/pg/facebook/about/. Accessed 18 March 2018.

8 Pauli Reinikainen. Suomen somekäyttäjien määrät julki: Twitter on ”elitistinen”, Instagram ja Whatsapp nousevat kohtisten [online]. 25 August 2017. URL: https://www.yrittajat.fi/uutiset/562158-suomen-somekayttajien-maarat-julki-twitter-elitistinen-instagram-ja-whatsapp-nousevat. Accessed 26 March 2018.

9 Facebook. Terms of Service [online]. 31 January 2018. URL: https://www.facebook.com/legal/terms. Accessed 30 January 2018.

10 Facebook. Facebook Platform Policy [online]. 2017. URL: https://developers.facebook.com/policy. Accessed 30 January 2018.

11 Facebook. Overview – Graph API [online]. 2017. URL: https://developers.facebook.com/docs/graph-api/overview/. Accessed 26 March 2018.

12 Facebook. Facebook Messenger Platform – Built-in NLP [online]. 2017. URL: https://developers.facebook.com/docs/messenger-platform/built-in-nlp. Accessed 30 January 2018.

13 Ron Miller. AWS continues to rule the cloud infrastructure market | TechCrunch [online]. 2017. URL: https://techcrunch.com/2017/10/30/aws-continues-to-rule-the-cloud-infrastructure-market/. Accessed 22 March 2018.

14 Amazon. AWS Lambda – Product Features [online]. 2018. URL: https://aws.amazon.com/lambda/features/. Accessed 18 March 2018.

15 Amazon. What is NoSQL? – Amazon Web Services (AWS) [online]. 2018. URL: https://aws.amazon.com/nosql/. Accessed 18 March 2018.

16 Amazon. Amazon Simple Storage Service (S3) – Cloud Storage – AWS [online]. 2018. URL: https://aws.amazon.com/s3/faqs/. Accessed 18 March 2018.

17 Amazon. AWS API Gateway – Product Features [online]. 2018. URL: https://aws.amazon.com/api-gateway/details/. Accessed 18 March 2018.

18 Amazon. Amazon Cloudwatch – Cloud & Network Monitoring Services [online]. 2018. URL: https://aws.amazon.com/cloudwatch. Accessed 18 March 2018.

19 Amazon. AWS Free Tier [online]. 2017. URL: https://aws.amazon.com/free/. Accessed 15 November 2017.

20 MDN. Promise – JavaScript | MDN [online]. 2005-2018. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed 30 March 2018.