UiT THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Making your devices speak

Integration between Amazon Alexa and the Managed IoT Cloud

—

Thomas Holden

Master thesis in INF-3981 June 2018

Abstract

Speech recognition and communication between humans and machines are increasingly popular today. Several companies already have products in this market segment.

The Managed IoT Cloud (MIC) platform is a complete ecosystem for management of Internet of Things (IOT) devices, data storage and analysis of data. However, the platform lacks an integration with a personal assistant to introduce voice control of the connected devices.

This study is about integrating Amazon Alexa and the MIC platform, with the aim of bringing voice control to the connected devices.

AlexaMIC is the result of this study. AlexaMIC supports querying and setting the current state of devices that are connected to the MIC platform using the voice commands that are defined in the architecture chapter of this thesis. In addition, AlexaMIC gives users the ability to name and group devices, and perform group operations, such as listing all of the devices associated with a group, and querying the current state of all the group members

Evaluations conducted in this study shows that neither Alexa or the MIC platform fulfills all the requirements for a good experience using speech recognition. Generally, the lack of metadata support in MIC, and the fact that Alexa utilizes syntactic instead of contextual speech recognition, creates a few issues in terms of the user friendliness of the application. Examples of issues that arose from this are that utterances must have invocation names, and that it is hard to think of every way a user might ask to turn off the light.

Speech recognition is a useful service. However, improvements are needed in terms of language understanding and context awareness.

Thanks to Kine Mari Karlsen, Arne Munch Ellingsen and Anders Andersen for their guidance throughout the work on this study.

Contents

List of Figures

1 Introduction

Speech recognition and communication between humans and machines are services which are increasingly popular today[8]. Several companies already have different smart products released to the market. For instance, Hue is a smart bulb designed by Phillips. The product is integrated with Apple Homekit and the smart assistant Siri, which introduces the possibility of controlling the behavior of the light using voice commands. An integration between a speech recognition service, a personal assistant, and an Internet of Things (IoT) device is what makes this possible. These products are popular because they simplify and automate tasks which potentially are tedious and repetitive. For example, consider an apartment where all parts of the house are connected to a smart assistant system. It would be as simple as asking the system to enable night mode, to turn off all the lights, lock all doors and windows, and turn on the security alarm.

The MIC platform is an offering by Telenor, created to help accelerate innovation in the field of IoT. The platform is an ecosystem for devices, supporting device provisioning, connectivity, data storage and inspection of measurement data, using a customizable dashboard, named the appboard. However, an important aspect of the ecosystem is missing. The platform does not support voice control of the connected devices. Alexa is a service offered by Amazon, which could be used to introduce voice control to the MIC platform.

Alexa supports custom integrations with third party service, such as MIC, through a feature known as custom skills.

1.1 Problem statement

The problem definition of this study is ‘to implement an integration between MIC and Alexa, to introduce voice control of the connected devices‘. Thus, the goal of this study is to develop and test a prototype which integrates Alexa and the MIC platform.

The prototype is named AlexaMIC.

Generally, IoT is about measuring parameters in the environment, and acting upon the environment, based on state changes originating from either the end user, or a rules engine. Therefore, the minimal requirements for AlexaMIC are the following features

1. Query a device for its current state
2. Alter the state of a device

1.2 Contributions

The following elements are included in this study

1. Proof of concept implementation of AlexaMIC that integrates MIC and Alexa.
2. Evaluation of the language model and interaction between the end users and Alexa.
3. Proposed features to improve the language model and interaction between users and Alexa.

1.3 Limitations

The following section outlines the limitations of the study

1. Potential security implications by allowing Alexa to control IoT devices are not part of this study.
2. The goal is not to develop a production ready system, but to create a proof of concept implementation that integrates the two mentioned systems.

1.4 Outline

This thesis is outlined as follows

Chapter 2 Background. Overview of the theoretical knowledge required to understand and repeat this study.

Chapter 3 Methodology. Describes the research method used in this study. This includes design of the prototype, the architecture, testing and evaluation.

Chapter 4 Design. Overview of the primary design goals of the implementation.

Chapter 5 Architecture. Detailed description of the final prototype, it's architecture, and how it communicates with MIC and Alexa.

Chapter 6 Testing the implementation. Describes the required steps to test the prototype using AWS.

Chapter 7 Evaluation. Evaluation of the two prototypes. Lessons learned are highlighted and discussed.

Chapter 8 Future work. Description of work that should be completed in the future, to improve the prototype

Chapter 9 Conclusion. Concluding remarks

1.5 Related work

This section gives a brief introduction to the related work. Lots of work are performed in the area of speech recognition, and many of the issues experienced from the work with AlexaMIC are addressed by different papers.

1.5.1 Personalized speech recognition for Internet of Things

Mahnoosh Mehrabani, Srinivas Bangalore and Benjamin Stern[12] deals with natural language interpretation in the paper Personalized speech recognition for Internet of Things. It discusses several of the problems that was faced during the development of AlexaMIC. For example, issues with sentences being parsed completely wrong because of missing context, and words misinterpreted as other similar words, such as e.g. "dim dining room light" and "jim dining room light".

The papers solution to the problem is a personalized language model, with the following accuracies:

- Task accuracy: 70 %
- Device accuracy: 80 %
- Intent accuracy: 90 %

Task accuracy is defined as a combination of device and intent accuracy. Thus, to successfully complete a task, the language interpretation service must get both the device and intent correctly interpreted from the speech.

The intent is defined as an action to perform. For example, an action could be to turn off the lights in a building. Using their language model, the action is interpreted correctly 90 % of the time.

The device accuracy is how accurately the language model determines which device to perform an action on.

1.5.2 One Solution for Voice Enabled Smart Home Automation System

One Solution for Voice Enabled Smart Home Automation System[11] is a paper which deals with the integration of an existing home automation system with the Google Cloud speech-to-text[1] service.

Their solution integrates the existing home automation system with Google's speech-to-text service by creating a layered architecture responsible for recording audio, communicating with the speech-to-text service, parsing the text from the analysis, and communicate with the existing home automation system.

The proposed architecture is verified and performance tested. It was found that 83% of the voice commands were parsed correctly and that the average time to get a response from the system was 1.72 seconds, which is acceptable.

1. `https://cloud.google.com/speech-to-text/`

2 Background

This chapter gives a brief introduction to the technologies used during the prototyping of AlexaMIC. The most important parts of this chapter are the sections about MIC and Alexa. These sections are described in detail to give an understanding of the two underlying systems that are integrated during the development of AlexaMIC.

This section also includes information about IoT, Lambda functions, Restful services, OAuth2 and MQTT.

2.1 Amazon Alexa

Amazon Alexa[1] is an intelligent personal assistant that supports a wide set of different services, such as weather reports, buying music from different music stores, home automation through integrations with systems such as Phillips Hue and Google Nest, ordering take out food among many others.

Third party integrations are supported using a feature known as custom skills. Developers use this functionality to develop integrations between the Alexa voice service and other services connected to the Internet.

The Amazon Echo is the hardware component of the personal assistant. It is a smart speaker that can be placed anywhere an Internet connection is available.

Users interact with the Echo, and Alexa in turn, processes the requests and return meaningful responses to the user.

2.1.1 Custom skills

Custom skills are defined in the Alexa dashboard. When a user is interacting with the Echo, for example, by asking for the next flight to Chicago. What happens, is that a custom skill processes the request from the Echo, by parsing the user's voice and dispatching the request via an intent to a cloud based processing service, responsible for processing the request. In this example, looking up the next flight, and returning a textual response to Alexa.

The textual response is returned to the Echo, and read back to the user.

Custom skills consists of the following components:

- Set of intents
- Utterances
- Invocation name
- Optionally images, different sounds etc
- Cloud based service used for processing intents
- Configuration

Intents represent actions that users perform when interacting with a custom skill. As an example, lets say a user requests the current state of a device. This implies the existence of a GetStatus intent, containing all the information required for Alexa to know where to dispatch the request for processing and how to recognize that the request belongs to the GetStatus intent.

Utterances are used to map requests from a user to the correct intent. Continuing with the previous example, an example utterance is: "What is the current status of {thingId}". When Alexa processes this sentence, and acquires a textual response of the speech, it is matched with the utterances stored in the different intent configurations, and if a match is found, the correct intent is invoked and processing can continue.

The last part of the sentence contains a slot type. Slot types are user defined variables. Alexa have several built in slot types, for example cities, literals, custom, numbers, actors and so on.

The custom type is the most useful, and is used to define any variable. In the example above, "thingId" is a custom slot type, used to get the identification number of a device.

An invocation name is used by Alexa to determine the correct skill to invoke. For example, lets say that, the GetStatus intent resides in a skill named MIC, with an identical invocation name. To invoke the GetStatus intent, the complete utterance is: "Alexa, ask MIC for the current status of device 0000123".

The sentence in the example above has all the information required for Alexa to invoke the MIC skill, dispatch the GetStatus intent and fill the custom slot variable with the id number of the device.

Images and sounds are supported in custom skills. However, this functionality is not important for the context of this study.

Intents are linked to a cloud compute service, used to process requests from the users. Alexa supports Lambda functions and custom hyper text transfer (HTTP) endpoints as cloud providers.

The preferred way of integrating an intent with a cloud provider is using the Amazon Web Services (AWS) ecosystem and Lambda functions. AWS provides all the configuration and setup out of the box, without the need worry about Transport Layer Security (TSL) certificates, which is a requirement for the communication between an intent and the cloud provider. Using a custom Hyper Text Transfer Protocol Secure (HTTPS) endpoint requires a TLS certificate deployed to the web server, from a small predefined list of approved Certificate Authorities (CAs).

Figure 2.1 illustrates the execution flow of an intent. Starting with the user asking for the status of a device, Alexa processing it, and dispatching the request to a cloud provider. The result is returned to the Echo.

2.1.2 Elicit response

Elicit responses is a feature used in situations where Alexa is unable to fill all the slot variables from a user utterance. This is common in situations where the speech is unclear, or the utterance has more than one slot variable.

The Lambda function or custom HTTPS endpoint linked with the intent, checks for undefined slot variables, and returns an elicit response directive, resulting

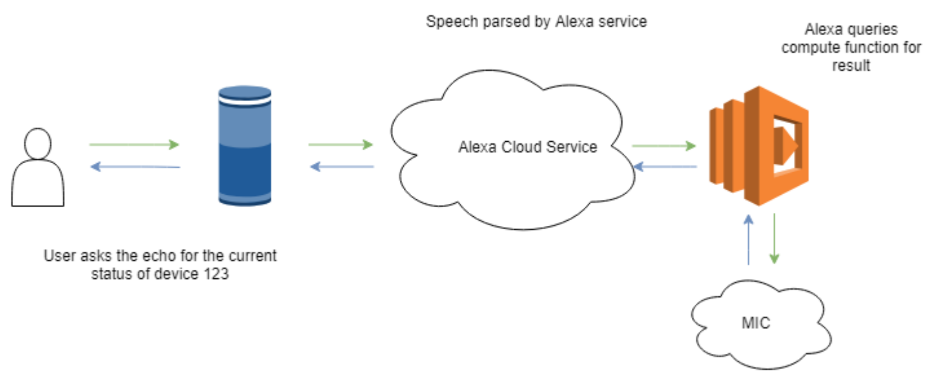
Speech parsed by Alexa service
Alexa queries compute function for result
Alexa Cloud Service
User asks the echo for the current status of device 123
MIC

Figure 2.1: Sample request from a user, to the GetStatus intent - the user is requesting the status of device 00000123

in the Echo asking the user for all the unspecified slot variables.

Consider the utterance: "Alexa, ask MIC to add testdevice to the bathroom group". The utterance consists of two slot variables, testdevice and bathroom, used to define the name of a device and group. The objective of the intent is to add the device to the bathroom group.

The compute function checks both variables, and finds that the group name is undefined. In this case, the elicit response directive is returned with an utterance describing how Alexa asks the user for the required information. As an example, this utterance is returned: "Please repeat the group name that you want to add the device to".

The user repeats the group name and Alexa returns control to the compute function. The new information is optionally confirmed with the user. When the information is confirmed, the device is added to the bathroom group.

2.1.3 Dialog delegate model

The dialog delegate model is a feature used to remove some of the boilerplate code responsible for confirming slot variables with the user. This works by defining re-prompt utterances in the configuration dashboard for the slot variables.

The compute function checks for undefined slots, and returns the dialog delegate directive if an empty slot is found. The Alexa service takes control, asking the user for the required slot values with the predefined re-prompt utterances, and confirms the slot variables if required. Control is given back to the compute function when re-prompting and optional confirmation of the slots are completed.

The main advantage of the dialog delegate model is that most of the boilerplate code regarding re-prompting of slot variables can be removed from the compute function.

2.1.4 Account linking

Account linking is a feature used to authenticate a compute function with a third party service. Since AlexaMIC requires reading privileges to the MIC platform to gather information about the devices that a MIC user has access to, account linking must be performed.

Account linking is based on the OAuth2 authentication protocol.

The process of linking an account is started from the Alexa mobile application or webapp. For account linking to function, the skill must be configured with an authorization and access token url.

The authorization url is used by the Alexa mobile application to redirect the user to a sign in website, responsible for authenticating the user with the third party service. The access token url is used to get an access token if the authentication was successful, or to request a new access token if the token is expired. The request and response flow adheres to the OAuth2 authorization code grant flow.

When account linking is completed, the access token is appended to the intent when control is given to the compute function. Thus, the compute function has access to the token, and can use it to make authorized requests to the third party service.

2.2 Managed IoT Cloud

The MIC[4] platform is an offering by Telenor developed to help accelerate innovation in the field of IoT. The platform is a complete ecosystem for devices. MIC supports provisioning, connectivity, data storage and inspection of the stored data using a special designed dashboard called the appboard.

The appboard supports different views, such as tables, text views, histograms etc. These are used to view and monitor data transmitted from different devices.

The platform consists of several APIs that the appboard use for it's functionality. The APIs are used by developers to create custom dashboards, tailored to the customer's requirements. However, without the need to worry about backend systems, data storage and so on.

The platform consists of three APIs:

- Thing API
- Cloud API
- Cloud REST API

The Thing and Cloud API's are accessible by invoking API calls using the AWS SDK. Cognito authentication is a requirement, before communication is possible.

The Cloud REST API is a restful implementation of the Cloud API. Authentication with Cognito is accomplished using an HTTP endpoint. Thus, no need to acquire and use the AWS SDK.

2.3 Internet of Things

The Internet of Things[2] (IoT) is about networking embedded devices with sensors and actuators measuring and acting upon the environment in which the devices are located. The goal is to simplify processes and peoples life's by the utilization of automation and integration between different services.

Consider an elderly person living alone at home. The family is constantly worried about him getting in a situation where he is injured and unable to alert the family or health care professionals. Using IoT, it is possible to place different sensors around the house, and on the elderly person. For example, a pulse clock, or an accelerometer used to detect critical situations such as hearth problems, or a bad fall. Whenever a problem is detected, the family or other health care professionals are notified, making it possible for the family to respond to the situation immediately, instead of discovering the person laying in the kitchen two weeks later.

Metadata from the sensors are added to the notification, which makes it possible to determine the situation’s degree of urgency. This informations is valuable to first responders, as well as the family of the person.

The example above shows how a process of notifying family and first responders are automated using devices with sensors connected to the Internet. The result is an automated process, which has improved a person’s life.

Another example is smart parking. Statistics shows that 30% of traffic congestion in big cities are caused by people driving around looking for free places to park their vehicles. What if devices could detect the free lots including their locations, and use this information to guide the drivers to choose the nearest and most efficient route, thus avoiding unnecessary driving and reducing the overall congestion.

Examples such as this exists for many different industries and applications. Key words are e.g. smart cities, smart industry, and smart buildings.

To make this possible, devices with sensors and actuators are required. The sensors are used to sense different parameters in the environment in which the devices are located. The actuators are used to act upon and change the parameters.

For example, an actuator could be a relay, controlling the state of an heat source.

The field of IoT is in constant expansion[3]. It is estimated that the number of devices connected to the Internet will surpass the worlds population by 2020.

2.4 Representational state transfer

Representational state transfer[7] (REST) is a stateless client-server architectural style.

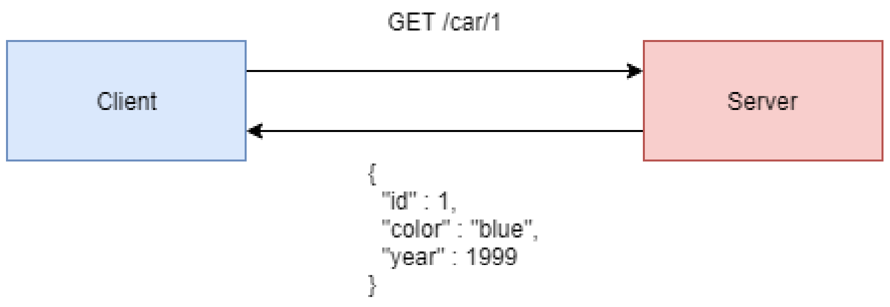

Figure 2.2: Client requesting a resource identified with the id 1, the server responds with a JSON formatted document describing all the attributes of the resource.

The REST architecture provides the following constraints

- Uniform interfaces
- Stateless
- Cacheable
- Client-server
- Layered system
- Code on demand

In the REST architecture, the different HTTP verbs are used to perform actions on resources. The POST, PUT and DELETE verbs are used to create, modify, or delete resources on the server. Whereas, retrieving resource are done using the GET verb.

Resources on the server are identified with the uniform resource identifier (URI).

2.4.1 Uniform interfaces

The uniform interface constraint is one of the most important aspects of the REST architecture. The purpose of the constraint is to simplify and decouple the architecture of a service to enable the different parts to evolve independently.

Data is organized into resources, for example, a car, with attributes such as color, model, year of production and so on, is called a resource. The main aspect of REST resources is that they are separate from their representation to the client. In other words, they might be stored in a text file or in a database, and returned to the client as Javascript Object Notation (JSON), Extensible Markup Language (XML) or HyperText Markup Language (HTML). No matter the representation, the resource is still the same.

Another aspect of the uniform interfaces constraint is that the representation of a resource contains enough information for the client to modify, or delete the resource, as well as the information required to parse the resource correctly.

2.4.2 Stateless

The communication between the client and server is stateless, meaning that the request must contain all the information required for the server to process and respond to the request. Examples of such information might be an access or refresh token used to authenticate the client performing the requests.

The REST architecture is capable of handling large amounts of requests because of the stateless property.

2.4.3 Cacheable

Clients and other intermediaries can cache responses to requests if the cacheable header is present in the HTTP request, which improves the scalability of the REST service.

2.4.4 Client-server

The client-server architecture allows for separation of responsibilities, which simplifies the code base for both the client and server.

For example, a client might implement a graphical user interface (GUI), which contains the current state of the application. The server is concerned with data storage and request handling, and contains no state whatsoever.

In the client-server architecture, the communication is always initiated by the client, and the server responds to the requests initiated by the client.

2.4.5 Layered system

The main aspect of a layered system is that the client does not know whether it is connected to the end server, or some intermediary, providing services such as load balancing, caching and so on.

This improves the scalability of the service, but might reduce the overall user perceived performance of the system.

2.4.6 Code on demand

Code on demand is an optional constraint to the REST architecture which allows clients to download and execute new code, and thus, acquire new functionalities. For example, a client might download and run some arbitrary Javascript code.

2.5 Lambda functions

Lambda[9] functions are general purpose compute resources offered by AWS. The main difference between lambda functions and ordinary compute resource such as a virtual private machine (VPC), is that the lambda is only running when it is processing a workload. The VPC on the other hand, runs until it is powered down. The two main advantages of using lambda functions are the substantial cost reduction due to the difference in uptime, and that the underlying infrastructure is maintained by AWS. This means that companies using lambda functions instead of ordinary servers or VPCs, do not have to worry about maintenance, such as e.g. upgrading software, and security patching of their compute resources.

As an example, consider a web service that serves a REST endpoint. The server runs continually, listening for incoming connections, processes the request and returns a response. The lambda on the other hand, is idling until it is executed by the underlying infrastructure in the AWS ecosystem. The lambda then processes the request and returns a response. When the lambda returns, it goes back to idling. The VPC on the other hand, continues running until it is manually powered down.

2.6 OAuth2

OAuth2[6] is the industry standard protocol for authorization of web, desktop and mobile applications. OAuth2 is developed by IETF.

2.6.1 Authorization code grant

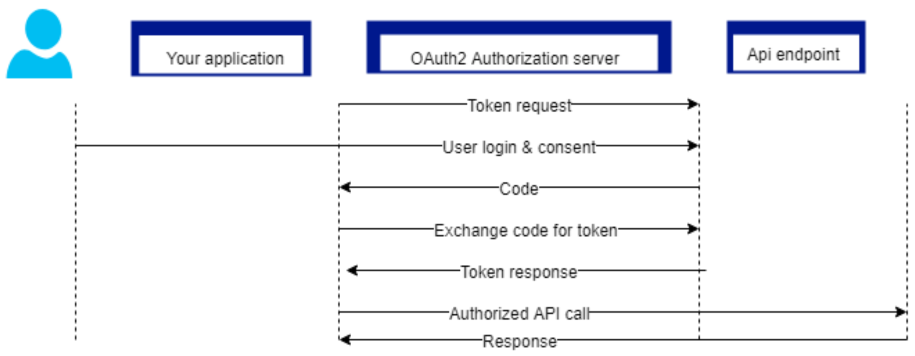

Figure 2.3: Authorization code grant - the alexa application requests a token by redirecting the user to an authentication website. The user authenticates with the third party service, and the server returns a code to the application. The application exchanges the code for the access and refresh tokens that was obtained in the previous step.

Important definitions:

Authorization server: Server that is responsible for authorizing a user with a third party service.
Application: In this context, the application is the Alexa account linking service.
Api endpoint: The MIC platform

The authorization code grant flow is initiated by the application. The application redirects the user to the authorization server, defined in the account linking configuration. This request contains important parameters, such as a redirect url as a query parameter. This parameter is used to redirect the user back to the application when the authentication with the third service is successful.

The user arrives at the authorization website where he is asked to input his credentials for the third party service. The user clicks sign in, and the credentials are authenticated. The third party service returns two tokens, that are stored internally in the authorization server.

A code is created, and returned to the application. This code is later exchanged for the access and refresh token that was obtained in the previous step.

The third party endpoint is now authorized, and the compute service can dispatch authenticated requests against it.

2.6.2 Implicit grant

Implicit grant is a simpler authentication flow compared to authorization code grant. The process is similar, except that the code step is completely omitted.

Refresh tokens are not supported by the implicit grant flow.

2.7 MQTT

MQTT[5] is an ISO standard based on the publish / subscribe messaging pattern. The protocol is designed for connections where the devices are constrained, or the network bandwidth extremely low.

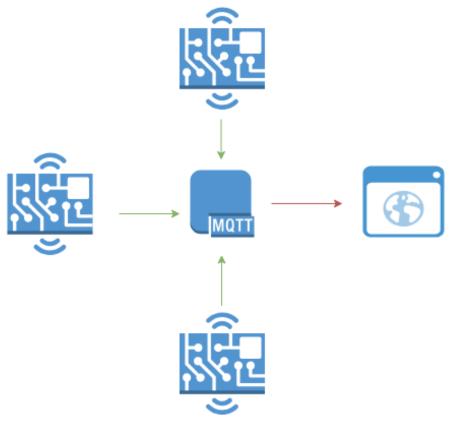

Figure 2.4: Example showing IoT devices publishing messages to an MQTT broker. The user interface is subscribed to the messages, and is updating it's interface based on the new data.

The pattern requires a message broker. MQTT clients connects to the broker and either subscribes or publish messages. The broker is responsible for distributing messages to interested clients based on the topic of the messages.

MQTT was first written by Andy Clark and Arlen Nipper of Cirrus Link Solutions in 1999.

2.7.1 Publish subscribe

The idea behind the publish subscribe messaging pattern is to decouple the sender and receiver. The decoupling is performed in three different layers: space, time and synchronization.

Space decoupling means that the sender and receiver never have to know the addresses of each other. The broker's address is always known, and the broker distributes the messages to the interested parties based on the topic of the message. MQTT clients are never connected directly with each other.

Time decoupling means that the participants does not have to be connected at the same time to communicate. The broker cache the messages, and distributes them when the different participants are available. The default behavior is real time communication, and caching is used as a fallback.

Synchronization decoupling means that the communication does not require synchronization.

The broker is highly scalable because of these three properties.

2.7.2 Message filtering

Message filtering in MQTT is performed on the topic of the messages. For example, a temperature and humidity sensor is publishing messages under the topics:

- myhouse/firstfloor/temperature
- myhouse/firstfloor/humidity
- myhouse/secondfloor/temperature

Subscribing to all the temperature sensors in the house is accomplished using a single level wildcard. In this example, the topic to subscribe to looks like this:

myhouse/+/temperature

Another option is to use the multi-level wildcard. An example is: myhouse/# which results in a subscription to all the sensors in myhouse.

The MIC platform uses the MQTT protocol to send and receive payloads from connected devices. The protocol is suited for this because of the decoupling described earlier.

3 Methodology

The methodology of this study is based on the spiral model[10]. The idea behind the model is that several prototypes are designed, developed and evaluated. Lessons learned from the previous round is used when designing and developing the next prototype.

The study is divided into a pre-study, and two rounds of design, development and evaluation, with the goal of arriving at the best possible prototype in the amount of time that is available.

This is illustrated by figure 3.1.

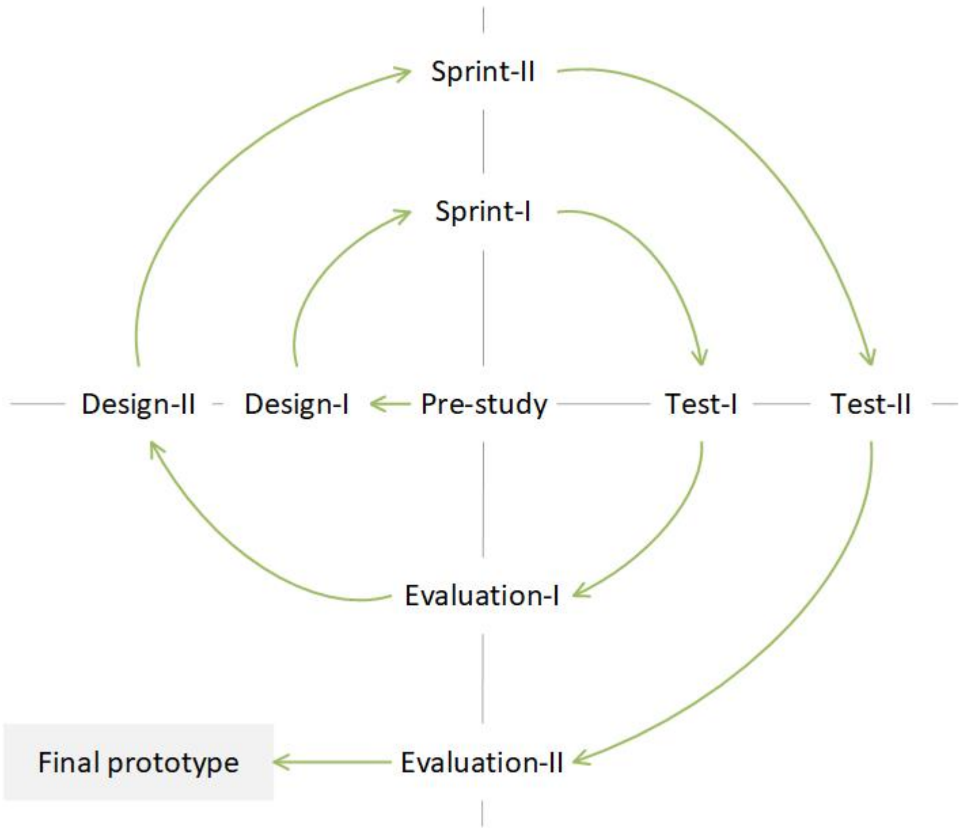

Figure 3.1: This study started with a pre-study that was used to learn Alexa and MIC, followed by two iterations of design, development, testing and evaluation. At the end, the final prototype is completed.

In the beginning of the study, a pre-study was carried out. In the pre-study, the focus was to learn about Alexa, and the AWS ecosystem with the goal of acquiring the knowledge required to complete the study.

The design phase was used to plan an overview of how the implemented prototypes should function, and what features to include. The goals derived during the design were used as guidelines during both sprints. By defining clear goals for the prototype, it was possible to use the evaluation phase to determine how good the design actually worked, and use this knowledge to correct and improve the design in the next phases.

The sprints were used to develop the prototypes based on the design goals defined for the current round in the spiral. Each of the sprints lasted for about a month, before the evaluation started.

The evaluation focused on determining if the design goals were archived and had the desired outcome. Other important aspects were the language model, the interaction between users and AlexaMIC, and how easy the current system is to use. Lessons learned during the evaluation were highlighted and improved in the next design.

The sections below describes these phases in more detail

3.1 Pre-study

This phase was used to research and learn about Alexa and the AWS ecosystem with the goal of using this knowledge to derive design goals for the first round of the spiral. Alexa was the first topic that was researched. This was decided to get an overview of what other requirements Alexa might have, in terms of developing a custom skill that integrates with a third party service.

The first requirement became apparent after reading about intents and intent processing. To process and return responses to the user, a lambda function or a custom HTTPS endpoint is required. Therefore, lambda functions and how they are debugged had to be researched. Debugging a lambda function is accomplished using a service called CloutWatch[1]. CloudWatch is Amazon's utility for debugging services in the AWS ecosystem.

Another important requirement was account linking. Account linking is used to authenticate and authorize third party services that are integrated with Aleza. Since AlexaMIC is an integration between MIC and Alexa, authentication is required for Alexa to access MIC on behalf of users and perform operations on devices. Account linking is an Alexa feature that complies with the OAuth2 authorization code grant flow. However, since MIC does not support OAuth2, a layer that sits between Alexa and the MIC platform, must be developed to handle the authentication and authorization of AlexaMIC's users.

3.2 Design 1

The first design was based around the idea of simplicity. The goal was to get the basic functionality of the integration working. However, some principals were defined at this stage, such as the need for a general purpose implementation. Basically, this was the idea that the integration should be usable on

1. `https://aws.amazon.com/cloudwatch/`

the connected devices, without having to alter the firmware of the devices. In addition, it was decided that the basic functionality of AlexaMIC should be the ability to query and set the current state of a device. These features were considered important because the basics of IoT is focused around data gathering and control of actuators on devices.

3.3 Sprint 1

The first prototype was implemented using the Javascript[2] programming language, and the Alexa-skills-kit-for-nodejs[3].

The implementation of the prototype involved the following steps

1. Defining the required intents, including the utterances and slot variables
2. Implementing the logic required for the intents, using lambda functions and AWS
3. Configure the lambda function for Alexa

The first stage of the development was to define the required intents. Two intents were defined, the first was the SetResource intent, which is used to alter the state of a connected device. To invoke the intent, an associated utterance had to be defined. This utterance was defined as "Alexa, ask MIC to set resource on device {thingId} {resourceName} {resourceValue}. This utterance consists of three slot variables named thingId, resourceName and resourceValue. The thingId is the identification number used by MIC to identify devices. The resourceName is used to identify the resource which is altered, and the resourceValue is used to set the actual value.

The second intent that was defined is the GetStatus intent, which is used to query a device for it's current state. The utterance for this intent was defined as "Alexa, ask MIC to get status of device {thingId}".

After the required intents were defined and published, the logic required to process the intents were implemented as a lambda function named IntentLambda. This function consists of an exported object, containing key value pairs, where the intent names are keys, and the handler functions are values. Two handler functions are implemented to handle the intent processing. These are named

2. https://en.wikipedia.org/wiki/JavaScript
3. https://github.com/alexa/alexa-skills-kit-sdk-for-nodejs

SetResource and GetStatus.

The handler functions authenticates with MIC using hard coded credentials (for simplicity), and communicates with the platform by dispatching asynchronous HTTPS calls to the exposed MIC APIs.

The final stage of development was to configure a trigger for the IntentLambda to communicate with Alexa. This was accomplished using the Lambda and Alexa dashboards. When the trigger was configured, the result was that the IntentLambda is called, each time an associated intent is invoked by Alexa.

3.4 Test 1

This stage of the study was used to perform usability tests on the implemented prototype. Important aspects of the testing, were the language model and the interaction between users and Alexa.

3.5 Evaluation 1

This stage was used to gather the results from the test, discuss the results and highlight important lessons learned.

3.6 Design 2

The second design was based on the evaluation of the first prototype. Largely, this meant improvements had to be made to the language model and interaction between users and AlexaMIC. Fixing most of the language issues were done by designing a database schema consisting of three tables. The ThingRoom table stores a many-to-many relationship between groups and devices, the Room table stores all the user defined groups, and the Thing table stores a one-to-one mapping between a MIC device, and a user defined speech friendly name. This is illustrated by figure 4.1.

Introducing a database to hold information about names and groups, meant that intents and logic had to be developed to support management of these tables. It was decided that group operations to query and set state should be included, to reduce the tediousness of controlling multiple similar devices.

3.7 Sprint 2

The final prototype was implemented using the Javascript programming language, the nodejs-skills-kit and the mysql[4] library.

The following steps were required to complete the development of the improved prototype.

1. Deploy RDS instance and the database schema

2. Change the utterances of the two intents and create the thingName slot variable

3. Implement the DatabaseService and MICService

4. Define intents required for groups and names, and define the required utterances

5. Implement the logic for the new intents

Deploying the MYSQL[5] database using the AWS RDS dashboard was the first stage in the implementation. The database was required, to store the names and groups of the different devices. After the deployment, the next step was to create the three tables, based on the schema illustrated by figure 4.1. The schema is migrated using MYSQL WorkBench[6]. In addition, the AWS firewall was configured to accept incoming and outgoing connections to the default MYSQL port. This was done to allow the lambda functions to communicate with the database, without having to configure a VPC[7].

After completing the first stage of development, the utterances from the previous round in the spiral, had to be altered to use the speech friendly name, instead of the thingId. Therefore, a new slot variable, called thingName was created, and swapped with the thingId in the utterances. This change was performed for both the SetResource and GetStatus intent. The device name is stored in the Thing table, and queried each time an action is performed on a device. This is a requirement to get the thingId for the MIC APIs.

In addition, two service layers were implemented to separate concerns, resulting in a cleaner code base, and improved maintainability. These service

4. `https://github.com/mysqljs/mysql`
5. `https://www.mysql.com/`
6. `https://www.mysql.com/products/workbench/`
7. `https://aws.amazon.com/vpc/`

layers are implemented in separate files, and imported to the IntentLambda when needed. The Database layer contains all the functionality to query the MYSQL database, whereas, the MICService layer is implemented to query the MIC API. The Database layer deals mainly with querying the database for information about device names, and which groups the devices are assigned to. However, to actually name and group devices, several new intents needed to be implemented in the IntentLambda.

The following intents were implemented to manage devices in terms of their names and groups. Utterances were also defined, but these are described in detail in the architecture chapter.

- AddThing
- AddThingRoom
- CreateRoom
- DeleteDevice
- DeleteGroup

The AddThing and AddThingRoom intents are implemented to name and organize logical groups of devices, making it easier for users to interact with the system. To Add a device to AlexaMIC, the MIC service layer must be invoked. This is done to confirm the existence of the device. To create a group, the CreateRoom intent is invoked, whereas, to delete a group the DeleteGroup intent is invoked.

Deleting a device is accomplished using the DeleteDevice intent.

In addition, intents to support group operations were implemented, such as for example, querying the current state of all devices, and listing all devices residing in a group.

The final stage of development was to implement a new lambda, called the AuthLambda, which allows multiple users to authenticate with MIC and use AlexaMIC. The AuthLambda was implemented using Axios and the Javascript programming language.

3.8 Test 2

This stage of development was used to perform usability tests on the system. Important aspects of the testing were the language model, and the interaction between users and Alexa.

3.9 Evaluation 2

This phase was used to gather the results from the test, discuss the results and highlight important lessons learned. In addition, custom skills and how they function in terms of syntactical language understanding is discussed.

4 Design

This chapter gives an introduction to the design goals of AlexaMIC. AlexaMIC was developed using the spiral method.

This chapter describes the first and final design of AlexaMIC. Refer to figure 3.1 for an illustration of the different phases in the study.

4.1 Design 1

This section describes the design goals of the first prototype.

The firs prototype was designed to support the most basic features required for a usable implementation of AlexaMIC. This meant that support for multiple users accessing MIC using Account Linking is omitted. In addition, support for creating logical groups, and naming devices are not implemented.

The basic functionality of AlexaMIC in this phase was therefore the following

1. Query a MIC connected device for its current state
2. Set state on a MIC connected device

These requirements meant that users should be able to issue voice commands to the Echo to get and set the current state stored on a device. Intents and custom logic had to be implemented to support these requirements.

4.2 Design 2

This section describes the design goals of the final prototype. Below are the three primary design goals of AlexaMIC

1. To develop an integration between the MIC platform and Alexa without imposing any rules on the communication protocols and payload formats used by the devices.

2. To increase natural voice interactions between users and devices using names and groups

3. To support important IoT features such as the retrieval of current state, and setting state on the devices

These design goals are described in depth below, starting with developing a general purpose integration, then, using names and groups and at the end, important IoT features which are implemented are described.

4.2.1 Develop a general purpose integration

Most of the commercially available voice control systems has some form of vendor lock in applied, or expensive license fees involved in using their systems. Examples of companies that utilizes this business model, is Apple with their Homekit[1] and Siri[2] integration, and their MFI[3] program. The problem with this approach is that it creates a barrier, which vendors must get by, in terms of costs. Another problem, is that equipment which is sold, rarely is compatible with other products on the market.

Therefore, the first design goal of AlexaMIC is to develop a general purpose integration between Alexa and MIC. This means that the system is not designed for a single and specific use case, and that no rules are imposed regarding the communication protocols between MIC and the connected devices, and how

1. https://developer.apple.com/homekit/
2. https://developer.apple.com/sirikit/
3. https://developer.apple.com/programs/mfi/

the devices formats and encodes their payload during data transmission.

This ensures compatibility between AlexaMIC and all of the devices connected to MIC, without the need to alter the firmware of devices. In addition, This makes it easier for vendors to start using AlexaMIC.

4.2.2 Using groups and names

The second design goal of AlexaMIC, is the ability to assign names and groups to the connected devices. The reason this is needed, is that the MIC platform identifies devices with an eight digit identification number referred to as the thingId of a device. The platform does not support arbitrary groups of devices. Thus, without designing AlexaMIC to support this feature, it would not be possible to create and assign devices to user defined logical groups. Without the support for user defined device names, users would have to refer to the thingId every time an action is performed on a device. Remembering an eight digit number for every smart device located in a user's home is hard, which underscores the importance of allowing users to define speech friendly names that are easy to remember and relate to.

The ability to group devices are an important feature for the possibility to perform group operations on a set of similar devices. For example, consider a lamp with ten different smart bulbs, each functioning as a separate device. Imagine that the user is turning off each bulb, one at a time. The user would have to ask Alexa repeatedly, to turn off each bulb associated with the lamp, which wastes time and makes the system unnecessary difficult to use.

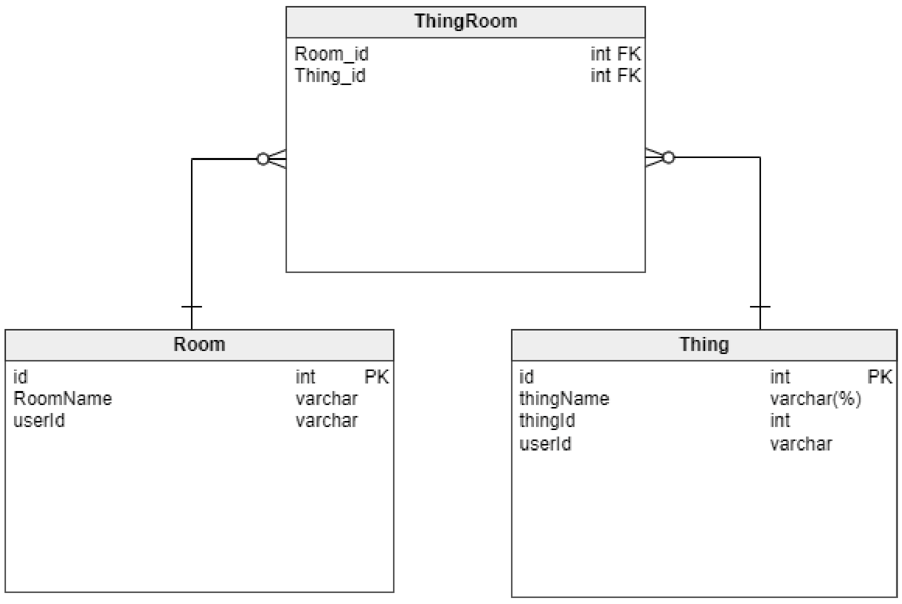

Figure 4.1: Illustration of a schema used to map the thingid to names, including grouping of devices, using a many-to-many relationship on group and thing.

Figure 3.1 is an illustration of how names and groups is used in the implementation of AlexaMIC. The figure shows three different SQL tables, named ThingRoom, Thing and Room. The Thing table holds information about the different devices which are added to AlexaMIC. This table forms a one-to-one mapping between the thingId in MIC and the user defined device name. This means that a thingId can only have a single device name associated with it.

The Room table is used to hold information about the different groups that the user have created. This information includes the name of the group.

The ThingRoom table is used to store an association between devices and groups. The ThingRoom table consists of two fields, named room_id and thing_id. The two fields directly relates to the id fields of the Thing and Room table. This makes it possible to look up a group, find the group id, and then check the ThingRoom table for all the devices which are related to the specific group. This works both ways, meaning that you can start with a device, find its id, and find all the groups which the device are assigned to. Refer to figure 3.1 for a complete overview of the database schema.

All of the tables, except for ThingRoom, contains a field named userId. This field is used to create a mapping between the current Alexa user, using the unique id that was created when the Echo was previously configured, and the Managed IoT Cloud user.

4.2.3 Supporting important IoT features

The last design goal is to support the most important features of IoT. This includes the ability to retrieve and alter the current state of a device. Allowing users to alter state, makes it possible to control actuators on the devices, resulting in the possibility to perform actions such as e.g. turning on and off light bulbs, adjusting the temperature on the thermostat and dimming the lights in the living room.

5 Architecture

This chapter gives an introduction to the architecture of AlexaMIC. This is a description of the final prototype, as outlined in figure 3.1.

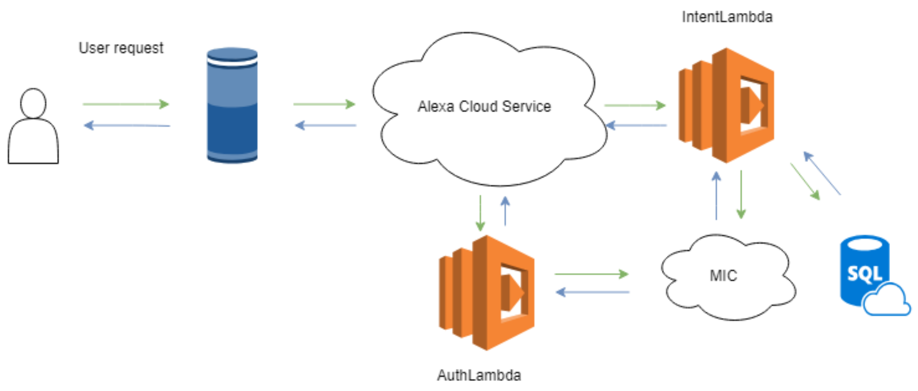

Figure 5.1: AlexaMIC architecture. All requests, except requests related to authentication originates from the Echo and are processed by the Intentlambda. Data from the local data store is fetched when required, and requests are dispatched at MIC. The AuthLambda handles authentication and re-authentication based on refresh tokens and the OAuth2 code grant flow.

Figure 5.1 illustrates a simplified version of the architecture, consisting of seven components. The Echo is a hardware component, described in the background section. The IntentLambda, AuthLambda and the SQL database are either developed or designed, during this thesis work, and the rest of the components, are third party services from either Alexa or MIC.

Alexa requires compute resources, such lambda functions or an external HTTPS endpoint, to process intents. The compute resources required by AlexaMIC, are implemented using lambda functions. Two lambda functions are implemented. These are the IntentLambda, used for intent processing and the AuthLambda, used for account linking and access token renewal. Thus, allow multiple users to use the system.

5.1 IntentLambda

When a user invokes an intent by asking Alexa to perform an action, the Alexa service processes the request and dispatches it to a lambda function implemented to handle processing, such as querying different databases for information and performing API calls to third party services. After the processing is completed, a textual response is returned to Alexa, and read back to the user by the Echo.

The IntentLambda is a function implemented to handle the processing for AlexaMIC. This includes querying the MYSQL database for devices, or groups and calling the MIC API to get or set device state.

The IntentLambda exports an object consisting of key value pairs. The keys are intent names. Thus, a single key exists for every defined intent. The values are handler functions, responsible for processing the requests from Alexa, and returning a textual response.

Figure 5.2 is an illustration of the exported object. The orange boxes denotes different keys. Whereas the blue boxes are handler functions.

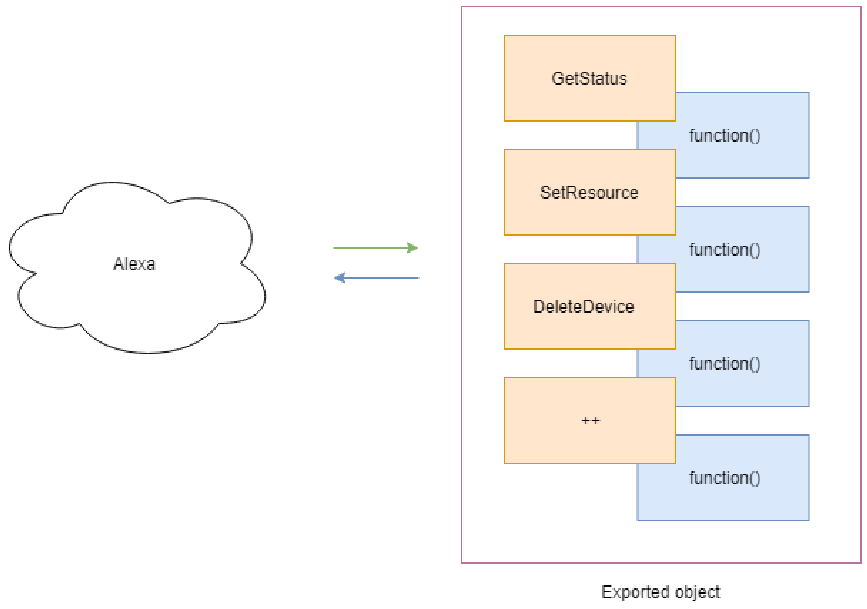

Figure 5.2: The IntentLambda is invoked by the Alexa service when an intent is dispatched - Alexa knows which function to call based on the key values that are stored in the exported JS object.

The IntentLambda is organized into different classes, based on functionality. Specifically, the database operations are contained within a class named DataStore, and the MIC API interactions are contained within the MICService class. This organization is performed to separate concerns and to improve the maintainability of the code.

5.1.1 MIC Service class

The MIC service class supports the basic functionality required for AlexaMIC to interact with the MIC platform. AlexaMIC communicates with MIC using asynchronous[1] HTTPS calls, and the communication flows from the MICService class to the MIC platform. Thus, the MICService class acts as the client. AlexaMIC utilizes the Axios[2] HTTP library to dispatch HTTP requests.

The following methods are implemented as part of this service layer

1. `https://searchwindevelopment.techtarget.com/definition/Ajax`
2. `https://github.com/axios/axios`

- verifyThing
- getStatus
- setResource
- getStatusThings

The verifyThing method is used to check if a thingId exists in MIC. This method works by dispatching an API call to MIC, directed at the /things endpoint, using the thingId as a query parameter. When the MIC API returns a response containing an error message, the device is considered to be non-existent.

The getStatus method is used to query the MIC platform for the current state of a device with a given thingId. The method works by dispatching an API call against the /things endpoint with the thingId as a query parameter assigned to the request. The response is an object containing the information that MIC has about the particular device.

The setResource method is used to alter the current state of a device. The method dispatches a PUT request to the '/things/resources' endpoint of of the MIC API. The response indicates if the operation was successful or not. The payload attached to the request contains the information required for MIC to know which device, and resource to alter.

The getStatusThings method is used to get the current state of a list of devices. This method works by calling the GetStatus method repeatedly for every device in the supplied list. The result of this operation is a list of devices, which are returned to the caller.

5.1.2 Data store class

The DataStore class provides all the functionalities required to perform database operations on the three tables that are defined for AlexaMIC.

The following methods are implemented in this class

- createRoom
- deleteGroup
- addThing

- deleteThing
- addThingRoom
- getRoomByName
- deleteThingsFromGroup
- getThingByName
- getThingsInRoom

The createRoom method is used to create a new group and store it in the database. The method requires a group name and the Alexa userId as arguments to the method call. The method works by using the SQL INSERT INTO statement, to insert the group name and userId to the Room table. The method returns a status code indicating the success of the operation.

The deleteGroup method is used to delete a specified group from the Room table. All references to the group from the ThingRoom table are also deleted. This method queries the Room table for the id of the current group. This id is then used to delete all records from the ThingRoom table with a room_id equal to the id of the group. The last step is deleting the group from the Room table. Both of these delete operations, are performed with the SQL DELETE and WHERE statements.

The addThing method is used to add a new record of a device to the Thing table. The method takes an Alexa userId, a user defined device name, and the MIC thingId as arguments. The method uses the SQL INSERT INTO statement, to insert the new record into the table.

The deleteThing method is used to remove a device from the Thing table, including all references from the ThingRoom table associated with the current device. This method first finds the id of the device record from the Thing table. Then uses this id to remove all records with a thing_id matching the id of the device, from the ThingRoom table. The last step is to delete the device itself from the Thing table. This is accomplished using the SQL DELETE and WHERE statements.

The addThingRoom method is used to create a relationship between a device and a group. This is accomplished by creating a new record in the ThingRoom table. The method takes the id of a group and the id of a device, as arguments. These values are added to the ThingRoom table using the SQL INSERT INTO statement.

The getRoomByName method is used to search for a specific group, based on the supplied group name. The method uses the SQL SELECT and WHERE statements to find the correct group from the Room table.

The deleteThingsFromGroup method is used to remove a device from a specific group. This method uses the SQL DELETE and WHERE statements with the room_id, userId and thing_id as parameters to the WHERE clause.

The getThingByName method is used to query a specific device from the Thing table, based on the supplied device name. This method uses the SQL SELECT statement, and the WHERE clause on the device name and Alexa userId. The result of this operation, is the device record for the device associated with the given device name and userId.

The getThingsInRoom method is used to get all the devices associated with a specific group. The method takes a group name and an Alexa userId as arguments. The correct group record is found using the SQL SELECT and WHERE clause on the userId and group name, resulting in the group record being fetched from the database. The next step is to select every record from the ThingRoom table, and join this record, with the Thing table, on the attributes ThingRoom.thing_id and Thing.id. The WHERE clause is used to filter the result to only fetch records that belong to the current Alexa userId and the current group id. The result of this operation, is a list of all devices associated with the given group.

5.1.3 Intents

The following intents are implemented for AlexaMIC.

- AddThing
- AddThingRoom
- CreateRoom
- DeleteDevice
- DeleteGroup
- GetStatusByRoom
- ListDevicesByRoom

- SetResource

These intents are described in detail below, starting with some common information which is true for all of the intents.

Validation and confirmation of slot variables

Invocation of the different intent handlers are done by the user speaking one of the defined utterances. Alexa uses a combination of the invocation name (MIC in this context) and the utterances, to check what custom skill to call, and then which intent to invoke. After Alexa has performed the initial processing, the correct handler is called. The first that happens in all of the handler functions are described in the next section.

Common for all of the handler functions are that slot variables are fetched from the intent header, validated and confirmed by the user. Part of the validation is to check if any of the slots are undefined, if so, return an elicit response directive, forcing the user to fill the variables correctly. The next step is to confirm every variable with the user. Confirming variables are performed by returning a confirmation directive from the handler function, supplied with a re-prompt utterance, such as for example "Did you say temperature sensor?". The user has the ability to answer with a positive or negative response, resulting in Alexa accepting the slot, or asking the user to fill it again. This process is repeated until all the slot variables are validated and confirmed.

Adding a device to AlexaMIC

To add a device to AlexaMIC, the user must invoke the AddThing intent using one of the two utterances defined below.

```
Utterances:

"to add thing {thingName} with id {thingId}"
"to add thing"
```

When the AddThing handler is invoked, the thingName and thingId slots are fetched from the intent header, validated and confirmed.

Before a device is added to the Thing table, the thingId is validated for existence using the MICService class. In other words, it is not possible to add devices, which does not exist in MIC.

When the validation is successful, a new database record is created and stored in the Thing table using the DataStore class.

The handler method returns a textual response containing the result of the operation, and potential error messages are included. The Echo reads the response to the user.

Adding a device to a group

To add a device to a group, the user must invoke the AddThingGroup intent using one of the two utterances defined below.

```
Utterances:

"to add thing to group"
"to add thing {thingName} to group {roomName}"
```

The first that happens after the intent handler is called, is that the slot variables thingName and roomName are fetched from the intent handler. The variables are then validated and confirmed by the user.

When the slot variables are validated, a new database record is created and added to the ThingRoom table, using the DataStore class.

When the device is added to the group, a textual response is returned to Alexa and read by the Echo.

Creating a group

Creating a group is accomplished by invoking the CreateRoom intent. This intent is invoked by the user speaking one of the two utterances defined below.

```
Utterances:

"to create room"
"to create room with name {roomName}"
```

When the intent handler is called, the roomName slot, is fetched from the intent header, validated and confirmed.

Upon successful validation, a new database record is created and stored in the Room table, using the DataStore class.

A textual response is returned to Alexa, indicating the status of the operation. The message is read to the user by the Echo.

Deleting a device from AlexaMIC

Deleting a device is accomplished by invoking the DeleteDevice intent. This intent is invoked by the user speaking one of the two utterances defined below.

```
Utterances:

"to delete device"
"to delete device with name {thingName}"
```

When the intent handler is called, the thingName slot is fetched from the intent header, validated and confirmed.

Before the device can be deleted from the Thing table, the references to the device, from the ThingRoom table must be deleted. The DataStore class has a method named deleteThing, which is invoked to handle the deletion of both the references from the ThingRoom and the device from the Thing table.

When the deletion is completed, a textual response, indicating the status of the operation, is returned to Alexa and read by the Echo.

Deleting a group from AlexaMIC

Deleting a group from AlexaMIC is performed by the user invoking the DeleteGroup intent. This intent is invoked by one of the following utterances.

```
Utterances:

"to delete group"
"to delete group with name {roomName}"
```

When the intent handler is called, the roomName slot is fetched from the intent header, validated and confirmed.

Before the group is deleted from the Room table, all the references to the group must be deleted from the ThingRoom table. The DataStore class has a method named deleteGroup, which handles the deletion of potential references to the group and the group itself.

The status of the database operation is returned to Alexa, and read to the user by the Echo.

Getting state from devices in a group

This intent is one of the supported group operations. The purpose of this intent is to get the current state of all the devices associated with a given group.

To invoke the intent, the user must speak one of the utterances that are defined below.

```
Utterances:

"to get status by room"
"to get status by room {roomName}"
```

When the intent handler is called, the roomName slot is fetched from the intent header, validated and confirmed.

The first step is to get all of the devices in the group, by calling the DataStore method getThingsInRoom with the roomName and the Alexa userId as parameters to the method call. The result of the operation is a list containing all of the devices associated with the given group.

The last step is to iterate the list of devices, and for every device, call the MICService method named getStatus, with the thingId of the current device as a parameter to the method call. The result of this operation is the state of the current device.

When the state of all the devices are fetched, a speech friendly response is constructed based on the contents of the information retrieved from MIC. This textual response is returned to Alexa, and read by the Echo.

List devices in a group

This intent is used to list all of the devices belonging to a particular group. The intent is invoked by the user speaking one of the two utterances below

```
Utterances:

"to list devices in group"
"to list devices in group with name {roomName}"
```

When the intent handler is called, the roomName slot is fetched from the intent header. The slot is validated and confirmed with the user before processing continues.

The DataStore method named getThingsInRoom is called with the roomName slot and the Alexa userId as parameters to the method call. The result of the call, is a list of devices associated with the current group.

A textual response is constructed based on the device list, and returned to Alexa. The response is read by the Echo to the user.

Set state on a device

This intent is used to alter the state of a device. The intent name is SetResource, and is invoked by the user speaking one of the two utterances below.

```
Utterances:

"to set resurce"
"to set resource on {thingName} {resourceName} to {resourceValue}
```

When the intent handler is called, the thingName, resourceName and resourceValue are fetched from the intent header, validated and confirmed.

The first step is to get the device from the database, using the DataStore method getThingByName, with the thingName and Alexa userId as parameters to the method call.

The next step is calling the setResource method, in the MICService class, with a payload as a parameter to the method call. The payload is created by the handler, and consists of three parameters, the thingName, the ThingType, and a list containing objects with the resource name and the new resource value.

The result of the operation is returned to Alexa, and read to the user by the Echo.

5.2 AuthLambda

The AuthLambda is implemented to handle account linking for Alexa and the MIC platform. The function adheres to the OAuth2 authorization code grant flow, and supports access token renewal, using a refresh token and authentication with MIC, using credentials inputted from the Alexa mobile application.

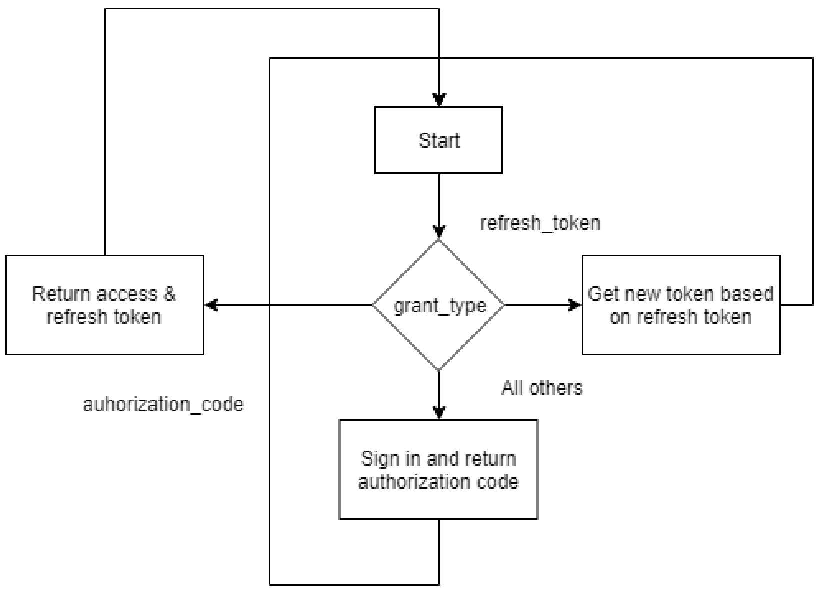
Start
refresh_token
grant_type
Return access & refresh token
Get new token based on refresh token
auhorization_code
All others
Sign in and return authorization code

Figure 5.3: Flow chart illustrating the execution flow of the AuthLambda function.

The AuthLambda first checks the grant_type and decides which action to perform based on this value.

When the grant_type is refresh_token, a new access token is acquired from MIC using the supplied refresh token. When the new access token is retrieved, it is returned to Alexa in the entity body of the HTTP response.

When the variable is authorization_code, the supplied code is decoded from Base64, to a refresh and access token, and returned as JSON to Alexa.

Figure 5.3 is a flow chart illustrating the execution flow of the AuthLambda. Figure 5.4 illustrates the authentication flow of the account linking process.

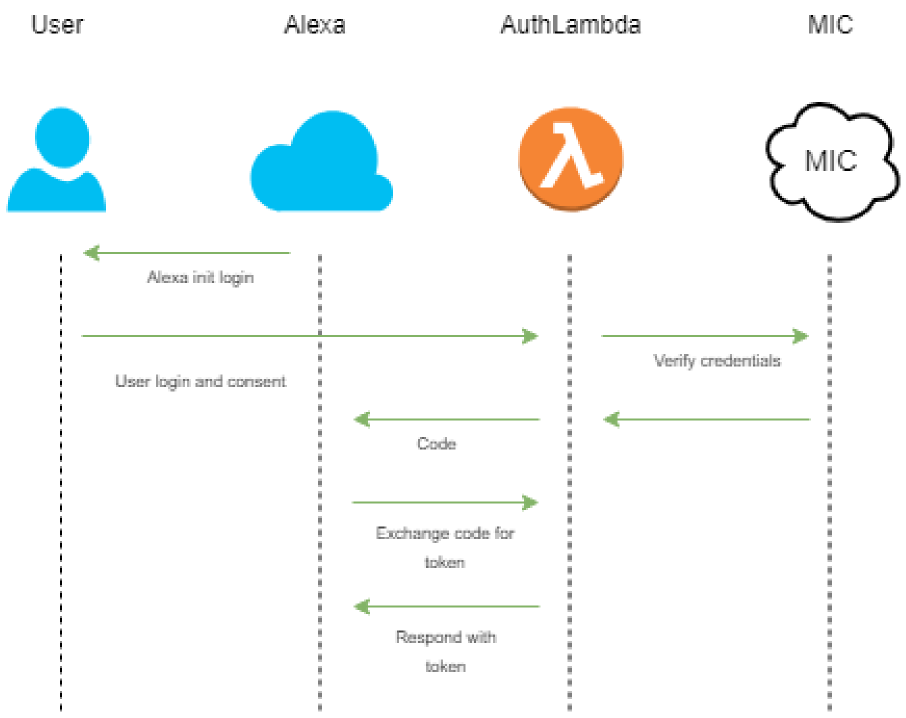

Figure 5.4: Account linking flow - Alexa is authenticated against a Managed IoT user.

5.2.1 Log in website

Another component that had to be developed to use Account Linking is a website containing a login form, used by the users to input their MIC credentials. When the user selects to start the Account Linking process, this login form is opened by the Alexa mobile application.

This website is implemented using the PHP[3] programming language, and must be hosted on an EC2 instance. The website consists of a simple login form, and custom logic to authenticate with the MIC platform using the supplied credentials. When the authentication is succesfull, the returned identifyId, access and refresh token is formatted as JSON and Base64[4] encoded. The Base64 encoded string is then returned to Alexa.

Encoding the access tokens are done to avoid storing the information in a database between the authorization_code and authentication step of the OAuth2 flow. However, this reduces the overall security of the authentication flow to some extent.

This is the first step of the Account Linking process. The AuthLambda is called for all the other steps.

3. `http://php.net/`
4. `https://en.wikipedia.org/wiki/Base64`

6 Testing the implementation

This chapter outlines the required steps to configure and test the implementation of AlexaMIC on AWS. The following steps are required to complete the setup

1. Create the required user accounts
2. Populate the intents using the included JSON
3. Create two lambda functions - IntentLambda & AuthLambda
4. Create a MYSQL database and allow access from the outside world
5. Configure the IntentLambda to communicate with Alexa
6. Configure the required OAuth2 endpoints
7. Create an EC2 instance to host static website

6.1 Create required accounts

To get started with AWS and Alexa, user accounts for both services must be created. Complete the sign up process for the services, and sign in to both of them. In addition, a MIC[1] user account must be created.

6.2 Populate the intents

Start by opening the Alexa developer dashboard[2], and create a custom skill. Input the contents of the attached JSON file, and click save. The intents should start to populate.

6.3 Create compute resources

Open the AWS console[3] and enter the Lambda dashboard. From here, start by creating two lambda functions with the names IntentLambda and AuthLambda. Uploading the source code for the functions are accomplished by navigation to the individual configuration dashboard for each of the newly created lambdas, choose upload and select the zip file for the associated function.

Clicking save initiates the upload of the source code.

When both of the uploads are completed, the lambda functions are ready to be used.

6.4 Configure API gateway

The API Gateway is a component from AWS that is used to route HTTP requests from the outside network to different AWS resources, such as for example, lambda functions.

Configuring the gateway is performed by opening the AWS console and navigating to the configuration dashboard of the API gateway. From here, choose to create a new API.

1. https://startiot.mic.telenorconnexion.com/
2. https://developer.amazon.com/alexa/console/ask
3. https://aws.amazon.com/console/

The API that is going to be created consists of a single endpoint ’/token’, supporting HTTP POST requests. Creating the endpoint is done from the dashboard by selecting to create a new resource from the drop down menu. Name the resource ’token’ and click save.

The next step is to create a POST method. Click the drop down menu and select to create a new method. Choose POST. A new window with additional settings will appear. In this windows, it is important to select the lambda proxy checkbox, and input the correct lambda function name in the lambda method field.

The lambda function name to input is AuthLambda.

The last step is to deploy the API. This is accomplished from the main drop down menu.

When the API is deployed, an endpoint address appears on the screen. This address must be inputted in the OAuth2 account linking configuration, in the Alexa developer dashboard.

6.5 Create MYSQL database

Create a MYSQL database using the Amazon RDS dashboard. When the database is created, get the endpoint address from the dashboard and use a suitable tool to run the attached queries against the endpoint, to create the required tables.

Access to the database must be given to the outside world, by editing the security policies of the newly created database. The settings regarding network access are found using the RDS dashboard.

6.6 Configure IntentLambda

Alexa must be given permission to invoke the IntentLambda. The correct permissions are given from the Alexa developer dashboard. Open the dashboard and navigate to the endpoint menu. From here, input the ARN of the IntentLambda. Finding the correct ARN is done by opening the lambda dashboard from the AWS console, and clicking on the IntentLambda. The ARN should be listed on this page.

6.7 Configure OAuth2 endpoints

Open the Alexa dashboard and the MIC skill. The menu on the left side contains an item named Account Linking. Click on this item to start the OAuth2 configuration.

The following fields must be filled:

- Grant type - Auth code grant
- Authorization uri - the url to the EC2 hosted website containing a login box
- Access token uri - the url to the AuthLambda '/token' endpoint
- Client id - input anything here
- Client secret - Input anything here
- Client authentication scheme - Credentials in request body

The client secret and id is not validated in AlexaMIC. Thus, the values are not important. However, in a production system, these values should be validated for security reasons.

6.8 Create EC2 instance to host static website

Create an EC2 instance running Ubuntu server, and install the Nginx web server. Deploy the static website that is attached to this study under a domain name of your choosing.

It is important to update the Account Linking configuration in the Alexa dashboard to this domain name, as this website is used for inputting the MIC credentials to authorize Alexa on behalf of a user.

7 Evaluation

This chapter is an evaluation of the prototypes that were developed during Sprint 1 and Sprint 2. Refer to figure 3.1 for an overview of the development process.

7.1 Evaluation 1

The first prototype is evaluated in terms of the language model and interaction between AlexaMIC and users. In addition, the ease of use is evaluated in terms of performing the operations that AlexaMIC supports.

7.1.1 Language model and interaction

The first prototype was implemented to support intents to query and set the current state of the devices connected to MIC. Four utterances were defined: "Ask MIC to get status of device {thingId}", "Ask MIC to get status", "Ask MIC to set resource on device {thingId} {resourceName} {resourceValue}" and "Ask MIC to set resource". The slot variable thingId, is the eight digit identification number from the MIC platform, used to identify different devices.

The largest problem with this approach was that users had a hard time remembering the thingId. In addition, Alexa's accuracy in terms of parsing numbers with long series of leading zeros was not satisfactory. Since the MIC thingID is in the following format xxxxabcd, were x is a series of zeros leading the actual id, denoted as abcd, this became a problem. In terms of statistics, the thingId was parsed wrongly about half of the times an intent was invoked.

Another problem was the invocation name used by AlexaMIC, which often was misinterpreted as microphone, instead of MIC, short for Managed IoT Cloud.

The following lessons were learned from these insights

1. Users should be able to name devices to avoid remembering long numbers, and to improve the accuracy of Alexa's speech recognition algorithm (avoid faulty parsing of leading zeros)

2. The invocation name is not ideal, alternatives should be tested.

7.1.2 Ease of use

The first prototype supports setting and querying the current state of a user specified device. Setting a resource is done using the utterance "Alexa, ask MIC to set resource on device {thingId} {resourceName} {resourceValue}. This works, and the resource is set together with the value using the MIC API. The only problem is that the variable type of the resourceValue is parsed as a string. Thus, it is not possible to set values with a type such as booleans and integers. Thus, it is the device's responsibility to typecast or parse the value into what is expected. This is a problem in terms of usability, and could lead to unexpected bugs in other parts of the system. A possible solution to the problem, could be to implement an intent for every variable type. However, this would increase the complexity of the system in terms of utterances which the user must remember.

Querying a device for its current state is done using the utterance "Alexa, ask MIC to get status of device {thingId}. This worked fine, expect for the issues already mentioned in regards to faulty parsing of the thingId slot.

The most important problem with the usability was the lack of group operations. Thus, interacting with a set of similar devices were tedious and repetitive. For example, turning off ten different LED lights, each functioning as a separate device, took time and involved changing the state of all the devices to turn off the lights. In addition, asking the devices from the previous example, for the state of their LEDs were equally repetitive.

The following lessons were learned from these insights

1. Group operations to query and alter state of similar devices are needed to increase the ease of use of the system.

7.2 Evaluation 2

The final prototype is evaluated in terms of the language model and interaction between AlexaMIC and the users, and the ease of use of the system. The following sections describes this in detail, and highlights some possible improvements.

7.2.1 Language model and interaction

In the final prototype, intents used to name and group devices were implemented. This solved most of the issues from the previous iterations, as users no longer had to remember long number series to interact with devices. Another problem was that Alexa had a hard time parsing number series with leading zeros. This issue is also resolved, as Alexa now only parses the speech friendly user defined device name. Figure 4.1 illustrates the database schema that was used to implement names and groups.

However, the language model and interaction is still not good enough. The main problems are the following

- Utterances must be constructed in a specific way, forcing users to speak the exact same utterance to be understood by Alexa.

- Invocation names leads to unnatural sentences

The first problem is that utterances defined for processing the intents, has to be specified strictly. As an example, consider the utterance "Alexa, ask MIC to add device 00001234 to the bathroom group". For the intent to be invoked, the user is required to speak the exact same sentence. Thus, omitting a single word, might be enough for Alexa to not recognize and invoke the intent. In other words, Alexa does not understand that "Alexa, ask MIC to add device 00001234 to the bathroom group" means the same as "Alexa, ask MIC to add device 00001234 to the bathroom". The reason this happens is because Alexa does not understand context, but only the syntax of the sentence. This is a problem since humans generally do not have a fixed and common way of speaking. The problem is reduced by defining many similar utterances. However, this is a guessing game, without any guarantees. Careful consideration is required to define all the utterances that a user might use in a specific situation.

Another issue is the invocation name that is required for Alexa to correctly determine which custom skill an intent belongs to. The problem is that all utterances must consists of the invocation name, leading to unnatural sentences. As an example, the user should be able to say "set resource on testdevice lightstatus off", but must say "Alexa, ask mic to set resource testdevice lightstatus off". However, even if all these issues were somehow fixed, the interaction is still not natural enough to be useful. Below is a description of a more suitable solution that could be implemented if Alexa or MIC had some understanding of the metadata and actions that every device supports.

Ideal solution

The ideal solution is to implement intents for different actions. For example, intents could be developed for turning on and off peripherals, dimming lights and locking / unlocking doors. An example of such an utterance is defined below

"Alexa, {action} the {type} in the {location}"

The complete sentence for turning off a light would then be "Alexa, turnoff the lights in the bathroom". Here the action is turnoff, the type is light and the location is bathroom.

To implement this, device metadata, describing the supported actions, the type of peripheral and the locations of the devices must be stored, either in AlexaMIC or the MIC platform. This metadata must be stored for all the devices used with the system.

7.2.2 Ease of use

The previous iteration had problems in regards of repetitive commands when wanting to perform the same operation on multiple devices. This issue is resolved to some extent in this iteration. Support for organization devices into groups, and some group operations are now supported. Users are now able to query multiple devices for their current state.

Support for setting a resource on multiple devices are still missing. However, the ground work is done, so it should be relatively easy to implement this feature.

7.3 Commercialization

This section deals with questions in terms of commercialization. John C. Mankins is the author of the technology readiness white paper, that discusses different technology readiness levels(TRL)[13]. TRL is a systematic metric system used to assess the maturity of different types of technologies. This approach has been used on-and-off in NASA space technology planning for many years to determine the readiness of the research and technologies that are utilized in their space program.

Mankins defined nine different TRL levels

1. Basic principals observed and reported
2. Technology concept and/or application formulated
3. Analytical and experimental critical function and/or characteristic proof-ofconcept
4. Component and/or breadboard validation in laboratory environment
5. Component and/or breadboard validation in relevant environment
6. System prototype demonstration in a space environment
7. Actual system completed and "flight qualified" through test and demonstration (ground or space)
8. Actual system "flight proven" through successful mission operations

Below is a paraphrased description of the different TRL levels.

TRL 1-2 is the most basic maturation level of technology. In this level, basic research are conducted.

In the TRL 3-6, the research from the lower TRLs are utilized to create proof-of-concept implementations of potential applications. The applications are tested in simulated environments, as close to the real environment as possible.

In the TRL 7-8, tests are performed in realistic environments. Not all applications reaches these levels, but mission critical application usually do.

TRL 9 is the successful completion of all tests. The application is ready to be used in a production environment.

7.3.1 Commercialization of AlexaMIC

AlexaMIC is an interesting technology, which potentially could be commercialized as a service available for people using MIC. However, in terms of the different TRLs defined by Mankins, the prototype is not yet ready for commercialization. AlexaMIC is somewhere between TRL-3 and TRL-4. In other words, AlexaMIC is a proof-of-concept implementation that shows what is possible using the two platforms that are integrated, but lacks many important features to be usable. Such as, a more natural language interaction, and support for metadata to construct better utterances. Implementing these improvements should be done as future work, if endeavors, such as commercializing of AlexaMIC is to be considered.

Before commercialization, AlexaMIC should be put through usability tests together with real users to make sure that utterances, and the correct features are implemented. This is part of the higher levels of the TRLs.

8 Future work

This chapter outlines the future work required to improve the implementation of AlexaMIC. The most important work is adding support for device metadata, which would make it possible to construct more natural sentences for the interaction between the users and AlexaMIC. The reason metadata is important, is that the metadata says something about the capabilities of the devices. In other words, which actions the devices supports, and what resources (state) that must be altered to perform these actions.

The next section is about using metadata to improve the language model for AlexaMIC.

8.1 Using metadata to improve the language model

Metadata about connected devices is important for the improvement of the interaction between the user and Alexa. Important metadata in this context is information about the location of devices and their capabilities, such as what actions are supported on the devices.

Below is an example of how this might look:

```
Example:

{
"location" : "bathroom",
"capabilities" : [
        "turnOn" : {
                        "resource" : "light",
                        "type" : "boolean",
                        "range" : ["True"]
                },
        "turnOff" : {
                        "resource" : "light",
                        "type" : "boolean",
                        "range" : ["False"]
                },
        "dim" : {
                        "resource" : "level",
                        "type" : "number",
                        "range" : [0, 100]
                }
        ]
}
```

The example shows a device with a bulb attached. The bulb supports both dimming of the light level, and turning the light on and off.

Intents are developed for every capability that the system supports. AlexaMIC thereafter, knows what action to perform based on the invoked intent, and the capability that is referenced during the invocation of the intent.

As an example, consider an intent named "turnOn", with the utterance "Alexa, ask MIC to turn on the {type}". AlexaMIC knows based on the metadata of the device, and the intent that was invoked, that it must alter a resource named light, to turn on the light.

Turning on the light described in the example is performed by the complete utterance: "Alexa, ask MIC to turn on the light".

This is extendable to other type of metadata such as locations. For example, a user might ask "Alexa, ask MIC to turn on the light in the bathroom".

The result of this, is that resources are altered in a cleaner and more natural way, resulting in a better interaction with AlexaMIC and the devices connected. This functionality is implemented as future work, either in AlexaMIC itself, or in the MIC platform. It makes sense that one of the underlaying ecosystem should supports the metadata describing a device.

8.2 Extending the MIC service class

The MIC Service currently supports small portions of the total API endpoints which are exposed by the MIC. Implementing the complete API introduces the possibility of extending the features of AlexaMIC to support device management, querying historical measurements and user account management. Thus, improving the overall experience of using AlexaMIC.

9 Conclusion

AlexaMIC is a proof of concept attempt at developing an integrating between MIC and Alexa. The problem definition of this study was to introduce voice control to the devices connected to the MIC platform.

AlexaMIC supports querying and setting the current state of devices that are connected to the MIC platform using the voice commands that are defined in the architecture chapter of this thesis. In addition, AlexaMIC gives users the ability to name and group devices, and perform group operations, such as listing all of the devices associated with a group, and querying the current state of all the group members.

Evaluations conducted in this study shows that neither Alexa or the MIC platform fulfills all the requirements for a good experience using speech recognition. Generally, the lack of metadata support in MIC, and the fact that Alexa utilizes syntactic instead of contextual speech recognition, creates a few issues in terms of the user friendliness of the application. Examples of issues that arose from this are that utterances must have invocation names, and that it is hard to think of every way a user might ask to turn off the light.

AlexaMIC is improvable by implementing some sort of metadata support, such as the example given in the future work section. This would allow the implementation of action specific intents, that leads to a more natural interaction with devices. An example of such an intent is the turnOff action, "Ask Alexa to turn off the light", where the light is a type of device supporting the turnOff action on a specific resource.

Speech recognition is a useful service. However, improvements are needed in terms of language understanding and context awareness.

Bibliography

[1] Amazon alexa. https://developer.amazon.com/alexa.

[2] Digitization driven by ai, iot and 5g. https://www.standard.no/Global/Bilder_webfase_2/IKT/Standard Morgen Digitalisering/02 Telenor.pdf.

[3] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. https://www.gartner.com/newsroom/id/3598917.

[4] Managed iot cloud. https://startiot.mic.telenorconnexion.com.

[5] Mqtt. https://mqtt.org.

[6] Oauth 2.0. https://oauth.net/2/.

[7] Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer.

[8] Voice assistant integration is the top smart-home trend at ces. http://www.businessinsider.com/voice-assistant-integration-top-smart-home-trend-ces-2018-1?r=US&IR=T&IR=T.

[9] What is aws lambda? https://docs.aws.amazon.com/lambda/latest/dg/welcome.html.

[10] TRW Defense Systems Group Barry W. Boehm. A spiral model of software development and enhancement. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=59.

[11] Milica Matić Igor Stefanović Istvan Papp Faculty of Technical Sciences University of Novi Sad Novi Sad Serbia Eleonora Nan, Una Radosavac. One solution for voice enabled smart home automation system. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8210611.

[12] Benjamin Stern Mahnoosh Mehrabani, Srinivas Bangalore. Personalized speech recognition for internet of things. https://ieeexplore.ieee.org/document/7389082/.

[13] John C. Mankins. Technology readiness levels. http://fellowships.teiemt.gr/wp-content/uploads/2016/01/trl.pdf.