

POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering



Master's Degree Thesis

Development and implementation of an automotive virtual assistant

Supervisor

Prof. Luciano LAVAGNO

Candidate

Andrea CELESTINO

Academic Year 2019-2020

Abstract

This thesis aims to the study of Intelligent Personal Assistants (IPAs) applied in the automotive field, proposing and testing an implementation.

In the first section, IPAs are studied from the first implementations to the current state of art, understanding how they became so popular, with an overview on the advances in speech and voice technologies.

Then an automotive personal assistant is designed and implemented, integrating some of the most popular technologies and expanding their functionalities.

The proposed approach is based on Alexa, one of the leader virtual assistant AI on the market; new capabilities, called *skills*, can be developed and customized to offer new voice experiences to the speaker and, in this case, the driver. Alexa will offer vehicle diagnostic and control features, profiles management and other services to showcase all the possibilities. The skill creation process is explained in detail with the definition of the *interaction model*, the voice-user interface, and the logic handling back-end code. The aim is to create a natural voice interaction, letting Alexa take some decisions, proposing to the driver assistance and letting the speaker talk in a more conversational way: for this, two non-canonical approach are proposed and implemented. Then, this thesis covers also the interaction between Alexa and the vehicle, with an infotainment system on board based on Android Automotive. After an overview on this operating system, an Android app, integrating the Alexa Auto SDK, is expanded to include a car status panel, providing and intuitive graphical interface through which both the driver and Alexa can interact.

Since the Alexa Auto SDK capabilities are limited, a communication system based on DynamoDB, a NoSQL database provided by Amazon Web Services (AWS) is implemented, so that the two systems can communicate and exchange more complex data.

Finally, in the last section of this thesis, the proper functioning of all the components are tested on an Automotive Development Platform, the SA8155P ADP air, with Android Automotive installed, the results of this project are shown and possible future implementations are discussed.

Table of Contents

List of Figures	III
1 Intelligent Virtual Assistants	1
1.1 State of Art	3
1.1.1 Trends with In-Car Virtual Assistants	4
1.2 Overview on speech recognition	8
1.2.1 Hidden Markov Models	9
1.2.2 Natural language processing	11
1.3 Proposed implementation overview	12
2 Alexa	14
2.1 "Who" is Alexa?	14
2.2 Alexa Skills	16
2.2.1 Alexa Skills Kit (ASK)	17
2.2.2 Development tools	19
2.2.3 How to host the service	22
2.2.4 Alexa Skills Kit Software Developer Kit	24
2.2.5 Other settings	24
2.3 The <i>interaction model</i>	25
2.3.1 Invocation name	25
2.3.2 Intents	26
2.3.3 Slots	28
2.3.4 Dialogs	30
2.3.5 The request	33
2.4 The logic	35
2.4.1 Intent handlers	36
2.4.2 Interceptors	44
2.4.3 Skill building	45
2.5 Location services and external APIs	45
2.6 Profile management: skill personalization	47
2.7 Control the vehicle	48

2.8	Complex and natural conversational flow	49
2.9	Alexa Auto SDK	51
2.9.1	Auto SDK Architecture and Modules	52
3	Android	56
3.1	Android overview	56
3.1.1	Android Automotive OS	59
3.1.2	Android Studio and the Apps	59
3.2	The Alexa app	66
3.2.1	Overview and GUI	66
3.2.2	Vehicle parameters	69
3.2.3	Alexa Car section	73
3.2.4	Notification Fragment	78
3.2.5	The skill card	79
4	DynamoDB	82
4.1	What is a NoSQL database	83
4.2	The connection between DynamoDB and the Android app	84
4.2.1	AWS Amplify	84
4.2.2	AWS and Amplify configuration	89
4.3	The connection between DynamoDB and the Alexa skill	90
5	Conclusions	93
5.1	Module Testing	93
5.2	System testing and results	98
5.3	Future improvement	99
5.4	Conclusions and Final Remarks	100
	Bibliography	101

List of Figures

1.1	HAL from "2001: A Space Odyssey"	1
1.2	Clippy the Paperclip	2
1.3	Overview of In-Car virtual assistants	6
1.4	Interest in voice services on vehicle, by demographic	6
1.5	Interest in same brand of in-home voice service on next vehicle	7
1.6	Modify likelihood to buy from a car company instead of a branded voice service	7
1.7	Basic block diagram of a speech recognition system	9
2.1	Amazon Alexa logo	14
2.2	Amazon Echo 1st Generation	15
2.3	How and Alexa skill works	17
2.4	Alexa developer console: skill main page	20
2.5	Alexa developer console: test page	21
2.6	Interaction model file	25
2.7	Some of the intents mapped	27
2.8	Intent definition with sample utterances and slot	29
2.9	Intent definition in JSON format	29
2.10	Dialogs settings for CheckIntent	32
2.11	A skill request sent by Alexa	33
2.12	Slot and slot resolution	35
2.13	Handler implementation for handling CheckIntent requests	37
2.14	Attributes included in a request	38
2.15	The manual configuration of the DynamoDbPersistenceAdapter	39
2.16	Definition of the interceptors to load and save attributes	40
2.17	JSON response body	41
2.18	JSON request with a <i>CheckIntent</i>	43
2.19	JSON response with a <i>HelpUserIntent</i>	43
2.20	Request flow and interceptors	44
2.21	Interceptor to log requests.	44
2.22	How a skill is built	45

2.23	How the Lambda function performs the reverse geocoding given the coordinates	46
2.24	How the skill react when an unknown user speaks.	48
2.25	How the skill react when an known user, with a profile already saved, speaks.	48
2.26	Intent for opening and turning on car component	49
2.27	Flowchart showing intents, variables and directives through which the conversation could go	51
2.28	Alexa Auto SDK architecture and (some of the available) modules .	52
2.29	High-level message flow for displaying visuals	54
2.30	The TemplateRuntimeHandler implementation extending the TemplateRuntime	55
3.1	Android logo.	56
3.2	Android OS architecture	58
3.3	Vehicle Hal architecture	59
3.4	Activity lifecycle	62
3.5	Hierarchy of views	63
3.6	The same fragment could be reused in different ways	64
3.7	Activity lifecycle with a fragment	65
3.8	Alexa Android app: weather card	66
3.9	Alexa Android app: languages menu	67
3.10	Alexa listening state	68
3.11	Layer structure analysis	68
3.12	jsonschema2pojo converter main page	70
3.13	Attributes Java Class	71
3.14	STATUS Java Class	71
3.15	Layer structure analysis	72
3.16	Example use of the <i>set</i> and <i>get</i> methods	72
3.17	Car status panel	73
3.18	How the <i>alexaCar</i> button is defined in the XML file and how the fragment layout is displayed as a preview in Android Studio	74
3.19	The button declaration and its click listener	74
3.20	The function that puts the car panel on the screen.	75
3.21	Car status panel main layouts structure.	75
3.22	Car status panel: the fragment layout blueprint	76
3.23	The listener creation.	76
3.24	Setting fragment inflated to control the mirrors	77
3.25	Warning menu fragment inflated.	77
3.26	The notification fragment displayed on the top of the screen.	78

3.27	The function that displays programmatically the notification fragment, setting the argument values.	79
3.28	How the app distinguish between a vehicle-skill card and a generic card.	80
3.29	The card sent when asking the skill if there is a problem with the battery, rendered in the car section.	81
3.30	How a simple request is rendered placed in the general section of the app.	81
4.1	DynamoDB logo.	82
4.2	The DynamoDb console with the available table	83
4.3	The table	83
4.4	AWS Amplify logo.	84
4.5	GraphQL schema	85
4.6	<i>VehicleData</i> auto generated class by Amplify	86
4.7	Function to retrieve vehicle parameters from the database.	87
4.8	Function to update vehicle parameters to the database.	87
4.9	How AppSync works	88
4.10	Subscription creation.	88
4.11	<i>onUpdate</i> behavior.	89
4.12	How the Lambda function performs a GraphQL mutation.	91
5.1	<i>CheckIntent</i>	93
5.2	The longest conversation possible	94
5.3	<i>GetPositionIntent</i>	95
5.4	How the skill cards are displayed on any Alexa device with a screen.	96
5.5	How the skill card is displayed on a smartphone	96
5.6	The SA8155P ADP.	97
5.7	The whole system architecture.	98

Chapter 1

Intelligent Virtual Assistants

An intelligent virtual assistant (IVA) is a software agent that assists people to perform basic tasks or provides services via natural language [1]. They are called in many different ways: intelligent personal assistants (IPA), digital personal assistants, voice assistants, or mobile assistants[2]. These entities combine set of technologies and tasks, such as speech recognition, language understanding, language creation, dialogue management and speech synthesis, to respond to user requests [3].

They have always been part of science fiction imaginary, and now they are par of our everyday lives: voice technologies have always been very fascinating. It is sufficient to mention Star Trek, HAL, from "2001: A Space Odyssey, or Samantha from "Her" to understand how they affected our culture and science fiction imaginary, building expectations and predictions for what speech recognition could look like in our world. And now, the diffusion of these smart assistants is constantly spreading.



Figure 1.1: HAL from "2001: A Space Odyssey"

Their development is not new: for nearly 100 years, electronics and computer industries have tried to reach the goal of voice interaction. In 1911, Radio Rex, a simple wood toy, was one of the first attempts in using speech recognition: when the owners would shout "Rex!", a little wooden dog would come out of its house [4]. But the first natural language processing computer program, or chatbot, ELIZA, was developed by MIT in 1960s, to demonstrate that the communication between man and machine was superficial [5]. This experiment gave name to the *ELIZA effect*, the anthropomorphisation of computer behaviors, assuming unconsciously that they are analogous to humans ones. From the 90s, digital speech recognition technology became a feature of personal computers.

And IVAs were already present almost 20 years ago. They were basic assistant and usually annoying: everybody should remember Clippy the Paperclip, the Microsoft office assistant [6].

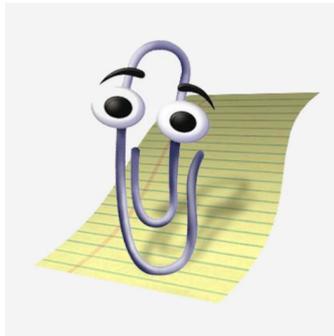


Figure 1.2: Clippy the Paperclip

The first modern digital virtual assistant, integrated on a smartphone, was Siri introduced by Apple on the iPhone 4S in 2011. Then, many companies decided to release their assistant, such as Alexa, Cortana, Google Assistant: now they are used everywhere and they have become part of our common used technologies.

Voice technologies are the natural evolution of human machine interfaces (HMI), that is a software that presents information to a user regarding the state of a process, accept commands and execute the operator control direction. Usually, information is displayed with a graphical user interface, but now voice interaction is becoming its key technology [7]. As the machine are becoming more complex to operate, HMI is focusing on reducing the human cognitive workload, to optimize the human task efficiency.

The new wave of mobile and ubiquitous virtual assistants is connected to the current shift of the form of interactions, from touch to speech-based interfaces [8] and to the incorporation of these assistants in “smart” devices such as smartwatches, smart televisions and smart cars.

1.1 State of Art

Virtual assistants are becoming more intelligent and complex, but also more and more integrated within our daily lives. Google Assistant, Apple's Siri, Amazon's Alexa or Microsoft's Cortana are just some of the more famous ones, integrated in many devices such as smartphones and speakers. These companies will only continue to integrate them more and more into the everyday products, fueling the "Internet of Things" movement. They use different techniques to design their IVA, based on the application and its complexity [9]. For example, Cortana dialogue system is improved using the Microsoft Azure Machine Learning Studio, while Google uses Deep Neural Networks (DNN) to highlight the main component of the dialogue system and then applies a new deep learning architecture. Alexa's case, the main focus of this thesis, will be deeply discussed in chapter 2.

All these companies are trying to focus in the core technologies for their dialogue systems, such as automatic speech recognition, text-to-speech, synthetic talking face and dialog management, to improve them more and more. They are also pushing themselves to multi-modal input modes, using the voice combined with touch and gesture, to provide a more immersive experience.

With the advance of machine learning and artificial intelligence, now IVAs are systems capable of learning the interests and behavior of the user and respond accordingly [10]. They can use AI-based functionalities to proactively perform action or, in most cases, suggest contents, that could be interesting for the user based on the analysis of previous choices. These behavior follows more the concept of "virtual butler" [11], that simply execute queries. But even in this role, users usually expect more and exceed the actual performance of IVAs, reducing the willingness to use them in a broader way [12]. The advances in these fields brought voice technologies and voice assistants in our lives with variety of applications: they are not only useful for one specific purpose, but provide many functionalities through an unified interface, that is very natural, designed for mobile contexts, bringing an ubiquitous interaction and that is integrate in an IoT ecosystem. [13]

The ease with which is possible interact with them makes the difference: recent technological developments in robotics, artificial intelligence, natural language processing (NLP) and user interfaces for mobile and ubiquitous interaction generated a renewed activity around virtual assistants. And the reasons about this sudden expansion have to be found in the technological progress that has increased the quality of results like accuracy, speed and, in particular, capability of recognizing different speakers on the basis of their accent, dialect, thus establishing a smooth and intuitive Human-Machine interaction: now a computer is capable of understanding a human without having to interact with a screen or a keyboard.

Although the technologies about all the virtual assistants are different and complex, they perform these four essential steps:

1. Voice Recognition: the voice analog signal is recorded and converted into a binary file, to be sent to the server.
2. The Speech record is sent to the IVAs server in the cloud: the recordings contain background and other noises, so the server filters and analyzes it. All these process are heavy to compute, for this reason it is the server that accomplishes this task, without weighting on the limited resources of the device.
3. The meaning is understood: natural language processing is performed, to interpret the audio.
4. The meaning is transformed into instructions.

So, virtual assistants are found in smartphones and homes, trying to accommodate user requests and helping them. But, what about cars?

1.1.1 Trends with In-Car Virtual Assistants

HMI technology is used by almost all industrial organization and in every field, especially in the automotive field: the number of in-vehicle infotainment systems (IVIS) is rapidly increasing. Those systems offer the potential to greatly enhance mobility and comfort: they combines traditional car functions with "smart" features, from the radio, to social networking. One of the most used IVIS is a big screen based navigation system. Manufacturers can offer various features in their products by choosing the HMI type that best aligns with their brand attributes and their customers' preferences. One of the latest trend is the addition of an increasing number of screens inside vehicles. All these screens increase the risk for excessive and, more important, dangerous level of inattention to vehicle control tasks. The majority of car accidents, about 93%, are caused by human error, with the inattention to the road as a contributing factor [14]. The development of new advanced driver-assistance systems (ADAS), combined with the use of the conventional vehicle-safety measures, such as seatbelts and airbags, offers great potential to improve road safety, reducing driver errors. Anyway, many studies show that interacting with the infotainment system divide the attention between the road and the desired task. For example, studying the glance analysis of the driver eye movement confirms that, during the radio tuning, the eye focus for longer time off road [15]. Indeed 80% of crashes and 65% of near crashes involve driver distraction form secondary task [16].

IVIs and user-friendly designs are a challenge that the automotive industry is facing right now, trying to overcome the potential increase in driver distraction. For wxample, Driver Distraction Test Rig (DDTR) is used to test protocols for evaluating the level of driver distraction imposed by infotainment systems [17].

IVIs innovated HMI systems, altering drivers' in-car interactions, but new problems arose. To overcome some of the safety problems with using LCD screens, voice technologies can be adopted, to reduce the time looking at screens.

Already from the 80s, the use of voice technologies and artificial intelligence to improve safety and reliability in vehicle was already discussed [18]. Surveys reveal that car manufacturers are implementing more and more voice-activation systems [19]. Voice-based interfaces introduce less distractions when executing in-vehicle secondary tasks, compared with visual and manual interfaces [20]. This improves vehicle control: for example, reducing the workload and improving the lane keeping performances [21]. However, compared with the other interfaces, voice interface has its own disadvantages, such as the lack of intuitiveness. Most of the traditional voice systems are designed with a limited vocabulary in order to reduce complexity and errors in the speech recognition processes. This makes it difficult to remember these specific commands, resulting in low usage of voice-activation systems. Using more intuitive commands, would decrease the time that the driver glances does not focus at the windscreen: [22].

Virtual assistants, fueled with their advanced AI and machine learning approaches, could break through this barrier, making the voice interactions more intuitive and natural, providing all the benefits of this type of interface inside vehicles.

They are already arriving inside the vehicles, but they are not really integrated with the vehicle. All automotive companies are establishing strategic partnerships with big ICT companies to deliver virtual assistants. Their deployment follows two main strategies, namely the integration of existing virtual assistants originally developed for smartphones and tablets, and the offer of virtual assistants specifically conceived for the car context. The former approach is currently the most common. They are focusing on three broad areas:

1. safety-related functionalities related to car navigation (including AI-based self-driving tasks);
2. customized infotainment;
3. gateway to IoT, such as car controls;

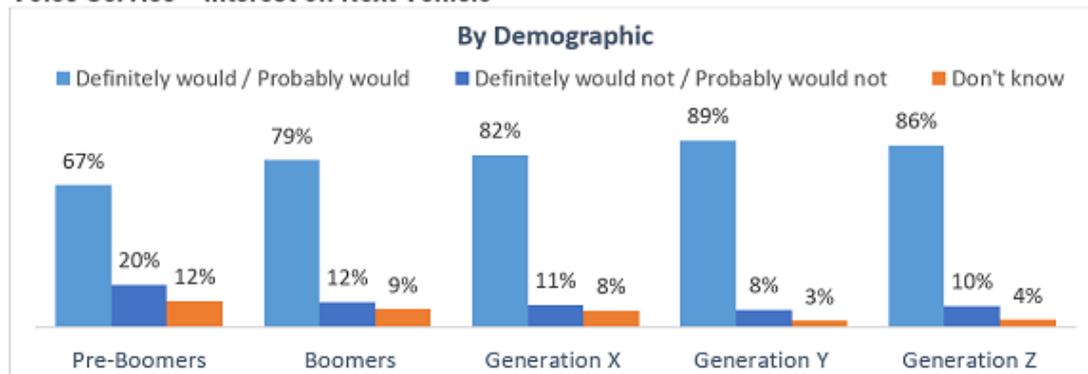
Virtual Assistant	Adopted by	Commercially Available	Focus Area
Google Assistant	Daimler Mercedes-Benz, Hyundai	Yes	Car Navigation; IoT applications
Cortana (Microsoft)	BMW, Nissan	Yes	Car Navigation
Alexa (Amazon)	Ford	Yes	Infotainment
OnStar Go (IBM, GM)	General Motors	Yes	Infotainment & Car Navigation
SIRI (Apple)	Most car manufacturers (via Apple CarPlay app)	Yes	Generalist
Yui (Toyota)	Toyota	No (concept stage)	Car Navigation; Virtual companion
HANA (Honda)	Honda	No (concept stage)	Car Navigation; Virtual companion
Sedric	Volkswagen	No (concept stage)	Virtual companion; Infotainment

Figure 1.3: Overview of In-Car virtual assistants

The design of virtual assistants is advancing more and more, however, their perceived usefulness and related social acceptance is still far from optimal. But the car industry is confident on the success of in-car virtual assistants, both in their role of “virtual butlers” supporting users’ requests and in the more advanced role of proactive agents with autonomous task execution and decision-making capacity [13].

The interest in voice technologies adopted inside vehicles is increasing as the time goes by, especially in the young generations, since they allow to maintain focus while keeping the driver connected to his digital life.

Voice Service – Interest on Next Vehicle



Generations are defined as Pre-Boomer (born before 1946); Baby Boomer (1946-1964); Gen X (1965-1976); Gen Y (1977-1994); Gen Z (1995-2004).

Figure 1.4: Interest in voice services on vehicle, by demographic

The familiarity with the voice technology is also an important component in this, because the desire to carry the same brand of in-home service on the vehicles is higher.

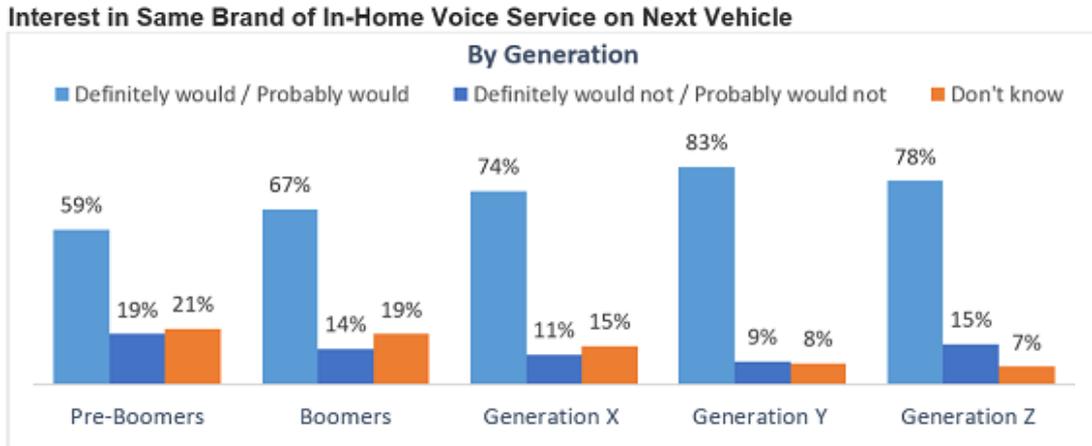
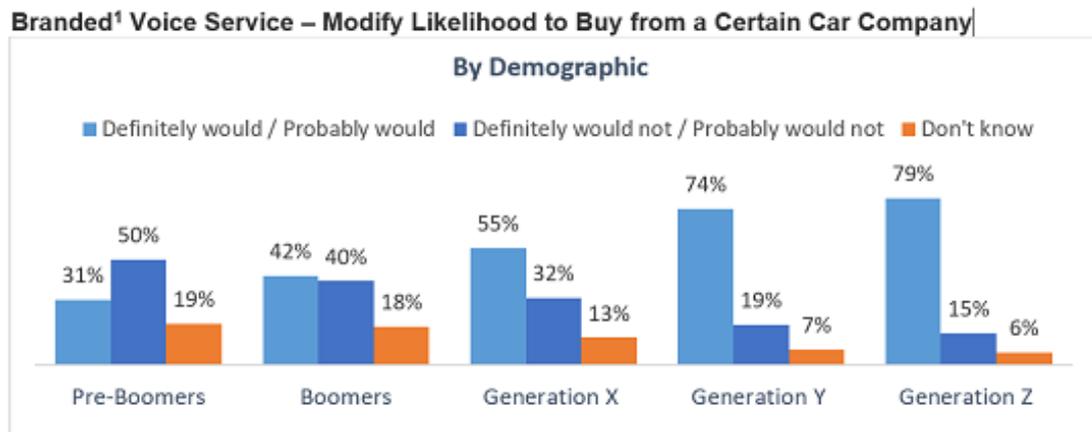


Figure 1.5: Interest in same brand of in-home voice service on next vehicle

They showed also more interest in remaining loyal to the same branded voice service other than to the specific vehicle brand.



¹Branded voice service examples include Alexa, Cortana, Siri, etc.

Figure 1.6: Modify likelihood to buy from a car company instead of a branded voice service

So, interest in voice technologies on vehicle is growing as the younger generation, that are the most interested ones. Automakers are aiming on this, to fidelize the young generations. [23] Summing up, virtual assistants are moving the first steps

inside cars, being directly integrated in the infotainment system, connected to the vehicle from the smartphone, or with dedicated devices, such as Echo Auto. In this last case, most of the customers complained when it was released for the absence of real automotive features: it was simply an echo speaker, with minimal actual integration with the car. As all the others implementation, they still are not well integrated.

So, in this thesis work, it will be shown how to implement an automotive virtual assistant that is an actual component of the infotainment system and that can interface with the car, being able to provide diagnosis features and others useful for automotive applications and to control the vehicle

1.2 Overview on speech recognition

Before moving on with the development of this thesis work, some basic information to explain how speech recognition work is essential to deeply understand the subject, even though the tools provided to developers allows to build voice experiences actually do not require this knowledge.

Speech recognition, also known as automatic speech recognition (ASR), is a cross-disciplinary subject, with the voice as the research object: the voice signal is converted into commands that a machine can understand. It involves many fields of knowledge, such as psychology, linguistics, acoustics, phonetics, signal processing and computer science, but also mathematics and statistics. The ultimate goal is to achieve natural communication between man and machine. It is considered to be one of the most complex area of computer science [24]. The variety of human speech makes development of this technology very challenging.

It can be achieved in many different ways, but the more advanced solution exploits artificial intelligence and machine learning, integrating grammar, syntax, composition of audio and voice signals to process human speech.

Speech recognition is essentially a *pattern recognition system* [25], consisting of four essential units:

1. a feature extractor, for selecting and measuring the most important properties of raw input data in a reduced form;
2. a pattern matcher, to execute comparison between the input pattern and a reference patterns using a distance measure;
3. a reference templates memory, against which the input pattern is compared;
4. a decision maker, to make the final decision as to which reference template is the closest to the input pattern.

The voice is recorded through a microphone, transforming the voice signal into an electrical signal. The system recognize a voice model according to the human voice characteristics, then analyzes the input signal, extracts required features and decides a template of speech recognition. The computer is used to compare the voice template and the characteristics of the input signal.

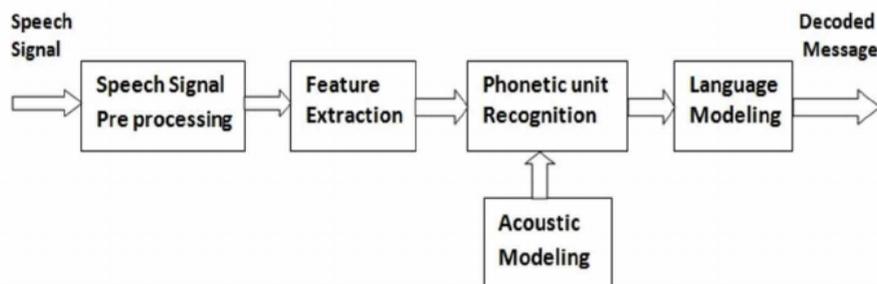


Figure 1.7: Basic block diagram of a speech recognition system

There are various algorithms and methods to recognize speech and improve the accuracy to transcript it into text. this. Two of the most used will be discussed, to give a general overview on these approaches. These two are at the fundamental pillars of Alexa, that is the main focus of this thesis work.

1.2.1 Hidden Markov Models

Hidden Markov Models (HMM), is a statistical model applied to the modeling of acoustic signal. Founded in the 1970s, so far, it is still considered to be the most reliable approach to achieve fast and accurate speech recognition.

In engineering, one of the most relevant problem is the representation of a signal of the physical world with a mathematical model, that allows to predict its behavior. There two types of models , given a fixed time instant t_0 and a signal represented by $x(t)$ [26]:

- deterministic models: given t_0 , the signal takes the value $x(t_0)$ that is a priori known;
- statistical models: the value taken $x(t_0)$ is describable only from a statistical point of view, by means of a stochastic variable and its probability density function.

Human voices can be modeled with the latter one and HMM is one of them. This model compares the waveform of a single word against what comes before, what comes after and against a dictionary, to figure out what has been said.

Markov Chains

A Markov Chain, at the basis of the construction of the HMM model, is a model that describes a sequence of possible events [27]. Markov chain makes a strong assumption: the probability of transitioning to any particular state depends only on the current state. The states before the current one are not relevant.

It is characterized by:

- $Q = q_1, q_2, \dots, q_N$ a set of N states, where q_i is a state variable;
- $A = a_{11}, a_{12}, \dots, a_{n1}, \dots, a_{nn}$ a transition probability matrix, where a_{ij} is the probability of passing from state i to j ;
- $\pi = \pi_1, \pi_2, \dots, \pi_N$ an initial probability distribution, where π_i is the probability that the chain starts in state i .

At each time instant, the *Markov assumption* is valid: when predicting the future state, the past does not matter; it matter only the current state. So the probability P of being in state q_i is

$$P(q_i|q_1, \dots, q_{i-1}) = P(q_i|q_{i-1})$$

and a state transition takes place in this way:

$$a_{ij} = P(q_{t+1} = j|q_t = i)$$

The entire process is observable as each state corresponds to a physical *observable* event.

The Hidden Markov Model

The HMM makes it possible to predict the probability of a certain sequence of states, considering not only the observable events, as in traditional Markov chain, but also *hidden* ones. It allows to incorporate hidden events, such as a tag associated to a part of the speech, into a probabilistic model [24]. Thus the observed events are the words in input, while the hidden events are what part of the speech that word is associated to. Labels, such as words, syllables, sentences, are assigned to the input sequence, mapping the input signal and allowing to determine the most appropriate label sequence.

This model is characterized by:

- $Q = q_1, q_2, \dots, q_N$ a set of N states, where q_i is a state variable;
- $A = a_{11}, a_{12}, \dots, a_{n1}, \dots, a_{nn}$ a transition probability matrix, where a_{ij} is the probability of passing from state i to j ;

- $O = o_1, o_2, \dots, o_T$ a sequence of T observation, each drawn from a vocabulary $V = v_1, v_2, \dots, v_V$;
- $B = b_i(o_t)$ a sequence of observation likelihoods, the *emission probabilities*, each expressing the probability of an observation o_t being generated from a state i ;
- $\pi = \pi_1, \pi_2, \dots, \pi_N$ an initial probability distribution, where π_i is the probability that the chain starts in state i .

Other than the Markov assumption:

$$P(q_i|q_1, \dots, q_{i-1}) = P(q_i|q_{i-1})$$

A second essential assumption is made, the *output independence*, that implies that the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other states or any other observations:

$$P(o_i|q_1, \dots, q_i, \dots, q_T, o_1, \dots, o_T) = P(o_i|q_i)$$

In 1989 [26], it was introduced the idea that HMM should be characterized by three fundamental problems:

1. *Likelihood*: Given an HMM $\lambda = (A, B)$ and an observation sequence O , determine the likelihood $P(O|\lambda)$.
2. *Decoding*: Given an observation sequence O and an HMM $\lambda = (A, B)$, discover the best hidden state sequence Q .
3. *Learning*: Given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B .

Summing up, the HMM is a complete expression of the acoustic model of the voice, using statistical methods to create a voice recognition search algorithm that obtain good results and that can be used for continuous speech recognition. The drawback are the long and sophisticated calculations and the long training sequence [28] [29].

1.2.2 Natural language processing

NLP is not a specific algorithm. Instead, it is the area of artificial intelligence which focuses on the interaction between humans and machines through language through speech and text. It gives the computers the ability of understanding in the same way humans can. NLP combines computational linguistics, in other words, the modeling of human language with rules, with statistical models, machine learning

models and even deep learning models. In this way, a machine can process human language, "understanding" its meaning.

Human language is full of ambiguities [30], but natural language-driven application should recognize and understand it even with the presence irregularities such as homonyms, idioms, metaphors, homophones, grammar exceptions and variations. Human speech or text is broken down and analyzed. Some of the task to accomplish this is are [31]:

- Speech recognition or speech to text;
- Grammatical tagging, to determine which word is which part of the speech.
- Word sense disambiguation, to distinguish the sense of a word based on its context;
- named entity recognition, or NEM, that identifies words and phrases as useful entities.
- Natural language generation, that is the opposite of speech recognition.

1.3 Proposed implementation overview

This master thesis project has been developed in collaboration with Marelli Europe and had as a goal to continue the development of an automotive infotainment system based on Android.

The main goal is to develop an automotive virtual assistant using one of the most used and famous virtual assistant AI, Amazon Alexa. The initial integration steps were already carried out by Marelli, with the development of an Alexa app for Android that provides Alexa's core functionalities. The aim is to implement a functioning system capable of interact with a vehicle with an Android system on board: Alexa must be able to retrieve data from a vehicle to allow the user to detect malfunctions and errors. Alexa becomes the voice interface between the user and the car, providing information about the status of the vehicle and, in some cases, assistance. This system aims specifically to overcome the base functionalities provided by Amazon and to provide the diagnosis aspects and features, to make it is possible to check the status of the car and of specific components, detecting if something is not working correctly. But the same infrastructure will be used also to actively control the vehicle, both from inside and outside the car, giving the possibility to remote control it. Also, to expand the experience of having a personal automotive assistant, additional features will be implemented to retrieve some information about the surroundings.

This project is composed by 3 components:

1. The Alexa skill, that implements the voice-user interface.
2. The Android app, that runs on the vehicle: collects and provides data; it also provides the graphical user interface for the driver.
3. The connection between the two through a database, DynamoDB, a NoSQL database, to which both the skill and the app are connected.

The proposed approach exploits all services offered by Amazon. This is only one of the many possible ways to implement a system like this, but it is one of the most intuitive and reliable way, because it relies on Amazon services that are well interconnected and integrated between each other.

The main features that are proposed are:

- Check if a single component works properly, giving information about possible source of the problem and possible solutions;
- check the whole vehicle to verify If something Is wrong;
- verify the status of a car part, for example, if the vehicle is locked;
- ask more detailed information for a warning message;
- helping find a solution for some car malfunctions;
- ask information of the car, as the date of the next car service;
- control the vehicle, remotely or not;
- personalize the driver experience by selecting and setting a driver profile, depending on the user speaking;
- location based features;
- general purpose; these are implemented to show the possibilities of an Alexa skill and expand the feeling of a personal assistant; since it can do HTTP requests, it can retrieve information from any APIs. In this case, some APIs of HERE Map are used.

After the definition of the proposed features, the design of the automotive virtual assistant can begin. The first step of the implementation of a voice assistant is the voice user interface, how the user can actually interact with the vehicle just by speaking. Alexa's capabilities must be expanded and customized.

Chapter 2

Alexa

2.1 "Who" is Alexa?

“Alexa is Amazon’s cloud-based voice service available on hundreds of millions of devices from Amazon and third-party device manufacturers”.[32]

Alexa is the virtual assistant AI technology developed by Amazon, arrived in 2014 with the launch of the smart speaker Amazon Echo, inspired by the Star Trek computer, with the aspiration of revolutionizing daily convenience using artificial intelligence. And now, in 2020, it is possible to say that they did it since now it is really common to have one of the Alexa powered devices at home. In just five years, Amazon has sold more than 100 million Alexa-powered devices.



Figure 2.1: Amazon Alexa logo

Alexa can be considered the leader between assistants at home. But the goal is not becoming the best assistant as a whole, competing with Siri and Google Assistant as mobile assistants, it is focusing on the idea of an “ambient user interface”. So Alexa started with the home, and now it is headed to cars. Amazon is focused on the places where it could make sense to speak out loud. The goal is to get as many devices to use Alexa as possible, and to make those devices do as much as possible. The focus, going forward is to make Alexa more conversational and become less awkward to use. [33]

Echo ushered in the convenience of voice-enabled ambient computing: Alexa-powered Echo broke the human-machine interaction paradigms, shifting how customers interact with lots of services, like find information on the Web and control smart devices.



Figure 2.2: Amazon Echo 1st Generation

What really made the difference in the launch of Alexa was how good she was in these four fundamental AI tasks [34] :

- Wake word detection: detect the keyword “Alexa” to get AI’s attention;
- Automatic speech recognition (ASR): convert audio streamed to the Amazon Web Services (AWS) cloud into words;
- Natural-language understanding (NLU): extract the meaning of the recognized words; in this way Alexa can take the appropriate action to satisfy customers’ requests
- Text-to-speech synthesis (TTS): convert Alexa’s textual response into spoken audio.

From the beginning, Alexa is continuously improving, reducing recognition errors, errors in NLU and even the gap in the naturalness between Alexa’s speech and human speech. All of this was achieved by combining machine learning, in particular, deep learning, with large-scale data and resources available through AWS.

Alexa is becoming more and more smarter, by reducing the reliance on supervised learning, i.e., building ML models on manually labeled data. Most of the advances in AI have been result of supervised deep learning, in which neural networks learn by analysing thousands of training examples labeled by a human annotator. Lately, Alexa uses a new self-learning paradigm that enables Alexa to automatically correct ASR and NLU errors without an annotator in the loop [35]. In this approach, ML is used to detect potentially unsatisfactory interactions with Alexa, detecting when a customer is unsatisfied with the response and interrupts the response and rephrase the request. Alexa is now self-correcting millions of defects per week.

Alexa's responses are becoming more natural, since neural networks are being used for text-to speech synthesis, resulting in more natural-sounding speech and making easier to adapt Alexa's TTS system to different speaking styles.

Thus Alexa, powered with these advanced technologies, provides assistance in our every day lives. It is capable of voice interactions, setting alarms, playing music, provide real-time information and also control smart devices, becoming a home automation system.

But how can Alexa deliver so many features?

2.2 Alexa Skills

Alexa's capabilities are called skills: the ability to look for something on the web, enjoying an audio book, or playing games are skills.[36]

It is also possible to teach Alexa new skills, to which customers can access. They are the equivalent of an app for a smartphone, a specific set of features, triggered by voice interactions. In the Alexa Skills Store, new skills can be discovered via the Alexa companion app on a smartphone and only if the user enables it, he can actually invoke it and use it. The skill is linked to the Amazon account and it can be invoked by any device associated to that account.

But skills can be installed by voice too or even Alexa can propose them to satisfy the customer requests, when she thinks that a particular skill could satisfy the customer requests. Thus, Alexa's capabilities can be continuously expanded.

When a customer speaks, Alexa processes the speech to determine the customer request, providing a response. The voice recording is sent over the internet to the Alexa Voice Services (AVS), the Amazon cloud-based service that is the real brain of Alexa. This converts the audio recording into commands: the AVS breaks the recording down to convert into instructions, detecting essential words, compatible with skills or functions, and invoking those skills; then the request is processed and a response is sent back, allowing Alexa to speak out with the relevant information. Everything is done within a fraction of a second.

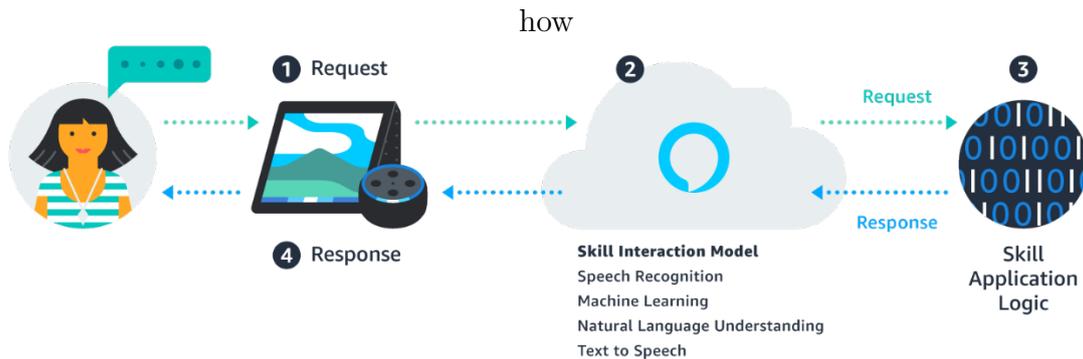


Figure 2.3: How and Alexa skill works

In other terms, Alexa is just an interface between the user and the skill logic: she allows developers to use a powerful and complete system to recognize and understand natural speech and to speak back to the user, without having to actually have deep knowledge of machine learning techniques and voice technologies.

In this process, it is possible to see how the voice interface and the logic-handler are two separate entities. Indeed, they are the two main components of a skill: one, the interaction model, that defines the voice user interface; the other is the skill logic, the back-end code. Alexa's job is to collect the data required to compose a correct request through the interaction model and then to send it to the back-end code.

Alexa communicates with the skill service via a request-response mechanism using HTTP over SSL/TLS. When the skill is invoked, the service receives a **POST** request, consisting of a JSON body and generates a JSON-formatted response. Thus, to expand Alexa capabilities, a new skill is needed for this project. Amazon allows everyone to create and publish new skills with the Alexa Skill Kit (ASK), providing guidelines for the skill creation.[37]

2.2.1 Alexa Skills Kit (ASK)

The ASK is a collection of the self-service APIs, tools, and sample codes that allows an easy and quick way to develop an Alexa Skill.

The first step is deciding what the skill should do. There are different types of skill that can be created, depending which functionalities are wanted: this determines how they integrate with the Alexa service and what is needed to be built [38]. The ASK supports the creation of many types of skills, that differ both in the final goal and in the implementation of the interaction model, but they can be grouped in two main categories:

1. Skills with a *pre-built model*, as Smart Home Skill API, Video Skill API, Flash

Briefing Skill API and Music skill API. These skills types give less control over the user's experience, but the development is simplified because the voice user interface does not need to be created and take advantage of pre-built model. They are also easier to be invoked by users, because they don't need to be "activated" with an invocation name: they are invoked with simple requests, such as "Alexa, turn off the lights". These are skills with functionalities very limited in a specific field, like controlling a smart home device or select and listen to audio content streamed through an Alexa-enabled device. The developer, in this case, has just to define how the skill responds to a particular directive or provides the data needed for the user.

2. Custom skill with a *custom interaction model*. It gives the most control over the user's experience and it can handle any type of request, as long as the developer define the words users say to invoke the requests and create the code to fulfill them. It is the most versatile type of skill, but also the most complex, since the interaction model has to be built from scratch.

Due to the nature of this project, a *custom skill* is the best choice: it allows to personalize as much as possible the user experience and can be used to satisfy any type of request: an automotive personal assistant should be as versatile as possible. The components of a *custom skill* are:

- The *interaction model*, the voice user interface by which the user can communicate with the skill. It is the mapping between sets of *samples utterances* that a user can say to invoke the *intents*.
- The *intents*, part of the interaction model, are the actions that user can invoke, the core functionality of the skill. They are the requests that the skill can handle.
- The *invocation name*, the name that Alexa uses to identify the skill and is included by the user when initiating a conversation with the skill.
- Set of images, audio files and video files to be retrieved when they are wanted to be included in the skill; they must be stored on a publicly accessible site by a unique URL.
- A cloud-based service that accepts those intents as a structured request and elaborates them. It must be accessible over the internet. The skill is linked to this service.
- A configuration that connects all the above components together, so that Alexa, when detecting the invocation of the skill, can interpret the speech according to the *interaction model*, routing the requests to the service of the

skill, the back-end code, set as endpoint. This is created in the developer console.

When speaking to an Alexa-enabled device, this is what happens:

1. The user's speech is recorded and sent to the Alexa service in the cloud.
2. Alexa recognizes that the recording contains an intent for a specific skill.
3. Alexa structures this information into a request and send it to the cloud-based service defined for the skill.
4. The service gets the request and elaborates it
5. The service sends back to Alexa a structured response, with text to speak to the user.
6. The Alexa-enabled device speaks the response back to the user.

2.2.2 Development tools

Skills can be created in many different ways, so it is worth spending some words to sum up all the different methods and tools that can be used, to show the differences and the advantages of each approach.

Alexa Developer console

It is the first resource that can be used and it is the most intuitive and easy way to create and manage skills. It guides the developer, providing a streamlined experience to create, manage and publish skills.

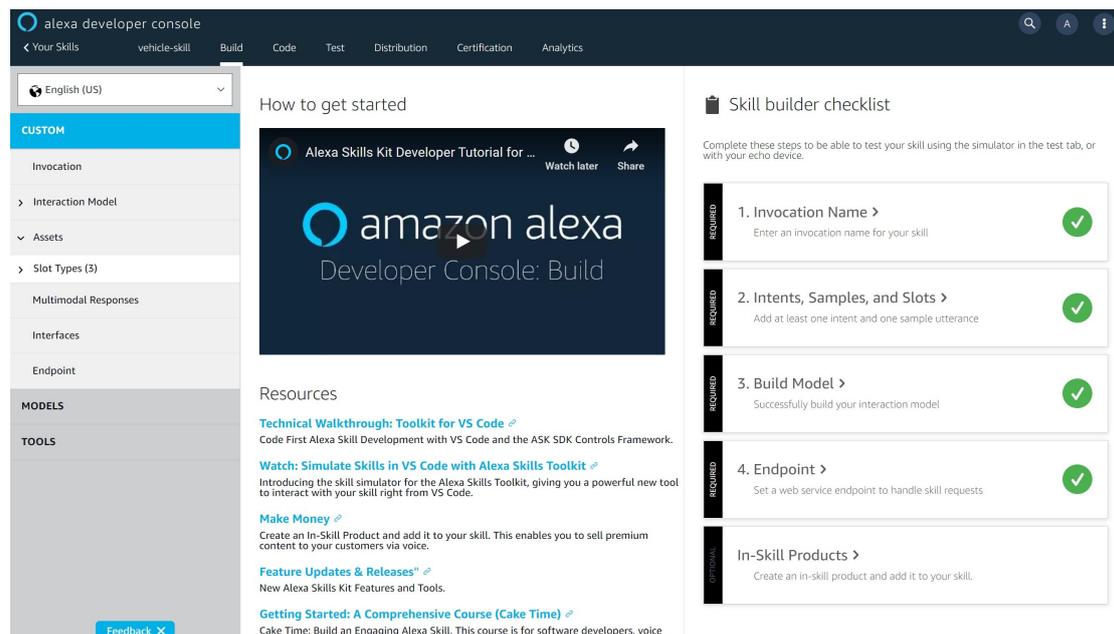


Figure 2.4: Alexa developer console: skill main page

It allows to set up all the skill configurations, the interaction model and specify the endpoints of the service, where the skill code is. This console let define the interaction model and manage all permissions required in an easy and intuitive way. It can also let the developer code online, but only for a specific type of skill, an Alexa-hosted skill: in this case, the backend code is completely managed by Alexa and Amazon services, but this will be explained better later. The problem is that only this type of skill code can be accessed through the console. Indeed, for any other option, this console feature is not accessible and the development must relays on other tools.

The developer console also allows to test all the skills with either text or voice, submit them for certification and also use analytics to review metrics, such as number of customers and intents invoked.

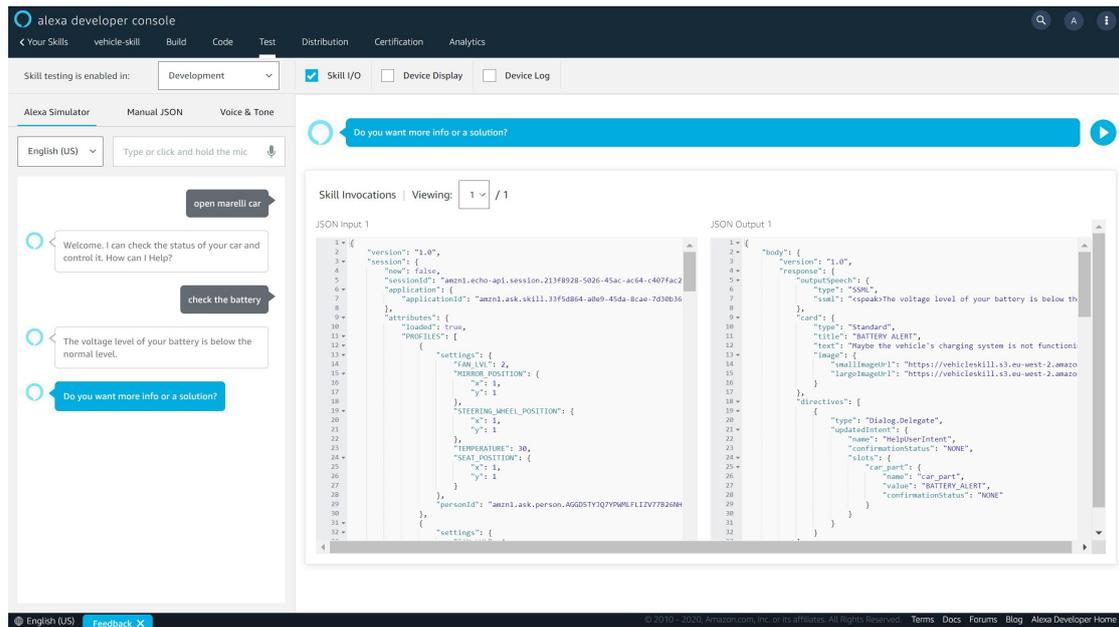


Figure 2.5: Alexa developer console: test page

Considering that there are limitation for the coding part, but remembering that the interaction model and the logic-handler code are two separate entities, the Alexa Developer Console remains the best tool for the voice-user interface design for its ease to use. Usually it is used in combination with Alexa Skills Kit Command Line Interface (ASK CLI).

Alexa Skills Kit Command Line Interface (ASK CLI)

The ASK CLI is a tool for managing a skill and the related resources, such as AWS lambda functions, Programmatically from the command line. It can be configured with AWS credentials if AWS services, such as AWS Lambda, are used to host the skill backend logic.

The ASK CLI allows to create new skills and deploy them to the developer console; if the skill is configured to use AWS Lambda, it can deploy and upload the skill code directly. It allows also to test skills and submit for certification and publishing by command line. Through the ASK CLI, developers can create a skill starting from scratch or from a model and can manually set every detail of the skill by changing the correct files. The developer has complete freedom over the skill, but needs to know very well how a skill is composed, all its parameters and how it works: every permission, configuration, must be correctly formatted in the right files. The drawback is that it is not user-friendly at first and developer has to have a little experience with the skill management in the console.

A combination of the developer console and ASK CLI can be one of the best options, to exploit the first one to configure easily a skill and create the interaction model, while using the latter one to manage the code in any way wanted.

Skill Management API (SMAPI)

The Skill Management API (SMAPI) provides RESTful HTTP interfaces for creating new skill or updating an interaction model. It allows any developer to build tools or services that can create and update Alexa skills.

The ASK CLI is one of such tools.

Others

Amazon is continuously expanding the ways in which is possible to create a skill, releasing and integrating new services to make them: recently an extension for Visual Studio Code, one of the most used code editor was released, the Alexa Skills Toolkit, that provides a dedicated workspace for Alexa skills; but it is also possible to use many AWS clouded-based tools, such as AWS CodeStar, a cloud-based development service to develop and deploy applications on AWS.

2.2.3 How to host the service

The logic handler consists of a web service, capable of process incoming *requests* to provide an appropriate *response*. The cloud-based service of the skill can be host in three main ways:

- The skill can be a web service, as a server running on a computer, listening for requests over the network. It can be host with any hosting provider. The web service must to accept request over HTTPS and can be written in any language. Alexa sends the request to the web service and it sends back a response.
- AWS Lambda, an Amazon Web Service offering, is a service that runs the code in the cloud only when it's needed and there is no need to manage servers. The Lambda function, the code that is run on AWS Lambda, is executed only in response to Alexa voice interactions and automatically managing the compute resources. It can be written in Node.js, Java, Python, C# or Go. It is the easiest way to host the service for a skill. The developer needs only to upload the code to a Lambda function and Lambda does the rest. It eliminates the complexity of setting up and managing an own endpoint. In this way, the developer does not have to take care of:
 - Managing the compute resources of the service;

- An SSL certificate: SSL is a protocol for encrypting Internet traffic and identify server identity that make an HTTP address an HTTPS, which is more secure. [39]
- Verifying the requests coming from the Alexa service, because accesses are controlled automatically.
- It is up to Alexa to encrypt the communication with lambda (Using TLS)

The Lambda function must be able to handle the request sent to the skill by Alexa. When the Lambda function is created, the developer must connect the function to the skill, to let it know where it has to send the requests, by updating the endpoint field in the skill configuration. Since the Lambda function is invoked through a trigger generated by events and requests, the invocation permissions to trigger the function must be granted to Alexa, usually by the skill ID verification: in this way, the function can be invoked only if the skill ID in the Alexa Skills Kit request matches the skill ID configured in the trigger. Most of these steps are guided and automatized using the ASK Command Line Interface (ASK CLI)

- Alexa-hosted skill, through the Alexa developer console: it is up to Alexa to store the code and all the resources used; it is the fastest way to create an Alexa skill and it is possible to access the code through the Alexa developer console in an online code editor. Alexa provides all the AWS resources needed, including a Lambda function. But, in this case, the developer does not have to take care of connecting the Alexa skill to the endpoint. This service is provided as free, but there are some usage limits, as the number of free requests, 1 million, or the amount of free compute time per month, 3.2 million seconds. To reduce perceived latency when accessing a skill, Alexa provisions 3 AWS Lambda endpoints, in 3 different regions. This makes the creation of an Alexa skill very easy, because the developer does not have to take care of everything that is behind the VUI and the logic handler code.

Regarding the development of the skill, the Lambda function and Alexa-hosted options are the easiest, especially the latter one, they are essentially the same thing. The creation of a Lambda function is very versatile and allows the developer to manage the resources used by the function, such as the quantity of memory allocated for compilation and number of acceptable requests, without the complexity of the management of the web service.

An Alexa-hosted skill has less flexibility, and the lambda function code is strictly linked to the skill and the Alexa developer console; the AWS resources are stored in an individual account, separate from the other users. But one of the biggest advantages, discovered in the development of this thesis work, is the management of the dependencies and the time used to upload the code to the Lambda function.

Basically, when updating the code of a lambda function, all the files have to be compressed in an archive before the upload. Before the upload, all the required modules and dependencies have to be downloaded and zipped. The problem is that there can be modules very heavy, from 40MB to 100MB. This makes the zipping and the upload process really slow: it arrived up to 8 minutes. This means that every time the code had to be changed, to build and simulate to see if everything is working fine, the process has to be repeated. Every single change or error correction is really slow, making the development and debug process really slow.

An Alexa hosted skill completely avoids this problem. Alexa provides AWS CodeBuild, a fully managed build service in the cloud that compiles, runs unit tests and produces artifacts that are ready to deploy. It automatically runs NPM as part of the build script, adding the libraries and dependencies only when the skill has to be deployed, directly in the cloud. Thus, the skill code can be directly written in the online code editor and all the dependencies are managed by the AWS CodeBuild. This allows to dramatically reduce the waste of time due to the download of the missing libraries, that could be very big, and the zipping process. Through the Alexa console and an Alexa-hosted skill, the same code with the same dependencies takes up to less than a minute to upload and deploy. This obviously speeds up the development process.

2.2.4 Alexa Skills Kit Software Developer Kit

Regarding the code, Amazon provides the Alexa Skills Kit (ASK) SDKs, a set of software development tools and libraries that gives programmatic access to Alexa features, making easier the process of building an Alexa skill. It is available for Node.js, Java and Python.

As showed before, Alexa builds a request containing the information about what intent the user wants to trigger and the optional slots required. Every interaction between Alexa and the skill code essentially is an exchange of a JSON object. So the skill code must be able to handle the request in the correct way, capturing the data and processing them: it must recognize each incoming request and return an appropriate response. The SDK makes this process easier and more intuitive. By including the ASK SDK in the Node.js project as an NPM module, it is possible to access all the functionalities provided.[40]

2.2.5 Other settings

The skill creation process consists also of additional settings that must be configured, all shown in the main page of the skill in the developer console. As well as all the model components, the most important settings to be configured are - the Endpoint: specify the endpoint of the skill, “where” Alexa sends the requests when the user invokes the skill. - Permissions: If a permission is enabled, the skill can

ask the user for permission to access specific information or perform certain tasks; the skill does not actually obtain this permission until the skill user consents to it. For example, the skill must be enabled to ask to access the name of the user, the device address, or to send reminders.

2.3 The *interaction model*

The *interaction model* consists of a file in JSON, JavaScript Object Notation, a lightweight format for storing and transporting data; the JSON format is syntactically identical to the code for creating JavaScript objects. Each data consists of a pair key/value, like JavaScript properties. The interaction model will have this format:

```
{
  "interactionModel": {
    "languageModel": {
      "invocationName": "marelli car",
      "intents": [{}],
      "types": [{}],
    },
    "dialog": {
      "intents": [{}],
      "delegationStrategy": "ALWAYS"
    },
    "prompts": [{}],
  }
}
```

Figure 2.6: Interaction model file

2.3.1 Invocation name

The first and one of the most important design choices is the *invocation name*: the user will say the it to begin an interaction with the custom skill. It is the actual identifier of the skill. The user can use a phrase including the invocation name to fulfill a request: the user can combine the invocation name with an action or a command, to invoke a specific command, an intent, of a specific skill.

There are many requirements for the invocation name: for example, it “ must not infringe upon the intellectual property rights of an entity or person”, “One-word invocation names are not allowed, unless” “The invocation name is unique to your brand/intellectual property with proof of ownership established through legitimate documentation”, “Invocation names that include names of people or places (for example, "Molly", "Seattle") are not allowed”, “The invocation name must not contain the wake words "Alexa," "Amazon," "Echo," or the words "skill" or "app"” and so on.[41]

In this case “*Marelli car*” has been chosen and the user can start interactions with the skill in 3 ways:

- By invoking the skill with a request:
"Alexa, ask *Marelli car* to check my car"
- Invoking the skill without any particular request:
"Open *Marelli car*"
- Just using the invocation name:
"Alexa, *Marelli car*"

Only the first way to invoke the skill will accomplish a specific task that the user wants. The other two do not trigger any specific intent. It is a special request that the Alexa skill can receive, a *LaunchRequest*, that inform the code that a user made a request without an intent, so it must be managed accordingly. Usually, it is used as “welcome” intent. It allows the skill to be invoked and, in the next requests, it is possible to call the skills intents without the need of specifying the invocation name.

So, there are two ways of asking Alexa to do something:

1. By doing a specific request to a specific skill:
User: “Alexa, ask Marelli car to check my car”
2. Launching first the skill, then making the request:
User: “Alexa, open Marelli car”
Alexa: “Welcome, how can I help you with your car?”
User: “Is my car ok?”

Note: Amazon is working to make the invocation of a skill more natural: an additional feature, released as a public preview and beta lately, only for English (U.S) skills, allows users the Name-free Interactions: this enables customers to interact with Alexa without invoking a specific skill by name, helping the users because they do not always know which skill is appropriate to accomplish a task. When Alexa receives a request without an invocation name specified, Alexa looks for skills that might fulfil the request, handing the request to the selected skill.

2.3.2 Intents

After the selection of the *invocation name*, it is time to convert the features outlined in the overview in *intents* to build the *interaction model*.

Summing up the functionalities to be implemented, this is the list with a generic phrase that the user could say to trigger them [42]:

- Ask Alexa to do the diagnostic of the vehicle :
“Is my car ok?”
- Ask to check the status of a specific component:
“Is there any problem with the battery?”
- Setting a driver profile in the vehicle with personalized settings:
“set my profile”
- Control the vehicle, from inside and outside the vehicle:
“Lock my car”
- Retrieve some information:
“When is the next revision?”
- General purpose features: to showcase how Alexa can interact with external APIs to offer even more complex features
“Save my location”
“Where’s my car?”
“Are there any parking lots nearby?”

Each of them is represented as an *intent* in the interaction model. For example, the will of the user to do a complete diagnosis of the vehicle is mapped to the *CheckUpIntent*: when the user says “Is my car ok?”, Alexa composes a request containing as intent value *CheckUpIntent*. Or, if the user wants to retrieve the location of his car by saying “where is my car”, the request contains the *GetParkingLocationIntent*.

In the creation of a new custom intent, the developer provides a name and a list of *utterances*, phrases, that the user would say to trigger it. To allow Alexa to understand what the user wants, the developer has to insert as many variations as possible of what the user can say to Alexa and mapped them to the

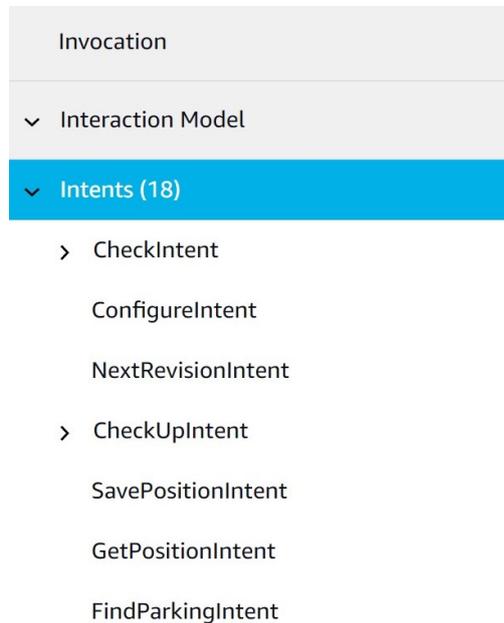


Figure 2.7: Some of the intents mapped

intents. There should not be any difference if the user says “where is my car?” or “where did I left my car?”: Alexa must be able to detect the *GetParkingLocationIntent* in both cases. The sample utterances must be unique and there can not be duplicate sample utterances mapped to different intents.

“Is there any problem with the battery?” => *CheckIntent*

The Alexa Skills Kit also includes a library of built-in intents that can be used and do not need sample utterances. They are classified in many categories, depending on what is their goal: there are Calendar intents, for asking about calendars and schedules, Weather intents, for requesting weather reports and forecasts and Standard Built-in intents, for general actions such as cancelling, stopping and asking for help; the latter one are required in every skills, because the user must always be able to interrupt or exit every skills. There are also Yes and No intents, already defined: in this way, the skill can be able to manage assertive or negative responses to Alexa’s answers.

Note: the built-in library of intent does not actually add functionalities to the skill: they are simply pre-built voice models, that allows to avoid the design of the voice user interface for those specific intents, since they are already defined. For example, if Yes intent is included, the skill will recognizes when the user says “yeah”, “yes”, “sure” without the need to manually provide them as sample utterances. More important, if a developer includes them in the skill, he has still to specify in the back-end code what to when the intent request arrives.

2.3.3 Slots

Intents can optionally have arguments, called *slots*: these are words or phrases that represent variable information. For example, the *CheckIntent* is used to check the status of a component of the vehicle or a warning. So, Alexa not only needs to know that the user wants to that intent, but also the specific component. The set of utterances that are mapped into the intents needs to contain the *slot* as a variable, inserting in the phrase the slot name in curly brackets `{ }`:

Intents / CheckIntent

Sample Utterances (10)  Bulk Edit  Export

What might a user say to invoke this intent?	
is {car_part} ok	
is {car_part} working	
is there any problem with {car_part}	
how is {car_part}	
check {car_part}	

[◀ 6 - 10 of 10 ▶](#) [Show All](#)

Figure 2.8: Intent definition with sample utterances and slot

```
{
  "interactionModel": {
    "languageModel": {
      "invocationName": "marelli car",
      "intents": [
        { },
        { },
        { },
        { },
        {
          "name": "CheckIntent",
          "slots": [
            {
              "name": "car_part",
              "type": "car_part",
              "samples": [
                "{car_part}"
              ]
            }
          ],
          "samples": [
            "is the {car_part} working",
            "is the {car_part} ok",
            "check the {car_part}",
            "{car_part}",
            "is {car_part} ok",
            "is {car_part} working",
            "is there any problem with {car_part}",
            "how is {car_part}",
            "check {car_part}"
          ]
        }
      ],
      { }
    }
  }
}
```

Figure 2.9: Intent definition in JSON format

In this way the user fills the gap in the sample utterance and Alexa understand what the user wants as the `{car_part}` value. This value is going to be part of the request. In this way, the back-end code can execute different tasks depending

on the slot value.

Each intent can have more than one slot and not all the sample utterances have to contain all the required slots: it is possible to delegate Alexa for filling all the slot values that the user may not provide. Indeed, for each slot, four configurations must be set:

- Slot type: it determines how the user input is handled and passed on to the skill. Every slot in each intent is associated to a *slot type*, set of values that the customer can say. It is possible to create *custom slot types* that defines what the skill expects from the speaker. For example, *{car_part}* possible values are “battery”, “engine”, “fuel”. Each of them can have an ID and synonyms to add variety and flexibility to the user.

Amazon also provides built-in slot type for dates, number, cities.

- Slot filling: this setting makes the slot required to fulfil the intent; if yes, the developer can specify what Alexa may say to prompt the user for the missing slot value.
- Slot confirmation: Alexa may ask back the user to confirm if the slot value is correct before continuing the dialog.
- Validation rule: to validate what the user may say. This avoids that Alexa misinterprets what the user wants or sets as slot values invalid words, that the skill could not be able to handle. Alexa could reject values that are not part of the slot type definition, or rejects values belonging to a set of words; in this way, Alexa could prompt back asking to make the request again, providing some examples to clarify to the user what are valid values.

A voice interface cannot prevent users from entering an invalid data and misunderstandings in spoken language may introduce errors in slot values. In this way, Alexa can validate slot values, reprompting and asking a new value.

Through the settings for slots, it is possible to see how Alexa can engage a real conversation, requesting and validating slot values. It is even possible to ask the user for confirmation of the whole intent. In this way a *dialog* between Alexa and the speaker can take place.

2.3.4 Dialogs

In a skill, a *dialog* with the user is a conversation with multiple turns in which Alexa asks questions to the user to collect data. But this conversation is tied to a specific intent, that represents the overall request of the user. It is intended only to gather and validate slot values and the whole intent: it continues only until all slots needed for the intent are filled and, in some cases, the intent is confirmed.

The dialog model is the part of the interaction model that manages the turns of the conversation and is represented in the JSON file; it identifies which slots are required for which intent, what Alexa says to ask for required slots, and all the settings for the slots mentioned before, such as validation rules. In short, it contains the structure of the conversation that Alexa carries on if the user does not make a complete intent request and if something has to be confirmed before sending it. In this way, it is Alexa that determines the next step of the conversation depending on what the user says and uses the prompts defined here to ask for the missing information. This delegation strategy relies on Alexa to manage simple dialogs. In this way, it is not up to the back-end code to check if the request is complete, without writing code to ask the user for slot values and confirmations. [43]

The conversation is left to Alexa and the request is sent only when the dialog and all the information are complete. Indeed, dialogs have a state property, *dialogState*, that Alexa uses to keep track of the conversation: *STARTED*, when the intent is invoked; *IN_PROGRESS* when Alexa doesn't have all the necessary information; *COMPLETE* when Alexa has everything to compose a request. Only in the latter state, the request is actually sent to the skill service.

The dialog can also be manually managed through code, to have more flexibility and make run-time decision, to take control of the dialogs and build more complex conversational flow. This will be discussed later, in the back-end code section with the ***Dialog.Delegate*** directive.

The following image shows how the slot *{car_part}* is set as required for the intent *CheckIntent* and how Alexa can asks some question to fill the incomplete request, accepting only the values belonging to the *car_part* type.

```

{
  "interactionModel": {
    "languageModel": {
      "invocationName": "marelli car",
      "intents": [{}],
      "types": [{}]
    },
    "dialog": {
      "intents": [
        {
          "name": "CheckIntent",
          "confirmationRequired": false,
          "prompts": {},
          "slots": [
            {
              "name": "car_part",
              "type": "car_part",
              "confirmationRequired": false,
              "elicitationRequired": true,
              "prompts": {
                "elicitation": "Elicit.Slot.1561425158292.469134112255"
              },
              "validations": [
                {
                  "type": "hasEntityResolutionMatch",
                  "prompt": "Slot.Validation.1091284506138.1460460928751.1288357834487"
                }
              ]
            }
          ]
        },
        {},
        {},
        {},
        {}
      ],
      "delegationStrategy": "ALWAYS"
    },
    "prompts": [
      {
        "id": "Elicit.Slot.1561425158292.469134112255",
        "variations": [
          {
            "type": "PlainText",
            "value": "Which component?"
          },
          {
            "type": "PlainText",
            "value": "What do you want me to check?"
          }
        ]
      },
      {},
      {}
    ]
  }
}

```

Figure 2.10: Dialogs settings for CheckIntent

While all these settings can be easily configured through the developer console, the resulting JSON file is much more complex: if done manually, it could be really time consuming and could lead to errors. For this reason, the console is the best tool for building complex voice experiences and interaction models.

2.3.5 The request

All of these settings, sample utterances, intents, slots, slot values, dialog model build the interaction model, the actual voice-user interface. The user will interact with the device, and in this case with the vehicle, through Alexa on the basis of this model, invoking the skill. Then Alexa builds a JSON request, containing what intent the user has requested, the slot values and all the information required for a correct request, and sends it to the skill, initiating a new skill *session*.

```

{
  "version": "1.0",
  "session": {
    "new": false,
    "sessionId": "amzn1.echo-api.session.0e285a9c-bf9b-41b8-a737-c8d48017612a",
    "application": {
      "applicationId": "amzn1.ask.skill.33f5d864-a0e9-45da-8cae-7d30b36ef671"
    },
    "attributes": {},
    "user": {}
  },
  "context": {},
  "request": {
    "type": "IntentRequest",
    "requestId": "amzn1.echo-api.request.f5f7017d-8097-49dd-ab29-084506333093",
    "locale": "en-US",
    "timestamp": "2020-10-09T14:48:07Z",
    "intent": {
      "name": "CheckIntent",
      "confirmationStatus": "NONE",
      "slots": {
        "car_part": {
          "name": "car_part",
          "value": "battery",
          "resolutions": {},
          "confirmationStatus": "NONE",
          "source": "USER"
        }
      }
    }
  },
  "dialogState": "COMPLETED"
}

```

Figure 2.11: A skill request sent by Alexa

Alexa can build different types of request. The user may want to invoke some skill functionalities, such as controlling a device or look for a recipe, or may be Alexa to send an event to the user [44]. Each type will send different information and the code should be able to manage all of them.

The possible types of requests are:

- ***LaunchRequest***: sent when a user invokes the skill without providing a specific command. Usually it is used for a welcome message.
- ***SessionEndedRequest***: sent when the skill currently open sessions is closed. This happens when the user exits the skill or when an error occurs.
- ***IntentRequest***: sent when a user sends a request that maps to one specific intent. The request includes also the name of the intent.
- ***Others***: there are other types of request, sent when special Alexa interfaces or functionalities are invoked, such as *AudioPlayer* directives. But these are not used for this project.

Other than that, the request contains many other information that the skill service can access. The main parameters of the request body are:

- ***"session"*** object, providing additional context about the request, such as if the session is new, *attributes*, a map of key-values, used to persist data during a session and are passed from responses to requests, or the application ID, unique for each skill, to verify that the request is meant for that service. It contains also some information about the Amazon account user.
- ***"context"***, that contains information about the state of the Alexa device at the time the request is sent, such as a device ID, the supported interface, like geolocation properties, and the allowed permissions.
- ***"request"***, with every detail of the user's request, such as the *request type* and the intent name. It contains also slots and slot values *resolutions*, the results of the slot entity resolution, a process in which Alexa attempts to resolve what the user said with possible slot values, synonyms and ID. In this way, when the user's speech is resolved, the skill receives the "standard" value defined for that slot type value, the actual value that the user spoke and the unique ID. This can be very useful, since it makes it possible to map multiple synonyms to a single ID and use that in the code, allowing more flexibility to the user speech.

```
"slots": {
  "car_part": {
    "name": "car_part",
    "value": "battery",
    "resolutions": {
      "resolutionsPerAuthority": [
        {
          "authority": "amzn1.er-authority.echo-sdk.amzn1.ask.ski",
          "status": {
            "code": "ER_SUCCESS_MATCH"
          },
          "values": [
            {
              "value": {
                "name": "battery alert",
                "id": "BATTERY_ALERT"
              }
            }
          ]
        }
      ]
    }
  },
  "confirmationStatus": "NONE",
  "source": "USER"
}
```

Figure 2.12: Slot and slot resolution

Now is up to the back-end code to accept the request and process it, providing a response

2.4 The logic

The back-end code is the logic handler behind the scene. Alexa works simply as an interface between the user and the skill code, to which the skill is connected by selecting the endpoint in the skill configuration in the Alexa developer console or manually with the ASK CLI.

The overall code, built with the ASK SDKs, can be summed up as a list of *intent handlers*, one for each intent of the interaction model, that accepts requests satisfying some requirements, processes them and returns a response. But first, each request must be identified.

The code needs first to recognize each incoming request to return an appropriate response. Before accepting a request, the lambda function verify that the request came from the correct skill. This prevents from configuring a skill with another endpoint, allowing others to exploit the code by sending own requests to someone other's code. Every request includes a unique skill ID, that can be used to ensure that the request was intended for the service. The ASK SDK can automatically handle the skill ID verification with a `.withSkillID()` directive.

2.4.1 Intent handlers

Each request handler needs first to identify the requests it can handle and then actually doing it. Usually, handlers accept requests based on the type and/or the intent requested, specific slot values or any other criteria derived from the data in the request, but there are also error handlers, responsible for error management when an unhandled error is thrown during the request processing. A skill must handle all of the intents defined in the *interaction model*.

The skill routes an incoming request to a particular request handler, processing it and returning a response. The code can do whatever the skill needs, such as analyzing data provided in the user request, calling Alexa APIs to get information about the user or to perform actions or even calling external APIs.

With the ASK SDK, a request handler can be implemented by using these two methods:

1. ***canHandle()*** method to identify if the incoming request can be processed, by imposing some conditions.
2. ***handle()*** is called by the SDK to take care of processing the data generating and returning a response. It can also provides the text that Alexa speaks to the user.

Both take a ***handlerInput*** object as input, which contains the complete JSON request sent to the skill, through which it possible to retrieve data in the request. This object provides also various entities to process the request. It contains the ***requestEnvelope***, the whole request body, so that the handler can access every detail of the request, as well as the ***responseBuilder***, that gives access to helper method to construct responses.

Once the handler completes its job, it sends a response back, a JSON structure that tells Alexa what to do next.

```
// /**
// * Intent Handler to check the status of a component
// */
const CheckIntentHandler = {
  canHandle(handlerInput) {
    return (
      Alexa.getRequestType(handlerInput.requestEnvelope) === "IntentRequest" &&
      Alexa.getIntentName(handlerInput.requestEnvelope) === "CheckIntent"
    );
  },
  handle(handlerInput) {
    //Process the request and formulate a response

    return handlerInput
      .responseBuilder
      .speak(speechText)
      .reprompt(repromptText)
      .getResponse();
  },
};
```

Figure 2.13: Handler implementation for handling CheckIntent requests

Session Attributes

The *handlerInput* gives also access to the *attributeManager*, through which is possible to manage *attributes*. They consists of a map structure of key and value pairs that the SDK allows to store and retrieve and that can be included as a property of a response. When Alexa sends the next request as part of the same session, the map is included in the request. Intent handlers can retrieve and elaborate them. For this thesis project, session attributes are essential to retrieve data of the vehicle, avoiding the need of continuously read values directly from the database connected to the car. This could lead to some efficiency problem: in case of connection problems, the skill may suffer from latency caused by the incapability of retrieving data. Instead, the car information can be loaded just once at the beginning of the session or just when it is needed, so that Alexa can elaborate them and answer to the driver requests faster, because all the information needed are already part of the requests and responses. Thus, once the attributes are loaded, handlers can execute any logic: it is possible to check the status of a single car component, execute a diagnosis of the whole vehicle, check how when is the next car revision and so on and then return an appropriate message to the driver, implementing most of the wanted features.

```

"session": {
  "new": false,
  "sessionId": "amzn1.echo-api.session.e6e28585-9408-4d9b-b28d-8fed2e0ad29a",
  "application": {},
  "attributes": {
    "loaded": true,
    "PROFILES": [],
    "STATUS": {},
    "CTRL": {},
    "PROBLEMS": "BATTERY_ALERT",
    "STATE": "CheckState",
    "VIN": {},
    "INFO": {},
    "WARN": {
      "LAMP_OUT": false,
      "COOLANT_TEMP_WARN": false,
      "TIRE_PRESSURE_WARN": false,
      "FUEL_WARN": false,
      "OIL_WARN": true,
      "DOOR_AJAR": false,
      "BATTERY_ALERT": true,
      "ENGINE_WARN": false
    },
    "NEXT_STATE": "HelpUserState"
  },
  "user": {}
},

```

Figure 2.14: Attributes included in a request

Once the session ends, all the attributes are lost: they are a session property and persist only throughout the lifespan of the skill session.

Persistent attributes

If the skill needs to remember data across multiple sessions, a persistent storage such as DynamoDB or S3, two AWS services, is needed. The Alexa skill can load *persistent attributes* from the persistent storage as session attribute, manipulate them and then save the new session one as persistent attributes.

So, to prevent the loss of session attributes when the session is closed, in this case, attributes are loaded to DynamoDB, that will be examined in detail in the dedicated chapter. But it is worth introducing now how the skill interacts with a database.

Alexa SKD provides the *PersistenceAdapter*, used by the *AttributesManager* to create tables, retrieving and saving attributes. Specifically, it provides the *DynamoDbPersistenceAdapter*, the implementation of *PersistenceAdapter* using AWS DynamoDB. This can be used in interceptors to automatically load and save data to a DynamoDB table.

The DynamoDB table can be automatically created when using an Alexa-hosted skill, but it is also possible to manually link a lambda function to a personal AWS resource, deciding actually where it can retrieve the data. In order to do this, the lambda function must be enabled to access resources modifying the *AWS IAM role*, that is a sort of AWS identity that has specific permission policies to determine what the identity can do in AWS. Then, in the code, it is possible to manually create a DynamoDB instance with the correct AWS resource role and scan/select the desired DynamoDB table. In the next image, the manual configuration of the adapter is presented, forcing also the name of the table and the id.

```

var persistenceAdapter;
const {
  DynamoDbPersistenceAdapter,
  PartitionKeyGenerators,
} = require("ask-sdk-dynamodb-persistence-adapter");
AWS.config.loadFromPath("./aws_config.json");
const DBClient = new AWS.DynamoDB({ region: "eu-west-2" });

persistenceAdapter = new DynamoDbPersistenceAdapter({
  tableName: "VehicleData-yyvj3urgxfg6zbd6ffefsvcouu-dev",
  createTable: false,
  partitionKeyGenerator: keyGenerator,
  dynamoDBClient: DBClient,
});

function keyGenerator(requestEnvelope) {
  if (
    requestEnvelope &&
    requestEnvelope.context &&
    requestEnvelope.context.System &&
    requestEnvelope.context.System.application &&
    requestEnvelope.context.System.application.applicationId
  ) {
    //return requestEnvelope.context.System.user.userId
    return "02d8f1c1-cc96-4648-8c4e-fdc5aabc814b";
  }
  throw "Cannot retrieve app id from request envelope!";
}

```

Figure 2.15: The manual configuration of the `DynamoDbPersistenceAdapter`

Then the `persistenceAdapter` is registered in the *SkillBuilder*, as in section 2.4.3. Now it is possible to access the `persistenceAdapter` functionalities to load and save data in the table with the *AttributesManager*. One common practice is to use request and response interceptors to do these operations automatically.

```

const LoadAttributesRequestInterceptor = {
  async process(handlerInput) {
    const { attributesManager, requestEnvelope } = handlerInput;
    const sessionAttributes = attributesManager.getSessionAttributes();
    if (Alexa.isNewSession(requestEnvelope) || !sessionAttributes["loaded"]) {
      const persistentAttributes =
        (await attributesManager.getPersistentAttributes()) || {};
      console.log(
        "Loading from persistent storage: " +
        JSON.stringify(persistentAttributes)
      );
      persistentAttributes["loaded"] = true;
      attributesManager.setSessionAttributes(persistentAttributes);
    }
  },
};

const SaveAttributesResponseInterceptor = {
  async process(handlerInput, response) {
    if (!response) return;
    const { attributesManager, requestEnvelope } = handlerInput;
    const sessionAttributes = attributesManager.getSessionAttributes();
    const shouldEndSession =
      typeof response.shouldEndSession === "undefined" ? true : response.shouldEndSession;
    const loadedThisSession = sessionAttributes["loaded"];
    if ((shouldEndSession || Alexa.getRequestType(requestEnvelope) === "SessionEndedRequest") && loadedThisSession) {
      sessionAttributes["sessionCounter"] = sessionAttributes["sessionCounter"] ? sessionAttributes["sessionCounter"] + 1 : 1;
      for (var key in sessionAttributes) {
        if (!PERSISTENT_ATTRIBUTES_NAMES.includes(key))
          delete sessionAttributes[key];
      }
      console.log(
        "Saving to persistent storage:" + JSON.stringify(sessionAttributes)
      );
      attributesManager.setPersistentAttributes(sessionAttributes);
      await attributesManager.savePersistentAttributes();
    }
  },
};

```

Figure 2.16: Definition of the interceptors to load and save attributes

These interceptors also check if the session is new, if the skill was stopped, or even exclude some attributes useful only for the session that do not need to be persisted. In this way, the Alexa skill is capable of read the vehicle data from the database. This also allows the skill to send some information to the vehicle. Indeed, the database is also use to send commands from the skill, just by changing some attributes: by detecting these changes , the vehicle could react. But the persistenceAdapter is not sufficient to build a real-time application, to inform the system in some ways that changes are done. But a solution will be found in chapter 4.

Response building

A lambda function, when used as endpoint, is only responsible to build the response in order for Alexa to answer to a customer. With the methods provided by ***responseBuilder***, Alexa can be instructed to speak, to show *cards* on a display or call Alexa APIs.

The output speech, set with ***responseBuilder.speak()*** command, can be a plain text or a *Speech Synthesis Markup Language* (SSML), a markup language that provides a standard way to mark up text for generation of synthetic speech. It is used to make Alexa speak with emphasis, change pronunciation and add other effects.

The response, in JSON format, consists of two main objects: *sessionAttributes*, already mentioned in the *request* definition, will be discussed later, since their management is very important; the other one is a *response* object, that actually define what Alexa has to do next. The latter object contains:

- an *outputSpeech* object, to set what Alexa must say next.
- a *card* object. These are graphical cards that describe or enhance the voice interaction; it contains a text and an image to be rendered on the screen. The image must be accessible through an URL. Voice responses need to be concise and look like a conversation. A card can provide additional and useful detail that would make a voice response too long or verbose.
- a *reprompt*, containing an *outputSpeech* to set reprompt message and instruct Alexa to listen for a response from the speaker. The user has few seconds, otherwise Alexa closes the session.
- *directives*, an array of *directives* specifying action that the device has to take, such as triggering the *AudioPlayer*. It can also instruct Alexa about what to say next with the *Dialog interface*.

```

"body": {
  "version": "1.0",
  "response": {
    "outputSpeech": {
      "type": "SSML",
      "ssml": "<speack>The voltage level of your battery is below the normal level.</speack>"
    },
    "card": {},
    "directives": [],
    "type": "_DEFAULT_RESPONSE"
  },
  "sessionAttributes": {},
  "userAgent": "ask-node/2.10.1 Node/v10.22.1 vehicle-skill"
}

```

Figure 2.17: JSON response body

The response sent is not always the end of the interaction: sometimes the skill needs to let the user say something in response, which becomes a new request for the skill. But if the user doesn't provide a response in 8 seconds, Alexa stop listening closing the microphone and, if the a reprompt is specified, Alexa reprompts the user to speak. In case there is still no answer, the session is closed. Otherwise, if the user says something that match the interaction model, the conversaion goes on and the session remains open.

The response can include *directives* to instruct Alexa to take other actions. Directives are organized in interfaces and the type determines the properties that

are needed to be provided. To include directives, the ASK SKD provides helper methods.

For example, there is the *AudioPlayer Interface* for playing audio and the *Dialog Interface*

Dialog Interface

This interface provides directives for controlling multi-turn conversation [45]. In the *dialog* section of the interaction model, it was shown how is possible to let Alexa complete intent requests when the user does not provide required data and sending a single request at the end. That was one of the possible strategy, the *auto delegation* of the dialogs. This interface allows to override it with a manual one, defined by code.

With manual delegation, instead of sending an *IntentRequest* only when the request is complete and the *dialogState* is **COMPLETE**, Alexa sends a request for each turn of the conversation, when the state is either in **STARTED** and **IN_PROGRESS**. It is possible to manually ask for a slot value or slot confirmation, with *Dialog.ElicitSlot* and *Dialog.ConfirmSlot* directives, providing also the prompt messages that Alexa speaks. This can lead to the exact same behavior and conversation that can be defined dialog model. Instead, with a *Dialog.Delegate* it is possible to leave the dialog management back to Alexa and the dialog model. The right delegation strategy depends on the complexity of the dialog. Auto delegation is simpler because the conversation does not need to be handled in the code. Manual delegation is more flexible, allowing the skill to make run-time decisions and directing the flow of the conversation.

The most important features of these commands is the possibility to manually select an intent and slot values, giving the code the capability to change the direction of the conversation. The speaker is not the only one that can request an intent, by directly invoking it, but Alexa can change the intent and have an active role, proposing or imposing an intent, as in natural and real conversation both speakers take decisions.

For example, in the next image, it is shown how the request consisted in an *CheckIntent* and then the response shifted the dialog to an *HelpUserIntent*.

JSON Input

```

1  {
2  "version": "1.0",
3  "session": {},
21 "context": {},
100 "request": {
101   "type": "IntentRequest",
102   "requestId": "amzn1.echo-api.request.c4aae0d1-255b-459f-a325-045754a189ef",
103   "locale": "en-US",
104   "timestamp": "2020-11-06T08:26:49Z",
105   "intent": {
106     "name": "CheckIntent",
107     "confirmationStatus": "NONE",
108     "slots": {
109       "car_part": {
110         "name": "car_part",
111         "value": "battery",
112         "resolutions": {},
113         "confirmationStatus": "NONE",
114         "source": "USER"
115       }
116     }
117   },
118   "dialogState": "COMPLETED"
119 }
120 }

```

Figure 2.18: JSON request with a *CheckIntent*

JSON Output

```

1  {
2  "body": {
3   "version": "1.0",
4   "response": {
5     "outputSpeech": {},
9     "card": {},
18    "directives": [
19     {
20       "type": "Dialog.Delegate",
21       "updatedIntent": {
22         "name": "HelpUserIntent",
23         "confirmationStatus": "NONE",
24         "slots": {
25           "car_part": {
26             "name": "car_part",
27             "value": "BATTERY_ALERT",
28             "confirmationStatus": "NONE"
29           }
30         }
31       }
32     }
33   ],
34   "type": "_DEFAULT_RESPONSE"
35 },
36 "sessionAttributes": {},
143 "userAgent": "ask-node/2.10.1 Node/v10.22.1 vehicle-skill"
144 }
145 }

```

Figure 2.19: JSON response with a *HelpUserIntent*

2.4.2 Interceptors

The last component of the skill code that are worth mentioning are *Interceptors*. These piece of code are special functions that are executed immediately before and after an intent handler. There are *request interceptors*, executed immediately before the handler for an incoming request, and *response interceptors*, invoked after. Just like handler, they accept *handlerInput* as input, so that they have complete access to the request and attributes and can alter them. They are very useful for debug e logging each interaction, sending all the requests and responses to Amazon CloudWatch, an AWS service for monitoring and observability service. But they can also be used to automatize any process, such as checking for permissions, saving attributes. All the interceptors must be registered in the *skillBuilder.addRequestInterceptors()* and *skillBuilder.addResponseInterceptors()*.

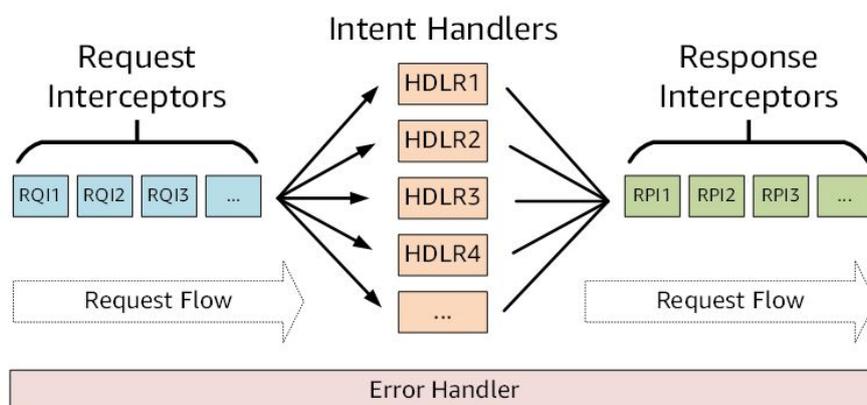


Figure 2.20: Request flow and interceptors

```
const LoggingRequestInterceptor = {
  process(handlerInput) {
    console.log(
      `Incoming request: ${JSON.stringify(handlerInput.requestEnvelope)}`
    );
  },
};
```

Figure 2.21: Interceptor to log requests.

For example the interceptor shown in figure 2.21 is used to log every incoming request to AWS CloudWatch, the only instrument that can be use for logging

purposes with a Lambda function. There is the equivalent one for incoming requests. Another important interceptor is the *LocalisationInterceptor*, that allows the skill to supports any language. When building a response, all the handlers use a string containing an identifier for the message instead of the text of the message. All the message texts are divided by language and stored in a separate file of the lambda function. and there is the translation of the text in each supported language. The interceptor, depending on the language, replace the message id providing the correct string in the correct language to the outgoing response.

2.4.3 Skill building

```
exports.handler = Alexa.SkillBuilders.custom()
    .addRequestHandlers( ...
    .addErrorHandlers(ErrorHandler)
    .addRequestInterceptors( ...
    .addResponseInterceptors( ...
    .withPersistenceAdapter(persistenceAdapter)
    .withCustomUserAgent("vehicle-skill")
    .withApiClient(new Alexa.DefaultApiClient())
    .lambda();
```

Figure 2.22: How a skill is built

This is how the skill is actually built, registering all components and integrating all the skill logic. With the *skillBuilder* it is possible to configure and construct a correct and complete instance of a skill. It also acts as the entry point for your skill, routing all request and response payloads to the handlers above. It must includes all handlers and interceptors defined. The order is important, since they're processed from top to bottom.

Once the code is uploaded to the lambda function, the skill is working and can be used and tested.

2.5 Location services and external APIs

Until now, it was shown how intent handlers can elaborate attributes to determine the status of the vehicle and accomplish the more basic features. But to provide a more complex and broad experience, the skill has to do more than that. In this section, two more complex intents are shown, demonstrating Alexa's capabilities. Considering the automotive environment of this skill, location services and location aware features are really important.

A skill can ask the user permission to obtain the real-time location of the Alexa-powered device. Once it is allowed, the geo-coordinates of the device are included and sent in the request, so that handlers can access them and provide new functionalities [46]. But the plain coordinates do not provide much information. Alexa skills can interact with remote APIs and web servers to get meaningful data by making HTTP requests.

In this project, HERE location services are used to process the coordinates: it provides developers many instruments to build location-aware apps and services. In particular, handlers access HERE REST APIs. REST is an acronym for Representational State Transfer and is an architectural style used for web services, providing interoperability between computer systems on the internet; one of their feature is the possibility to retrieve data, a *resource*, inside a *response* when doing a *request* to a specific URL. HERE provides many APIs and many different features, such as the **Geocoding API** and the **Off-street Parking API**.

The first one is used to retrieve the location of the car, previously saved on the database. When asking to Alexa "Where is my car?", the *GetPositionIntent* is triggered and it sends a request containing the coordinates to the **Geocoding API**. Then it receives in response a JSON with the address and builds the message that Alexa speaks. This can be particularly useful in those situations where the driver knows the place well enough to be able to orient himself and just needs some indications to remember where he parked. In this case, he just needs to ask to Alexa through the smartphone.

```

async function getAddress(position) {
  try {
    const response = await axios(
      `https://revgeocode.search.hereapi.com/v1/revgeocode?at=${position.latitude}%2C${position.longitude}&lang=en-US&apiKey=${API_KEY}`
    );
    console.log(response);
    const data = response.data;
    return data.items[0].title;
  } catch (error) {
    console.error(error);
  }
}

```

Figure 2.23: How the Lambda function performs the reverse geocoding given the coordinates

The latter API is used to find an available slot in the closest parking facility: it provides a list of facilities in a range around the received coordinates in order of distance. The handler check the availability of these facilities, finding the closest and available one and retrieving its address, so that Alexa can indicate it to the driver.

Note: these are demo features, to show the endless capabilities of an Alexa skill. Unfortunately, the location services are not available on all devices. But it is useful to show how it is possible to retrieve any information and builds a base for future

implementations.

2.6 Profile management: skill personalization

Alexa can provide personalized experiences [47] for recognized skill users and a virtual automotive assistant should be able to offer them.

An Alexa device, or in this case, a vehicle, might have two or more speakers who interact with it. If a skill supports personalization, it can differentiate an individual user by means of voice profiles. Alexa can save voice profiles by listening to the user's voice recordings and associating them with the Amazon account.

When a user speaks to Alexa, if the speaker is recognized and if the skill personalization settings are active, the skill can provide personalized content and actions. Each skill request from that speaker includes a *person* object and the skill can parse the *personID* from the request. Through that ID, the skill can use some of the person information, without actually passing it to the skill, protecting the privacy of the user. For example, the developer can include the use of the ID in the intent handlers to retrieve the person name, but he will never have real access to that information.

In this project, voice profiles are used to set some car configuration as soon as the driver speaks: it is common that the same vehicle is shared between family members and everyone may have different preferences regarding the seat position or the mirrors orientation. Thus, the skill can recognize speakers and setting the saved preferences before making other requests or propose to save and create a new driver profile, if it is not already defined before. In case the driver does not have a voice profile, the skill simply welcomes the driver.

Drivers can create personalized profiles, saving the current setting of the vehicle, that will be set every time they will launch the app. To do that, an attribute stores and adds all the profiles. In the following images, it is possible to see how, only when the *person* value in the input JSON is present in the *System* parameter, the speaker is actually recognized.

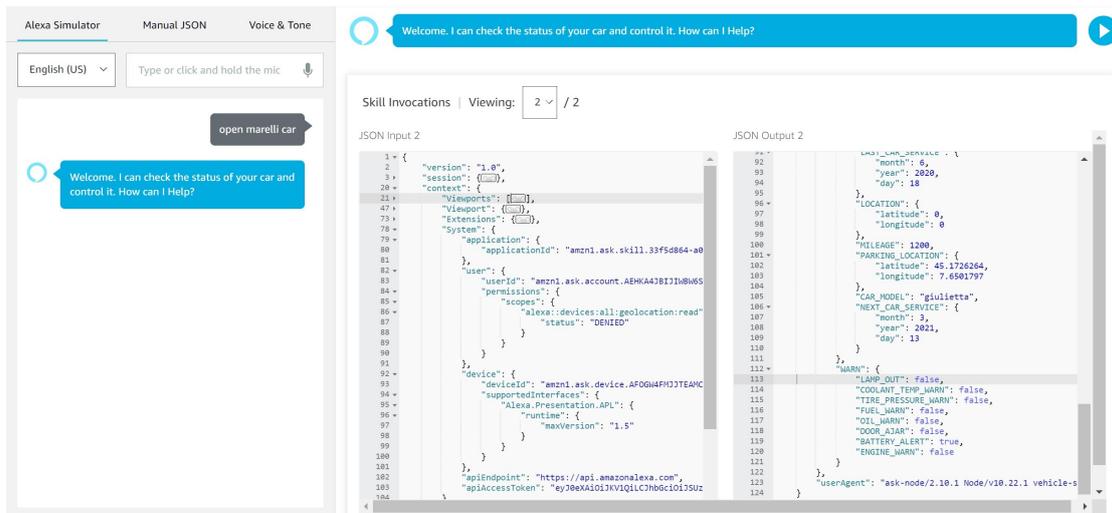


Figure 2.24: How the skill react when an unknown user speaks.

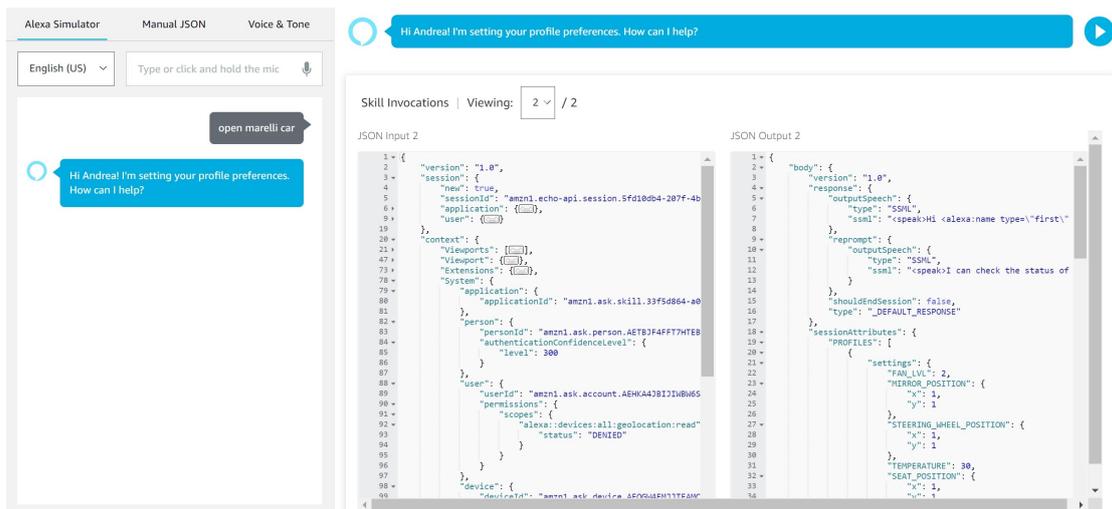


Figure 2.25: How the skill react when an known user, with a profile already saved, speaks.

2.7 Control the vehicle

To control the car with the Alexa skill, some intents detect what the speaker wants to control and how, then send these information to the database and the android app will be triggered. The details of how this happens will be explained later. Now, it is shown only how the lambda function set the commands, writing the

CTRL attribute values, *type* and *value*. The *type* contains the type of operation, such as **OPEN**, **CLOSE** or **SET_PROFILE**, while the *value* holds the ID of the component to be controlled or the personID of the profile to be set.

```
const ActivateIntentHandler = {
  canHandle(handlerInput) {
    return (
      Alexa.getRequestType(handlerInput.requestEnvelope) === "IntentRequest" &&
      Alexa.getIntentName(handlerInput.requestEnvelope) === "ActivateIntent"
    );
  },
  handle(handlerInput) {
    const { attributesManager, requestEnvelope } = handlerInput;
    const sessionAttributes = attributesManager.getSessionAttributes();
    const part = getSlotID(requestEnvelope, "controllable_part");

    sessionAttributes.CTRL.type = "OPEN";
    sessionAttributes.CTRL.value = part;
    attributesManager.setSessionAttributes(sessionAttributes);
    updateDB(sessionAttributes);

    return (
      handlerInput.responseBuilder
        .getResponse()
    );
  }
};
```

Figure 2.26: Intent for opening and turning on car component

2.8 Complex and natural conversational flow

Most of the intents and interaction shown before consists of simple dialogs: the user asks something, Alexa answer or complete the task. But is it possible to make more complex and natural conversation? Human speech can vary in forms and goals: a conversation could start from a point and end somewhere else.

This is not compatible with the “classical” Alexa approach, where all conversations are aimed to the fulfillment of a single intent requested from the user. Even the multi-turns dialogs, defined before with the *dialog model*, have as only goal, the gathering of data to do a single intent request. So, how is it possible to expand the conversation possibilities? In this thesis project, two approaches were used:

- Alexa is transformed in a state machine, to keep track of the current state. Alexa can take the lead of the conversation, switching intents based on the current and next possible state. Two new attributes, *STATE* and *NEXT_STATE* are created to keep track of the conversation. The *STATE* is usually linked to the current intent requested, while *NEXT_STATE* value can depends on the current state, the vehicle data and what the user may choose to do. This approach let Alexa propose new intents, exploiting Yes/No questions to advance in states. But since all the “yes” and “no” are mapped to the same *AMAZON.YesIntent* and *AMAZON.NoIntent*, so they are handled by the same two handlers, states are used to determine the actions that follow.
- Use of “complex” slot types: to avoid using only Yes/No questions to propose

new intents to the user, Alexa can collect whole utterances as slot value. This allows to exploit the slot elicitation rules, delegating most of the conversation to Alexa.

Let's consider the case in which there Alexa can propose two or more alternatives to the speaker. It would be necessary to ask directly confirmation for every intent that the user may want or not, in a precise order. If there were more than two possibilities, this could lead to a long list of Yes/No question. Instead, Alexa could directly ask which alternative is preferred, allowing a less awkward conversation with a list of possibilities. This is different from the simple intent invoking, because the dialog can be left to Alexa, allowing less coding.

With the combination of these two approaches, multi-turns conversations were achieved in this project: the skill code can update and change the intent without the need of the user to actually invoke it. This was used for the vehicle diagnostic, the most complex conversation of the skill: when a driver asks for the status of one component or when he asks Alexa to do the diagnosis of the overall vehicle and a problem is detected, the skill takes control of the situation. It let Alexa asks the user if he needs assistance, by asking if he wants more information about the problem or if he needs help to find a solution. In the first case, Alexa explains the problem in detail and what may have caused it, proposing also a solution if the driver confirms. In the second, Alexa gives practical advice, inviting in some cases to look for mechanics or gas station nearby. In the following diagram, the logic flow and the directives used are shown.

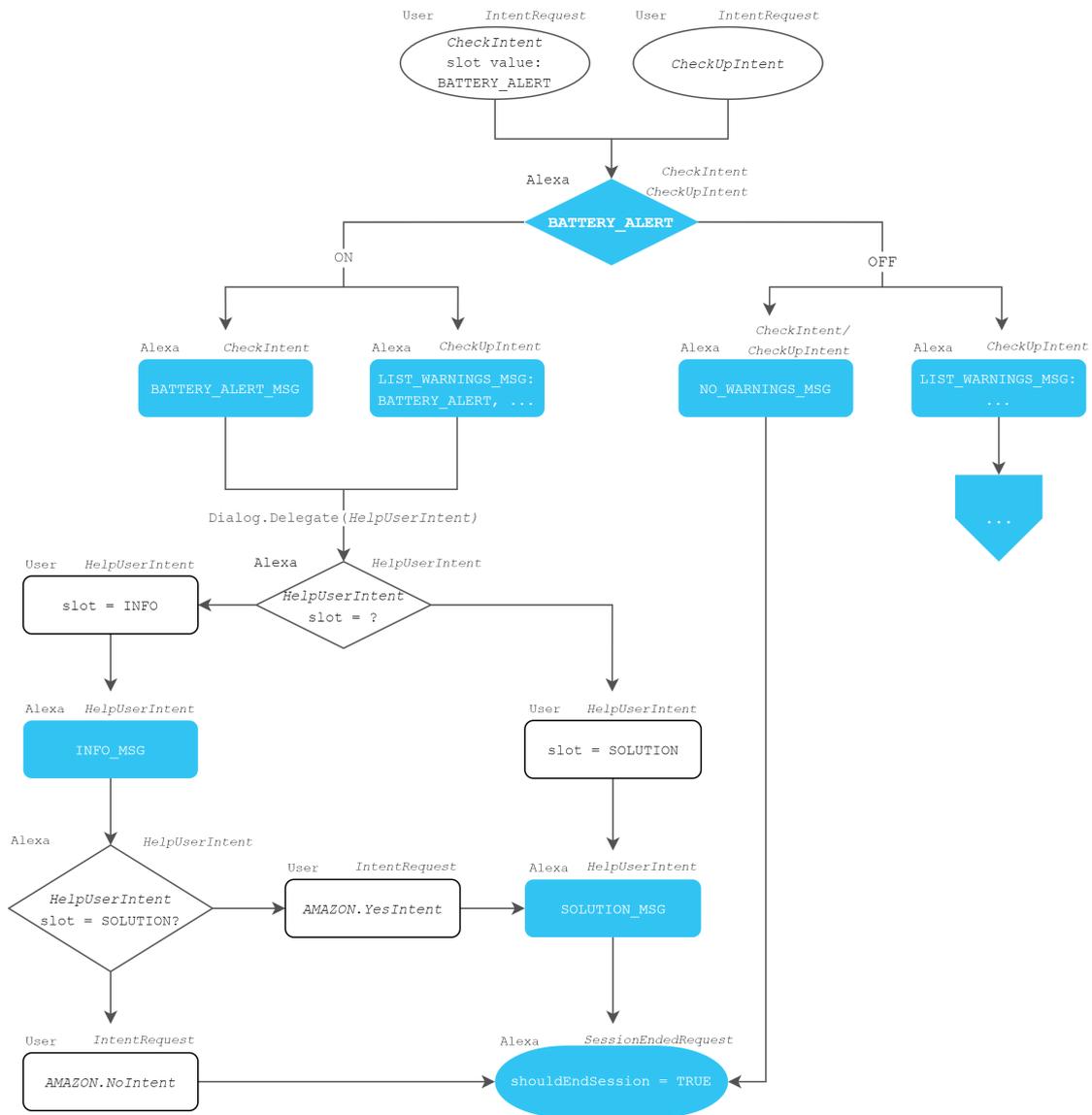


Figure 2.27: Flowchart showing intents, variables and directives through which the conversation could go

2.9 Alexa Auto SDK

Before moving on with the development of the Android app, it is worth spending some words about the Alexa Auto SDK.

This SDK (version 2.3) brings Alexa into the vehicle, adding automotive-specific functionalities. It includes libraries in C++ and Java that enable the car to process

audio inputs and to connect with the Alexa service, supporting Android, Linux and Automotive Grade Linux (AGL). The SDK includes core Alexa functionalities, such as speech recognition and synthesis, and can make any vehicle an Alexa-powered device. It is abstract and modular, providing the runtime engine to communicate with Alexa, but also interfaces to implement platform-specific behavior, such as media playback, template rendering, phone control and navigation control. This SDK can be include in the Android project. [48]

The next section is dedicated to the architecture of the SDK, to better understand how it works, focusing on how it can render data on a display, since it is a fundamental part of this thesis.

2.9.1 Auto SDK Architecture and Modules

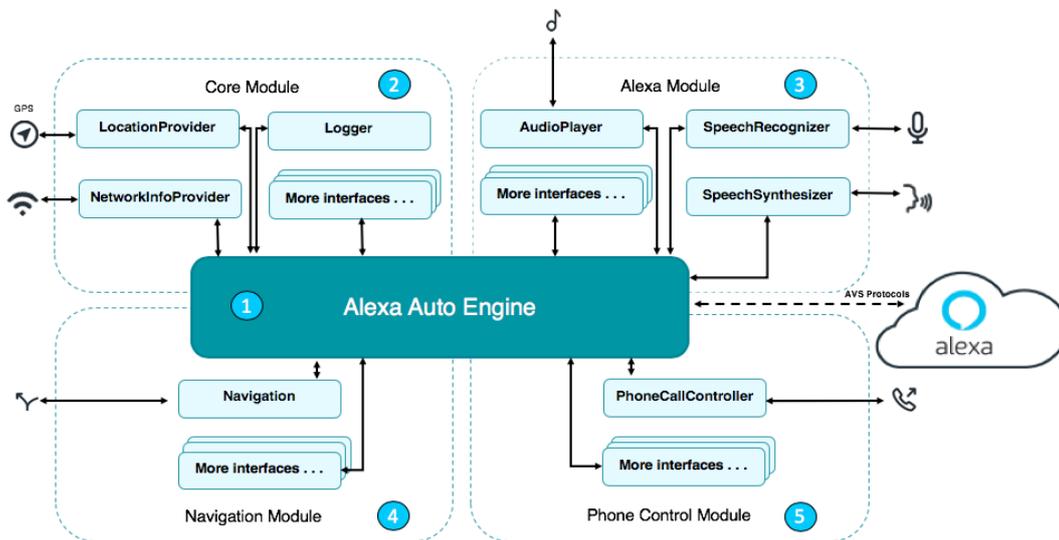


Figure 2.28: Alexa Auto SDK architecture and (some of the available) modules

1. **Alexa Auto SDK Engine:** it is the runtime implementation of the SDK. Its capabilities are expanded by the other modules, that implement platform-specific behavior, abstracted into interfaces called *platform interfaces*. These defines the API and how the application communicates with the Engine.
2. **Core Module:** gives an easy way to integrate the SDK into an application or a framework. It provides the Engine class, all the platform interfaces and the infrastructure for audio input and output, for logging, location and network information. All other modules are dependant from the Core Module.

3. Alexa Module: includes platform interfaces and support for Alexa features, such as speech and audio output, media playback, Alexa speaker and template rendering on screen.

These three contains the core functionalities of the SDK, while all the other module and extension are optional, such as the navigation and phone control. It is worth notice that there is a Local Voice Control (LVC) Extension, to provide some Alexa functionalities without an internet connection, running an Alexa endpoint inside the vehicle's head unit. [49]

To create an application based on this SDK, it is necessary to configure and build an instance of the Engine through the Core module. Then, it is possible to extend the Auto SDK interfaces by creating custom handlers for each interface wanted, registering in the Engine.

Since this thesis aims to give some visual feedback to the driver, it is necessary to understand how the Alexa Auto SDK can render images and cards. The Alexa module, providing interface for standard Alexa features, gives access to APIs to interact with the Alexa services, from speech input and output handling, authentication, to display card templates. Alexa can send visual metadata to be displayed, the *cards* mentioned in the Alexa chapter. The template information are sent and the platform implementation has to render them on the UI. Two main display card template types are present:

- **Templates** type, providing visual metadata in response to a user request. When Alexa wants to display some information on the screen, the Alexa Voice Service, other that sending a *Speak* directive, sends a *RenderTemplate* directive, to instruct the client to display some visuals. The *Render* directive supports these types of template:
 - *BodyTemplate1*, text-only cards.
 - *BodyTemplate2*,for cards that provides alsto an image with the text
 - *ListTemplate1*, to display lists and calendar.
 - *WeatherTemplate*, for weather data.
 - *LocalSearchDetailTemplate1*, for location information of interest.
 - *LocalSearchDetailTemplate1*, gives a list of navigation-based points of interest.
 - *TrafficDetailsTemplate*, to display travel distance and time.

The last three templates are available only for automotive products.

Each template type sends a unique *payload*, with parameters specific to the type of content that must be rendered, such as *title*, *subtitle*, *text*, *image URL* and so on.

- **PlayerInfo** type, providing visuals aimed to control media playing through the *AudioPlayer* interface, including control buttons.

The following diagram shows how the AVS-enabled product and AVS communicates to deliver visual metadata. [50]

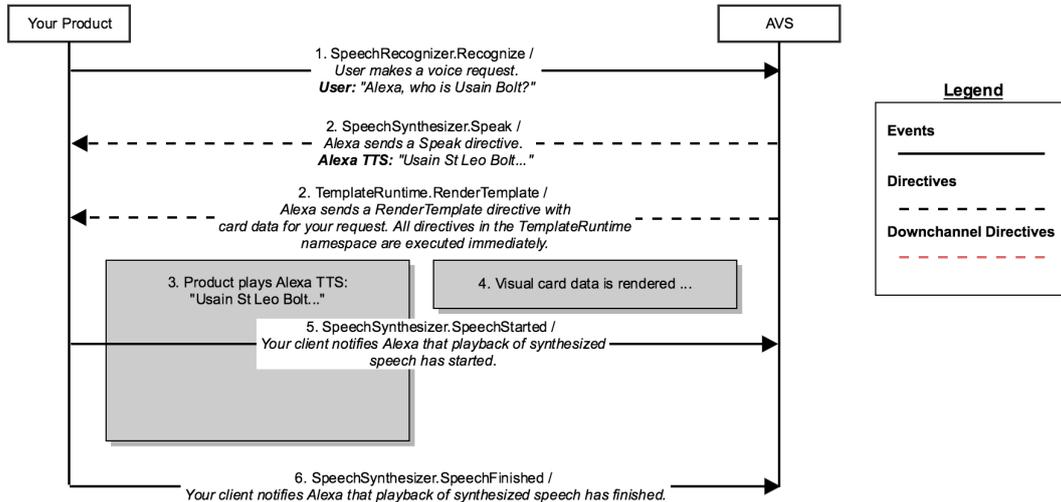


Figure 2.29: High-level message flow for displaying visuals

The *TemplateRuntime* class is in charge of rendering skill metadata and can be extended to implement custom handlers for GUI templates. The handlers take the the JSON string containing the visual metadata sent by Alexa, the *payload*, extract the useful information and then they can display them in a custom UI. [51]

```
public class TemplateRuntimeHandler extends TemplateRuntime {
    ...
    @Override
    public void renderTemplate( String payload ) {
        // Handle rendering the JSON string payload as a display card
    }
    ...

    @Override
    public void renderPlayerInfo( String payload ) {
        // Handle rendering the JSON string payload as a display card
    }
    ...

    @Override
    public void clearTemplate() {
        // Handle dismissing the display card
    }
    ...
    @Override
    public void clearPlayerInfo() {
        // Handle clearing the player info display card here
    }
}
```

Figure 2.30: The TemplateRuntimeHandler implementation extending the TemplateRuntime

Chapter 3

Android

3.1 Android overview

Android is the most widespread mobile Operating System in the world. Google launched the first Android commercial device in 2008, and, from that moment, this platform started its growth till obtaining nowadays the 72.92% of the market [52]. Android is open source and this made it so popular: the huge ecosystem of developers, very productive and with a lot potential, made this system so successful. Although Android is a mobile operating system, it is not only oriented to smartphone and tablets but also to smartwatch, wearable devices, televisions (Android TV) and is now coming to cars, with Android auto.



Figure 3.1: Android logo.

This OS is based on a complex architecture built upon a Linux kernel and it has been designed with the main goal of being easily updatable and flexible [53]. The architecture is divided in six basic components:

- The **Linux Kernel** is the base of the Android platform. It manages all the hardware components, the memories and the drivers, providing the most fundamental services. It is a monolithic kernel, where all the components share the same memory, the *User Address Space*. This implementation of the memory management allows the system to be faster in executing system calls or calls between operating systems components, with the drawback of being less reliable: the amount of code running in kernel mode and the ease

with which malfunctions can propagate among the components, corrupting the whole system, make it less secure.

- The **Hardware Abstraction Layer (HAL)** handles hardware components. It consists of multiple libraries that provide standard interfaces that expose the capabilities of the hardware component to the higher-level Java Api framework. Basically when an API makes a call to access a hardware component, the library module for that specific device is loaded, creating an interface.
- A layer composed the **Android Runtime (ART)** and **Native C/C++ libraries**. The first is used to compile the code, minimizing memory footprint, is the application runtime environment used by Android OS; the latter, is a set of native libraries required from many core system components, such as ART and HAL. Some of these functionalities are also exposed to apps from the Java framework API.
- The **Java API Framework** includes all the set of classes, packages and interfaces used for creating apps. This layer is the most important one, since it handles the interface with lower layers and avoids the programmer to taking care of them. Both official Java APIs and unofficial ones, third-party APIs, can be used. It contains services that enable access to the Android core features such as graphical components, activity managers and so on.
- The **System Apps**, the highest level, is a set of default apps that can provide key capabilities, to which programmers can access them from their own app.

Most of these layers are not directly used by developers, but it is worth to know the internal architecture of the system in order to better understand how it works. In this way, it is possible to develop an app that handles properly the available resources.

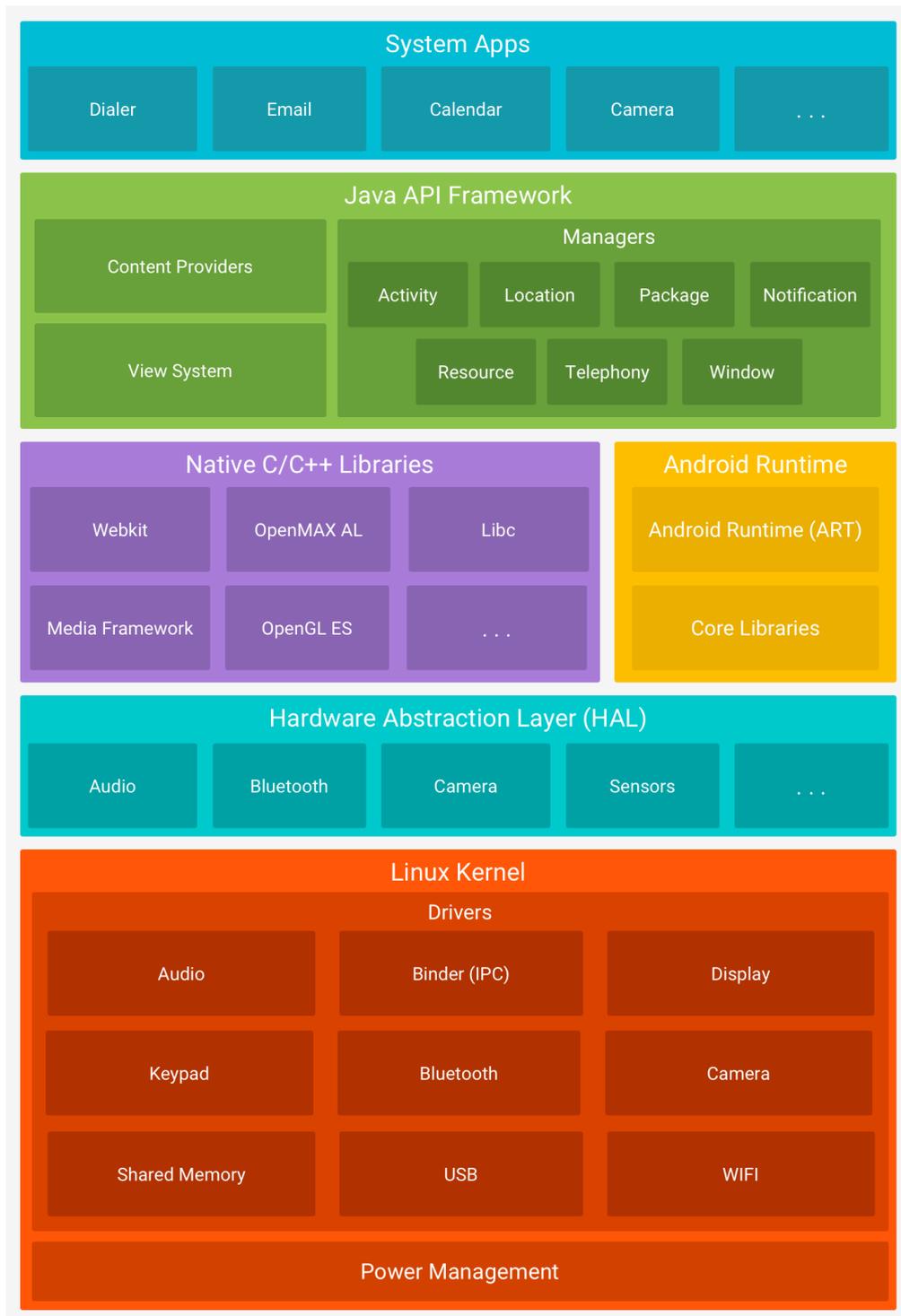


Figure 3.2: Android OS architecture

3.1.1 Android Automotive OS

Android Automotive OS is an Android-based operating system designed for vehicles: it is an infotainment platform based on Android, tightly integrated with the characteristics of a car. The OS is built directly onto cars and, besides the infotainment, such as navigation, music playback and messaging, it handles vehicle-specific functions. It provides a driver-optimized experience and allows user to install apps directly in the car. [54]

Many car subsystems are interconnected with each others and the In-Vehicle Infotainment (IVI) system through many bus topologies and protocols, such as Controller Area network (CAN) bus, Local Interconnect Network (LIN) bus, or automotive Ethernet. The Android Automotive HAL provides a unified interface regardless of the physical transport layer. A **Vehicle HAL (VHAL)** module can be implemented by connecting a specific platform HAL interface with technology-specific network interface (e.g. CAN bus). For example, it may include a dedicated MicroController Unit (MCU), running a proprietary real-time operating system to access the CAN buses, connected to the CPU running Android Automotive via a serial link. Or, instead of a dedicated MCU, bus access could be implemented as a virtualized CPU.

Summing up, the vehicle HAL (VHAL) defines the interface between the car and the vehicle network service, defining the properties that OEMs can implement. It is based on accessing properties that are an abstraction for a specific function. [55]

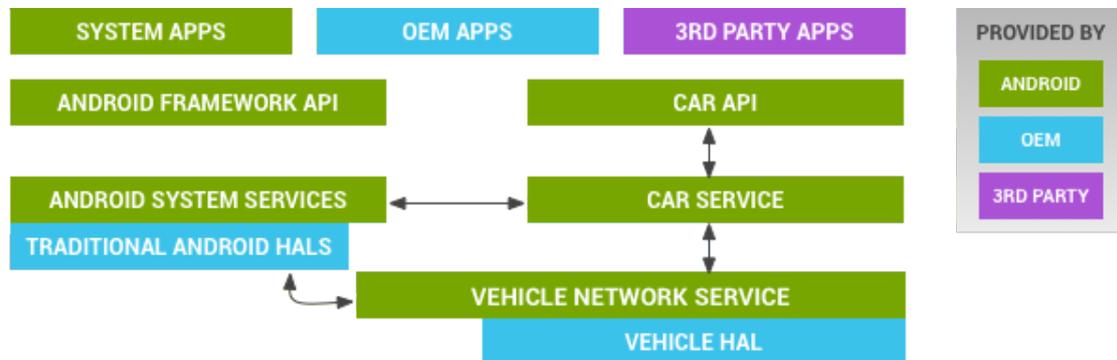


Figure 3.3: Vehicle Hal architecture

3.1.2 Android Studio and the Apps

After the overview on the internal architecture of the whole Operating System, it is worth spending some words on the Integrating Development Environment (IDE) Android Studio and the applications.

Android Studio includes everything necessary for designing a complete project, such

as the Android Software Development Kit (SDK), with all the Android libraries, and even the infrastructure to test and debug the application, from the Android emulator, for testing without a real device, to the Android Debug Bridge (adb), a command-line tool that let communicate with a real device [56]. Android apps can be written in Java, C++ and Kotlin and, when the Android SDK tools compiles the code, an Android package, the APK, is generated. The APK contains all the contents of the app and is the file that Android devices use to install the app. An Android project is characterized by four components [57]:

- The *activities* represent a single screen with a graphical user interface and are the base of each Android project. They are the main Java classes in which all the actual code is implemented, performing tasks inside the app. Usually, an Android project contains more than one Activity, to facilitate readability and to optimise the app callings. Activities work together to provide a cohesive user experience, but each one is still independent of the others. Most of the Java code is contained in the activities and links the graphical user interface implemented in the Layouts with an actual action.
- A *service* is a component that can run in the background, usually used to perform long-running operations. It does not provide a user interface.
- A *broadcast receiver* enables the system to deliver events outside of the regular user flow: this let an app respond to system-wide broadcast announcements, waiting for messages. It can generate notifications in the status bar
- *Content providers* manage the application data that can be stored in many ways.

A typical Android app contains multiple instances of these components, declared in the *app manifest*, an XML file in the root of the project source, that describes the essential information about the app, listing all the components, the permissions and the configuration information. Indeed, the Android system implements the *principle of least privilege*: each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission. However, there are ways for an app to share data with other apps and for an app to access system services

Android apps are built on a modular mechanism, to enable the creation of synergies, avoiding strong interdependencies, but letting them cooperate with each other. Applications use a particular kind of messages, *intents*, to activate each others or to exchange data. Intents define actions to be performed and sets of data on which to operate, as they are an abstract description of which operation is wanted to be performed: it is the OS that finds and instantiates the corresponding components

that can handle the required action. They are asynchronous messages and are used in an activity to request an action from another activity, or from some other app components.

Activities

Activities are the base class that provide GUIs with which users can interact. The OS creates the activity and manages its life cycle invoking some specific methods, such as *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, *onDestroy()* [58]. An activity manages user interactions, acquiring the required resources, builds and configure the graphical user interface, reacts to events triggered by users, manages notification regarding its own life cycle. It may start another activity by creating an *intent* object.

For each task started in the home screen of the device, the OS builds an *activity stack*, a memory region to add and remove data in a last-in-first-out manner, initialized with the default activity. During its lifecycle, an activity may request the system to start a new activity: the latter is created and inserted on the top of the stack, becoming visible and interactive; the previous is shifted back and remain in the background. Android allows the execution of several tasks at the same time and activities can be interrupted and paused when given events are triggered.

The first method that is called is the *onCreate()*, in which all the initializations and most resources allocation have to be done. This method also contains all the associations between the components created in the graphical Layout and the Java variables that will be used to perform a specific function. This method is followed by the *onStart()* that is called every time the activity becomes visible to the user. Then, very quickly, the activity enters the *Resumed* state and the system invokes the *onResume()*, called every time the activity starts interacting with the user, when it is on the top of the activity stack. When the activity has to be moved to the second position of the stack, it enters the *Paused* state, and the system invokes the *onPause()*. In this particular situation, the activity is going into the background, releasing all the unnecessary resources, but it has not been killed yet. The last two methods to be called are *onStop()* and *onDestroy()*. The former is called when the activity is no longer visible to the user because is going to terminate. The latter is called only and only if the activity has to be destroyed. This can happen because it has finished its operations or because of memory or energy optimization. The last method that has to be analyzed is *onRestart()* that is separated from the rest of the flow. This method is called after the activity has been stopped with *onStop()* as when the user presses the home button in the application. When this happens *onPause()* and then *onStop()* are invoked, and the Activity is moved to the background without being destroyed. In other words, it can restores the activity after been in background, without destroying it,

reducing access time. As already explained this method can be called only if there are no problems of memory or energy optimization otherwise the *onDestroy()* method is called. Usually, during the development of any Android project, the only method that has to be written by the programmer is the *onCreate()* because all the other methods are handled and optimized automatically by the compiler, unless specific actions are wanted when one of these other methods is called. In the *onCreate()* method, a *view* must be prepared and made visible: it is in charge of presenting content to the user and to allow interactions.

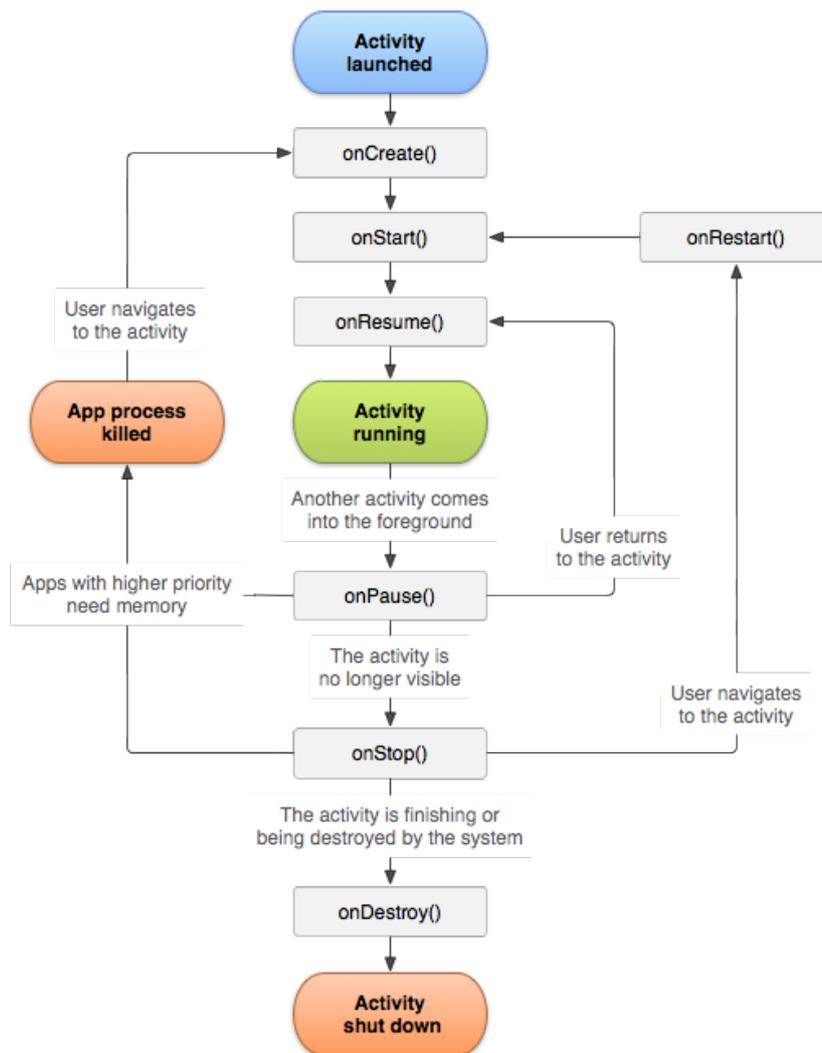


Figure 3.4: Activity lifecycle

The graphical user interface: *Views* and *ViewGroup*

The graphical user interface is usually made of a set of elementary widgets, connected together to form a hierarchical structure that reflects the usage of space inside the display. Android, as most of other GUI framework, relies on the composite pattern to model a hierarchy of visual components, *display-tree* [59].

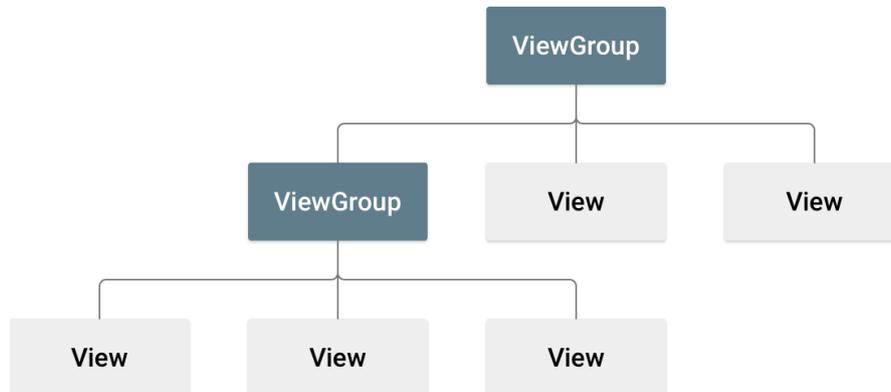


Figure 3.5: Hierarchy of views

A *view* is an instance of the *View* class, the basic building block for user interface components, such as buttons, texts, switches, images and so on. Views occupy rectangular areas on the screen and are in charge of visual resources and event handling. They are also the base class for *widgets*, the interactive graphical components. A subclass of the *View*, the *ViewGroup* is the base class for *layouts*, invisible containers that hold others Views or ViewGroups. Views can be created programmatically, described in a layout XML file or use a combination of the two approaches. In the first approach, the different visual components are directly instantiated and connected to each other to form the desired hierarchy, configuring the contents and registering event handlers from the code. This allows more flexibility and control, but maintenance is very difficult. An XML file to create a view is easier to maintain and can be visualized and designed with the visual editor provided in Android Studio; the only drawback is that the managing of dynamic contents might be difficult. With the hybrid approach, the hierarchy is described via the XML and all the elements can be manipulated from the code. This takes the advantages of the previous two approaches; the only drawback is that if the visual hierarchy is very complex, it may be difficult to maintain it.

Each view can have an *ID* property, an integer associated with them. The IDs are usually assigned in the layout XML file and are used to localize views within the view tree. Then, from an Activity, the element can be instantiated by finding the associated ID. From the Activity, views can be modified: properties can be set, such as the text for a *TextView*; they can be hidden or shown using the *setVisibility()*

command; *listeners* can be set up, allowing clients to be notified when something happens, as for buttons, to execute some commands when they are clicked.

Fragments

Now that the overview on an android app, activities and the GUI is given, it is worth spend some words on *fragments*, that are an essential component of this project. Fragments are a behavior or a portion of the user interface. Multiple fragments can be combined in a single activity to build a multi-pane UI for better maintenance and organization. A fragment can be considered as a modular section of an activity, with its own lifecycle, its own input events, that can be added or removed. Fragments encapsulates components in a reusable way, handling all the user interaction within the component. Fragments must be always hosted within an activity and are affected from its lifecycle. But when the activity is running, each fragment can be managed individually. They can be reused in multiple activities and simplify the task of adapting an interface to different screen types. But it is up to the activity to keep together the different blocks. They are added as a part of the activity layout in a *ViewGroup* inside the activity *hierarchy view tree*. [60]

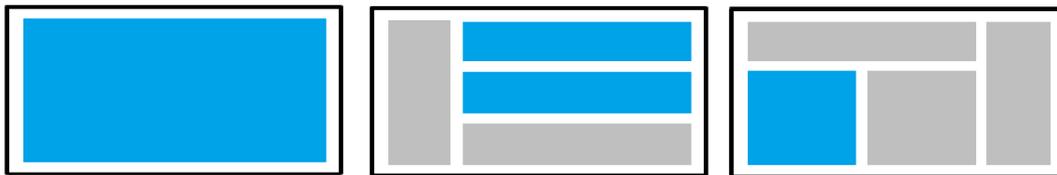


Figure 3.6: The same fragment could be reused in different ways

Fragments can be seen as a fusion between an Activity and a View: they have a complex life cycle, but also owns a hierarchy of views that can become part of the host activity visual tree. Differently from activities, that are completely managed by the Android framework, fragments can be manipulated by the programmer, creating, removing, showing them. When the Android system has to draw the fragment layout, it calls the *onCreateView()* callback method of that specific fragment and so here the layout must be provided. The method must return the root *View* of the layout: this can be done by inflating a layout resources defined an an XML. Fragments can be declared directly inside the activity fragment or can be added programmatically to an existing *ViewGroup* with the *FragmentManager* and *FragmentTransaction* APIs. And then, they can be removed with the *onDestroyView()*.

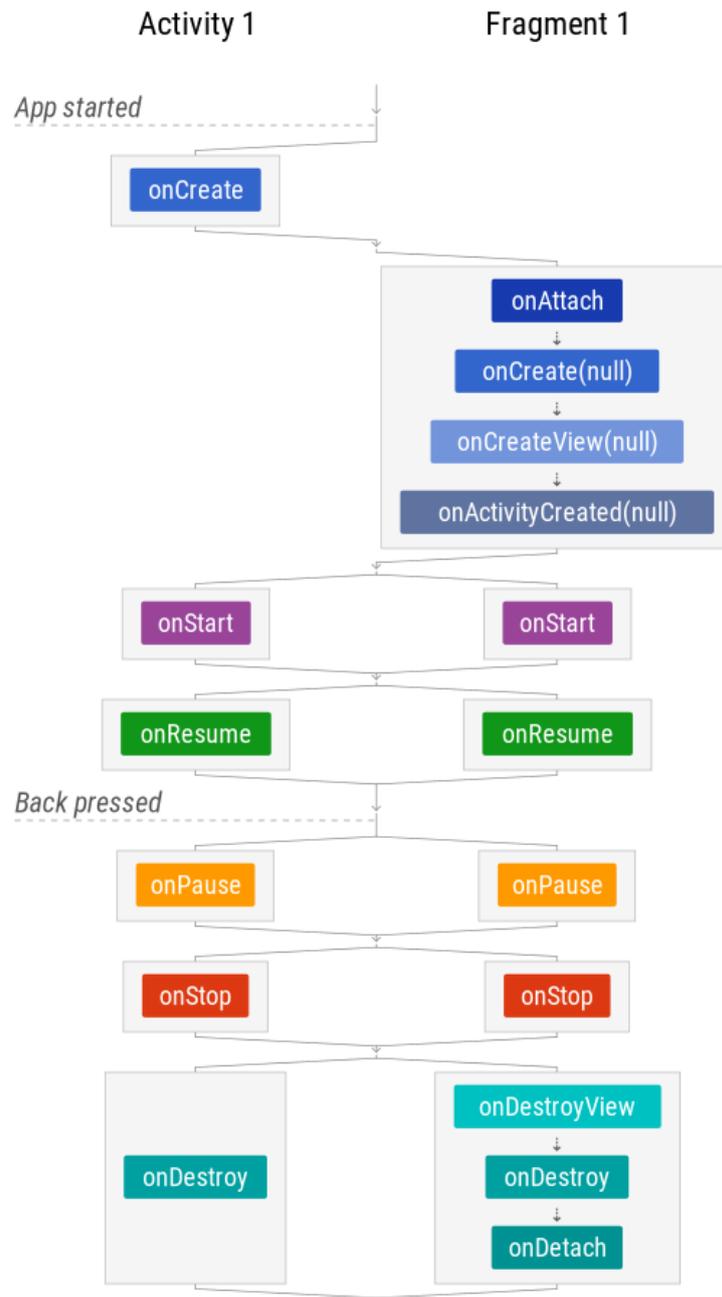


Figure 3.7: Activity lifecycle with a fragment

3.2 The Alexa app

3.2.1 Overview and GUI

The goal of this project is to integrate the automotive virtual assistant features of the skill in the Alexa app already developed by Magneti Marelli. As mentioned before, a skill is automatically active on all Alexa powered devices connected to the same account. Since this app is based on the Alexa Auto SDK, Alexa's core functionalities are already implemented and the skill is usable from the first moment, once the user logs in with the account on which the skill is activated. But the app should also provide an interface to expose the information about the vehicle and give some visual feedback. First, it is necessary to understand the structure of the app, both from how it works and how the GUI is implemented, to integrate new functionalities and graphic contents in a coherent and cohesive way.

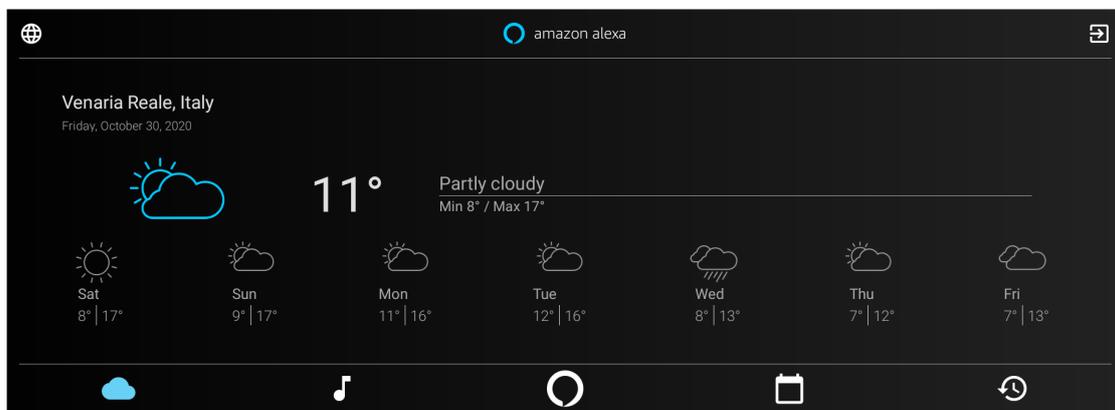


Figure 3.8: Alexa Android app: weather card

The app is constituted of a single Activity, the *MainActivity*, that instantiates and controls all the Alexa modules. All core functionalities are implemented starting from it.

The GUI consists of a single ViewGroup container that presents many layout inside, to host other fragments and other Views. The whole main screen consists of a *LinearLayout*, a type of ViewGroup layout that can hold others *child views* aligning them in a single direction, in this case, vertically. It contains three layers, from top to bottom: the first one is a *RelativeLayout*, a layout that displays children in relative positions and it is used so hold the Alexa logo as an *ImageView*, view object used to display an image resource, and the two buttons on the sides, used for selecting the language and logout from the Amazon account. These two are implemented as simple *ImageView* objects, but two *click listener* functions, one for each of them, describe the behavior and what happens when they are clicked. At the

center, a *Frame Layout*, a viewgroup used to display a single item, is used to show cards upon an interaction with Alexa or when the bottom grid button are pressed. These cards are simple XML layouts describing an empty structure for each type of Alexa interaction that can be displayed, and then they are populated with the data received in the Alexa response with the ***TemplateRuntimeHandler***, that receives the JSON *payload* and, depending from the type, fill the corresponding XML template. Each type of card has its own XML template, that the Alexa Auto SDK fill with data, and that the android app shows in the relative section of the app: there are the weather, the music player, the calendar, and the generic purpose section for Alexa cards. Hidden inside this frame layout, there is another *FrameLayout* used as a fragment container, to hold the fragments of the two menus for the language and the logout.

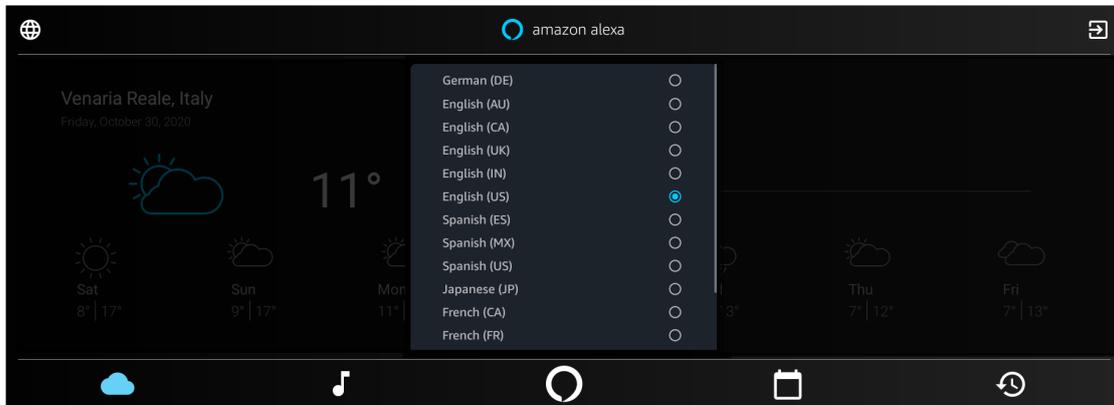


Figure 3.9: Alexa Android app: languages menu

One last *FrameLayout* is present on the bottom of the screen. Usually it holds the fragment that manages the buttons that allow to switch from one card to the other, except when the central one: when pressed, it activates Alexa, turning on the microphone and letting the driver speak. The Alexa state is shown through a new fragment that enters the screen with an animation from bottom to top, hiding the buttons grid. Alexa can be in three state, listening, thinking and speaking.

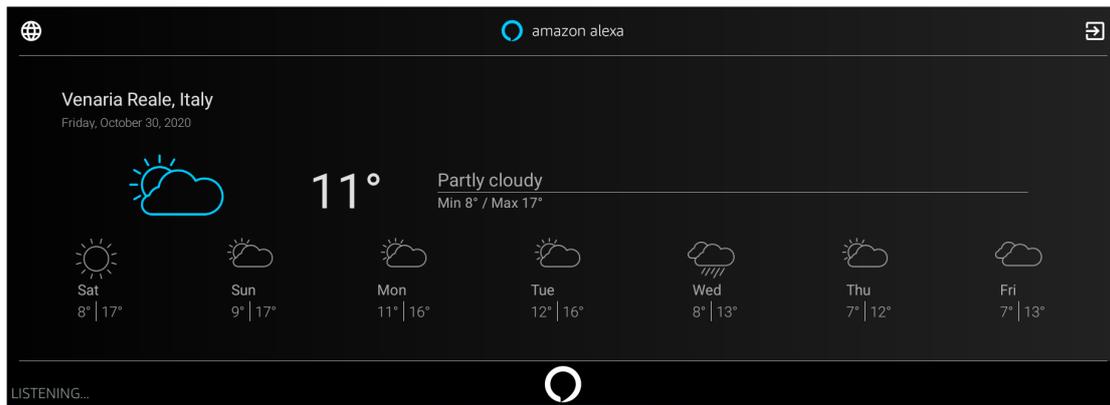


Figure 3.10: Alexa listening state

Here, the whole layouts structure is summed up and displayed, showing the background `LinearLayout` of the activity, the three `ViewGroups` hosted and also the empty fragment container at the center.



Figure 3.11: Layer structure analysis

After this overview on how the app GUI is structured and how everything is displayed, it is possible to create and add new screens and views and integrate the Alexa the skill developed before. But first, vehicle information must be modelled and stored in the app.

3.2.2 Vehicle parameters

The app manages the vehicle parameters, that must be abstracted and structured in a cohesive way with how the VUI could use them. Considering the skill, *attributes* were an essential components to provide data to the speaker and to handle data. The vehicle data should reflect that structure, to avoid the need of converting data in many different way, and instead to create a unique way of handling them. Here, it will not be discussed how these parameters are saved or loaded to the database, just how they are converted from a JSON structure to be represented in the Java code.

The attributes in JSON format are usually transmitted as string, that can be converted back to a **JSONObject**, the Java class used to represent them, that gives access to *put()* and *get()* methods. But JSONObjects lead to boilerplate code: it is not possible to access directly nested elements and it is necessary to encapsulate all JSON operation inside a **try/catch** that throws a **JSONException**, to indicate a problem with the JSON API; such problems include attempts to parse or construct malformed documents, such as NaNs (not a number) or infinities and using an out of range index or nonexistent name. To simplify the code development but also the readability, the best choice is to convert the JSONObject into a Java object, meaning that it is necessary first to build a Java class that can accommodate all the JSON attributes, and then mapping them to the Java object.

Attributes Java classes

The Java classes, to which the JSON should be converted, could be manually written, but this could lead to missing some pieces. By analyzing the structure of the attributes, there are 6 main types of attributes, with 40 parameters in total. Writing manually the Java classes is not the best idea. And, if any changes is done to the data structure, such as one single ID change or even the addition of one value, could lead to problems and errors. To solve this, the best way is to find an instrument capable of converting a JSON structure automatically. Online there are many tools capable of doing this. In this case, **jsonschema2pojo** was used: it generates *Plain Old Java Objects* from JSON, that can be copied from the DynamoDB table. Many settings can be tuned, to decide what the output java classes will be.

jsonschema2pojo

Generate Plain Old Java Objects from JSON or JSON-Schema.

The screenshot displays the main configuration page of the jsonschema2pojo converter. On the left, a JSON schema is shown with line numbers 1 through 56. The schema defines an 'attributes' object with nested objects like 'CTRL', 'INFO', 'LOCATION', 'MILEAGE', 'NEXT_CAR_SERVICE', 'PARKING_LOCATION', and 'PROFILES'. The 'PROFILES' array contains two profile objects with various settings like 'FAN_LVL', 'MIRROR_POSITION', 'SEAT_POSITION', 'STEERING_WHEEL_POSITION', and 'TEMPERATURE'. Below the schema are 'Preview' and 'Zip' buttons.

On the right, configuration options are provided:

- Package:
- Class name:
- Target language: Java Scala
- Source type: JSON Schema JSON YAML Schema YAML
- Annotation style: Jackson 2.x Jackson 1.x Gson Moshi None
- Generate builder methods
- Use primitive types
- Use long integers
- Use double numbers
- Use Joda dates
- Use Commons-Lang3
- Include getters and setters
- Include constructors
- Include `hashCode` and `equals`
- Include `toString`
- Include JSR-303 annotations
- Allow additional properties
- Make classes serializable
- Make classes parcelable
- Initialize collections
- Property word delimiters:

Figure 3.12: jsonschema2pojo converter main page

From this, the classes are generated, maintaining the hierarchy of the original attributes and providing all useful methods. In the next figure, it is possible to see, on the left, all the auto-generated classes, and how the *Attributes* class is defined as a set of sub-classes. For each of them, *get()*, *put()*, *toString* and other useful methods are defined.

```

1 package com.attributes;
2 import ...
11
12 @{...}
21 public class Attributes {
22     @JsonProperty("CTRL")
23     private CTRL cTRL;
24     @JsonProperty("INFO")
25     private INFO iNFO;
26     @JsonProperty("PROFILES")
27     private List<PROFILE> pROFILES = null;
28     @JsonProperty("STATUS")
29     private STATUS sTATUS;
30     @JsonProperty("VIN")
31     private VIN vIN;
32     @JsonProperty("WARN")
33     private WARN wARN;
34     @JsonIgnore
35     private Map<String, Object> additionalProperties = new HashMap<->();
36
37     public Attributes() {}
38
39     public Attributes(CTRL cTRL, INFO iNFO, List<PROFILE> pROFILES, STATUS sTATUS, VIN vIN, WARN wARN) {
40         super();
41         this.cTRL = cTRL;
42         this.iNFO = iNFO;
43         this.pROFILES = pROFILES;
44         this.sTATUS = sTATUS;
45         this.vIN = vIN;
46         this.wARN = wARN;
47     }
48
49
50     @JsonProperty("CTRL")
51     public CTRL getCTRL() { return cTRL; }

```

Figure 3.13: Attributes Java Class

```

119
120 @JsonProperty("FUEL_LVL")
121 public String getFUELLVL() { return fUELLVL; }
122
123 @JsonProperty("FUEL_LVL")
124 public void setFUELLVL(String fUELLVL) { this.fUELLVL = fUELLVL; }
125
126
127 public STATUS withFUELLVL(String fUELLVL) {
128     this.fUELLVL = fUELLVL;
129     return this;
130 }
131
132
133 @JsonProperty("LOCK")
134 public Integer getLOCK() { return lOCK; }
135
136 @JsonProperty("LOCK")
137 public void setLOCK(Integer lOCK) { this.lOCK = lOCK; }
138
139
140 public STATUS withLOCK(Integer lOCK) {
141     this.lOCK = lOCK;
142     return this;
143 }
144
145
146 @JsonProperty("MIRROR_POSITION")
147 public MIRRORPOSITION getMIRRORPOSITION() { return mIRRORPOSITION; }
148
149
150 @JsonProperty("MIRROR_POSITION")
151 public void setMIRRORPOSITION(MIRRORPOSITION mIRRORPOSITION) {
152     this.mIRRORPOSITION = mIRRORPOSITION;
153 }
154
155
156
157
158
159

```

Figure 3.14: STATUS Java Class

JSONObject to Java object conversion

Then, a JSON object can be converted to a java object with ease.

```
private Attributes convertFromJSON(String jsonStr) throws JsonProcessingException {
    ObjectMapper objectMapper = new ObjectMapper();
    return objectMapper.readValue(jsonStr, Attributes.class);
}

private String convertToJSON(Attributes att) throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    return mapper.writeValueAsString(att);
}
```

Figure 3.15: Layer structure analysis

From the main activity, the data can be accessed through the methods provided by the Java classes, easily, with *get()* and *set()*. For example, in the next figure, it is shown how the values in the hidden menu for manually setting the warnings, the switches values are read and the corresponding attribute is set with ease.

```
private void getSwitchesValues() {
    att.getWARN().setBATTERYALERT(switchBattery.isChecked());
    att.getWARN().setLAMPPOUT(switchLamp.isChecked());
    att.getWARN().setDOORAJAR(switchDoor.isChecked());
    att.getWARN().setENGINEWARN(switchEngine.isChecked());
    att.getWARN().setCOOLANTTEMPWARN(switchCoolant.isChecked());

    String value;
    switch (radioTire.getCheckedRadioButtonId()) {
        case R.id.tireLow:
            value = "low";
            att.getWARN().setTIREPRESSUREWARN(true);
            break;
        case R.id.tireMid:
            value = "mid";
            break;
        default:
            value = "high";
            att.getWARN().setTIREPRESSUREWARN(false);

            break;
    }
    att.getSTATUS().setTIREPRESSURE(value);

    switch (radioOil.getCheckedRadioButtonId()) {...}
    att.getSTATUS().setOILLVL(value);

    switch (radioFuel.getCheckedRadioButtonId()) {...}
    att.getSTATUS().setFUELLVL(value);
}
```

Figure 3.16: Example use of the *set* and *get* methods

3.2.3 Alexa Car section

First, a new section of the app is added, to be able to find all the vehicle information in a single panel.

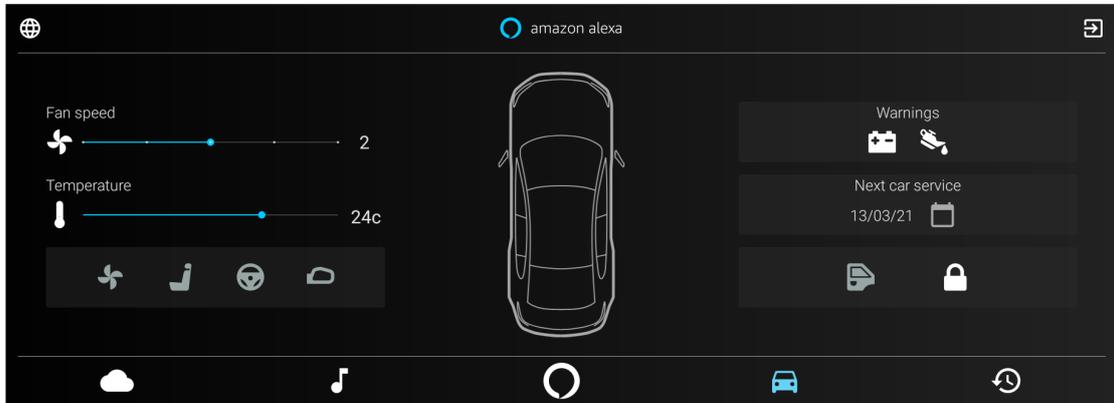


Figure 3.17: Car status panel

To do this, the calendar section of is replaced with a car one for this demo project. A new button, with the icon of a car, takes the place of the calendar button. It is implemented as an *ImageView* in the XML file of the *fragment_grid_bottom* layout, that displays the car icon and associates the id *alexaCar*.

Then in the Java class of the grid bottom fragment, an *ImageView* object is instantiated and associated to the one defined in the XML layout through the id, with *R.id.alexaCar*, and a click listener is set to detect touches: when the icon is pressed, if the car status panel is not already displayed on the screen, it is opened with *openFragmentCarStatusFragment()*, that displays a newly created fragment, *CarStatusFragment*, on the central layout of the main activity, after.

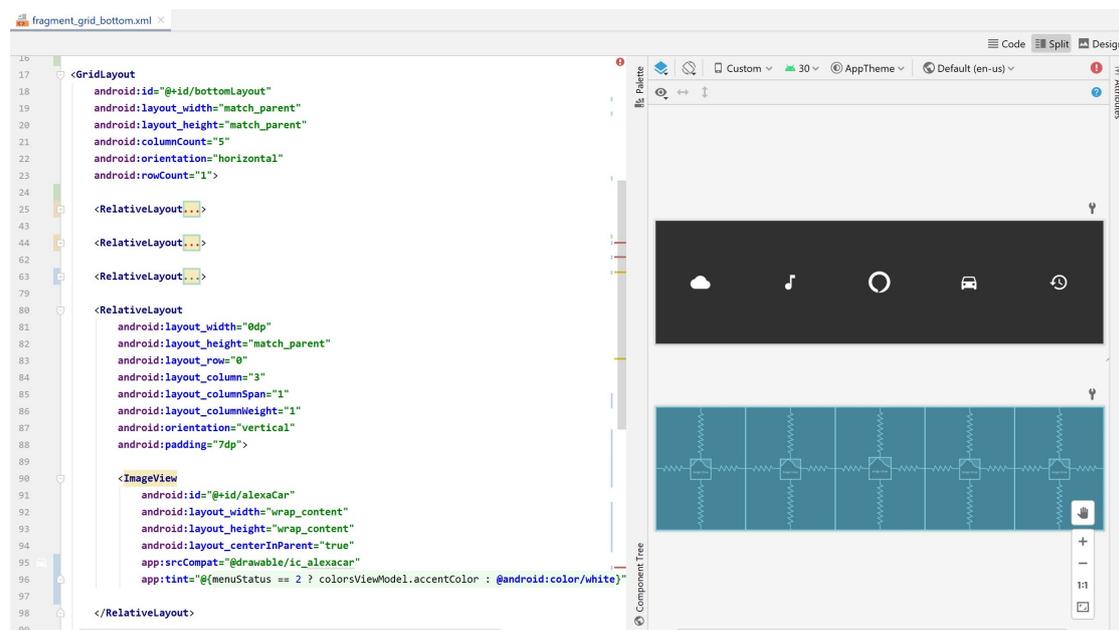


Figure 3.18: How the alexaCar button is defined in the XML file and how the fragment layout is displayed as a preview in Android Studio

```

alexCar = (ImageView) fragmentGridBottomBinding.getRoot().findViewById(R.id.alexCar);

alexCar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (!mActivity.getCurrentView.equals("alexCar")) {
            final FrameLayout centerFrame = (FrameLayout) (mActivity).findViewById(R.id.centerFrame);
            final FrameLayout centerCard = (FrameLayout) (mActivity).findViewById(R.id.centerCard);
            if (centerFrame != null)
                centerCard.removeAllViews();

            mActivity.setGetCurrentView("alexCar");
            mActivity.openFragmentCarStatusFragment();
            fragmentGridBottomBinding.setMenuStatus(2);

            active = alexaCar;
            if (getArguments() != null)
                getArguments().clear();
            Bundle b = new Bundle();
            b.putString("active", "alexCar");
            setArguments(b);
        }
    }
});

```

Figure 3.19: The button declaration and its click listener

The function also makes sure to clear the screen to avoid overlapping views.

```

public void openFragmentCarStatusFragment() {
    if (!openCarStatus) {
        Log.i(Thread.currentThread().getStackTrace()[2].getClassName(), Thread.currentThread().getStackTrace()[2].getMethodName());
        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.setCustomAnimations(R.anim.fade_in, R.anim.fade_out, R.anim.fade_in, R.anim.fade_out);
        transaction.addToBackStack(null);
        transaction.add(R.id.centerCard, carstatusfragment, tag: "CARSTATUS_FRAGMENT").commit();
        openCarStatus = true;
    }
}

```

Figure 3.20: The function that puts the car panel on the screen.

In this function, it is possible to see how fragments can be added to an activity inside a specific ViewGroup: the *carstatusfragment*, an instance of the *CarStatusFragment* Java class that defines the fragment, is added to the *centerCard* layout of the main activity, with an animation, thanks to a *FragmentTransaction* object and its native methods.

This fragment, the *CarStatusFragment*, contains the panel showing the status of the vehicle. The XML layout of the fragment is made of a *GridLayout*, a ViewGroup that places its children equally spaced columns and rows.

Here, it is possible to see the structure and the main activity on the screen once the status panel is displayed. The blue layout is the actual fragment base *GridLayout*, with one row and three columns, each of them containing another layout.

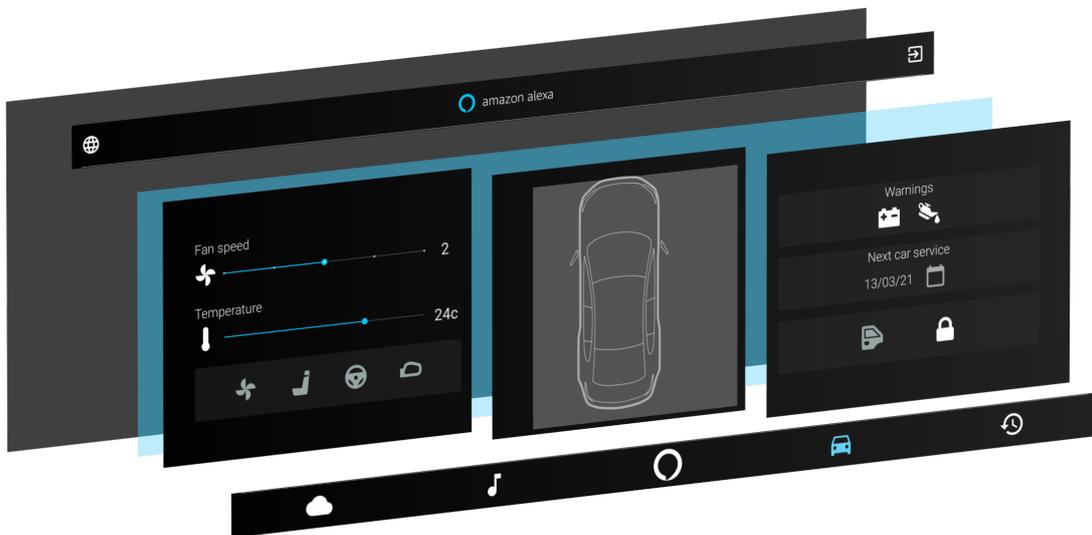


Figure 3.21: Car status panel main layouts structure.

Here, the blueprint of the fragment shows all the single components.

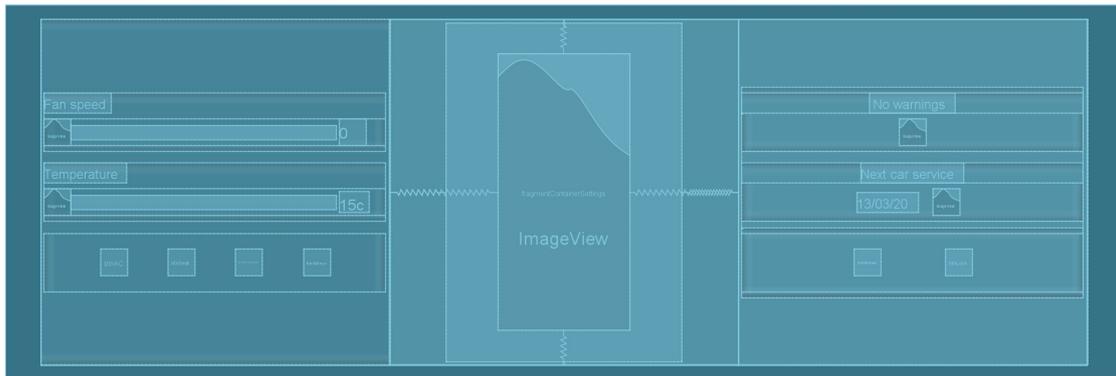


Figure 3.22: Car status panel: the fragment layout blueprint

The first is dedicated to the vehicle settings, such as the climate control features and the mirror, the seat and the steering wheel settings. The fan speed and the temperature controllers are implemented with a *SeekBar*, with a draggable thumb: the user can touch it and drag it, setting the current progress level as desired. For the other settings, it is possible to access a menu to set the desired position of those components. Such menus are instantiated as a fragment, that is added in the central empty container. This demonstrate how the fragments are modular components: there is only one *fragment_settings* layout and Java class, used multiple times with a different implementation. Based on the selected component, the fragment is personalized with the correct icon and the correct current position of that specific component is shown. The button is first found in the layout with its id: with *R.id.btnMirror* it is possible to access the id, then and the view is found and associated to a *ImageButton* object.

```
MirrorButton = (ImageButton) v.findViewById(R.id.btnMirror);
MirrorButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) { mActivity.openFragmentSettings( type: "MIRROR_POSITION"); }
});
```

Figure 3.23: The listener creation.

The function *openSettingFragment()*, as the name suggest, is a custom method instantiated in the Main Activity that opens or closes the *fragment_settings*. It takes *type* as input, a string indicating which component is selected. This value is passed to the fragment as an *argument* inside a *bundle*. The latter is a mapping of key-value couples, that can be passed to the fragment with the *setArguments()*, that supply the construction arguments for the fragment. In the *onCreateView()*

method of the fragment, the type value is read and used to select the correct icon and values.



Figure 3.24: Setting fragment inflated to control the mirrors

The second column host an image used for decorative purposes and to allow future implementation, such as adding animations.

In the last column, there is the status panel, showing warnings, some relevant information such as the next revision date, and the status of the vehicle windows and the locks. All of these value depends on the vehicle parameters.

In this thesis, these parameters are simulated and inserted manually through a secret menu accessible by long-clicking the word *WARNINGS*.

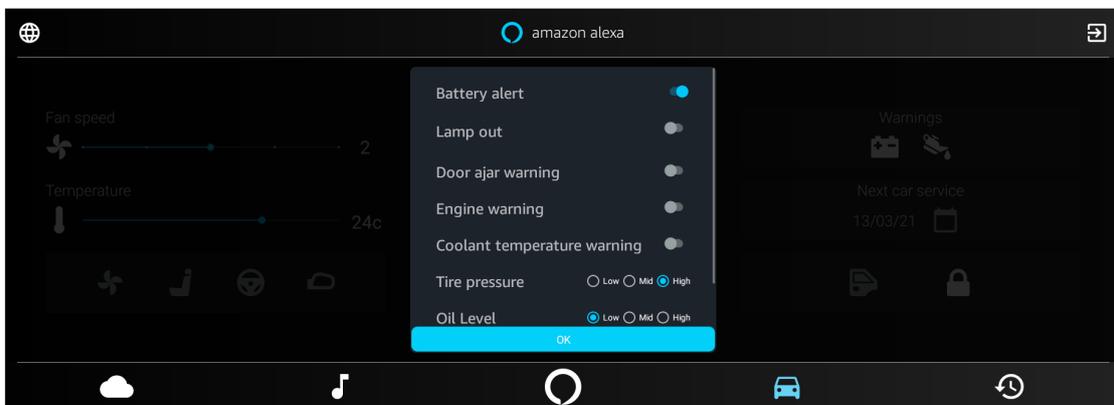


Figure 3.25: Warning menu fragment inflated.

It is worth noting how the button are implemented in these cases, to showcase some of the Android Views widgets: the first ones are *Switches*, two states toggle switch widgets, while the others are *Radio buttons*, that allow o select only one option from a set, a *Radio Group*.

3.2.4 Notification Fragment

The Alexa skill is capable of controlling the vehicle and change its parameters, such as turning on the fans or locking the vehicle. Amazon guidelines suggest that, when Alexa execute a command to control a device, Alexa should not confirm the action. But the GUI could notify these commands in some way, to confirm the correct execution and give a visual feedback, other than changing the vehicle parameters. To do that, a new fragment is created, mimicking the fragment that shows the Alexa state at the bottom of the screen, when invoked.

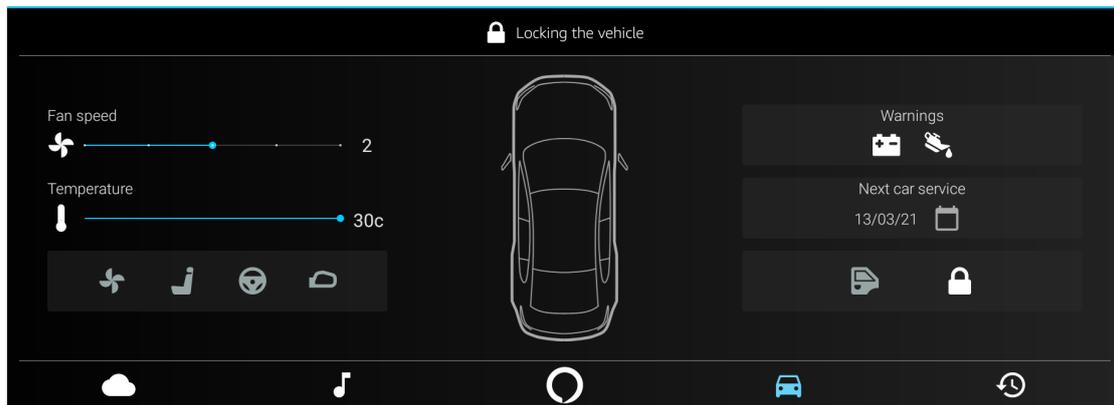


Figure 3.26: The notification fragment displayed on the top of the screen.

For this, a new java class is created, the *NotificationFragment*, extending the *Fragment* class. Its *onCreateView* method returns an XML layout that will host an icon and a text communicating what action is being executed, depending on the value passed to the fragment as arguments. The *MainActivity* instantiate a *NotificationFragment* and, when Alexa control a vehicle component through the database, the activity is notified: depending on the type of control action, the fragment is opened object by passing the as arguments the *type*, such as open or close, and the value *value*, such as lock or fans. The fragment is added with an animation, in this case, entering from the top of the screen, with a ***FragmentTransaction*** object. In addition, the two buttons for logout and languages are deactivated during the. The fragment is automatically closed after 3 seconds, invoking the closing fragment function.

```

public void openFragmentNotificationFragment(String type, String value) {
    View notChrome = findViewById(R.id.notification_chrome_view_container);
    final ObjectAnimator backgroundColorAnimator = ObjectAnimator.ofObject(notChrome, propertyName: "backgroundColor",
        new ArgbEvaluator(), Color.parseColor( colorString: "#00000000"),
        Color.parseColor( colorString: "#00CAFF"));
    backgroundColorAnimator.setDuration(300);
    backgroundColorAnimator.start();

    Bundle bundle = new Bundle();
    bundle.putString("value", value);
    bundle.putString("type", type);
    // set FragmentClass Arguments
    notificationFragment.setArguments(bundle);
    Log.i(Thread.currentThread().getStackTrace()[2].getClassName(), Thread.currentThread().getStackTrace()[2].getMethodName());
    FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
    transaction.setCustomAnimations(R.anim.enter_from_top, R.anim.exit_to_top, R.anim.enter_from_top, R.anim.exit_to_top);
    transaction.addToBackStack(null);
    transaction.add(R.id.notificationFragmentContainer, notificationfragment, tag: "NOTIFICATION_FRAGMENT").commit();
    this.findViewById(R.id.Logout).setClickable(false);
    this.findViewById(R.id.ty).setClickable(false);
    final Handler handler = new Handler();

    handler.postDelayed(new Runnable() {
        @Override
        public void run() { closeFragmentNotificationFragment(); }
    }, delayMillis: 3000);
}

```

Figure 3.27: The function that displays programmatically the notification fragment, setting the argument values.

3.2.5 The skill card

In the overview, it is explained how the *TemplateRuntime* is responsible for rendering visual metadata coming from Alexa, but only specifics type of templates can be rendered. The Alexa skill can send a card inside its response, but the Android app can't distinguish between the skill card and another card send, for example, when asking "How far is the moon from the Earth?". All generic cards are mapped as *BodyTemplate1* or *BodyTemplate2*, so, in theory, there is no distinction between the *vehicle-skill* cards and the other ones. But, to provide a coherent user interface, the custom skill cards should be displayed inside the AlexaCar section. It is necessary to find a workaround to distinguish the content of the cards.

By analyzing the templates payloads, it is possible to notice that, when a custom skill sends a card, a string with name of the skill is included in the JSON as a *sub-title*: this can be used to create custom functionalities relative to the skill. Upon receiving a *TemplateRuntime.RenderTemplate* directive with a card, the Alexa app can use *renderTemplate()* to read the skill name inside the JSON *payload* and execute different commands.

```
@Override
public void renderTemplate(String payload) {
    try {
        // Log payload
        JSONObject template = new JSONObject(payload);
        String type = template.getString( name: "type");
        String subTitle = "";
        if (template.getJSONObject("title").has( name: "subTitle")) {
            subTitle = template.getJSONObject("title").getString( name: "subTitle");
        }
        Log.i(sTag, subTitle);
        Log.i(sTag, payload);

        if(mActivity.getCurrentView.equals("alexaCar"))
            mActivity.closeFragmentCarStatusFragment();
        switch (type) {
            case "BodyTemplate1":

                if (subTitle.equals(new String( original: "vehicle-skill"))) {
                    viewAlexaCar.updateView(template);
                } else {
                    viewAnswer.updateView( type: "BodyTemplate1", template);
                }
                break;
            case "BodyTemplate2":
                if (subTitle.equals(new String( original: "vehicle-skill"))) {
                    viewAlexaCar.updateView(template);
                } else {
                    viewAnswer.updateView( type: "BodyTemplate2", template);
                }
                break;
            case "ListTemplate1":
                viewList1.updateView(template);
                break;
            case "WeatherTemplate":
                viewWeather.updateView(template);
                break;
        }
    }
}
```

Figure 3.28: How the app distinguish between a vehicle-skill card and a generic card.

In this way, the app is able to distinguish between any card and the vehicle-skill ones, and can also display them in different ways, even though they are formally the same object. The visual metadata is extracted from the payload as a *template*, a JSON object, and passed to a *viewAlexaCar* object, instance of the custom Java class *ViewAlexaCar*, that extracts data from the template to populate the associated XML layout, displaying it on the screen.

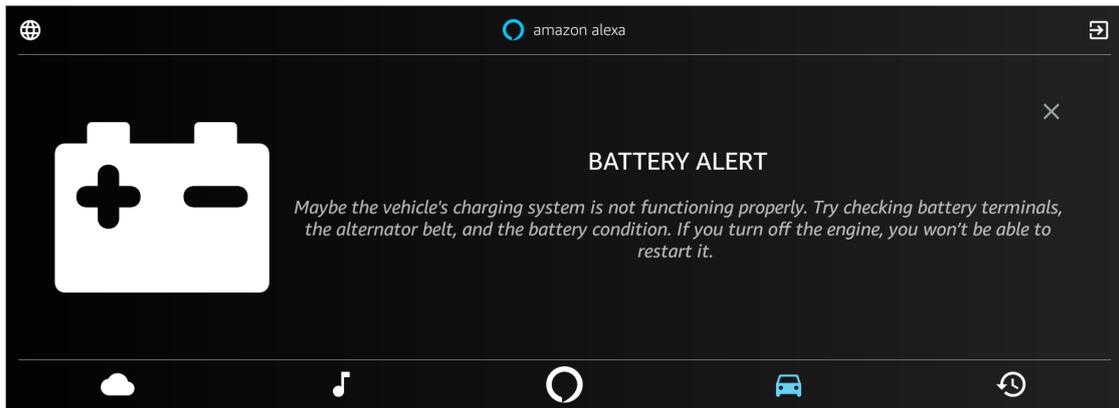


Figure 3.29: The card sent when asking the skill if there is a problem with the battery, rendered in the car section.

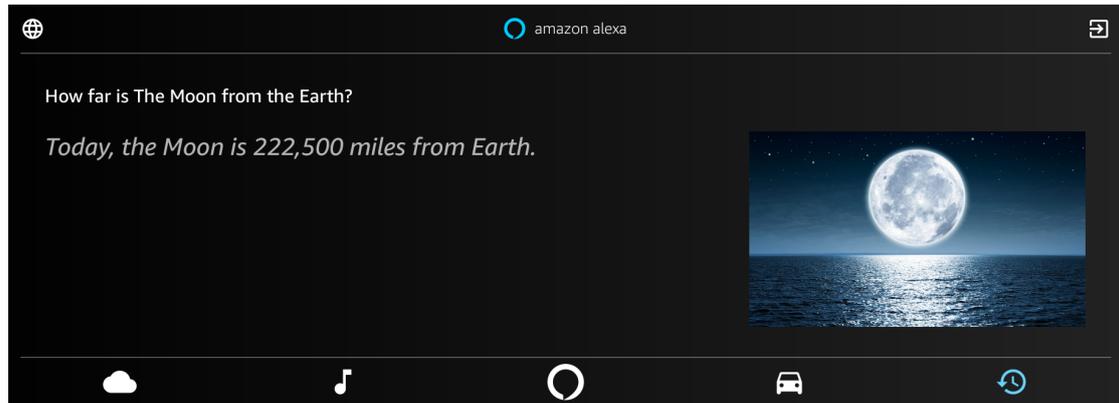


Figure 3.30: How a simple request is rendered placed in the general section of the app.

Chapter 4

DynamoDB

Both the Alexa skill and the Android app need a place to store the vehicle information and a way to communicate with each other. The proposed solution to these problems utilize Amazon DynamoDB, an AWS service. It is a durable, multimaster and fully managed database. This allows to focus on building the application, instead of spending time to configure the database and updating it. It also comes with a built-in security and backup system and in-memory caching for internet-scale applications. This service can handle more than 10 trillion requests per day, supporting even peaks of more than 20 million requests per second. It is used by many big enterprises, such as Toyota and Samsung.[61]

DynamoDB is a *NoSQL* database, designed to run at high performances even for that applications that would overburden the classical relational databases. It stores key-value data and document, usually used for web, mobile, IoT and all other application that need low-latency accesses.



Amazon DynamoDB

Figure 4.1: DynamoDB logo.

Through the DynamoDB console, the tables can be easily managed, from creating them, to adding, deleting and querying data. But it also provides instruments to connect other types of application to it, with AWS SDKs. DynamoDB stores *tables*,

a collection a *items*, that are also collection of *attributes*. With a primary key, each item of the table is uniquely identified in a table, and with other additional secondary keys, it is possible to have more querying flexibility.

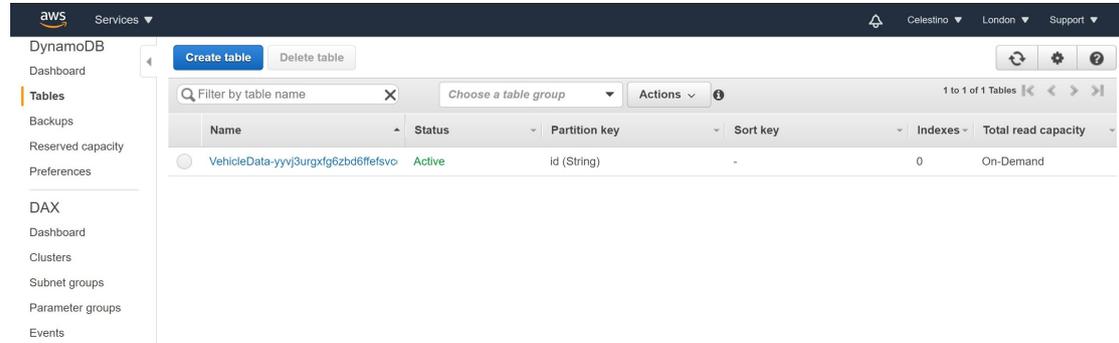


Figure 4.2: The DynamoDb console with the available table

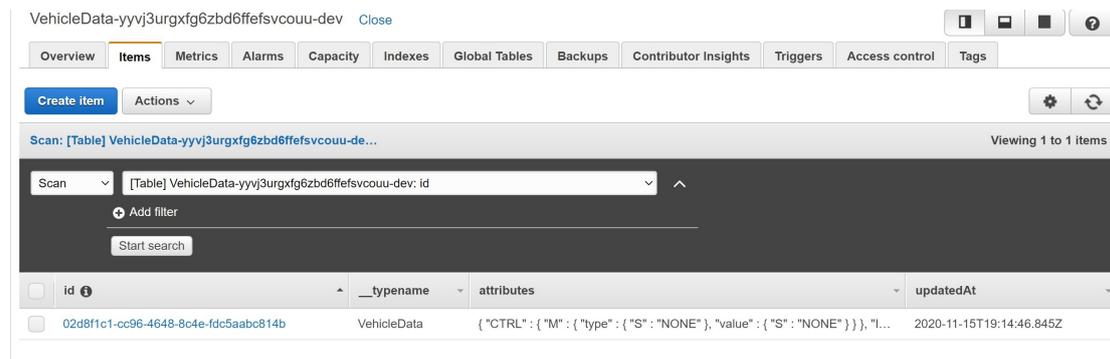


Figure 4.3: The table

An application, to work with Amazon DynamoDB, must use Some simple API operation, for *control plane* operation, to create and manages tables, for *data plane* actions, to execute action on data in a table.

In the next sections, it will be shown how this database is used to store data from both the Alexa skill and the Android app, creating a link between two different worlds.

But, before moving on, few words are spent to describe non-relational databases.

4.1 What is a NoSQL database

Relational databases, accessed by *SQL*, Structured Query language, are the traditional databases that were predominant until 2000s. From the 70s, when SQL

databases rose in popularity, storage space was very expensive, so developers normalized the databases to reduce data duplication as much as possible.[62] SQL is the standard language used for relational database management, such as update or retrieve data. A relational database contains objects called *tables*, where data and information are stored, each of them identified with a unique name.

A *NoSQL* database is built for specific data models with a flexible schema [63]. They are recognized as one of the most easy to develop, with the most functionalities and for their performance at scale.

They are *non-relational* database, that stores data in other formats, not only relational tables. This does not mean that data can't have some relationship: they can store relationship data, just in a different way. NoSQL data models let to nest all the related data within a single data structure, without the need of splitting them between tables. The data models are tailored for the specific use case, so they can support bigger workloads better, providing better performance than relational ones for the same use case. They are very scalable, making them able to maintain performances. This means that, with the growing amount of work, the potential of the system can be enlarged to accommodate that growth.

4.2 The connection between DynamoDB and the Android app

To connect the Android app to DynamoDB, AWS services help developer with Amplify.

4.2.1 AWS Amplify

AWS Amplify is a set of tools to build scalable and secure full stack applications with AWS services, for web and mobile applications [64]. It allows to configure app backends in an easy and fast way. It will be used to connect the existing cloud backend to the mobile app. The Amplify libraries and APIs let the android app to connect to a DynamoDB table: then the car parameters can be loaded, making them available for the skill. It is available for Android, iOS, Javascript and Flutter.

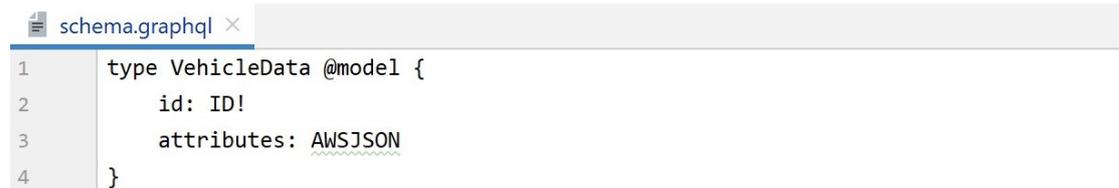


Figure 4.4: AWS Amplify logo.

The amplify libraries can be installed in the Android project as dependencies in the build configuration [65]. Then, Amplify must be initialized in an *Application*

class, that must be included in the manifest of the app. Amplify simplify the provisioning of backend resources across the different Aws services. This tool provide also APIs to connect the android app to a DynamoDB table. These APIs provides interfaces for retrieving and persisting model data, with a built-in support for AWS AppSync, a fully managed service to develop and handle GraphQL APIs, connecting to data sources such as DynamoDB: these will provide CRUD operations (Create, Read,Update, Delete) and also real-time functionalities. Where GraphQL is a language for APIs to query and manipulate data, providing a description of the data and a way to retrieve exactly what is wanted. [66] The API category of Amplify provides solution to make HTTP requests to GraphQL endpoints. Usually, to integrate AWS AppSync, it is necessary to set up the API endpoint and the authentication settings, generate the code from the API schema and then write the code to manipulate the data. Amplify provides all the tools to make this process really easy.

Everything start with a GraphQL schema, that will be the base model of how the data are structured. In this case, the schema is really basic, to accommodate the structure of the Alexa skill attributes in the JSON format:

A screenshot of a code editor showing a GraphQL schema. The editor has a tab labeled 'schema.graphql' with a close button. The code is as follows:

```
1 type VehicleData @model {
2   id: ID!
3   attributes: AWSJSON
4 }
```

Figure 4.5: GraphQL schema

From this, Amplify can automatically generate code to manipulate the data in the format of *VehicleData*, creating a java class.

```
VehicleData.java ×
18  |  /** This is an auto generated class representing the VehicleData type in your schema. */
19  |  /all/
20  |  @ModelConfig(pluralName = "VehicleData")
21  |  public final class VehicleData implements Model {
22  |      public static final QueryField ID = field("id");
23  |      public static final QueryField ATTRIBUTES = field("attributes");
24  |      private final @ModelField(targetType="ID", isRequired = true) String id;
25  |      private final @ModelField(targetType="ANSJSON") String attributes;
26  |      public String getId() { return id; }
27  |
28  |
29  |
30  |      public String getAttributes() { return attributes; }
31  |
32  |
33  |
34  |      private VehicleData(String id, String attributes) {
35  |          this.id = id;
36  |          this.attributes = attributes;
37  |      }
38  |
39  |      @Override
40  |      public boolean equals(Object obj) {...}
41  |
42  |
43  |
44  |      @Override
45  |      public int hashCode() {...}
46  |
47  |
48  |
49  |      @Override
50  |      public String toString() {
51  |          return new StringBuilder()
52  |              .append("VehicleData {")
53  |              .append("id=" + String.valueOf(getId()) + ", ")
54  |              .append("attributes=" + String.valueOf(getAttributes()))
55  |              .append("}")
56  |              .toString();
57  |      }
58  |
59  |
60  |
61  |
62  |      @Override
63  |      public static BuildStep builder() { return new Builder(); }
64  |
65  |
66  |
67  |
68  |
69  |
70  |
71  |
72  |
73  |
74  |
```

Figure 4.6: *VehicleData* auto generated class by Amplify

With this class, it is possible now to retrieve the data from the backend resource and convert them in a *vehicleData* type object, with an ID and a JSON containing all the attributes. Now it is possible to query data, fetching them from the table and converting the attributes in a readable JSON with the Amplify framework, accessing the API.

```

public void getVehicleData() {
    Amplify.API.query(
        ModelQuery.get(VehicleData.class, ID),
        response -> {
            String answer = ((VehicleData) response.getData()).getAttributes();
            Log.i( tag: "MyAmplifyApp", msg: "vehicleInfo: LOADED");
            try {
                att = convertFromJSON(answer);
            } catch (JsonProcessingException e) {
                e.printStackTrace();
            }
        },
        error -> {
            Log.e( tag: "MyAmplifyApp", error.toString(), error);
        }
    );
}

```

Figure 4.7: Function to retrieve vehicle parameters from the database.

```

private void updateVehicleData() throws JsonProcessingException {
    //To avoid triggering the notification
    att.getCTRL().setType("NONE");
    att.getCTRL().setValue("NONE");

    String attStr = convertToJSON(att);

    VehicleData data = VehicleData.builder()
        .id(ID)
        .attributes(attStr)
        .build();

    Amplify.API.mutate(
        ModelMutation.update(data),
        response -> Log.i( tag: "MyAmplifyApp", msg: "Updated data: " + response.getData().getId()),
        error -> Log.e( tag: "MyAmplifyApp", msg: "Update failed", error)
    );
}

```

Figure 4.8: Function to update vehicle parameters to the database.

This framework also provide a very interesting feature: subscription. They allow to create real-time clients, capable of listening mutations on the data and execute code only when triggered [67] . This is the key feature to Allow the Alexa skill to communicate in real-time with the android app, making it capable of control the vehicle. GraphQL natively support subscription to perform real-time operation and AppSync can push data to all the clients listening. Thi means that it can make any data source real-time with no effort, since the connection management is handled automatically by AppSync libraries. So, whenever the Android app create a GraphQL subscription operation with the Amplify client, a secure WebSocket connection is established and will remain connected constantly, allowing to receive

real-time data from the skill, that becomes a data source.

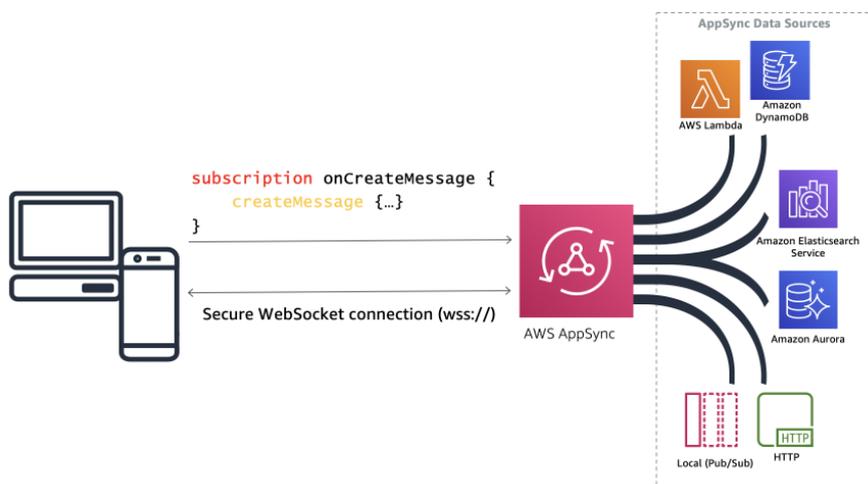


Figure 4.9: How AppSync works

Different types of subscriptions are supported, to detect when data are created, deleted or updated.

Thus, in the *onCreate()* method of the *MainActivity*, the app is subscribed data updates on DynamoDb:

```
ApiOperation subscriptionOnUpdate = Amplify.API.subscribe(
    ModelSubscription.onUpdate(VehicleData.class),
    onEstablished -> Log.i( tag: "MyAmplifyApp", msg: "Subscription on update established"),
    onUpdate -> {...},
    onFailure -> Log.e( tag: "MyAmplifyApp", msg: "Subscription on update failed", onFailure),
    () -> Log.i( tag: "MyAmplifyApp", msg: "Subscription completed")
);
```

Figure 4.10: Subscription creation.

and with the behavior of the specific cases are define. With *onUpdate*, it is specified what happens when data are updated. The Alexa skill can send commands through the database, the android app client reacts to it through the subscription, read the command and execute what it is necessary.

```

onUpdate -> {
    Log.i( tag: "MyAmplifyApp", msg: "VehicleData update subscription received: " + ((VehicleData) onUpdate.getData()).getAttributes());
    runOnUiThread(new Runnable() {
        public void run() {
            try {
                att = convertFromJSON(((VehicleData) onUpdate.getData()).getAttributes().toString());
            } catch (JsonProcessingException e) {
                e.printStackTrace();
            }
            String type = att.getCTRL().getType();
            String value = att.getCTRL().getValue();

            if (!type.equals("NONE")) {
                openNotificationFragment = true;
                openFragmentNotificationFragment(type, value);
                switch (type) {
                    case "OPEN":
                        switch (value) {
                            case "WINDOWS":
                                att.getStatus().setWINDOWS(1);
                                break;
                            case "LOCK":
                                att.getStatus().setLOCK(1);
                                break;
                            case "FAN_LVL":
                                att.getStatus().setFANLVL(1);
                                break;
                        }
                        break;
                    case "CLOSE":
                        switch (value) {...}
                        break;
                    case "SET_PROFILE": {...}
                        break;
                }
            }
        }
    });
}

```

Figure 4.11: *onUpdate* behavior.

When an update is detected, the DynamoDB table is retrieved and casted first into a *VehicleData* object and the attributes JSON is extracted and converted to a string with the native methods. Then, the attributes string is converted to an *Attributes* object, allowing to use more intuitive methods to access nested elements. The *CTRL* attribute is read, to understand what the Alexa skill wants. When all the operations are complete, the app takes care of cleaning the *CTRL* attribute values, to be sure that even when the app updates the data of the vehicle on the database, no residual command risks to be triggered.

4.2.2 AWS and Amplify configuration

It is worth spending some words about the AWS configuration necessary to make Amplify work.

The Amplify Command Line interface (CLI) is the unified toolchain to create cloud services based on AWS [68]. Once downloaded, the first thing to do is to configure it, logging into the AWS console using the correct IAM user. Amazon IAM (Identity and Access Management) is used to manage user and user permissions within AWS services. The user has to have administrator access to the account to provide all the

resources for Appsync from AWS. This user will be identified with an *accessKeyId* and the *secretAccessKey* and the CLI will connect using these keys. Then, the CLI is configured and Amplify can be initialized in any project with it running the correct command from the terminal: all the necessary configuration steps will be automatically performed. After that, Amplify features can be manually implemented, such the GraphQL API used here.

4.3 The connection between DynamoDB and the Alexa skill

The Alexa skill already can communicate with the database with the persistence adapter, as mentioned in chapter 2.

With the integration of the AWS Amplify framework, it was shown how the subscription service can automatically detect GraphQL mutations and push data to all connected clients. But the persistence adapter utilized by the skill lambda function does not actually execute GraphQL mutation: thus, it is necessary another way to communicate with the database, when a command is wanted to be sent from the skill to the car.

A GraphQL mutation must be manually done by the lambda function.

```
const axios = require('axios');
const gql = require('graphql-tag');
const graphql = require('graphql');
const { print } = graphql;

const updateVehicleData = gql`
mutation updateVehicleData($input: UpdateVehicleDataInput!) {
  updateVehicleData(input: $input) {
    id
    attributes
  }
}
`

exports.handler = async (attributes) => {
  try {
    delete attributes["loaded"];
    delete attributes["STATE"];
    delete attributes["NEXT_STATE"];
    delete attributes["PROBLEMS"];

    const graphqlData = await axios({
      url: 'https://6acs5mmorbe3bdbpmltiwazzcy.appsync-api.eu-west-2.amazonaws.com/graphql',
      method: 'post',
      headers: {
        'x-api-key': "da2-biko6os16rhgznghsf6dhblgdu"
      },
      data: {
        query: print(updateVehicleData),
        variables: {
          input: {
            id: "02d8f1c1-cc96-4648-8c4e-fdc5aabc814b",
            attributes : JSON.stringify(attributes)
          }
        }
      }
    });
    const body = {
      message: "successfully updated VehicleData!"
    }
    return {
      statusCode: 200,
      body: JSON.stringify(body),
      headers: {
        "Access-Control-Allow-Origin": "*"
      }
    }
  } catch (err) {
    console.log('error creating todo: ', err);
  }
}
```

Figure 4.12: How the Lambda function performs a GraphQL mutation.

The GraphQL mutation must be manually written in the correct syntax, then the lambda function as to do an HTTP request directed to the correct URL of the AppSync API used by Amplify and the correct credentials must be included in

the request, such as the *accessKeyId* and the *secretAccessKey*. In this way, when this module is invoked, a mutation is executed, uploading the attributes and the command. When the speaker wants to control the vehicle triggering the correct intents, one attribute category is set: the **CTRL** attribute, with a *type* and a **value**, is used to **OPEN**, **CLOSE** a component, passed with its ID as a *value*, or **SET-PROFILE**, with the *personID*.

Chapter 5

Conclusions

5.1 Module Testing

All the single components must be tested to verify if they works as a single components and as a whole system.

Alexa skill

The Alexa skill can be tested almost everywhere with any device. The simulator on the developer console allows to test the VUI and the backend logic, especially to achieve the natural conversation as it is implemented in this thesis.

All the intents are checked, to see if they trigger the correct intent handler and elaborate the data as they should, if they execute the correct API calls. In the next few images, some of the intents are invoked through the developer console simulator.



Figure 5.1: *CheckIntent*

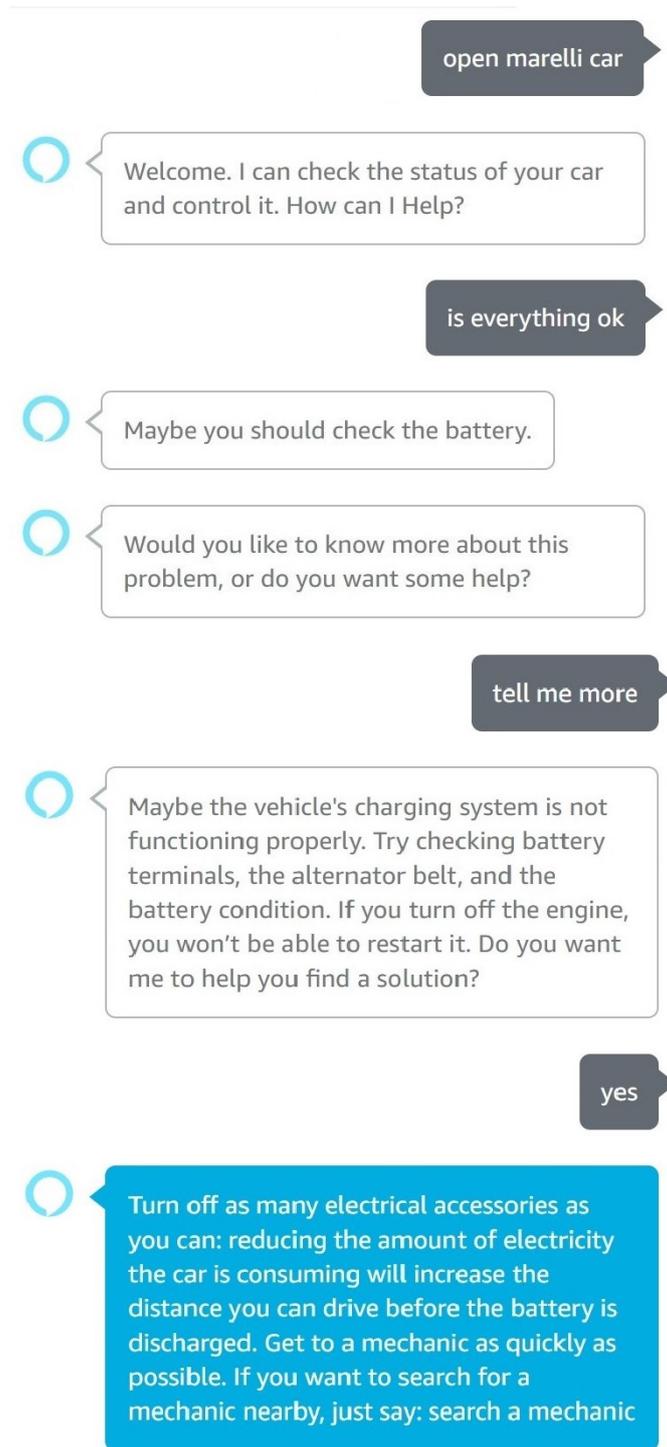


Figure 5.2: The longest conversation possible

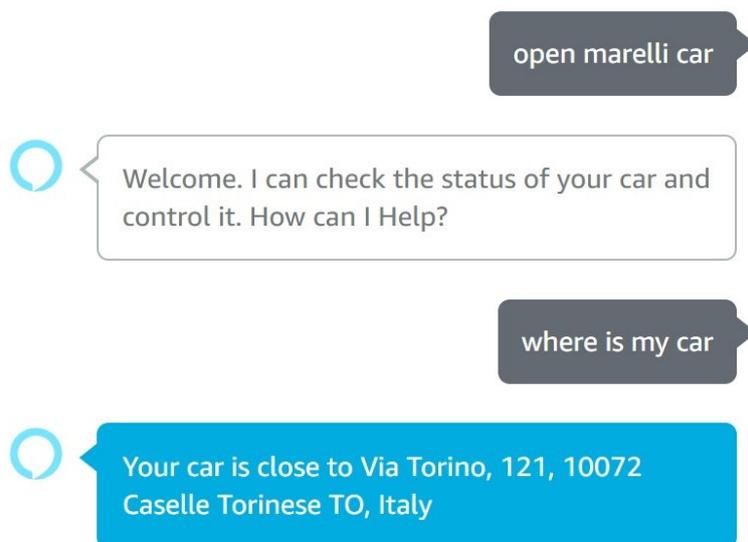


Figure 5.3: *GetPositionIntent*

The simulator shows also the input and output JSON, as shown in the chapter dedicated to Alexa. These are also available in AWS CloudWatch, a monitoring and observability service provided by Amazon. Here, with the interceptor defined, each JSON request is logged. It makes also possible to use it to log information from the Lambda function, allowing the debug and troubleshoot of the code. The vehicle parameters on the database can be manually changed in this phase, to verify all the cases.

The skill instantly usable on any Alexa-powered device, answering the speaker requests about his vehicle and sending cards too:

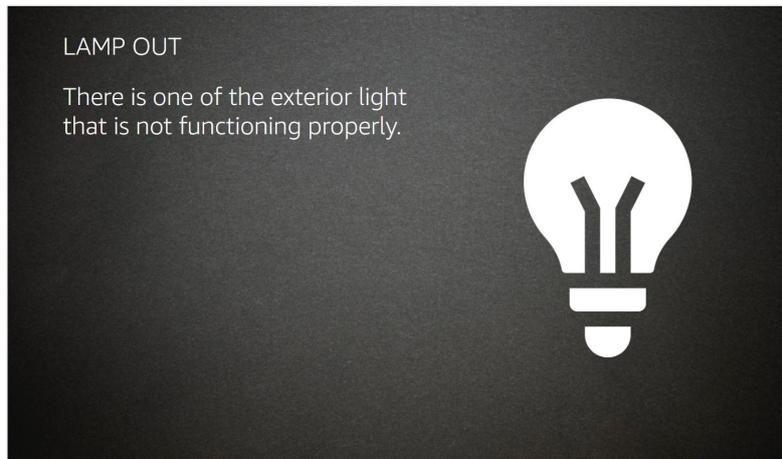


Figure 5.4: How the skill cards are displayed on any Alexa device with a screen.



Figure 5.5: How the skill card is displayed on a smartphone

Android app

The app is tested with an Automotive Development Platform (ADP), SA8155P, an embedded solution that provides access to a high-performance automotive infotainment, advanced driver assist platform for developing, testing, optimizing and showcasing in-vehicle infotainment solutions. Connecting a microphone, one speaker and a touchscreen, it provides all the necessary functionalities.

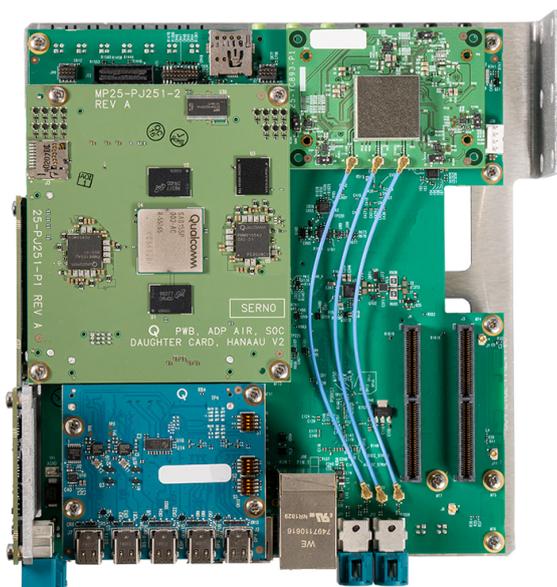


Figure 5.6: The SA8155P ADP.

Android Automotive OS is installed on the platform and, with Android Studio running on the computer, it is possible to install the app directly on the device, to test it. The device is connected through the previously mentioned adb, a client-server program: the client runs on the development machine and sends commands; a daemon (adb) runs the command on the device, running as a background process on it; a server, that manages the communication between the daemon and the client, running as a background process on the development machine [69]. Once the connection is established, Android studio can install directly the apk on the device.

The results are the screens shown in chapter 3, that were directly captured from the screen of this test environment.

5.2 System testing and results

To check if all the modules deployed into the final architecture leads to the expected functionalities, all of them are connected and many tests are run to verify if they works. In reality, most of the previous tests already had some types of interaction with other components.

Here, it is presented the final architecture of the whole system, how every components and AWS service interact with each others:

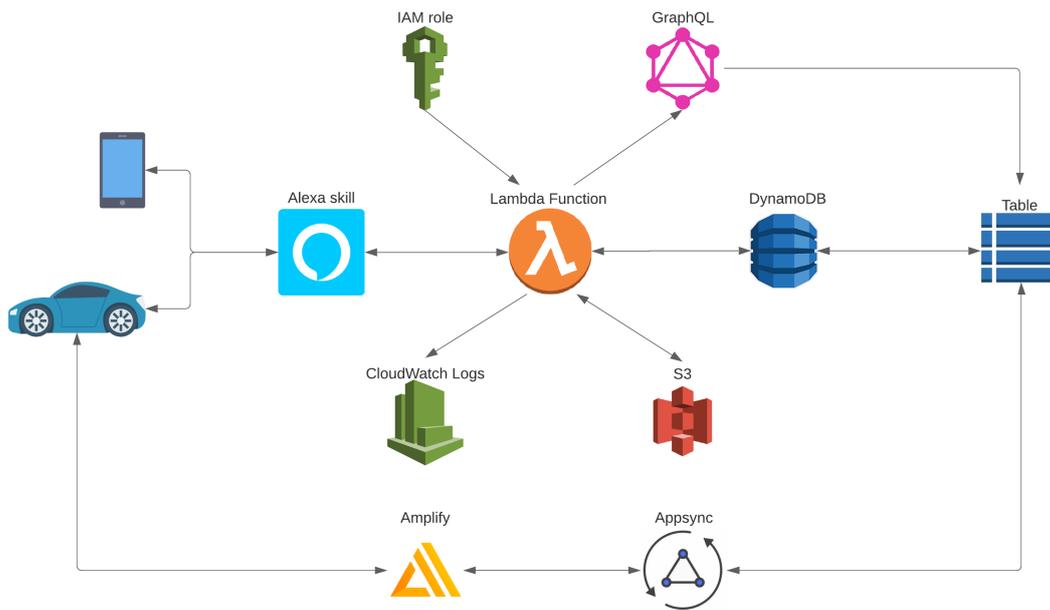


Figure 5.7: The whole system architecture.

From the vehicle or the smartphone, with an Alexa app installed, the skill is invoked and requests are processed by AWS Lambda. Lambda interfaces with many different services to provide the the skill functionalities: IAM to manage all the accesses to AWS resources, DynamoDB to retrieve persistent attributes, S3 to load he images for the cards, CloudWatch to log events; it also execute HTTP requests to send commands to the vehicle via GraphQL mutations. The vehicle, update its data on the table through Amplify and Appsync, and detect the mutations performed by the skill.

As it is possible to see, it is a complex structure with many different components and services, that works together to present a functioning automotive virtual assistant. All interconnected, AWS services are capable of interacting with each other, automatizing some processes.

5.3 Future improvement

The one presented in this thesis, it is a functioning system. But many improvement could be done: during the study and the development, deepening the knowledge of all of these services, many choices were done because they were the less complex choice. Every component could be implemented in a different way. This application relies on AWS services as much as possible, gaining reliability and an integrated environment.

Alexa features are continuously expanded, every month new features are released. The development should be a continuous process to integrate the new capabilities to offer a better experience. For example, interaction without using an invocation name is currently released as a beta feature, as *Alexa conversation*, a new method to delegate dialogs to Alexa, to take care of more complex interactions.

Even Alexa Auto SDK are constantly updated, releasing version 3.0 almost at the end of the development of this project.

Anyway, some future implementation and suggestion related to all the system could be:

- Increase the number of sample utterances and slot synonyms to increase the possibility of the VUI.
- Use more external API to offer many different features, all within the same skill.
- Better management of the skill attributes and GraphQL mutations: essentially, they do the same thing in different way. But the latter was necessary to use the real-time features of Amplify.
- Both the skill and the app are manually linked to one DynamoDB table: it would be better to automatize this process.
- An authorization mode should be implemented to protect the access to table: as it is now, anybody, with the correct keys, can access it. It is fine only for development purposes.
- Update from Alexa Auto SDK v2.03 to v3.0.
- Utilize the LVC of the Alexa Auto SDK: at the moment, every command sent from the skill has to pass from the cloud. The SDK supports local voice commands, for offline controls, but they would be commands not integrated within a skill, but with the embedded system on the vehicle.
- Vehicle parameters could be modelled in a more reliable way.

- Vehicle data are manually set in this project: they could be generated or simulated differently.

5.4 Conclusions and Final Remarks

The work done for this master thesis has given the opportunity to evaluate the potential of Alexa combined with Android, applied to the automotive environment. As it is, this project shows how to connect very different systems and showcase some of Alexa's capabilities, but also demonstrate some of its limitations, that needed a workaround to achieve what was wanted.

Alexa is a new technology and is moving the first steps in this field, while Android is new too, compared to others IVI systems, but comes with a great experience behind it and a large slice of the market for mobile devices such as smartphones and tablets. This is a great benefit because the end user is already familiar with the systems components, its applications and its operation, so it will be possible to find inside the vehicle the same functionalities with which he interacted so far at home or with its portable devices.

The community of developers, growing more and more, makes it possible to develop applications for this environment because there are no substantial differences with what there was before. In this way these developers can reuse their knowledge even in a new field of application.

The biggest of all the benefits is for the manufacturers that don't have to develop an ad-hoc VPA or an operating system, that would entail many a big amount of resources. So, new experiences can be delivered to the drivers with new products, using familiar technologies and creating a complex, but user-friendly, system.

Bibliography

- [1] C.W. Kim and C.K. Suh. «Factors Affecting the Intention to Use the Intelligent Personal Assistant». In: *Proceedings of International Academic Conferences (No. 5807971)*. 2017 (cit. on p. 1).
- [2] M. McTear. «The Dawn of the Conversational Interface». In: Switzerland:Springer International Publishing (2016) (cit. on p. 1).
- [3] N. Pantidi B.R. Cowan. «What can I help you with?: infrequent users' experiences of intelligent personal assistants?» In: Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services (2017) (cit. on p. 1).
- [4] Tom Krazit. *Google finding its voice*. URL: <https://www.cnet.com/news/google-finding-its-voice/> (cit. on p. 2).
- [5] W. D Epstein J; Klinkenberg. «From Eliza to Internet: a brief history of computerized assessment». In: *Computers in Human Behavior* (2001) (cit. on p. 2).
- [6] L. Swartz. «Why people hate the paperclip». In: Labels appearance behavior and social responses to user interface agents (2003) (cit. on p. 2).
- [7] Yu Tiecheng. «The current development of speech recognition». In: *Communication World* (2005) (cit. on p. 2).
- [8] M. McGregor M. Porcheron J. E. Fischer. «Talking with conversational agents in collaborative action». In: *Companion of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing* (2017) (cit. on p. 2).
- [9] Gamal Bohouta Veton Képuska. «Next-generation of virtual personal assistants (Microsoft Cortana, Apple Siri, Amazon Alexa and Google Home)». In: *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)* (2017) (cit. on p. 3).
- [10] A. Deotale L. Manikonda and S. Kambhampati. «What's up with Privacy?» In: *User Preferences and Privacy Concerns in Intelligent Personal Assistants* (2017) (cit. on p. 3).

- [11] S. Payr. «Virtual butlers and real people: styles and practices in longterm use of a companion». In: *Your Virtual Butler*. 2013 (cit. on p. 3).
- [12] E. Luger and A. Sellen. «The Gulf between User Expectation and Experience of Conversational Agents». In: Proc. of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16) (2016) (cit. on p. 3).
- [13] G. Lugano. «Virtual assistants and self-driving cars». In: (2017) (cit. on pp. 3, 6).
- [14] T. A. Dingus V. L. Neale. «An overview of the 100-car naturalistic study findings». In: (2005) (cit. on p. 4).
- [15] Sodhi. «Glance analysis of driver eye movements to evaluate distraction». In: Behavior Research Methods, Instruments, & Computers (2002) (cit. on p. 4).
- [16] Liang. «Nonintrusive Detection of Driver Cognitive Distraction in Real Time Using Bayesian Networks». In: Transportation Research Record (1988) (cit. on p. 4).
- [17] D. Shirley; A. Greenwood; C. Böttcher; Zhenwei Cao. «Driver Distraction Test Rig for HMI studies». In: 2009 IEEE International Conference on Systems, Man and Cybernetics (2009) (cit. on p. 4).
- [18] S.S Awad. «Voice technology in the instrumentation of the automobile». In: IEEE Transactions on Instrumentation and Measurement (1988) (cit. on p. 5).
- [19] SBD. «European Telematics & ITS: HMI trends in Europe - balancing functionality with safety». In: SBD Report SBD/TEL/950 (2006) (cit. on p. 5).
- [20] J. L. Harbluk T. A. Ranney and Noy Y. I. «Effects of voice technology on test track driving performance: Implications for driver distraction». In: Human Factors (2005) (cit. on p. 5).
- [21] N. Yoshitsugu K. Itoh Y. Miki. «Evaluation of a voice-Activated system using a driving simulator». In: SAE paper 2004-01-0232 (2004) (cit. on p. 5).
- [22] P. Zheng; M. McDonald; C. Pickering. «Effects of Intuitive Voice Interfaces on Driving and In-vehicle Task Performance». In: 2008 11th International IEEE Conference on Intelligent Transportation Systems (2008) (cit. on p. 5).
- [23] A. Walker. *Looking into the Future of Voice Services in the Car*. URL: <https://developer.amazon.com/en-US/blogs/alexa/alexa-auto/2019/05/looking-into-the-future-of-voice-services-in-the-car> (cit. on p. 7).
- [24] *Speech Recognition*. URL: <https://www.ibm.com/cloud/learn/speech-recognition> (cit. on pp. 8, 10).

- [25] Fatih A.Unal. «Neural Networks and Pattern Recognition». In: Proc. of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16) (1998) (cit. on p. 8).
- [26] L. R.Rabiner. «A tutorial on hidden markov models and selected applications in speech recognition». In: 1989 (cit. on pp. 9, 11).
- [27] D. Jurafsky and J. H. Martin. «Speech and language processing». In: 2019 (cit. on p. 10).
- [28] Zhang Qiong Zhang Ping. «Based on HMM and BP neural network for speech recognition». In: 2008 (cit. on p. 11).
- [29] Haoquan Zhao Jianliang Meng Junwei Zhang. «Overview of the Speech Recognition Technology». In: (2012) (cit. on p. 11).
- [30] Y. Goldberg. «Neural Network Methods in Natural Language Processing». In: 2017 (cit. on p. 12).
- [31] *Natural language processing*. URL: <https://www.ibm.com/cloud/learn/natural-language-processing> (cit. on p. 12).
- [32] *What Is Alexa?* URL: <https://developer.amazon.com/en-US/alexa> (cit. on p. 14).
- [33] Dieter Bohn. *AMAZON SAYS 100 MILLION ALEXA DEVICES HAVE BEEN SOLD — WHAT'S NEXT?* URL: <https://www.theverge.com/2019/1/4/18168565/amazon-alexa-devices-how-many-sold-number-100-million-dave-limp> (cit. on p. 14).
- [34] Rohit Prasad. *Alexa at five: looking back, looking forward*. URL: <https://www.amazon.science/blog/alexa-at-five-looking-back-looking-forward> (cit. on p. 15).
- [35] Chenlei Guo. *How we taught Alexa to correct her own defects*. URL: <https://www.amazon.science/blog/how-we-taught-alexa-to-correct-her-own-defects> (cit. on p. 15).
- [36] *What Are Alexa Skills?* URL: <https://developer.amazon.com/en-US/alexa/alexa-skills-kit> (cit. on p. 16).
- [37] *Get Started with the Guide*. URL: <https://developer.amazon.com/en-US/docs/alexa/alexa-design/get-started.html> (cit. on p. 17).
- [38] *Build Skills with the Alexa Skills Kit*. URL: <https://developer.amazon.com/en-US/docs/alexa/ask-overviews/build-skills-with-the-alexa-skills-kit.html> (cit. on p. 17).
- [39] *AWS Lambda*. URL: <https://aws.amazon.com/lambda/> (cit. on p. 23).

- [40] *ASK SDK for Node.js*. URL: <https://developer.amazon.com/en-US/docs/alexa/alexa-skills-kit-sdk-for-nodejs/overview.html> (cit. on p. 24).
- [41] *Choose the Invocation Name for a Custom Skill*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/choose-the-invocation-name-for-a-custom-skill.html> (cit. on p. 25).
- [42] *Create Intents, Utterances, and Slots*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/create-intents-utterances-and-slots.html> (cit. on p. 26).
- [43] *Delegate the Dialog to Alexa*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/delegate-dialog-to-alexa.html> (cit. on p. 31).
- [44] *Request and Response JSON Reference*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/request-and-response-json-reference.html> (cit. on p. 33).
- [45] *Dialog Interface Reference*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/dialog-interface-reference.html> (cit. on p. 42).
- [46] *Location Services for Alexa Skills*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/location-services-for-alexa-skills.html> (cit. on p. 46).
- [47] *Add Personalization to Your Skill*. URL: <https://developer.amazon.com/en-US/docs/alexa/custom-skills/add-personalization-to-your-skill.html> (cit. on p. 47).
- [48] *Alexa Auto Software Development Kit*. URL: <https://developer.amazon.com/en-US/alexa/devices/alexa-built-in/development-resources/auto-sdk> (cit. on p. 52).
- [49] *Auto SDK Architecture and Modules*. URL: <https://github.com/alexa/alexa-auto-sdk> (cit. on p. 53).
- [50] *TemplateRuntime 1.2*. URL: <https://developer.amazon.com/en-US/docs/alexa/alexa-voice-service/templateruntime.html#rendertemplate> (cit. on p. 54).
- [51] *Handling Display Card Templates*. URL: <https://github.com/alexa/alexa-auto-sdk/blob/2.3/platforms/android/modules/alexa/README.md#handling-display-card-templates> (cit. on p. 54).
- [52] *Mobile Operating System Market Share Worldwide*. URL: <https://gs.statcounter.com/os-market-share/mobile/worldwide> (cit. on p. 56).

- [53] *Platform architecture*. URL: <https://developer.android.com/guide/platform> (cit. on p. 56).
- [54] *Android Automotive OS*. URL: <https://developers.google.com/cars/design/automotive-os> (cit. on p. 59).
- [55] *Android Automotive OS architecture*. URL: <https://source.android.com/devices/automotive> (cit. on p. 59).
- [56] C. Aliferi. *Android programming cookbook*. 2016 (cit. on p. 60).
- [57] *Application Fundamentals*. URL: <https://developer.android.com/guide/components/fundamentals> (cit. on p. 60).
- [58] *Activity Lifecycle*. URL: <https://developer.android.com/guide/components/%20activities/activity-lifecycle.html> (cit. on p. 61).
- [59] *Layouts*. URL: <https://developer.android.com/guide/topics/ui/declaring-layout> (cit. on p. 63).
- [60] *Fragments*. URL: <https://developer.android.com/guide/components/fragments> (cit. on p. 64).
- [61] *DynamoDB overview*. URL: <https://aws.amazon.com/dynamodb> (cit. on p. 82).
- [62] Lauren Schaefer. *NoSQL Explained*. URL: <https://www.mongodb.com/nosql-explained> (cit. on p. 84).
- [63] URL: <https://aws.amazon.com/nosql> (cit. on p. 84).
- [64] AWS Amplify. URL: <https://aws.amazon.com/amplify> (cit. on p. 84).
- [65] AWS Amplify SDK for Android. URL: <https://docs.amplify.aws/lib/q/platform/android> (cit. on p. 84).
- [66] The GraphQL language. URL: <https://graphql.org/> (cit. on p. 85).
- [67] Ed Lima. *New features that will enhance your Real-Time experience on AWS AppSync*. Nov. 2019. URL: <https://aws.amazon.com/blogs/mobile/appsync-realtime> (cit. on p. 87).
- [68] Amplify CLI. URL: <https://docs.amplify.aws/cli> (cit. on p. 89).
- [69] *Android Debug Bridge (adb)*. URL: <https://developer.android.com/studio/command-line/adb> (cit. on p. 97).