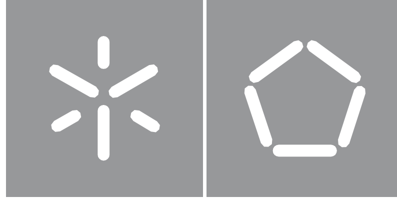
Universidade do Minho
Escola de Engenharia

Rita de Moura Machado Coelho Canavarro

Development of a process for the creation of cross-platform voice applications for Amazon Alexa and Google Assistant

December 2019



Universidade do Minho
Escola de Engenharia

Rita de Moura Machado Coelho Canavarro

Development of a process for the creation of cross-platform voice applications for Amazon Alexa and Google Assistant

Master dissertation
Master Degree in Computer Science
Software Enginner

Dissertation supervised by
Professor Doutor António Nestor Ribeiro

December 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Atribuição-NãoComercial
CC BY-NC
https://creativecommons.org/licenses/by-nc/4.0/

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Os dispositivos conectados pertencem à área de ”Ambient intelligence” (AmI) e são dispositivos inteligentes que podem fornecer diversos serviços ou através de comandos por voz ou de forma autónoma. Estes dispositivos conseguem ser autónomos, devido ao facto de conseguirem capturar informação do ambiente através dos seus sensores e depois processá-la, de modo a que consigam ativar a ação necessária (”Context-aware Computing”). Os assistentes digitais também pertencem à área de AmI e são programas de software baseados em ”Natural User Interfaces”, o que significa que estes funcionam com recurso a comandos por voz para efetuar uma determinada ação [46]. Os assistentes podem estar presentes em dispositivos conectados e foram desenvolvidos para ajudar as pessoas nas suas tarefas diárias.

Devido ao aumento no uso de assistentes digitais, surgiu a necessidade de atender às exigências de uma gama mais ampla de utilizadores, dado que as funcionalidades básicas, para as quais os assistentes haviam sido programados, já não eram suficientes. Esta necessidade levou a uma nova abordagem em relação à expansão das funcionalidades dos assistentes digitais, que consistiu na criação de aplicações por voz.

As aplicações por voz ainda são relativamente recentes e como tal ainda não existem muitas ferramentas, padrões arquiteturais que tenham sido estabelecidos ou uma metodologia ”standard” que possa ser usada no processo de desenvolvimento. Este problema é ainda maior se abordarmos as aplicações por voz ”cross-platform”, dado que hoje em dia existe uma abundância de diferentes assistentes digitais integrados. A inexistência de uma metodologia ”standard” significa que os programadores irão acabar por usar a(s) metodologia(s) que lhes pareçam as mais adequadas tendo em conta o seu objetivo de obter um produto estável.

A falta de standardização e de suporte ao desenvolvimento ”cross-platform” de aplicações por voz é a motivação desta dissertação de mestrado. O objetivo desta dissertação é o desenvolvimento de um processo de construção independente de plataforma, que irá promover a criação de aplicações por voz ”cross-platform” e a automatização do mesmo. Este processo vai estar disponível atráves de uma plataforma, com um editor visual incorporado, que irá permitir a criação de um template de modelo de linguagem que mais tarde irá ser usado para gerar modelos específicos a uma plataforma de modo a que se possa definir o ”frontend” e código ”boilerplate” para o desenvolvimento inicial da funcionalidade do ”backend”. Ao usar esta plataforma, os programadores irão ser capazes de criar e fazer o ”deploy” de aplicações por voz para a Amazon Alexa e para o Google Assistant a partir de uma única fonte de informação, apesar das diferenças que existem entre os seus modelos aplicacionais e, mais importante, recorrendo principalmente aos requisitos pretendidos e não somente aos aspectos tecnológicos.

Palavras-chave: Aplicações por voz, Assistentes digitais, Dispositivos conectados, Engenharia de Software

ABSTRACT

Connected devices belong in the Ambient intelligence (AmI) area, and are intelligent devices that can provide various services through voice commands or autonomously. These devices can be autonomous due to the fact that they can gather information from the environment through their sensors, and then process it in order to trigger the necessary action (Context-aware Computing). Digital assistants also belong to the AmI area and are software programs based on Natural User Interfaces, which means that they work via voice commands to perform a certain action [46]. The assistants can be present in connected devices and were developed in order to help people on their daily tasks.

Due to the growth in the usage of digital assistants there was a need to cater to a wider range of users and their necessities given that the basic functionalities, that the assistants had been programmed to, weren't enough. This need led to a new approach regarding the expansion of the digital assistants functionalities, which consisted in the creation of voice applications.

Voice applications are still relatively new and as such there are still not that many tools, established architectural patterns or even a standard methodology that can be used in the development process. This problem is even bigger if we address cross-platform voice applications, given there is nowadays a plethora of different vendors of integrated digital assistants. The lack of a standardized methodology means that the developers will end up using the methodology(ies) that seems the most adequate concerning the purpose of obtaining a stable product.

This lack of standardization and support to the cross-platform development of voice applications is the motivation for this master's dissertation. The goal of this dissertation is the development of a platform-independent construction process that promotes the creation of cross-platform voice applications and its automatization. This process will be made available via a platform, with an incorporated visual editor, that allows the creation of a language model template that will later be used to generate platform-specific models to define the frontend and boilerplate code for the initial development of the backend functionality. By using this platform, the developers will be able to create and deploy voice applications for Amazon Alexa and Google Assistant from a single source of information despite the differences in their application models and, most important, resorting primarily to the intended requirements and not only to technological aspects.

Keywords: Connected devices, Digital assistants, Software engineering, Voice applications

CONTENT

1 Introduction 1
1.1 Context 2
1.1.1 Interactive voice response 4
1.1.2 Automatic Speech Recognition 5
1.1.3 Speech synthesis 5
1.1.4 Natural Language Processing 6
1.2 Motivation 7
1.3 Goals 9
1.4 Implementation challenges 10
1.5 Dissertation outline 11
2 State of the art 12
2.1 Introduction 12
2.2 Digital assistants 12
2.2.1 Samsung Bixby 12
2.2.2 Microsoft Cortana 13
2.2.3 Apple Siri 15
2.2.4 Google Assistant 16
2.2.5 Amazon Alexa 17
2.3 Comparison study regarding the digital assistants 18
2.4 Frameworks and Platforms 20
2.4.1 Frameworks 22
2.4.2 Platforms 23
2.4.3 Comparison between Jovo and Violet 25
2.4.4 BotTalk support in the construction process 30
2.5 Voice design patterns 31
2.5.1 Guidelines and Design patterns 32
2.6 Voice applications development phases 35
2.6.1 Specification 36
2.6.2 Development 37
2.6.3 Programming models 41
2.7 Summary 41
3 Problems statement and challenges 43
3.1 Introduction 43
3.2 Problem statement 43

3.3 Challenges 45
3.4 Summary 45
4 Development 47
4.1 Introduction 47
4.2 Approach and decisions 47
4.2.1 Definition of the conversation model 49
4.2.2 Backend development 51
4.2.3 Specification of the platform-independent construction process 53
4.3 Implementation 56
4.3.1 High-level activity diagrams 57
4.3.2 Language model template 64
4.3.3 Platform-specific language model generators 69
4.3.4 Platform-specific boilerplate code generators 78
4.3.5 Deployment 82
4.3.6 VoicePrint 88
4.4 Summary 91
5 Case studies 92
5.1 Introduction 92
5.2 The voice applications market 93
5.3 News voice application - The Informed 95
5.4 Shopping voice application - Bag it 103
5.5 Results 110
5.6 Unit testing 112
5.7 Summary 113
6 Conclusion 115

a Support material 125
a.1 VoicePrint requirements 125
a.1.1 Functional requirements 125
a.1.2 Nonfunctional requirements 127
a.2 Requirements of the trivia application 128
a.2.1 Functional requirements 128
a.2.2 NonFunctional requirements 129
a.3 Requirements of the shopping voice application 129
a.3.1 Functional requirements 129
a.3.2 Seller 130
a.3.3 Shopping cart 130
a.3.4 NonFunctional requirements 131
a.4 Requirements of the news voice application 132

a.4.1 Functional requirements 132
a.4.2 NonFunctional requirements 132
a.5 Personas for the news voice application 133
a.5.1 Main persona 133
a.5.2 Short description 134
a.5.3 Users personas 134
a.6 Personas for the shopping voice application 135
a.6.1 Main persona 135
a.6.2 Users personas 136
a.7 Apache velocity templates 137
a.7.1 Alexa Velocity template 137
a.7.2 Google Assistant Velocity template 137
a.7.3 Jovo Velocity templates 138
a.7.4 BotTalk Velocity template 139
a.8 Built-In Intents 139
a.9 Built-In Input Types 141
a.10 Execution times 142
a.11 Case studies - Examples 143

LIST OF ACRONYMS

AmI Ambient Intelligence. iv, v

API Application programming interface. 8, 13, 22, 24, 26, 28–30, 38, 89, 90, 95, 99, 100, 103, 104, 110

ASR Automatic speech recognition. 4–6, 17, 33, 37, 39

AVS Alexa voice services. 5, 6, 17

AWS Amazon web services. x, 17, 28, 30, 39, 44, 82–87, 89, 101, 102, 109–111, 143

CALO Cognitive Assistant that learns and organizes. 15

CSS Cascading Style Sheets. 89

DARPA Defense Advanced Research Projects Agency. 1, 15

FXML FX Markup Language. 89, 90

GUI Graphical user interface. 7

HCI Human-Computer Interaction. 6

IoT Internet of Things. 2, 3, 7

IVR Interactive voice response. 1, 4, 5, 50

MVC Model-View-Controller. 89

NLP Natural language processing. 4, 6, 33

NLU Natural language understanding. 4–6, 17, 18

SDK Software development kit. 1–3, 7, 8, 12–16, 18, 20, 21, 36, 37, 39, 41, 55, 84

STM Short term memory. 34, 102, 109, 110

TTS Text-to-Speech. 4, 5

UI User Interface. 88, 89, 91, 117

UML Unified Model language. 44, 47, 48, 50, 51, 57, 64, 111

VUI Voice user interface. 4, 7, 18, 19, 23, 24, 26, 31–38, 50

YAML YAML Ain't Markup Language. xiii, 24, 55, 56, 69, 70, 76, 77, 81

LIST OF FIGURES

Fig. 1 Growth of connected devices between 2014 and 2017 [56] 4
Fig. 2 Jovo versus Violet 25
Fig. 3 Specification of a voice application [18] 36
Fig. 4 Flow of a voice application [50] 37
Fig. 5 Components of the language model 48
Fig. 6 Components of the backend of a voice application 51
Fig. 7 Components of the language model template 54
Fig. 8 Main conversation flow of e-commerce application 58
Fig. 9 Search product Intent flow 59
Fig. 10 Add product cart Intent flow 60
Fig. 11 Ask for next result Intent flow 61
Fig. 12 Main conversation flow of news application 61
Fig. 13 News Intent flow 63
Fig. 14 Components of the generic language model template 65
Fig. 15 Google Assistant directory hierarchy 72
Fig. 16 Deployment process for Amazon Alexa 83
Fig. 17 Deployment process for Google Assistant 84
Fig. 18 Deployment process with Jovo 85
Fig. 19 Deployment process for AWS Lambda 87
Fig. 20 VoicePrint Layout 88
Fig. 21 Alexa App store - Volume of Applications 93
Fig. 22 Google Actions App store - Volume of Applications 94
Fig. 23 VoicePrint - MainPage 96
Fig. 24 VoicePrint - Create custom template 96
Fig. 25 VoicePrint - Custom template main page 97
Fig. 26 VoicePrint - Create Intent 98
Fig. 27 VoicePrint - Built-in Intent 98
Fig. 28 VoicePrint - News Intent 99
Fig. 29 VoicePrint - Built-in Input type 100
Fig. 30 VoicePrint - Custom Input Type 100
Fig. 31 VoicePrint - Deploy pt.I 101
Fig. 32 VoicePrint - Deploy pt.II 101

Fig. 33	VoicePrint - Search product	105
Fig. 34	VoicePrint - Ask for another result	107
Fig. 35	VoicePrint - Position Input type	107
Fig. 36	VoicePrint - Add product to cart	108
Fig. 37	VoicePrint - Specify quantity of product to add to the cart	108
Fig. 38	VoicePrint - Bag it deploy	109
Fig. 39	The Informed - Alexa developer console test	144
Fig. 40	The Informed - Actions console test	145
Fig. 41	Bag it - Alexa developer console test	146
Fig. 42	Bag it - Actions console test	147

LIST OF TABLES

Table 1 Pros and Cons of using each digital assistant 21
Table 2 Documentation 26
Table 3 Development of the language model 27
Table 4 Development of the business logic - hosting and database 28
Table 5 Development of the business logic - Coding 29
Table 6 Testing phase 30
Table 7 Design patterns for voice applications defined in [71] 34
Table 8 Design patterns for voice applications defined in [54] 35
Table 9 Development of the Frontend of voice applications 40
Table 10 Development of the Backend of voice applications 40
Table 11 Built-in Intents of Amazon Alexa 140
Table 12 Built-in Intents of Google Assistant 140
Table 13 Built-in Input Types of BotTalk 141
Table 14 Built-in Input Types of Amazon Alexa 141
Table 15 Built-in Input Types of Google Assistant 142
Table 16 Execution times 143

LIST OF LISTINGS

4.1 Default language model template . . . 65
4.2 Custom language model template . . . 66
4.3 Alexa - Checking for the existence of platform-specific functionalities example . . . 70
4.4 Alexa - Checking for the use of built-in Intents example . . . 71
4.5 Alexa - Checking for the use of built-in Input Types example . . . 71
4.6 Assistant - Checking for the existence of platform-specific functionalities example . . . 73
4.7 Assistant - Checking for the use of built-in Intents example . . . 73
4.8 Assistant - Checking for the use of built-in Input Types example . . . 73
4.9 Jovo - Checking for the existence of platform-specific functionalities example . . . 75
4.10 Jovo - Checking for the use of built-in Intents example . . . 75
4.11 Jovo - Checking for the use of built-in Input Types example . . . 75
4.12 BotTalk - Apache velocity template for the YAML Intent file . . . 77
4.13 BotTalk - Apache velocity template for the YAML Input file . . . 77
4.14 BotTalk - Platform-specific Intents . . . 77
4.15 BotTalk - Built-In Intents . . . 78
4.16 BotTalk - Built-in Input Types . . . 78
4.17 Boilerplate code generation . . . 79
4.18 Alexa - Steps for deployment . . . 83
4.19 Google Assistant - Steps for deployment . . . 85
4.20 Jovo - Steps for deployment . . . 86
4.21 Lambda - Steps of the deployment of the boilerplate code . . . 87
4.22 Example of passing information between controllers . . . 90

A.1 Alexa boilerplate code template example . . . 137
A.2 Google Assistant boilerplate code template example . . . 137
A.3 Jovo configuration template example . . . 138
A.4 Jovo boilerplate code template example . . . 138
A.5 BotTalk boilerplate code template example . . . 139
A.6 BotTalk test file template . . . 139

1

INTRODUCTION

Despite the fact that only in recent times connected devices controlled by voice, that are integrated with a digital assistant, such as the Amazon Echo or the Google Home started to be more used by people all over the world [62] [49], this type of technology is not recent. An example of it is that it only takes us a call to the call center of a cable television company or to a bank for us to be assisted by a system of interactive voice response (IVR). Another examples of this technology can also be seen in old pop-culture movies or tv shows like The Jetsons, Star Trek or even Knight Rider [4].

The efforts for the development of devices that can understand what has been said and with that obtain data, that can or cannot be used to perform a certain action, dates back to the early 19th century. The phonograph, invented by Thomas Edison, was the first appliance to be able to record and reproduce sound [63]. The first computer to be able to perform voice recognition and accomplish something with that data was the IBM Shoebox, that could recognize and understand 16 words (the digits 0 through 9, mathematical operations such as addition, subtraction, among others and the request of the total of the operation) [55].

A few years later in 1971, the Defense Advanced Research Projects Agency (DARPA) founded an investigation project in the voice recognition area, that led to the development of Harpy, by Carnegie Mellon. Harpy was a machine that could recognize 1011 words [67]. Already in 2003, DARPA started to develop a project named CALO (Cognitive assistant that learns and organizes), that resulted in a digital assistant that could learn while it interacting with its users. This project allowed, a few years later, the development of the digital assistant Siri by the company Siri Inc. [55], that was later bought by Apple.

After the appearance of Siri in 2011, other digital assistants started to be developed and released to the market or improved, such as Siri, do to the progress on the fields of voice recognition and machine learning [59]. Such advances resulted in the development of digital assistants that work only through voice commands, which in turn allows for a more natural interaction with the user.

Smart personal assistants (SPAs) or digital assistants are defined as systems that use ”input such as the user’s voice [...] and contextual information to provide assistance by answering questions in natural language, making recommendations and performing actions” [10]. Digital assistants have as a main objective to help people on their daily tasks such as setting up alarms or sending a text message [38] or through voice commands or autonomously due to the gathering and processing of a particular type of data.

In order to extend their functionalities, making them more appealing to the consumer market, companies such as Amazon Alexa and Google Assistant [48] decided to provide SDKs so that developers could develop

voice applications. Voice applications can act as an interface for other programs, that already have a mobile/web counterpart, through voice commands [38] or be independent applications that exist only for voice-first platforms. Furthermore, along with the SDKs, there were also some frameworks developed by third-parties that appeared in the market such as Jovo [51] and Voxa[1].

Nowadays, what is happening in the field of voice applications is similar to what happened when mobile applications first appeared. When developers try to develop a voice application they feel tempted to apply, for instance, the methodologies that they use when developing web/mobile applications, which might not be a good idea due to the transient and invisible nature of voice applications.

In this field, there are already some development tools and methodologies available yet a standard methodology, that defines a set of rules and the models that can be developed to specify certain components of the application such as the conversation model, still hasn't been defined. The focus on platform-specific tools/methodologies and the absence of a standard methodology makes the development of cross-platform voice applications more complex and time-consuming and might make the developer more focused on the different technological aspects rather than on the application requirements.

Therefore, the main goal of this master's dissertation consists in the development of a construction process of cross-platform voice applications, where from a generic specification it should be possible to generate an application and deploy it to various systems. This construction process will be the basis for the development of a platform, with a visual editor, that will promote the platform-independent development of cross-platform voice applications and its automatization for the digital assistants, Amazon Alexa and Google Assistant, given that the differences on their application models don't prevent the use of the proposed development process.

1.1 Context

Ambient intelligence is one of the pillars of the 4.0 industrial revolution and as it progresses together with Internet of Things (IoT), and the area of machine learning [89], digital assistants, that make use of the tools that this areas offer, will have more capacities both in terms of task realization and in terms of autonomous gathering and processing of data. Digital assistants were developed with the purpose of being integrated with connected devices, that can either be, for example, smartphones [24] or devices such as the Amazon Echo or the Google Home.

The first version of digital assistants had as a main objective to help people on their daily life by helping them perform simple and routine tasks such as, for instance, setting up alarms, send a text message [38] or ask for the weather forecast [59]. Those tasks could either be performed through voice commands (hands-free) while the user did another (related or not) task or they could be performed autonomously due or to their pre-planning by the user or because the connected device gathered and processed data that activated the execution of said task(s), eliminating the user's need to use the device manually [57].

The most valuable characteristic of digital assistants is their efficiency and pleasantness, as they can help the users perform everyday tasks [19] or tasks that they don't feel comfortable doing [55], in an

1 http://voxa.ai/

easy to use way as the users only have to verbally express what they want to do as if they were talking with another person [43]. Such interaction between the user and the digital assistant is possible because the later uses technology based in Artificial Intelligence, Cloud computing (speech recognition, natural language processing, among others) [15] and bidirectional voice communication.

The digital assistants that are proposed as the research target for this master's dissertation are Amazon Alexa and Google Assistant. These two assistants are software agents activated by voice, that are constantly trying to detect if the keyword that activates them was spoken or not. If the keyword is detected, the assistant will start to record the user's voice and when its finished it will send the user's request to a specialized server. This server will process and interpret the request and depending on the type of command, it will provide the assistant with the appropriate response, that will be spoken back to the user [38].

Regarding their functionalities, even though these assistants offer unique characteristics, that differentiates them, they also share some similarities by having both the capability of performing various basic tasks such as controlling ubiquitous devices (IoT) like thermostats and other aforementioned tasks. Furthermore, these digital assistants can have their functionalities expanded by the development of voice applications. Voice applications can be developed by any third-party developer and can either act as an interface for other programs via voice commands [38] or be applications that are only for voice-first platforms.

Voice applications can be developed for many categories (e.g education, family, lifestyle, etc) similar as mobile applications, unless there is a conceptual or functional hindrance. These hindrances could be, for instance, that the SDK doesn't provide support for the development of the functionalities of the application or the concept simply isn't viable in a voice-only interaction. It should also be mentioned as a curiosity that in terms of application categories, there are more voice applications in the market for the News, Games and Trivia, Meteorology and Lifestyle categories [84].

Regarding the voice application market, Gartner [77] foresaw that the market growth would be of 2 billion dollars until 2020, comparing it to the 360 million dollars growth that was reached in 2015, a year after the appearance of Amazon Echo in the market in 2014. VoiceLabs wrote a report in 2017 [84] concerning what they labelled the voice-oriented ecosystem, where they foresee and discuss the growth of connected devices, digital assistants and voice applications. VoiceLabs foresaw that in 2017 24.5 millions of connected devices, that are solely dedicated to voice interactions, would be sold, which would lead to a total of 33 million of voice-only devices in circulation.

In relation to voice applications, VoiceLabs concluded that it would be necessary for the companies to offer a good quality open-source ecosystem to attract developers to this field, something that at the time only Amazon offered given that Google would only make available its SDK in April 2017. The decision of offering tools to the developers made Amazon have more than 7000 voice applications in its app store and a growth of 500% on the second half of 2016 [84].

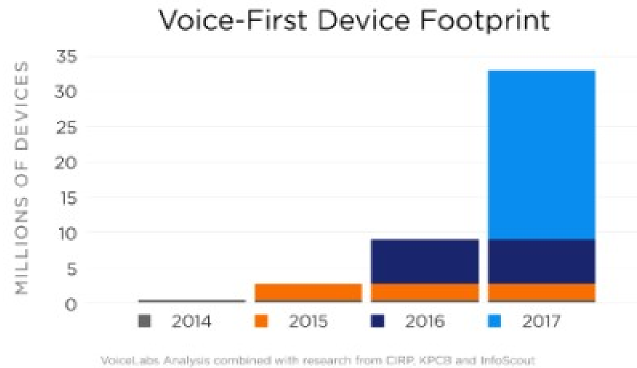


Fig. 1.: Growth of connected devices between 2014 and 2017 [56]

Concerning the development of voice applications, each digital assistant uses and provides a myriad of different tools to the developers such as, for instance, tools for text-to-speech (TTS), automatic speech recognition (ASR), natural language processing (NLP) and natural language understanding (NLU). Typically, the digital assistants obfuscate the definition of the aforementioned tools from the developer, which allows them to work in a higher abstraction level. This translates, for instance, on the developers not having the need to worry about developing, teaching and training neural networks to fulfill the processes of ASR and NLU and in retraining them again when the functionalities of the application are modified/expanded. It should also be mentioned that Google, opposed to Amazon, allows the developers to use other NLU platform without it being the provided one (DialogFlow) [4].

Lastly, there are already some frameworks, open-source and proprietary, in the market that provide tools for the development of voice applications. However, there aren't that many frameworks that provide support for cross-platform development or that are focused both in the development of the frontend, the voice user interface (VUI), and of the backend. The lack of full-stack development support for voice applications is sometimes due to the fact that the frameworks prefer to abstract the development of the backend by giving the developer ways to create the business logic in a very high-level manner at the same time they are developing the VUI.

1.1.1 Interactive voice response

The phone-tree or interactive voice response dialogues can be thought as the ancestors of voice applications do to the fact that they allowed the user to utilize a system of automatic information by voice (interactive voice system). This type of systems were and still are more used in call-centers of various types such as of banks, TV companies, among others [65], where through a phone call the user is informed of the options he can choose by the means of a fixed voice menu. When the user chooses the option that he sees fit he can either pass to another voice menu that is more specific or be redirected to a real assistant. Therefore, the IVR systems function by presenting a fixed menu by voice, in real time, to the customer [4], who can answer, after the end of the presentation, or by pressing a certain button on its phone or by speaking a certain word or short sentence.

Nowadays, voice applications don't work nor should they work like an IVR system, since those are known for being voice systems with very rigid rules that don't take into account the needs of the users and the context of the conversation, and that are somewhat painful to use because the user must wait that all the fixed menu is presented before he can select the option that he deems more adequate (taking the chance that there isn't an option that is appropriated for him). However, it should also be mentioned that some of the more modern IVR solutions can already take into account the context of the call and try to offer personalized options to the user.

1.1.2 Automatic Speech Recognition

Automatic speech recognition is a subfield of computational linguistics, that is concerned with the modelling of natural language in a computational perspective. In this field, methodologies and tools for the recognition and translation of the human speech [37] to text are developed in order for the human speech to be understood by computers.

The algorithm that is used for speech recognition varies between digital assistants as, for instance, Amazon uses the algorithm present on the Alexa voice services (AVS), that provides a cloud-based service of ASR and NLU, and Google uses, for instance, the algorithm that is present on Dialogflow, that abstracts all the NLU process from the developer. The complexity of these algorithms varies in terms of the needs of the systems that will use them.

The voice recognition process consists in three distinct phases, being that the first phase consists in the transforming the analog waveform of what the user said into a digital representation. In the second phase, the digital representation is divided in distinct sound units called phonemes and breaks [72]. In the third and last phase, the phonemes are provided as input data to an algorithm, whose fulfillment will result in a text that specifies what the user requested [12].

1.1.3 Speech synthesis

Speech synthesis or text-to-speech (TTS) can be seen as the opposite process of speech recognition, given that it leads to the production of an artificial human speech [37]. This artificial speech will be spoken by a machine, such as a digital assistant, that exists, for instance, in a smartphone or in a connected device.

The transformation process of text to voice consists in two distinct phases, being that the first phase consists in using an algorithm to process the text to phonemes and breaks. In the second and last phase, the sounds that the phonemes should produce when spoken are created and put together in order for the artificial speech to be produced [12].

1.1.4 Natural Language Processing

Natural language processing is a subfield of computer science, computational linguistics and artificial intelligence, that focus in the interaction that computers have with natural language [11], in terms of how computers can analyze what has been said or written by the user [64]. Afterwards, the computers can work with the data, that resulted from the analysis.

It should also be mentioned that NLP is inserted in the area of Human-Computer Interaction (HCI), since it's used to allow computers, connected devices, etc to have the capacity of interpret and derive meaning from the voice commands spoken by the user and to communicate back by using natural language [37], and also read texts and determine which parts of what the user said or wrote are or aren't important.

The NLP algorithms are based in machine learning more specifically in statistical machine learning [11]. However, the statistical machine learning algorithms are not the only existing approach on NLP given that rule-based methods can also be used. The NLP algorithm that is used by the digital assistants varies between them given that, for example, Amazon uses the algorithm that is present in AVS while Google uses the algorithm that is present in Dialogflow.

Lastly, NLP algorithms includes various phases such as the following ones:

- Processing the text into tokens and parsing said tokens;
- Lemmanization, that consists in creating the tokens as they are present in the dictionary;
- Stemming, that consists in converting the tokens in their roots (non-changeable constituent parts);
- Detection of the language that was used;
- Identification of the semantic relationships and the combination of the syntactic sentence structure and semantic relationships [30].

In the end, a text, whose constituent parts were simplified to their most fundamental level, is obtained in order to be able to apply the Natural language understanding process.

Natural Language Understanding

Natural language understanding is a subtopic of Natural language processing [27] that consists in processing the text, obtained by the process of ASR and NLP, to an ontology in order to be able to extract its meaning [42] and the relationships that exist between its constituent parts.

This process occurs after the use of the NLP algorithm and utilizes the context that was gathered by the recognition devices such as the microphone or the device that fulfilled the ASR. The context will be used in order to be able to distinguish the meaning and the relationship between the fragmented parts of the text and also which parts of the text contain the commands that the user spoke.

Lastly, by giving meaning to the commands that the user spoke, the NLU allows the definition of the action, that must be fulfilled, without having had the need to restrict the user to utilize a formal syntax to issue the command [59].

1.2 Motivation

Motivated by the rapid technological advances in the areas of ambient intelligence, IoT and machine learning [82] and by the growth in the usage of digital assistants [49], companies such as Amazon, Google and Apple decided to provide their own software development kits (SDKs) for the development of voice applications. By doing so they will be able to extend the functionalities of their digital assistants and also cater to a wider range of users and their necessities. Furthermore, along with the SDKs, there were also some frameworks, developed by third-parties, that appeared in the market such as Jovo [51] or Voxa[2].

Voice applications are still relatively new and as such there are still not that many tools, established architectural patterns or even a standard methodology that can be used in the development process. This problem is even bigger if we address cross-platform voice applications, given that there is nowadays a plethora of different vendors of integrated digital assistants (e.g Amazon, Google and Apple).

Nowadays, there are already some methodologies that the developers can follow in order to initiate the development of voice applications and also some tools that they can use in it. Typically, the methodologies begin by defining, each in his own way, what is necessary to start the development of an application of this type, in terms of guidelines for the definition of the conversation model, the SDKs that exist, voice design patterns, among other information.

Furthermore, there is also the attempt, akin to what happened when mobile applications first appeared, of trying to apply the development methodologies that are used for web or mobile development when developing voice applications, which might not be a suitable approach. Such is due to the fact that voice applications have a transient and invisible nature and the development of a VUI is very different from a GUI. For instance, in voice interfaces the developer has to be careful with the amount of information he tells the user in each turn of the dialogue in order to not overwhelm its short-term memory [74]. Contrarily, in graphical interfaces the developer can expose any amount of information he deems necessary without worrying about the user's memory, because he will be able to quickly scan the page and find what he needs.

However, despite the existing development tools and methodologies, a standard methodology for the development of voice applications, that defines a set of rules that the developer should follow and the models that can be developed to specify certain components of the application such as the conversation model, still hasn't been developed [48]. Due to the absence of a standard methodology and the existence of a myriad of research papers [62], books [4] [65] and tutorials [6] [31], that explain their own approach in terms of requirements, phases of development, auxiliar models, etc, each developer will end up following the methodology(ies) that seems the most adequate concerning the purpose of obtaining a stable product. The dilemma is that obtaining a stable product doesn't necessarily means the same as obtaining a product whose final state is optimum for both the user and the developer.

The existence of a standard methodology, whose phases were well defined, would allow the developers to not only be focused, during the development phase, on the myriad of technological aspects that differ among assistants and that need to be handled separately but also on the application requirements. Another

2 http://voxa.ai/

set back due to the lack of a standard methodology is that if the developers have to explain the development of the voice application or to their clients or to their peers, with or without experience in the area of voice, they will not have a common “language” to use to share their ideas or developments steps.

Lastly, if a developer decides he wants its application to appear in another digital assistant, there is the possibility that he might have to restructure a great portion of its application. Such can happen due to, for instance, not being able to reuse the components of the business logic because they were not developed to be able to deal with the differences that exist between the assistants application models. This would make the expansion of the application to another assistant time-consuming and perhaps complex depending on the documentation and tools available to help in this process.

Regarding the conversational aspect of voice applications, the definition of the conversational model between the user and the application usually diverges a great deal from methodology to methodology. Such is due to the fact that there are some methodologies that prefer to advocate an approach where the developer starts by creating high-level flow diagrams [34], that expose the application conversational flow and the paths that the user can take in it. These diagrams can be developed using the tool(s) and notation(s) that the developer sees fit. Other methodologies prefer to advocate an approach where the developer doesn’t create diagrams but scripts [65], that expose the possible turn-based dialogues that can happen between the user and the application (sample dialogues).

The conversational aspect of a voice application, particularly if the application is not of the request-response type, is its most important characteristic [37]. If this aspect of the voice application is not well conceived and developed it may condemn the success of the application among the users [43]. Such is due to the fact that users, akin to when they are talking with other persons [73], don’t like to talk with a digital assistant that doesn’t understand the current context of the conversation [14], doesn’t remember information that was previously mentioned, that could make the interaction more pleasant, and that is always asking for the same information, among other aspects that can make the dialogue feel less natural.

Regarding the programming languages, the companies that develop the digital assistants currently provide SDKs for a wide set of languages such as JavaScript (Node.JS), Java, Python, C#, among others. However, the developers can also find and use SDKs, that were developed by third-parties developers, for programming languages that are still not supported by the companies. Due to the fact that there are already many programming languages that have SDK support, the developers can likely find one that they feel more confident in to start developing their voice applications.

The availability of templates for voice applications varies according to the digital assistant that the developer chose to develop for. For instance, Google provides a few excel templates (Google Sheets) [33], that allows the developer to create applications in just a few minutes. However, those template are only focused on the definition of the frontend and they remove from the developer the possibility of having control over the backend of their application. Regarding Amazon, it also provides templates, that are focused in the definition of both the frontend and the backend of the voice application. This means that the developer may only have to provide information about the application such as its name, description and the API endpoint or he may have to develop both the backend and the frontend almost from scratch.

In conclusion, it was decided that streamlining the development of voice applications for different digital assistants will be the motivation for the development of a construction process of cross-platform voice applications, with focus in the assistants Amazon Alexa and Google Assistant. This process will be the basis of a platform that will promote the platform-independent development of cross-platform voice applications and its automatization, without disregarding the differences that exist between the digital assistants application models. Furthermore, by providing the developer with a structured platform-independent development process and tools, he will be able to be more focused on the requirements and fundamental parts of the voice application (e.g functionalities, user experience, etc) rather than on platform-specific details and the development of cross-platform voice application won't be so time consuming and convoluted.

1.3 Goals

The goals that were defined for this master's dissertation are the following:

- Definition of process for cross-platform voice applications development that targets the digital assistants, Amazon Alexa and Google Assistant. In this process, it should be possible to develop and use a generic specification of a voice application to generate and deploy it to various systems.
- Define a generic specification for the voice applications, that the developers will have to use to build their applications. This specification will have to be generic enough to feed a generation and deployment process for different voice-based systems;
- Allow the developer to specify, in the aforementioned voice application specification, an application functionality as platform-specific;
- Provide the developer with built-in functionalities and argument types, provided by the digital assistants, as if they were platform-independent;
- Development of a platform, recurring to the aforementioned construction process, that will promote a platform-independent development of cross-platform voice applications and its automatization. This platform will conceal the generation and deployment process of the applications from the developers by providing an easy to use visual editor. This visual editor will allow the developer to define the specification of the voice application in an iterative way;
- The platform should be practical, answer to the needs of both technical and non-technical users, user-friendly and eliminate the necessity of manually writing the frontend language model specification in JSON;
- Perform a research concerning the digital assistants, that currently exist in the market, in terms of the functionalities that they offer both to the users and the developers and the similarities and differences that exist in their application models;

- Perform a research concerning the existence of frameworks that provide help in the development of voice applications;
- Research and comprehend the current development process of voice applications, that is, how these type of applications are specified and created;
- Develop a tool to help the developers in the definition of the voice application conversation model. Voice applications models are very different from the remaining web and mobile applications models that exist in the market. Such is due to the fact that they have a transient and invisible nature and function through voice commands. These commands will have to be processed in order to the backend service to able to understand them.

Therefore, a study regarding the necessary requirements to develop voice applications and the similarities and differences in the application models of Amazon Alexa and Google Assistant will be conducted so the proposed construction process for these digital assistants can be developed. After the development of the proposed tools, they will be used for the development of two cross-platform voice applications. These two applications will serve as an example of what can be done with the platform and, thus, the construction process.

1.4 Implementation challenges

The development of the proposed construction process of cross-platform voice applications and the platform that will abstract its use, entails some expected challenges. Therefore, throughout this dissertation the following challenges are expected to be found and solved or handled in the best way possible:

- Develop the construction process in a way that makes updating it less complex when faced with updates in this technological area;
- Understand the specification and development process of voice applications for digital assistants, given that each assistant has its own application model and functionalities that they can offer [29] [51]. By understanding the differences and the similarities that exist among the digital assistants application models, the process of gathering the adequate requirements for the proposed construction process of voice applications can be accomplished;
- Define a generic representation for the voice applications, that the developers will have to use to build their applications. This representation will have to be generic enough to feed a generation and deployment process for different voice-based systems;
- Develop a platform-independent voice application generation and development process, that will use the aforementioned representation. This process will alter the representation accordingly to the different structural rules that each digital assistants imposes. This representation will be used by the process to generate both the frontend definition and the boilerplate code for the initial development of the backend functionality of the voice application.

- At the same time promote a platform-independent construction process but still allow the developer to use some platform-specific functionalities, like built-in argument types, and also to be able to mark functionalities as platform-specific.

1.5 Dissertation outline

This dissertation is outlined in six chapters, which describe the steps that were taken towards the development of the proposed platform-independent construction process. In the first chapter the problem that motivated this dissertation was presented and contextualized along side with the goals and challenges that will be accomplished and solved, respectively.

In the second chapter, it's presented a study regarding the state of the art, which was composed by an analysis of the current state of the digital assistants market and a comparison between the existing assistants application models. Additionally, it was also analysed the existing tools and solutions for the development of voice applications, such as, for example, frameworks and voice design patterns.

In the third chapter, the problem that motivated this master's dissertation and which the platform-independent construction process aims to help solving will be presented in a more detailed manner. Furthermore, the possible challenges that are expected to be found and handled during this dissertation will also be presented and analysed.

In the fourth chapter, the relevant decisions and the approach that was taken in order to achieve the proposed goals will be presented. This chapter will present the studies that were conducted in order to define how the problem and challenges presented in the previous chapter will be dealt with. Afterwards, the specification of the construction process and the way the process and the other tools were implemented will be presented in detail to conclude this chapter.

In the fifth chapter, two case studies are presented. These case studies consist in two voice applications being that one of them will use a request-response approach and the other will have a more conversational approach. Additionally, they will allow the verification of the viability of the developed construction process and platform and if the results that they produce are the ones that were intended.

In the fifth chapter, two case studies are presented. These case studies consist in two voice applications being that one of them will use a request-response approach and the other will have a more conversational approach. Additionally, they will allow the verification of the viability of the developed construction process and platform and if the results that they produce are the ones that were intended. In this chapter, it's also presented and discussed the tests that were performed to the code, that constitutes the proposed construction process, in order to verify if it was working as intended.

Lastly, in the sixth and final chapter, a conclusion regarding the work that was developed will be presented along side a few thoughts regarding the possible future work that can be done.

2

STATE OF THE ART

2.1 Introduction

In order to be able to develop a master's dissertation based on well-founded principles and with knowledge regarding the current state of the field of voice applications and digital assistants, a study concerning the state of the art in said fields must be conducted. By analyzing the state of the art, one can acquire knowledge about what was already been done or is currently being done in the research area and also obtain a global vision about the current state of existing methodologies, design patterns and tools.

Therefore, in this chapter the state of the art for voice applications and digital assistants will be presented. In addition, the necessary assumptions to be able to advance to the development of the proposed construction process of cross-platform voice applications will also be made.

2.2 Digital assistants

Nowadays, the most known digital assistants in the market are Alexa from Amazon, Google Assistant from Google, Siri from Apple, Cortana from Microsoft and Bixby from Samsung. Currently, this market is dominated by Amazon, being that its strongest competitor is Google [36], however some market researches have shown that in the upcoming years Amazon will lose it's place at the top for Google [2] [21], being that one of the reasons might be the fact that the former is present, by default, in all Android smartphones and tablets.

In order to develop the proposed construction process, the aforementioned digital assistants should be known, in terms of their characteristics and application models, so the ones the proposed process will target can be chosen. Furthermore, it's also important to assess how much support these digital assistants provide to the developers through their SDKs. Therefore, in this section the results of a study conducted upon the aforementioned digital assistants is going to be presented and analysed.

2.2.1 Samsung Bixby

Bixby is a digital assistant produced by Samsung for its devices such as smartphones, tables and other connected devices fabricated by Samsung. This assistant was officially released in March 20, 2017 [68], which makes Bixby the most recent digital assistant on the market. It should also be mentioned that

this assistant is a reformulation of S Voice, a digital assistant that Samsung launched in 2012 with the Samsung Galaxy 3.

Samsung allows the development of voice applications by third-party developers [47], in order to increase the number of functionalities provided by Bixby. The voice applications will have to be developed using a SDK and an API provided by Samsung, being that these were made available in November 2018. Before that date, there was only a beta program for the development of voice applications for Bixby, available via invitation. Samsung decided to develop these tools in order to be competitive with the rest of the digital assistants [47] present in the market, which already provided tools for the development of voice applications.

Therefore, Bixby provides a developer studio (Bixby Studio), that supplies, in addition to the aforementioned SDK and API, a programming language called Bixby language, which is a declarative language whose structure is similar to JSON. This language is used in conjunction with another API, that contains functionalities in JavaScript for Bixby.

These tools allow the developers to be able to develop their capsules, term used by Samsung to refer to voice applications, in terms of the requests that the capsules should be able to answer and moments. Moments correspond to the responses that Bixby will provide to the user and can be of three types: input, confirmation and result [13]. It should also be mentioned that the definition of the language models, that define the frontend of a voice application, is quite specific to this digital assistant, given that the developer must define a model using the Bixby language.

Lastly, in regard to the base functionalities that Bixby provides to its users, some of them are the following:

- Send text messages or write emails;
- Schedule or reschedule appointments;
- Work with third-party applications in order to improve its responses to the user's requests;
- Open the applications in split-screen mode in the user's smartphone;
- Realign photos;
- Allow the user to make requests that contain two distinct actions as, for example, open the Uber application and give a certain rating to a driver.

2.2.2 Microsoft Cortana

Cortana is a digital assistant produced by Microsoft for its devices such as computers, speakers, Xbox One, among others. Microsoft officially launched this assistant in April 2, 2014 [44]. Due to the set back that this digital assistant suffered in 2018 in comparison with Google Assistant and Amazon Alexa, Microsoft CEO Satya Nadelle admitted earlier in 2019 that Cortana will no longer aim to be a direct competition of the aforementioned two assistants [86].

Microsoft Cortana will have a new vision that entails repositioning it as a skill that can run across multiple platforms [85] (e.g in Alexa or Google Assistant) and especially for Microsoft 365 subscribers. The first step of Microsoft with this new approach towards Cortana seems to be targeting it as a virtual assistant with a great support for conversational interactions for business workers to organize their days. The conversational element of this assistant will be improved by combining various skills and the conversation context to respond to the user's request as Microsoft believes that everything that the user says and does is interconnected.

Regarding the development of voice applications, Microsoft provides the Cortana skills kit so third-party developers are able to develop them. By providing a SDK there are more chances that a more rapid expansion of Cortana's functionalities will occur. The skills, term used by Microsoft to refer to its voice applications, are viewed as bots [28], which ends up being a very different approach from the remaining digital assistants targeted in this study. The other assistants in this study view voice applications as applications and not bots.

The development of skills is made through the use of the Bot Builder SDK and other Azure services, being that Microsoft also allows the developers to repurpose code that they have already developed for voice applications for Amazon Alexa [80]. In terms of programming languages, developers can use C# or JavaScript (Node.JS).

However, these tools aren't totally free of charge unlike what happens in the digital assistants Alexa and Google Assistant. Such is due to the fact that Microsoft only provides a credit, that can be used to acquire the paid services, during 30 days and the services that are actually free of charge or only last a year and then start being pay as you go or can only be used for a certain amount of times.

Lastly, in regard to the base functionalities that are provided to the users, Cortana is well known for its deep integration with Outlook [87], a scheduling and emailing application developed by Microsoft, in comparison with the rest of the assistants, whose integration with Outlook is more shallow in terms of functionalities available.

In addition to the ability of assisting the user with the scheduling, rescheduling or cancelling of its appointments [53] and the ability of sending emails, Cortana also provides the following functionalities:

- Allow the user to make online searches by using the Bing search engine;
- Create notifications that have the ability to analyze the user's context. For instance, if the user creates a notification, that must be triggered when he gets off work, when the user walks away of its workplace Cortana will send him the notification [9];
- Play music or videos;
- Find recently used files.

2.2.3 Apple Siri

Siri is a digital assistant produced by Apple for its operating systems such as IOS, watchOS, macOS [53] and for the devices that operate with said operating systems like the iPhone. Siri was acquired by Apple in 2010, when Apple bought the company Siri Inc., a company that had developed Siri based on the knowledge that was acquired during the CALO project [55]. This project, as it was mentioned in Chapter 1, started being developed in 2003 by DARPA. Although Siri was acquired in 2010, Apple only officially released Siri on its devices on October 14, 2011.

Apple allows the development of shortcuts, that can be viewed as voice applications, since the launch of the iOS 12 and also the extension of mobile application functionalities to Siri through a SDK [1]. Since iOS 13, Apple now provides a shortcuts built-in application [66] that the user can use to check all the shortcuts that he has and also set them up to run automatically.

Creating shortcuts for a mobile application is different from extending it, given that when an application is extended to allow communication with Siri, it can only implement functionalities that are provided by it, like messaging, calling, etc.

The development of shortcuts consists in the creation of custom functionalities for mobile applications. These custom functionalities can be less generic and more focused on what the mobile application wants to accomplish. It should also be mentioned that Apple allows its users to create their own shortcuts for a native or third-party (if there is already support for that) application and that they can personalize the voice commands that will activate them [17].

Regarding the development process, in order to develop shortcuts or extensions of mobile applications for Siri the developers will have to use the Siri Kit SDK, the XCode IDE, Swift or Objective-C and the intents and Intents UI app extension frameworks [20]. Lastly, in regard to the base functionalities that Siri provides to its users, some of them are the following:

- Create notifications and take note of what the user is saying when requested;
- Do online searches;
- Create alarms;
- Make calls or video-calls via FaceTime;
- Write text messages, that the user dictated, and send them;
- Make restaurant reservations;
- Calculate conversions;

1 https://developer.apple.com/siri/

2.2.4 Google Assistant

Google Assistant is a digital assistant produced by Google for the Android operating system [4] and later for IOS, even though in the later operating system the Assistant can not function at full-capacity. Furthermore, this assistant is also present in devices developed by Google such as Google Home, Wear OS by Google, Android TV, among others. This digital assistant was officially released in May 18, 2016 [52].

The Google Assistant is the successor of Google Now, being that inherited all of its research capacities and still added some new functionalities [55] due to Google advances in the area of artificial intelligence. Moreover, the Assistant also improved the interaction with the user by being able to talk with him [53] and at the same time learn the user preferences [55]. Google Assistant will be able to use said information in the future to, for example, filter the results of a certain request.

Google, in order to be competitive in the digital assistants market, also allows the development of voice applications, to which it refers as Actions [60]. Regarding the development tools that were made available, Google opted to follow an open-source route by providing various tools such as the Actions SDK, templates to develop Actions and, lastly, Dialogflow.

Dialogflow [4] allows the development of the conversational flow between the user and the application (the interaction model) and offers tools to deal with the transformation of the user's request to something that the application can understand. Normally, the user's requests are transformed in JSON requests, that contain all the necessary information for its fulfillment. In regard to the programming languages, Actions can be developed using Java, JavaScript (Node.JS), Python, among others. Lastly, in regard to the base functionalities that Google Assistant provides to its users, some of them are the following:

- Play music or start a podcast depending on the user's request;
- Check when a certain movie session starts or when a certain flight takes off;
- Make restaurant reservations or find a coffee shop that is along the user's daily route and that serves what he requested;
- Send text messages, emails or make phone calls;
- Control connected devices such as lamps or thermostats (smart-home devices);

Google I/O 2019

At this year Google I/O conference some new features and changes to Google Assistant were announced to the public. One of the most notable announcements is that the newest version of this digital assistant will process user's requests up to 10 times faster than its older version, because of the use of on-device computation power.

This improvement of latency will happen because Google reduced their artificial intelligence models, that are used for speech recognition and natural language processing, making them small enough to efficiently run directly on the user's device [83]. By having the models run directly on the device there will

be no need to send data to remote servers for processing and the user will be able to quickly perform its tasks.

Another announcement was the "How-to markup language" [40] that allows a developer to tell Google that its voice application responds to an "How to" question (e.g. "How do I tie a Tie?"), for example, by providing the user with a set of instructions accompanied by a video or images.

Lastly, another interesting announcement was "Duplex for the web" [39]. Duplex was first presented to the public in 2018 as a Google Assistant feature that could make reservations by phone for the user but instead of letting him talk with the, for example, hair salon employee, it was Duplex that spoke for the user with an AI-based voice. Given that Duplex sounded almost human, even introducing pause breaks and words like "humm" to sound more human, and could comprehend what the receiver of the call said and correctly answer him back, there were several security issues that emerged such as the receiver of the call not knowing it was talking to a machine, etc.

This year, Google decided to make Duplex a feature for the web, which means that Google Assistant will be able to continue making reservations but instead of making a phone call, Duplex will automatically fill the details needed for the reservation and also pay for it in just a few steps. Additionally, Duplex will prompt the user for confirmation in each step in order for him to be able to change something if need be.

2.2.5 Amazon Alexa

Alexa is a digital assistant produced by Amazon for its connected devices such as Amazon Echo, Fire TV [53], among others. Furthermore, Alexa is also present on the IOS and Android operating systems as a mobile application. This assistant was officially released in November 2014 [25], which makes it one of the oldest digital assistants next to Siri and Cortana.

Amazon was the first company to provide open-source tools to developers in order for them to be able to develop voice applications, which Amazon named Skills. By making this decision, Amazon allowed Alexa to grow rapidly in terms of the number of functionalities available to its users in comparison with the rest of the digital assistants present in the market [84].

Regarding the development tools that were made available, Amazon provides the Alexa skills Kit for the development of skills and the Blueprints tool, that provides skills templates that don't require coding. Moreover, Amazon also provides the AVS, that is a cloud-based service of ASR and NLU, and, lastly, the AWS, that allows the deployment of skills and the use, for example, of DynamoDB. DynamoDB is a NoSQL database that can be used, for example, to store information in between user sessions[4]. In regard to the programming languages that can be used, skills can be developed using Java, JavaScript (Node.JS), C#, among others.

To conclude, in regard to the base functionalities that Alexa provides to its users, some of them are the following:

- Control connected devices such as tv's, lamps, thermostats and alarms;
- Make restaurant reservations or order a meal to be delivered to the user's location;

- Do online shopping via the Amazon website;
- Create and send notifications or create alarms;
- Do online searches using the Bing search engine;
- Learn the user's preferences in terms of, for example, restaurants and music and use them to improve the user experience.

Amazon re:MARS

This year, Amazon hosted a new conference dedicated to Artificial Intelligence named re:MARS (Machine learning, Automation, Robotics and Space) [2], that was inspired on the MARS event. The MARS event was hosted by Amazon founder and CEO Jeff Bezos and was invitation-only. At re:MARS 2019, Amazon announced Alexa conversations, a new deep learning-based approach to the creation of natural dialogues on Alexa [78], that aims to allow the creation of skills with natural and flexible voice experiences and the chaining of multiple skills in a single conversation.

Nowadays, Alexa skills consist of two components: the NLU, to which the developer provides the language model, and the conversation structure and fulfillment logic, that are provided entirely by the developer. With Alexa conversations, the skills will still consist of the two aforementioned components however the developer will not need to provide the conversation structure, given that it will be provided by a recurrent neural network (RNN) [78].

For this RNN to work, the developer will have to provide a handful of dialogues (called Golden Dialogues), that will be used to generate simulated ones. The simulated dialogues will be used to train a RNN, that will model the skill dialog flow. At runtime, the RNN will take into account the session's dialog history and predict the optimal next step in the dialogue, which improves the skill accuracy and reduces the developer design and development efforts.

Lastly, it should also be mentioned that, Alexa conversations will take into account the dialog context in order to proactively predict the user's latent goal from the direction the dialog is taking and proactively recommend additional steps [58]. An example of it can be an user asking to book a flight and after having booked the flight, Alexa will prompt him to also book an hotel, enabling the conversation flow to expand to other topics and, most likely, other skills.

2.3 Comparison study regarding the digital assistants

The aforementioned digital assistants possess similarities regarding some of the provided core functionalities and the specification of the voice user interface (VUI). Bixby and Siri are the exceptions, due to the fact that they have a different way of specifying the language model, that defines the frontend. [70] [20].

However, the assistants also possess differences regarding the technologies and programming languages that can be used, the development capacities that are provided via SDK and the way that the

2 https://remars.amazon.com/

language model components must be structured. A study concerning the application models of each digital assistant, presented in this section, was conducted with the purpose of better comprehend their differences and similarities.

During the development of the VUI, an aspect that is common to all the assistants, is that the developer must specify in the language model, the application functionalities, the most common ways that the user has of requesting said functionalities, and the arguments that the user must provide in order for its request to be fulfilled. However, each digital assistant has its own structural rules and notation language for the specification of the language model.

In regard to the notation language, the language models for Alexa, Assistant and Cortana can be specified using JSON. The language model for Bixby must be specified using the Bixby language. The language model for Siri must be specified by creating a Intent File definition for each functionality. Regarding the components of the language model, each assistant has its own designation for them, that may or may not be the same as that of another assistant(s). For instance, the component sample invocation phrases can have the designation utterances, that is used by Alexa, Cortana and Bixby, or the designation user phrases, that is used by Google Assistant.

Concerning the components that each assistant needs in their language model, it can be stated that there are some similarities and differences among them. Alexa, Google Assistant and Cortana all use the same two types of components minus the different designations in some of them:

- A component to define the functionalities, where the developer will also define the various ways a functionality can be invoked (sample phrases) and the arguments that are needed for it to be executed;
- A component that is used to define the type of the arguments, that were defined in the previous component.

The components of the Bixby language model consist in:

- Concepts, that are used to define what Bixby must know during runtime;
- Actions, that define what can effectively be done in the application;
- Sample phrases (utterances) constituted by goals, that can either be concepts or actions;
- Values, that the user must provide in order for its request to be fulfilled.

The Siri language model consists in two components:

- One that will allow the definition of the application functionalities. In this component, the developer will have to define various informations such as the functionality category, arguments, arguments types and if it will have shortcuts or not and if so which ones;
- One that will allow the definition of the functionalities vocabulary. In this component, the developer will have to provide the name of the functionality and an array of sample phrases, that the user can say to invoke it.

Another difference regarding the language model is the way that it must be specified. In Alexa, the developer must create a single JSON file to specify the whole language model. In Google Assistant, the developer must define the model by creating two JSON files to specify each functionality and two other JSON files to specify the type of the arguments that the user must provide for his request to be fulfilled. In Cortana, the developer must define the model as a LUIS (Language Understanding Intelligent Service) bot [4].

In Siri, the developer must use the XCode editor to create a definition file for each functionality, which translates in having the language model separated in many files as the number of functionalities [20]. In Bixby, the developer must use the Bixby Studio to define examples of invocation, which they denominated Goals, for each functionality that is going to be developed [70].

Regarding the programming languages that can be used in the development phase, the digital assistants Alexa and Assistant are the ones that offer more SDKs for a larger number of languages such as JavaScript (Node.JS), Java, Python, among others. Cortana offers two SDKs, one for JavaScript (Node.JS) and another one for C# (.NET).

The assistants Bixby and Siri provide an editor that the developers must use, thus restricting the programming languages that can be used. If the developers want to develop a voice application for Bixby they will have to use the Bixby language, developed by Samsung, and JavaScript. In case they prefer to develop a voice application for Siri they will have to use either Swift, developed by Apple, or Objective-C.

The study concerning the digital assistants allowed for a more profound comprehension of the base functionalities that they offer, if they provide tools for the development of voice applications and the similarities and differences that exist between their application models. The pros and cons of using each digital assistant are presented in Table 1. Furthermore, this study also helped validate the choice of Amazon Alexa and Google Assistant as the target digital assistants for which the proposed construction process is going to be developed.

2.4 Frameworks and Platforms

The expansion of voice-based technologies is still relatively recent [82] and there are only a limited amount of tools for the development of voice applications for digital assistants, that aren't provided by the companies that developed them. Normally, these tools or aren't totally free of charge or are only focused on a single digital assistant, which doesn't allow a developer, to develop a cross-platform voice application from a specific idea and a generic description.

In this subsection, some of the frameworks and platforms that currently exist on the market will be presented. The focus will be in both single and cross-platform frameworks in order to be able to assess what type of tools are available to help in the development of voice applications.

To conclude, a comparison about the frameworks, that are going to be supported by the proposed construction process, will be presented. The explanation about why it was decided to also offer support for BotTalk in the construction process will also be presented.

Table 1.: Pros and Cons of using each digital assistant

<table>
<tr><th></th><th>Pros</th><th>Cons</th></tr>
<tr><td>Bixby</td><td>It might gain more followers in the developer community over time due to the development environment that it offers and because Samsung is planning to integrate Bixby with a vast array of its products.</td><td>Restricts the development of voice applications to its own tools. Doesn't provide SDKs for other programming languages aside its own. Defined an application model that is very distinct from the ones that the other digital assistants use.</td></tr>
<tr><td>Cortana</td><td>Allows the use of code developed for Alexa skills. Shares a large number of similarities with Alexa and Assistant.</td><td>Defines its voice applications as bots. It's more restrictive in its use free of charges. Microsoft recent plans for Cortana decided to restructure it as a skill and not as a digital assistant.</td></tr>
<tr><td>Siri</td><td>The user community of Siri can be quite big due to the global number of Apple users, which might attract developers to the development of voice applications for it.</td><td>Apple restricts the programming languages that can be used to its own language (Swift) and Objective-C. It only allows the extension of mobile applications or the creation of shortcuts for them and not the creation of voice-only applications.</td></tr>
<tr><td>Alexa</td><td rowspan="2">Offers various free of charge and open-source tools for the development of voice applications. Vast documentation and tutorials to help the developers. Application models are similar, and the differences between them don't prevent the use of a generic construction process. For instance, they have a similar vision and definition of the language model components.</td><td>Recent launch of important updates and changes to the way Alexa works, which might result or not in a completely different application model for voice applications in the near future.</td></tr>
<tr><td>Google Assistant</td><td>A lot is going on in terms of new features and updates to Google Assistant and that might reflect or not in their application model for voice applications.</td></tr>
</table>

2.4.1 Frameworks

Jovo

Jovo is an open-source cross-platform framework that was created to allow a developer to develop voice applications, from the same codebase, for Amazon Alexa and Google Assistant. Given that these two digital assistants share some structural similarities, this framework makes use of that characteristic and allows developers to develop generic JavaScript (Node.JS) code without losing the ability to use platform-specific functionalities [50]. It should also be mentioned, that Jovo provides a service that helps developers in the creation of webhooks. This service creates a link to a local web server in order for the developers to be able to debug the application locally instead of having to update the code all the time on the chosen hosting service and then debug the application recurring to the digital assistant console.

Lastly, by being platform agnostic, this framework helps increasing the efficiency of the development process as the developers won't have the need to maintain two separate codebases for the two assistants and also testing the voice application business logic and maintaining a consistent user experience across platforms will be easier.

However, even though Jovo provides a proprietary language model for the development of the frontend, that can later be transformed into language models for Alexa and Google Assistant, the developer will have to specify it entirely by hand in a JSON file, which can become quite cumbersome the more complex the application becomes. Additionally, if there is the need for platform-specific functionalities, the developer will have to specify them in one of the two additional sections of the Jovo language model. These two sections consist in the language models of Alexa and Google Assistant, respectively. For instance, if a functionality should only be present in Google Assistant, the developer will have to specify it in its language model, inside the Jovo model. Such implies that if in the future the developer wishes this functionality to also be present in Amazon, that he might have to refactor a considerable portion of the overall Jovo model by hand.

Violet

Violet is an open-source cross-platform framework that was created in order to ease the development of high-end voice user experiences by helping developers in the definition and use of conversational flows [69]. This framework offers support to the development of conversational bots and voice applications. Violet uses JavaScript (Node.JS) and a HTML-inspired language to develop the conversational flow between the user and the application.

Furthermore, it also allows the use of plugins to ease the call to external APIs, such as a plugin to make a connection to a PostgreSQL database. Regarding the deployment of the applications, the developer can deploy them to mobile, web pages, Amazon Alexa or Google Assistant.

It should also be mentioned that the language model, generated by Violet, for Google Assistant, at the time this dissertation was developed and written, was not well structured. The generated model for Google Assistant lacks the arguments and their types, that are defined in the backend code, which means that

the developer will have to specify them by hand on the DialogFlow console and also correct the phrases where they were supposed to appear.

Voxa

Voxa is an open-source cross-platform framework that is focused on providing a way of organizing and developing voice applications as state machines. This framework provides support for the development of Amazon Alexa, Google Assistant and Cortana voice applications. Voxa claims that no matter how complex a VUI is, it can always be represented through a state machine [5] in the backend and that this type of representation brings the flexibility that the VUI needs to be rigid in specific states but also to be able to move around states when it's necessary.

An adaptation of the MVC architectural pattern for voice applications is used by Voxa, which means that the developer will have to structure its application code in three distinct parts: models, views and controllers. The model is the layer that will save/retrieve/delete data from, for example, a database but it can also be a data structure that will hold the current session state of the application. The session state will only persist until the user closes the session. The view is the response that the developer will code in the backend and that the application will return to the digital assistant, which in turn will say it back to the user.

In conclusion, the controller is where the developer will code the application as a state machine by separating the business logic in "states", that will each return an adequate response to the digital assistant. In terms of the development of the frontend, Voxa is currently developing a spreadsheet approach to the definition of the language model. Voxa spreadsheet approach can be somewhat unusual given the fact that, normally, language models are defined, for example, in JSON files. This approach might not appeal to all developers.

2.4.2 Platforms

BotTalk

BotTalk is a markup language and an online platform that allows developers to create cross-platform and multimodal voice applications for Amazon Alexa and Google Assistant [81]. This platform is focused in providing a simple and easy way for developers to create voice applications without them having to resort to Amazon Lambda or to the codification of a hosting server to host the application code. It also allows developers with little or no backend coding knowledge to develop voice applications as they will only have to use the BotTalk language to develop all the business logic and the applications responses, that are going to be said back to the users.

Having said that, it should be noticed that BotTalk doesn't provide support to the definition of platform-specific functionalities and the way it supports the use of built-in argument types, that the digital assistants provide, is still somewhat poor given that it only provides support to a selection of them. For instance, this

support could be expanded if this framework allowed the definition of two types for an argument, one to be used in Amazon Alexa and other to be used in Google Assistant.

BotTalk voice applications are organized in three distinct parts, that are specified in YAML files, and they are the scenario, the intents and the slots [23]. The scenario is where the developer will specify information concerning the voice application (e.g its invocation name) and all its business logic. The development of the business logic will consist in the definition of the several steps and actions that need to happen in order for the application to be able to fulfill the user's request. The intents is where the developer will define the sample phrases of the functionalities, which are phrases that the user can say to invoke them. Lastly, the slots is where the developer will define the arguments that a functionality needs from the user in order to fulfill its request.

Invocable

Storyline, afterwards renamed Invocable [75] was a framework developed in order to ease the creation of voice applications for Amazon Alexa, given that it allowed the developers to develop an application without having to resort to coding. In order to develop the voice application, the developers were provided with a visual editor with a drag-and-drop interface. This type of interface allowed the definition of a set of connected blocks, that represented the flow of the application, being that each one of the blocks defined what the application should say to the user and what the user could say in response and vice-versa.

However, even though the founders of Invocable had recently restructured the framework, it's no longer available [76] given that it was integrated with VoiceFlow, a platform for the development of voice applications for Amazon Alexa and Google Assistant. VoiceFlow, like Invocable, is focused on the development of voice applications via a drag-and-drop editor.

VoiceFlow

VoiceFlow is a platform for the design, prototype and development of cross-platform voice applications for Amazon Alexa and Google Assistant that is designed to be non-technical (doesn't requires coding). This platform offers a free of charge and a paid plan to the users, being that the free plan limits the user to the creation of only three applications.

The development process consists in using a drag-and-drop interface to either design a high fidelity prototype of the voice application or to develop the VUI of the application and, in case it's needed, connect to APIs or third-party services [79] in order to provide a certain functionality. VoiceFlow also allows the developer to directly launch its voice application to the app stores of the digital assistants and manages all that is related to the hosting of the application.

2.4.3 Comparison between Jovo and Violet

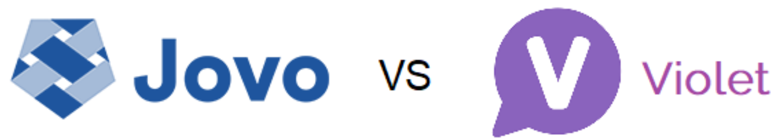


Fig. 2.: Jovo versus Violet

The construction process, aforementioned in chapter 1, is going to support the specification and generation of voice applications, whose development can be done recurring to the cross-platform frameworks Jovo or Violet.

Prior to this decision, a more in-depth study regarding these two frameworks was conducted in order to attest if the support for the development of their language model, that is used to define the frontend, and the boilerplate code for the initial development of the backend, had purpose in the construction process. Regarding the way the study was conducted, it was decided to develop a quiz voice application using both Jovo and Violet. Such decision is due to the fact that the most fitting way to make a well grounded study concerning these frameworks was to use them to develop a voice application instead of only reading about them, which wouldn't give the necessary knowledge to take well-founded conclusions.

In regards to the quiz voice application, it was decided that the application should allow the user to perform general knowledge quizzes in topics such as movies and sports, and that he would gain points for every correct answer he gave. The points will allow the user to level up through a rank designed for this application, that goes from apprentice to grand master. Furthermore, this application will require the user to maintain a continuous dialog with the digital assistant, which allows the assessment of the support that Jovo and Violet provide to the development of voice applications that are dialogue driven instead of request-response driven and that require the persistency of data, and the maintenance of the current conversation state between the user and the application. The conversation state will allow the definition of functionalities that are specific to a certain state of the application, for instance, the definition of a stop functionality that only stops the execution of the current quiz and not of the application.

For the development of the application, first it was established its requirements (Annex A.2) and specified its conversation model. The following phase consisted in the development of the frontend and backend of the application. The development of the application using both frameworks and what each one offers to the developers is going to be presented in the following tabular explanations: Table 2, Table 3, Table 4, Table 5 and Table 6.

Table 2.: Documentation

	Jovo	Violet
Documentation and tutorials	Thorough documentation regarding how to develop voice applications and how to use Jovo, and which functionalities are provided. A few tutorials are available in order to explain how to utilize Jovo and what it can be done with it. Additionally, documentation explaining Jovo API is also available.	Documentation is still under development. No tutorials are provided but some simple examples of voice applications developed using Violet were made available. Furthermore, the documentation lacks some important explanations concerning, for instance, how to change the launch mode (e.g if it's mandatory the specification of voice scripts along with the language model) and how to add a third-party database using their API.

In regards to the development of the language model using Violet (Table 3), it should also be mentioned that this framework imposes the definition of voice scripts along with the model. Such is due to the fact that there are some components of the application, that Violet generates, that seem to be set to default like the invocation phrase and the way the application interacts with the user. In case the developer wants to change those components, he has to create a voice script using the Violet HTML-like language, which translates into having to define the VUI twice instead of only once, which adds redundancy to the development phase.

Table 3.: Development of the language model

	Jovo	Violet
Specification	Definition of the application invocation name and functionalities in a proprietary JSON model. The invocation phrases and the necessary arguments for the functionalities also had to be defined. Lastly, the type of the arguments, that the user will provide, were also defined.	Definition of the functionalities and their invocation phrases and arguments in the backend code with the method respondTo(). The developer needs to create a respondTo() method for each functionality. The definition of the types of the arguments, that the user will provide, will also be in the backend code with the method addInputTypes().
Advantages	The Jovo language model can be converted in two platform-specific models, one for Alexa and one for Assistant, by using the tool Jovo Cli and the build command.	The platform-specific models, one for Alexa and one for Assistant, are automatically generated after the voice application is deployed. After the deployment of the application, Violet provides an endpoint to access a web interface, where tests to the application can be performed, and the generated models.
Disadvantages	Specification of the model entirely by hand in the JSON file. If there is the need for platform-specific features they will have to be specified in the proprietary language model of the targeted platform, that is also part of the Jovo model. The presence of various language models in one somewhat increases the complexity of the overall model and of the possible future improvements.	Mixing the definition of the frontend and backend in a single source file, which might translate in a more complex debugging phase and may also make maintaining the application a laborious task due to the lack of separation of concerns. Incorrect generation of the language model for Google Assistant in terms of the definition of the arguments and their types. Doesn’t let the developers name the functionalities and instead provides them with generic names like IntentA, which makes the debugging task more complex than it needs to be.

Table 4.: Development of the business logic - hosting and database

	Jovo	Violet
Hosting	It was used AWS Lambda as an hosting service given that it manages and automatically scales the application, and offers the services of various databases. Additionally, it also allows the creation of an API Gateway in order to connect to DialogFlow, eliminating the need to use a second hosting service to launch the application in Google Assistant.	Following the examples present in Violet documentation, it was decided to host the application in Heroku. AWS Lambda couldn't be used because at the time Violet didn't complied with the way that Lambda requires the routing handler, that routes the HTTP requests that the application will receive, to be defined.
Database	Given that the application is going to be hosted in AWS, it was decided to use DynamoDB given that it can be easily integrated with Lambda and is supported by Jovo.	A FileDB, more specifically a JSON file, had to be used, even though Violet provides a plugin to access data in a Postgres database. Such is due to the fact that the explanation that Violet provides on how to setup the plugin on the code is extremely lacking and the various attempts to successfully set it up were damaging the time management for this study and the benefits of using Postgres wouldn't compensate it.

Regarding the development of the business logic, the application will provide different types of quizzes depending on the digital assistant that is being used. Such is due to the fact that Google Assistant imposes a limit of 200 values in its argument values, and this affected the answer argument, that will contain the possible answers that the user can give, because its number of values surpassed this limit. In order to solve this dilemma, it was decided that Alexa would have true or false and multiple choice quizzes while Google Assistant would only have true or false quizzes. This decision led to the necessity of knowing, during run time, with which assistant the user was talking to.

Table 5.: Development of the business logic - Coding

	Jovo	Violet
Handling the different types of quizzes	Jovo provides a method that returns the type of application (Alexa skill or Google Action) that the user is talking to, which solved the dilemma of knowing, during runtime, with which assistant the user was talking to. By using this method it was possible to use the same codebase for both assistants, given that to correctly provide the quizzes there was only the need to check which type of application was being used and according to that the user's path would diverge.	Violet doesn't provide a method to check which digital assistant is the user talking to and implementing one would be a possible useless task due to the fact that at the time Violet API was undergoing changes that could potentially nullify the developed method. It was then decided that the voice application developed with Violet would only provide true or false quizzes for both assistants.
API	Jovo API was considered to be remarkably complete and offered some methods that were very helpful in the development of the functionalities. For instance, Jovo provides methods to access the database or the session attributes, a method to move to another functionality when needed, without requiring user input, and another one to setup the current conversational state that the application is in (e.g quiz state).	Violet API left much to be desired given that it lacked some basic methods such as one to get the userID (in order to find a way to get the userID the source code had to be studied and the only possible method available was soon to be deprecated). This lack of methods is comprehensible due to the fact that Violet at the time was still in beta, however there are some basic methods that should be provided first hand.

Table 6.: Testing phase

	Jovo	Violet
Advantages	Offers a webhook for local debugging, that allows the invocation of the functionalities in order to test the application responses and also lets the developer choose the assistant that is going to be used during the tests.	Offers a web interface with a integrated debugging tool. This tool allows the developer to mimic the invocation of functionalities by the user and also the user interaction with the application.
Disadvantages	The webhook only works with a FileDB and given that DynamoDB is being used the application had to be tested via the Alexa developer Console/DialogFlow and AWS.	Alexa developer console and DialogFlow had to be used to fully debug the application.

Even though both frameworks allowed the development of the purposed voice application, it can be stated that Jovo offered more methods and tools, that helped and accelerated the development process. Regarding Violet, it's still a maturing framework and needs more improvements, given that , despite the fact that a functional voice application was also developed, the development process was impaired by the lack of methods in their API and the incorrect generation of the language model for Google Assistant.

In conclusion, concerning the development of the backend, it can be stated that Jovo offered a more powerful API than Violet. Both frameworks offered a somewhat similar support to the development of the frontend.

It will be in the development of the frontend that the integration of the proposed construction process with these two frameworks will be most useful. Such is due to the fact that this process will provide a way for the developer to only have to define one language model and then have the option to generate it to multiple platforms. This one to many generation of language models will allow the developer to be more focused on what he wants his application to do and not on the different structural rules, and will streamline the cross-platform definition of the frontend. Furthermore, in case of Jovo, this process will eliminate the necessity of having to develop the language model by hand and, in case of Violet, will eliminate the necessity of correcting the Google Assistant language model by hand.

2.4.4 BotTalk support in the construction process

The support to the platform BotTalk by the proposed platform-independent construction process wasn't initially planned for, being that this decision was taken after the construction process was developed and implemented. If this decision had been taken beforehand this platform would have also been present in the comparison study that was performed and that was presented in the previous section.

Adding support for the integration of the construction process with BotTalk arose from the need to verify if the process was indeed developed in a way that made it extensible to new systems. Even though BotTalk

uses its own language to specify the voice application, it also uses the same components present in the language models of the approached digital assistants. Such similarity made it possible to offer support to the generation of the BotTalk language model and the generation of boilerplate code for its backend.

In conclusion, by adding support to BotTalk, the construction process will allow its users to be able to define and generate voice applications for one more system, without losing the possibility of, for instance, one day use the same definition to generate a language model and boilerplate code for Jovo, in order to develop the application using said framework.

2.5 Voice design patterns

As it was stated in Multimodal Interfaces of Human-Computer Interaction by Karpov and Yusupov [45] the interaction between humans and computers has three main aspects: Communication, information exchange between the participants; Interaction, exchange of actions and reactions; and Perception, the cognition of each other by the participants of the conversation.

Given that voice applications will "have" a conversation with its users, the definition of a voice user interface will be centered in its conversational aspect. This interface can be defined recurring either to both voice design patterns and existent guidelines for VUI design or to one of them.

In regard to the design patterns, when a new technological platform appears there is always an attempt to apply the interface design patterns that were used in other existing platforms [4]. An example of it was when mobile applications first appeared and the developers tried to apply the design patterns that existed for the development of web interfaces.

A similar attempt also happened when it came to voice applications, where developers tried to apply mobile design patterns to the development of VUIs. Such attempts revealed themselves fruitless, given that voice interfaces have an invisible and transient nature [71], unlike graphical interfaces that are always visible and permanent. Those interfaces are permanent in the sense that the users can see the effects that their actions had on the graphical interface after they were fulfilled. Such is not possible on voice interfaces, given that after the fulfillment of the user's request and the communication of its result by the assistant, the commands and the previous applicational state of the voice application can no longer be accessed.

Regarding the guidelines, there are several research papers such as Multimodal Interfaces of Human–Computer Interaction by Karpov and Yusupov [45], User interfaces for voice applications by Kamm, C. [43], among others or books such as Designing Voice User Interfaces by Pearl, Cathy [65], that provide some guidance to what should and shouldn't be done in the development of VUIs. Additionally, the developers can also find some guidelines in web articles or, for instance, in the development methodologies of voice applications provided by Amazon and Google for their digital assistants.

Therefore, it was decided to conduct a study regarding both the existence of voice design patterns and the already existent guidelines.

2.5.1 Guidelines and Design patterns

At the beginning, due to the novelty that was to develop voice applications and because most of the applications that existed in the market were either web or mobile applications, there weren't any design patterns that helped in the development of the VUI, only guidelines. The existent guidelines provided tips for the developers about what should and shouldn't be done in the definition of the VUI.

Nowadays, in the voice applications field there are already various guidelines that should be followed, such as:

- Avoid overloading the user short term memory with irrelevant information [61] and don't force him to remember large quantities of information between interactions [43];
- Develop ways to recover from error situations that are in accordance with the current interaction context. If it's possible, it should be explained to the user why its request couldn't be fulfilled and the alternative solutions [43];
- Be careful in the development of the confirmation questions, in terms of how the user is asked to confirm a certain information that he gave and also of when he is asked to confirm it. The developer needs to make sure that the interaction is not tedious, inefficient and long [43].

A few research papers, such as User interfaces for voice applications by Kamm, C. [43], Multimodal Interfaces of Human–Computer Interaction by Karpov and Yusupov [45] or Classifying Smart Personal Assistants: An Empirical Cluster Analysis by Knote, Robin and Janson, Andreas and Söllner, Matthias and Leimeister, Jan Marco [48], and books, such as Designing Voice User Interfaces by Pearl, Cathy [65], are centered in either the proposal and discussion of basic guidelines for the definition of VUI's and digital assistants or in the correct designing and testing of the VUI's. In Multimodal Interfaces of Human–Computer Interaction [45], it's stated that a proper VUI should focus on meeting the end users requirements and also apply a few basic guidelines such as:

- The interface must be intuitive to the user and feel natural to him. This means that there shouldn't be the need to instruct the user on how to work with the VUI;
- The interface must allow the user to use the minimal possible number of operations to reach its goal. The voice application must carry out the user's requests as swiftly as possible, which will improve the overall efficacy of the application;
- Assure, as much as possible, that the interface is robust in terms of the functioning of the VUI components and accurate in regard to, for instance, being able to detect the normal ways a user can say its request. This will help diminish the errors that can happen during the operation of the interface and make it reliable;

The work presented in Classifying Smart Personal Assistants: An Empirical Cluster Analysis [48] addresses, for instance, the importance of the anthropomorphization of digital assistants and of the use of the conversational context to improve the voice application interaction with its end users (context-awareness).

Regarding anthropomorphization, this notion refers to ”a conscious mechanism wherein people infer that a non-human entity has human-like characteristics and warrants human-like treatment” [88]. The realization of this notion entails creating a persona for the voice application in order to define the way it will speak with the users. The existence of a persona will make the users consider the application to be more user friendly [88] and will raise their expectations of a more human behaviour and responses [19].

In regard to context-awareness, the notion of context is defined as ”all aspects of an entity's (i.e a person's, place's or object's) physical and logical environment”. If the developer makes its voice application able to detect the surrounding context, the application will be able to react to it and trigger an adequate action, which may result in providing the user with a more complete response to its request or in suggesting an action that may be relevant in that context.

In User interfaces for voice applications by Kamm, C. [43] it's stated that when specifying the VUI of their voice applications, the developers should take into account the task requirements of the application, the capabilities and limitations of the existing technology such as ASR and NLP, and the characteristics of the end users in terms of expertise, expectations and preferences.

The design strategies proposed in User interfaces for voice applications [43] are focused in alleviating the existing technology limitations and in improving the usability of voice applications. These strategies refer to the use of ”dialogue flow strategies using appropriately directive and informative prompts and the use of subdialogue modules to provide access to instructions, to confirm that the user's input has been correctly recognized, and to detect errors and recover from them.”.

Regarding the use of feedback by the application and the use of confirmation questions, these two strategies are said to limit the possible wrong actions that the application might make by providing the user with feedback concerning its state and by requesting confirmation about its interpretation of the user's request. Nevertheless, it's also stated that being too meticulously in the implementation of these strategies often results in tedious and inefficient interactions, which will lower the user's satisfaction with the application.

In regard to error recovery procedures, they are developed in order ”to prevent the complete breakdown of the system into an unstable or repetitive state that precludes making progress toward task completion.”. By initiating the error recovery procedure, the application will allow the user to understand that something went wrong and he will have the chance to take actions that can put the application back to the correct state.

It should also be mentioned that in User interfaces for voice applications [43] it's referred that the design of the VUI is more efficient as an iterative process, that should empirically test the interface with groups, that represent the end users, for instance, with the use of Wizard of Oz tests.

In the book Designing voice user interfaces - Principles of conversational experiences [65], Pearl offers her expertise in designing VUIs and guide us through several VUI design guidelines, how to test and measure the VUI performance and how it can be improved, among other topics. In regard to the VUI design guidelines some are centered in domains such as confirmation strategies, error handling, use of context, commands that every voice application should support, among others.

When it comes to the use of context, it's stated that in a conversation the current context matters and "having a conversation with a system that can't remember anything beyond the last interaction makes for a dumb and not very useful experience" [65]. As a result, the application should maintain a context throughout its interaction with the user (e.g in a database) in order to know what happened before and how such information can be useful when performing the current task.

In regard to the confirmation strategies, the same opinion that was found in User interfaces for voice applications [43] can be found in this book, over-confirming a piece of information might ensure its veracity but will, most likely, drive people away from the application. In regard to the commands that every voice application should support, commands such as repeat, help or goodbye are suggested.

Regarding voice design patterns, there were a few patterns that were developed by Schnelle, Lyardet and Wei in [71] and later only by Schnelle and Lyardet in [54]. These patterns were developed in order to provide a centralized, well-structured and clear methodology [71], that the developers could use when they are developing the voice user interface of their voice application. The design patterns for VUI's created in [71] are focused in three aspects (Table 7), the ministration of data (Data entry), the level of experience of the users (Level of experience) and the limits of the users short term memory (Limits of STM).

For each one of this aspects, the following patterns were developed:

Table 7.: Design patterns for voice applications defined in [71]

Design pattern	Focus	Goal
Index	Data entry	Allow the user to know what can be done in the application.
Scripted interaction	Data entry	Provide ways for the user to discover what can be done in the application and guide him through them.
Escalating detail	Data entry	Provide increasing levels of detail if the user requests so.
Decoration	Level of experience	Decorate the options with more information when it's the user first time in the application.
Detailed information	Level of experience	Provide detailed information about the current interaction topic upon user's request.
Information spreading	Limits of STM	Provide the correct amount of information.
Active reference	Limits of STM	Tell the user where he is in the application, what is happening, what can be done next, among other cardinal aspects to his orientation in the application

The design patterns for VUI's created in [54] are focused in three aspects (Table 8), the strategy used for defining the dialogues (Dialog strategy), the way that the system responds to the user (System response) and the usability of the application (Usability). For each one of this aspects, the following patterns were developed:

Table 8.: Design patterns for voice applications defined in [54]

Design pattern	Focus	Goal
Form filling	Dialog strategy	Method to gather information that is structured sequentially from the user.
Menu hierarchy	Dialog strategy	Method to gather information that is structured hierarchically from the user.
Mixed initiative	Dialog strategy	Method to gather information, that contains interdependencies among them, from the user.
Persona	System response	Define how the application must sound to the user (its "personality").
Structured audio	System response	Provide structured information to the user.
Busy waiting	Usability	Provide information to the user if he needs to be kept waiting for the execution result of a functionality.
Language selector	Usability	Provide support to the selection of various languages, mainly the language of the target audience.
Context aware call	Usability	Use the current contextual information to present more data to the user.

2.6 Voice applications development phases

The first phase in the development of a voice application consists in starting by specifying how the voice user interface and voice user experience of the application should be. This specification normally consists in a five step process (fig.3). Throughout this phase, the developer will spend most of its time in the steps of defining, describing and refining the dialogues between the application and the user, that constitute the conversation model. Such is due to the fact that the conversation model is a very important aspect of a voice application and most be well defined and developed [43]. Additionally, when the developer is testing the VUI via, for instance, Wizard of Oz tests it might be necessary to either refine the way that some interaction was defined or define an interaction that was missing.

It should also be mentioned that Wizard of Oz tests consist on having a person role playing the part of the voice application, that is, reading the answers that the application will provide to the user's requests and on having another person role playing the part of the user, that will make the requests to the application [4].

2.6.1 Specification

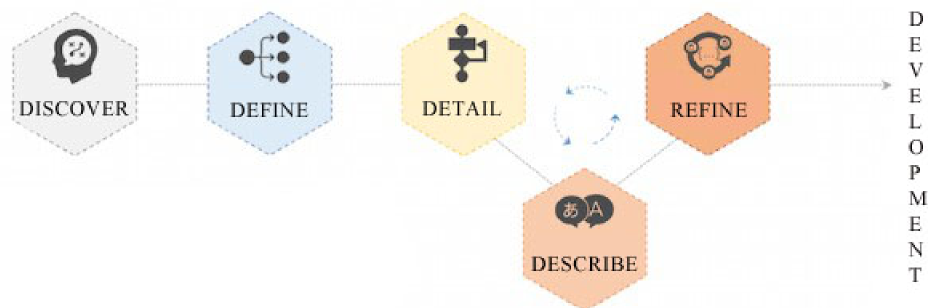


Fig. 3.: Specification of a voice application [18]

The first phase of the specification process consists in answering the question "What do the users need?" [18]. This is due to the fact that only the idea that the developer has and its respectively viability is not enough of an assumption to start planning a voice application. The developer should also find if what he is going to develop is indeed interesting and necessary at the eyes of its potential users [35].

The second phase of the specification consists in defining two of the most important aspects of a voice application, its personality or persona and its functionalities. The developer must define a persona for its application because the users are always going to, unconsciously, project the idea of a person when they are using the application [65]. Thus, the developer should purposely conceive a persona in order to try to influence what the users will perceive in terms of the personality of the application and also because researches shows that the personification of VUIs improve the user's experience [41].

Regarding the development of the application persona, unlike in web/mobile applications, with voice applications, visual elements can't be used. Except in the cases where the connected device, where the digital assistant is present, has a screen and there is support of the SDK to the development of multimodals voice applications [65] [73]. Therefore, the persona must be reflected in the name of the application, in its icon, description and invocation name. Furthermore, the persona should also be reflected in the way that the application converses with the user in terms of the tone of voice it uses and in the way that its sentences are structured.

Lastly, in the second phase of the specification, the developer must define the application functionalities. To define them, the developer must first understand which functionalities should always be present in a voice application (e.g a help functionality) and also which functionalities would be most valuable to the users in the type of application he is developing. This research is needed due to the fact that, even though some ideas might sound great, they might, for instance, end up not being what the users really need or they aren't able to function in a voice-only application because to achieve their full potential they also need the support of visual elements [65].

Furthermore, the developer should also verify if the application functionalities are going to be innovative in the category that the application will belong to and also take into account that digital assistants are

normally present in connected devices, such as the Amazon Echo or Google Home, which are inserted in shared spaces and which is something that might lead to the need of security measures [19] [82] (e.g account linking).

The third and fourth phases of the specification consist in detailing and describing the application conversation model, that is, the way that the application and the user will interact. Their interactions can be defined, for instance, by creating dialog scripts and/or flow diagrams [65], and later by defining a language model. This model will "teach" the application the various phrases that the user will normally say to express its request (phrase-mapping) and if there will be arguments that the functionalities will need to receive from the user in order to fulfill its request [4].

In terms of the definition of the conversation model, the developer can start, for example, with the definition of dialog scripts that detail interactions that did not result in any errors or exceptions. After the definition of those scripts, the developer must define those where exceptions did happen due to errors such as the lack of information (Design for failure). Errors can occur, for example, when the application couldn't understand what the user said (e.g a possible problem of ASR) [43] [46] or couldn't do what was requested (e.g don't have the requested functionality).

The two aforementioned phases are the most important in this specification process because the users will always end up being able to ask anything to the application and in a format of their choice. Therefore, by providing the application with what the user can say when he wants the application to fulfill a certain request, the developer can try to redirect the conversation when the user asks something that is not in accordance with the available functionalities.

Lastly, the fifth and last phase of the specification process consists in validating and testing the aforementioned scripts, in order for them to be refined before the developer begins the application development phase. For instance, to fulfill this phase the developer can use Wizard of Oz tests. Concluded the specification process, the developer is apt to proceed to the application development phase, that will consist in the development of the frontend (VUI) and the backend.

2.6.2 Development

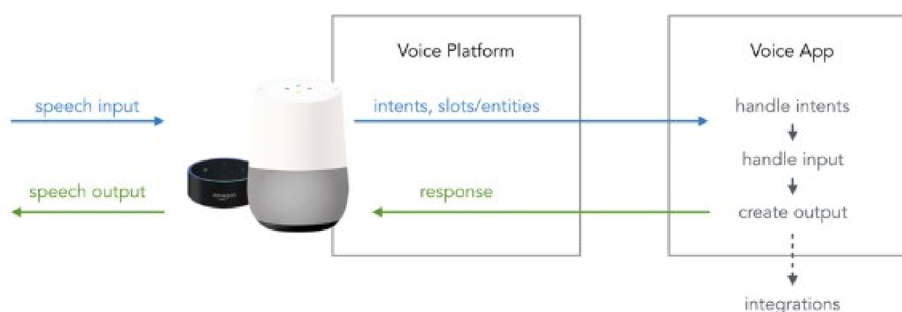


Fig. 4.: Flow of a voice application [50]

Regarding the development of the voice application, the developer is restrained either by the SDKs, that exist for the chosen digital assistant, or by the functionalities of the framework that he is going to use. The

development process usually begins with the definition of the frontend (Voice user interface). Such is due to the fact that the VUI is the most important aspect of a voice application and it's where the developer will specify what the user can ask of it.

The approach taken during the definition of the VUI can influence the target audience feelings towards the application. This means that the VUI can make the target audience either be fan of the application because they don't feel like they are talking with a machine or think that the interaction its monotonous and frustrating due to the way that their requests are dealt with [43].

Concerning the development of the VUI, it will consist, as it was aforementioned, in defining the persona of the application and its conversation model, that will later be adapted to a language model. In the language model, the developer will have to define the invocation name of the application, the functionalities that the application will offer (Intents), how the user can request said functionalities (Sample phrases) and if there are arguments that the functionalities need from the user in order to satisfy its request (Slots) [4].

Concluded the definition of the VUI, the developer will proceed to the development of the backend of the voice application. The backend will support all the business logic in terms of how the application functionalities must be performed. Regarding how the backend will be composed, it will be constituted by handler functions, that will support the defined functionalities [4], and by a routing handler, that will handle all the incoming HTTP POST requests from the digital assistant.

There might also be the need to store information in a database that will, for instance, improve the future use of the application or store the current conversational state that the application is in, so that it may know which handler function to execute [4]. Additionally, the backend might also have to use external APIs in order to have access to more resources and/or extra functionalities.

Frontend

The first phase of the frontend development process, aforementioned in the specification section 2.6.1, consists in making a clear and detailed specification of the persona of the application and of its functionalities and how they can be invoked. The second and last phase, consists in transposing the VUI specifications, that were developed in the previous phase, to a language model [4].

The definition of the language model, should start with the choice of the invocation name, that is, the name of the application. The invocation name shouldn't be either too long nor have present predefined words like ask, run, start, among others. A guideline for choosing the name of the application is that it most contain at least two words.

The next step in the definition of the language model is the creation of the intents, that is, the application functionalities. The intents scheme will be used by the assistant every time that the user invokes the application and requests something of it, in order to know which functionality must be invoked. Additionally, in case there isn't an adequate functionality to deal with the user's request, the assistant will use the scheme to be able to invoke one of the intents that deals with exceptions/errors.

In the intents scheme, the developer should also define the sample phrases, that is, the various ways that the user normally uses to request each one of the intents. When defining those phrases, the developer

should provide the largest possible amount of examples, in order to cover the largest number of cases and to feed the machine learning ASR process with more examples.

The last element that needs to be defined in the scheme, if necessary, is the arguments that the user must provide in order for the functionality to be able to fulfill its request. In the cases where the users don't provide said data beforehand, the assistant must request it from the user, for instance, by the means of a conversation funnel, that was previously defined by the developer. A conversation funnel consists in guiding the user towards its goal by providing him with fewer and fewer options each time[1].

The final step in the definition of the language model consists in the definition of the type of the arguments, that were defined in the intents scheme. These types will be custom types defined by the developer, given that the ones that the digital assistants provide don't have to be defined in the model, and they will only have to be defined if there are arguments that will use them. Their definition will consist in specifying their name and the values that the arguments might take, and if possible synonyms for those values in order to increase the number of ways a user has to provide an argument.

Backend

The first phase of the development of the backend, consists in deciding which programming language is going to be used and possibly in verifying if there is a framework that can assist with the development process and future maintainability of the application. The choice of the programming language that is going to be used lies in diverse factors, such as for which languages the chosen digital assistant provides SDKs and with which of those languages is the developer more productive with. The choice of a framework lies in factors such as which framework best suits the application to be developed and if the developer intends to develop a single or cross platform application.

The second phase of development consists in making decisions concerning the deployment of the application. For instance, the application can be hosted in a cloud-service like AWS or the developer can build a webhook or a dedicated server. The development of webhooks is a method used to expand the behaviour, for instance, of a web application via customizable HTTP callbacks, in real time, when a certain type of event occurs that requires a system to provide information to another [3]. In this case, the webhook would be used to connect the digital assistant to the business logic. Furthermore, the developer should also decide if there is going to be the need for data persistence and if so which database is going to be used. For example, if developer chooses to host its voice application in the AWS he will have access to the DynamoDB, a no-SQL database [4].

The third phase consists in the development of the application functionalities (e.g the handlers [4]) in the chosen programming language. Lastly, the fourth and last phase consists in testing the application and in fixing the errors that are found and in improving the functionalites if needed. Having reached the end of the testing and improvements phase, the developer can make a distribution request of its application to the digital assistants app store.

Single platform development

Regarding the voice applications, that will be developed in this dissertation with the use of the proposed development process, they will be targeted to the digital assistants Alexa and Assistant, as it was mentioned before. As such, it was decided that the application models of both assistants should be presented (Table 9, Table 10) in order to expose the similarities and differences that exist among them and also the general development of a single-platform voice application.

Table 9.: Development of the Frontend of voice applications

Digital assistants	Language Model	Application responses
Amazon Alexa	Definition through a visual editor or a single JSON file.	Define the responses on the backend code and send them on the response output. Can also define a dialog model to handle the confirmation of an Intent and the gathering and confirmation of an Input value.
Google Assistant	Definition through a visual editor or multiple JSON files (two for each Intent and Input).	Define the responses on the backend code and send them on the response output or define them on the model schema.

Table 10.: Development of the Backend of voice applications

<table>
<tr><th>Digital assistants</th><th>Similarities</th><th>Differences</th></tr>
<tr><td>Amazon Alexa</td><td rowspan="2">Start with importing all the necessary modules and performing the needed configurations. To define the functionalites the developer needs to define a handler for each one of them. End with the definition of a routing handler to establish the entry point for HTTP POST requests.</td><td>Each handler is going to be composed by two different functions. A function denominated canHandle, that allows the definition of the activation rules, and a function denominated handle, that has the code that will fulfill the user's request.</td></tr>
<tr><td>Google Assistant</td><td>The handler is composed by only one function, that fulfills the user's request, given that there is no need for activation rules.</td></tr>
</table>

2.6.3 Programming models

The programming models, that the developers can use when they are developing their voice applications varies in terms of which digital assistant are they being developed to. Regarding the programming languages that can be used, there are already a wide range of SDKs, either provided by the companies that develop the digital assistants or by third-party developers, in various languages. The most used programming languages are JavaScript (Node.JS) and Python, followed up by Java.

However, the choice of the programming language that is going to be used in the development process should always take into account which programming language(s) the developer feels more confident and productive with and the complexity of the voice application. For instance, JavaScript (Node.Js) can be more suited for small voice applications (e.g request-response applications) and Python can be more adequate for mid-range applications, that, for example, need to maintain a dialogue between the user and the application in order to fullfil their request. However, applications that are developed in Python will require a more intense test and debug phases given that the majority of the errors will only be detected in runtime. Lastly, Java can be more suited for high-range applications, that have complex interactions between the user and the application, given that it provides a wide range of tools and libraries.

In regard to the development platforms, the developers can use their preferred IDE or, in case one is made available, the IDE that is provided by the company that develops the digital assistant. For instance, Apple provides the IDE Xcode [3] and Samsung provides the Bixby developer studio [4]. Furthermore, the developers can also use a framework such as Jovo [51] or Voxa [5] in order to accelerate and streamline the development process of the application. The use of a framework can assure that the application will be well structured and that there are tools that will maintain it throughout time and accordingly to the future updates of the digital assistants.

Therefore, in regard to the programming models, the developers can use the SDK for the programming language that they feel more productive and comfortable with. Furthermore, they can also use a framework to achieve an application development that is more spry and, if possible, cross-platform.

2.7 Summary

This chapter concludes with a few thoughts on the state of the art regarding the digital assistants and the development of voice applications. The areas of voice applications and digital assistants are still evolving, in terms of technological advances, and that implies that the respective state of the art is still in a phase of maturation, In regard to discoveries, existency of voice design patterns, available tools for the application development, among other aspects.

Regarding the digital assistants, there are a plethora of assistants in the market due to the fact that there are a myriad of companies that are trying to be pioneers in this area [48] and, therefore, the first

3 https://developer.apple.com/develop/
4 www.bixbydevelopers.com/
5 http://voxa.ai/

consumer choice when the time comes to choose an assistant or to buy a connected device [56] such as the Amazon Echo or the Google Home.

However, in regard to development tools, voice design patterns and development methodologies there is still a long way to go in these two areas. For instance, the frameworks and platforms that are available in the market or are payed or sometimes aren't focused on the full-stack development of a voice application. Even the frameworks Jovo, Violet and Voxa, that are open-source and cross-platform, lack support or need to provide better support to the definition of the frontend of a voice application.

Regarding the voice design patterns, there are already some patterns developed by Schnelle and Lyardet in [54] and also by Wei in [71]. Nonetheless, these patterns are still not prevalent in the existent methodologies given that, for instance, high-level guidelines for the development phase are still preferred. Lastly, in regard to methodologies, as it was mentioned in chapter 1, a standard methodology, that the developers can use to obtain a well structured voice application, whose specification is general enough to work for more than one digital assistant, still doesn't exists.

To sum up, the study and elaboration of the state of the art allowed for a better comprehension of what was already been done until now in the area of voice applications and what is still there left to do or improve. Furthermore, this study also helped in the confirmation of the utility of the work purposed in this dissertation in the area of voice applications and their development for more than one digital assistant.

3

PROBLEMS STATEMENT AND CHALLENGES

3.1 Introduction

In this chapter, the problem that the proposed platform independent construction process for voice applications aims to help solving will be defined in detail. By defining the problem statement, the subset of problems that the construction process will have to solve will be determined and will serve as basis for the implementation phase.

In addition to the problem statement, the challenges that are expected to be found during the development of the construction process will also be analysed in detail. By analyzing the challenges that might appear during this dissertation, a set of possible approaches to deal with them can be planned ahead and serve as guidance in the implementation phase.

3.2 Problem statement

The absence of standardization makes the development of cross-platform voice applications more complex and time-consuming for the developer due to the current plethora of different digital assistants, that have their own application models. Furthermore, by having to work with more than one application model, the developer might be more focused on the different technological aspects rather than on the requirements of the application and its development.

Nowadays, what is happening in the field of voice applications is similar to what happened when mobile applications first appeared. When developers tried to develop mobile applications they didn't have a standardized development methodology, so they tried to apply the knowledge and development phases that they used in the development of web applications. Regarding the development of voice applications, what the developer encounters when he starts researching about it's documentation and a set of best practices guidelines provided by each digital assistant. Additionally, at the time this master's dissertation was developed, the developers also found an absence of a solid base of examples that teach how to put those guidelines into practice or that teach how to start with the definition of a specification and then efficiently develop a cross-platform voice application.

However, the main issue with the lack of standardization is that the developers don't have a common "language" to use to share ideas or development steps with one another and platform independent rules to

guide them towards the development of voice applications that have a good user experience across digital assistants. Furthermore, by not having a standard methodology to follow the developers might end up using the methodology(ies) that seems the most adequate concerning the purpose of obtaining a stable product. Additionally, the developers might also try to apply some development strategies used in the development of web/mobile applications, which won't be a good idea due to the transient and invisible nature of voice applications.

In the present day, the few existing frameworks and platforms, aforementioned in section 2.4, aim to deal with the absence of standardization by proving tools that can conceal the differences between the digital assistants application models. Having said that, the existing frameworks and platforms aren't foolproof given that, for example, some are more oriented to the development of the frontend while others are more oriented to the development of the backend. However, there are also a few frameworks/platforms that offer support to the full-stack development of voice applications.

In order to try to streamline the development of cross-platform voice applications, the proposed construction process will allow the definition of a generic specification of a voice application, that will then be used to generate the application components, and will also allow its deployment to the targeted platforms. The targeted platforms, as it was aforementioned in section 2.3, will be the digital assistants Amazon Alexa and Google Assistant. The cross-platform frameworks Jovo and Violet and the platform BotTalk, will also be approached in order to allow the developer to be able to use this construction process and one of those tools together in the development of a voice application.

The generic specification, that the construction process will use, must be flexible enough to allow the declaration of single-platform functionalities and also conceal the use of some platform-specific information, like built-in functionalities. This specification will contain all the necessary information to generate the frontend of the application and the boilerplate code for the initial development of the application backend. Regarding the deployment of the generated voice application, it will consist in uploading the application backend code to a cloud-service such as AWS and in uploading the language model, that constitutes the frontend, to the consoles of each digital assistant, Amazon Alexa Console and DialogFlow.

Another goal of this master's dissertation is the development of high-level UML activity diagrams, that will help in the modeling of the application conversation model, and of a platform, with a visual editor incorporated, that will abstract the use of the construction process. High-level UML activity diagrams will be defined with the purpose of providing the developers with a platform independent tool that can allow them to more easily model the frontend and outline the application functionalities, and communicate ideas with others (e.g client, other developers, etc).

The platform will abstract and promote the construction process and also make it suitable to both technical and non-technical users by providing a layer of abstraction on top of it, via the visual editor. Regarding the platform functionalities, the developers will be able to define the specification of the voice application and then use it to generate the necessary components for its development. Optionally, the developer can also choose to deploy the specification to the targeted digital assistants, which already generates all the needed components of the application.

3.3 Challenges

The main challenge of this master's dissertation is the somewhat high innovation speed of this technological area. This translates into sometimes having groundbreaking changes in the features of the digital assistants (e.g in the application model that they defined for their voice applications) or into having a regular and iterative release of updates. This is also verified in the cross-platform frameworks and platforms presented in section 2.4, given that they try to accompany the changes that occur in the area of voice applications. This challenge is not one that is easily dealt with yet there are some precautions that can be taken into account during the development of the construction process, in order to try to make its modification less complex when faced with future updates regarding the digital assistants application models.

Therefore, an architecture with emphasis on the separation of concerns should be used, in order to minimize the repercussion that the changes in one component might have in others. For instance, the component that deals with the definition of the specification of the voice application should be separate from the components that generate the platform-specific specification of the frontend for the targeted platforms. This is due to the fact that if any one of those components needs to be altered, the others won't have that same need, unless it's an important update, either on the specification defined by the construction process or on one of the platform-specific specifications, that implies changing the generic specification.

The differences that exist between the application models of Amazon Alexa and Google Assistant can also present a challenge to the development of the construction process. When the study regarding the digital assistants, presented in section 2.3, was conducted, similarities were found between the assistants application models. However, that doesn't guarantee that there will be enough common ground between them in the future to maintain the proposed construction process cross-platform. Such is due to the fact that they are owned by different technological companies, that develop them at different rates and with different mindsets and goals.

To conclude, the last challenge refers to the inherent difficulties of trying to develop platform independent processes and tools for an ever changing technological field. As it was previously mentioned the digital assistants have different application models and their level of difference when it comes to the components, that are needed to define the voice application, and their structural rules might end up being an hindrance to the development of the construction process. Additionally, maintaining the process platform independent, while trying to provide the developer with the possibility of marking an application functionality as platform-specific and also with the possibility of using platform built-in functionalities and argument types as if they were platform independent, might bring additional challenges.

3.4 Summary

Throughout this chapter the problem that this master's dissertation aims to help solving and its main goals was presented. By having a clear definition regarding the problem, it was possible to start defining in detail the goals for this dissertation and the potential solutions for them. Furthermore, it was also inferred how

the challenges, presented in Chapter 1, could be solved and how much they affected the potential solutions for the proposed goals.

Only by knowing the possible challenges that might appear and how they can be dealt with, can the implementation phase be safely approached with a clear set of possible solutions to those challenges and produce results that fit the goals presented in Chapter 1 and defined in detail in this chapter. In the next chapter, how the solution for the proposed goals was implemented and how the challenges that were encountered were solved, is going to be presented.

4

DEVELOPMENT

4.1 Introduction

In this chapter, the significant decisions and the approach that was taken in order to develop the proposed construction process, the platform and the high-level activity diagrams are going to be presented in detail. The final result is expected to have functionalities that can resolve or handle in the best way possible the aforementioned challenges (section 3.3), with regards to the current state of the art.

The construction process will be defined by a generic specification and by a couple of modules. The specification will be a language model template that will serve as the starting point to the generation of the platform-specific language models and boilerplate code. There will be four distinct modules, being that three of them will provide functionalities related to the generation of the platform-specific language models, the generation of the boilerplate code, and the deployment of the voice application to the targeted digital assistants, respectively. The fourth module will abstract the three previous modules and provide their functionalities to the platform components. These modules will be developed independently of one another in order to achieve separation of concerns and will use the previously mentioned specification so as to be able to complete their purpose.

The high-level UML activity diagrams, will be an auxiliary tool to the construction process, given that they have the purpose of helping the developer in the establishment of the application conversation model and in the outlining of its functionalities. The platform is going to be developed in order to provide the construction process with a user interface that will abstract its behaviour, streamline the specification of the language model template, and make the use of some features of the process, such as using platform-specific functionalities, more effortless to the developers.

To sum up, throughout this chapter, the chosen development approach, and how it was applied towards the achievement of the purposed goals of this master's dissertation, will be presented.

4.2 Approach and decisions

In order to be able to develop the construction process, firstly a study about the necessary components both in the frontend and backend of a voice application for Amazon Alexa and Google Assistant had to be conducted. This study will complement the study presented in section 2.6.2.

The frontend of a voice application is developed by defining a language model, that describes how the application can be invoked, what it can do and how the user can ask for the functionalities available. By defining this model, the developer will provide the assistant with a wide range of important information along with a mapping tool that allows the assistant to map each one of the user requests to its corresponding function on the backend. The information can be, for instance, the phrases that can be used to request a certain functionality or the data that is necessary, from the user, for its request to be fulfilled.

The conducted study led to the conclusion that both assistants have the same components in their language models, even though the models have different structures and some components have different denominations. In order to present the language model, it was developed a UML domain model diagram (Fig.5) that specifies its components and the relations that they maintain among them. The components of the model are the following:

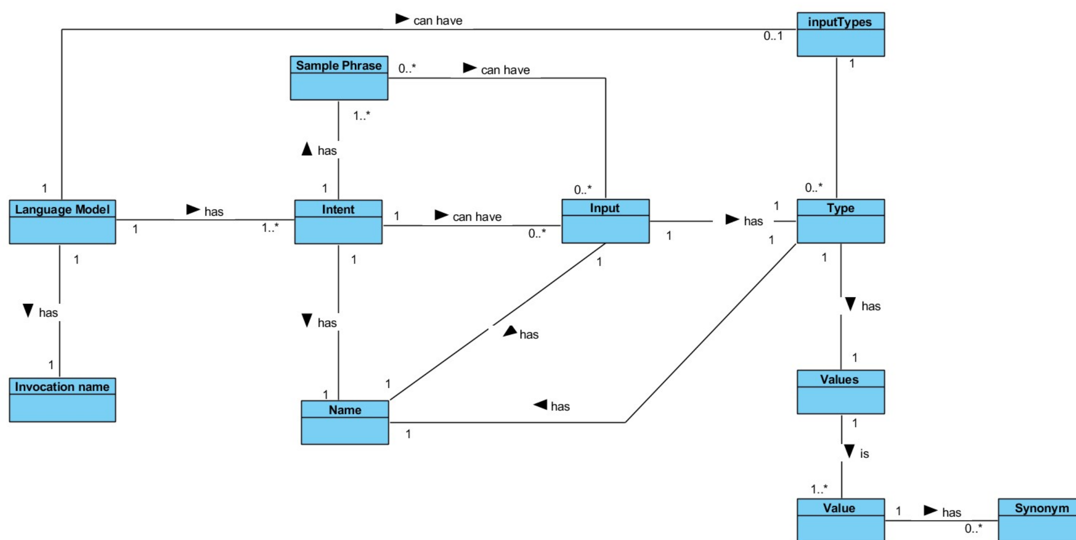


Fig. 5.: Components of the language model

- Invocation name, that corresponds to the name that the user will use to invoke the voice application;
- Intent, that represents a functionality that the application will offer to the user. Each one of the Intents will be composed by a name and by a set of sample phrases, which Amazon denominates of utterances and Google of User expressions, that represent what the user will usually say to invoke the functionality;
- Inputs, which Amazon denominates slots and Google entities, that represent the data that the user must provide in order for the application to be able to fulfill its request. The Inputs are associated with the sample phrases as they must be placed in the phrase in the place that is most likely for the user to say them. Each one of the Inputs is composed by a name and a type;
- Input Types, allow the definition of developer defined types, given that digital assistant provided types don't need to be defined;

- A type that is composed by a name and by a set of values that an Input might take. Furthermore, these values can have synonyms, that are defined in order to increase the number of ways that the user has of saying a certain word and therefore provide a certain field [4]. This makes the dialogue more flexible.

The backend of a voice application will be constituted by handler functions, that will support all the functionalities that must be provided to the user, and by a routing handler, that will handle the incoming HTTP POST requests. There might also be the need to persist information in a database that will, for instance, improve the future use of the application or store the conversational state that the application is in, so that it may know which handler function to execute, if necessary. Additionally, the backend might also have to use external APIs in order to have access to more resources and/or extra functionalities.

To sum up, this study led to the discovery that the language models of both assistants share the same structural components and complemented the study presented in section 2.6.2. Both these studies proved that there are enough common elements both in the development of the frontend and the backend of a voice application, for the targeted assistants, to support the proposed construction process. Additionally, these studies also lead to a conceptual decision regarding the generic specification that the process will use.

The conceptual decision consisted in specifying a platform-independent template that generalizes the language models of both assistants and that will allow the developer to specify the frontend of a voice application. This template will allow the generation of platform-specific language models due to the fact that it will be composed by all the common components of the assistants language models (Fig.5).

However, this template will also have some information that is not present in the Fig.5 and that will allow the process to provide more options to the developer in the definition of the frontend, such as the possibility of marking a functionality as platform-specific. Furthermore, the way that the template was defined and how it will be used to generate the platform-specific language models will be explained in detail in a following section.

4.2.1 Definition of the conversation model

Typically, when developers begin the definition of the conversation model of their voice application, they are faced with several recommended approaches. These approaches can consist in the following:

- Conduct a few market researches [35] to better comprehend the target population and the way that they usually speak in terms of phasic construction, expressions and abbreviations;
- Outline and develop dialogue scripts, that define and expose the intended interaction between the user and the voice application [65];
- Definition of high-level flow diagrams, that outline the conversation flow of the voice application [34] [35].

It should also be mentioned, that the developers can choose to follow more than one of the presented approaches and deliver, for example, both dialogue scripts and high-level flow diagrams instead of only one of those. This choice depends on what the developers think is necessary to develop the VUI and on the development methodology(ies) that they chose to follow.

Regarding the high-level flow diagrams, not all developers feel confident in developing them due to the high probability of committing the mistake of iterating too much the definition process. This mistake happens when the developer tries to describe every possible next step in the conversation between the user and the application. Such task can end up being a potential infinite exercise given that the users have the freedom to say whatever they want whenever they want when they are using the application. If this mistake isn't revised, the voice application will be very similar to an IVR and not to a natural dialogue with another person [65].

Until now the construction process will offer a language model template for the development of the frontend of a voice application. However, a tool for the definition of the conversation model, that will help the developer in the specification of the language model template, and, therefore, of the frontend, will also be developed. Such decision was based in two factors, one related to the third challenge and the development of platform-independent tools, presented in section 3.3, and the second one related to the development of a tool that provides a common language for developers to share ideas with one another.

This approach consists in the development of high-level UML activity diagrams in alternative to the previously mentioned high-level flow diagrams [34] [35], given that they provide a well-defined notation and development rules, and will also be a platform-independent tool that will help in the definition of the application conversation model.

Nonetheless, the development of activity diagrams follows a certain set of rules that might not give the developer enough freedom to express what its voice application is going to do. Additionally, if the developer has never used UML he will have to take some time to learn the notation and rules inherent to the development of activity diagrams. Nevertheless, it's considered that, despite the possible disadvantages of using UML activity diagrams, they were still be an acceptable approach when compared to the high-level flow diagrams.

The activity diagrams will be used to specify the main flow of conversation between the user and the application, the application functionalities, and an example of a phrase that can be used to invoke the functionality. These diagrams will help in the development of the language model components because the developer will already have a solid notion of the functionalities (Intents), that the application will provide, and their sample phrases, and if they will need to receive arguments (Inputs) from the user in order to fulfill its request or not.

Furthermore, the activity diagrams can also be used to demonstrate the interactions between the user and the application during the fulfillment of a request (a more detailed flow of conversation), so that the developer can specify what he wants its application to respond back to the user when a certain functionality is being performed.

Therefore, by developing a tool for the definition of the conversation model, that uses a well-known and established modeling language, it's expected that the developers will more easily avoid the mistake

of iterating too much the definition process of the diagrams and that they end up developing a voice application that sounds natural and human (as much as possible) to the users. Regarding the chosen modeling language, as it was aforementioned, it was decided to use UML for the development of this tool. This choice falls into an attempt to develop a tool that the developers could use regardless of the digital assistant that they choose to develop the voice application for.

Lastly, the high-level activity diagrams will be presented and explained in detail in a following section (section 4.3.1), in order to better expose how they help in the definition of the frontend and how they were specified.

4.2.2 Backend development

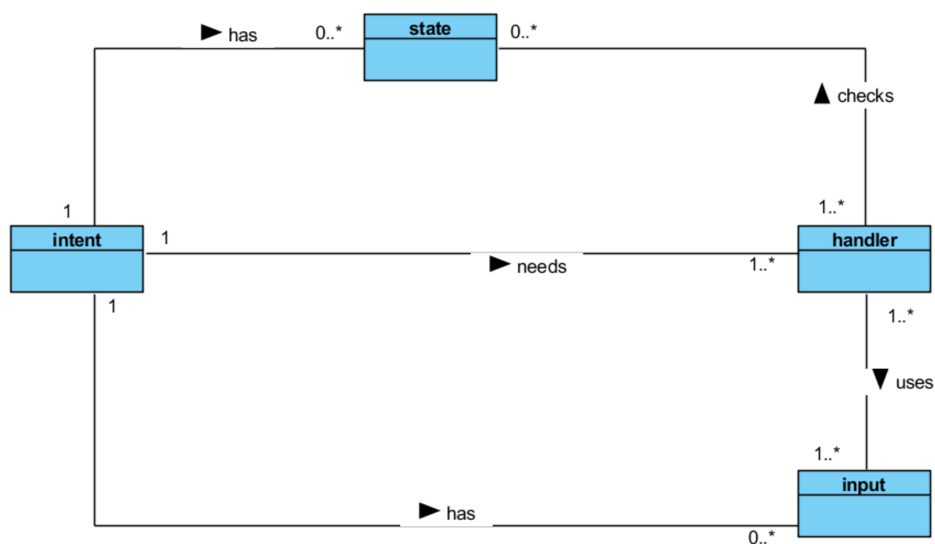


Fig. 6.: Components of the backend of a voice application

In order to better understand the similarities and differences of the assistants application models in regards to the development of the backend, a study concerning the necessary components in it and how they are related was conducted. In Fig.6, the components that are or that can be used to define a functionality (Intent) in the backend of a voice application are presented.

To develop the backend, the developer must define a handler for every Intent, that the application will provide to its users. The difference between Alexa and Google Assistant, regarding the development of the handlers, is that in Alexa they must be divided in two functions (section 2.6.2), being that one handles the activation of the Intent and the other handles its fulfillment. In Google Assistant, there is the need for only one handler, that already states which Intent it answers to. It should also be mentioned, that Jovo follows Google Assistant approach, which means that only one handler is required for the definition of each Intent.

Regarding the workflow of the handlers, they will perform the necessary logic to fulfill the Intent, that the user requested, and afterwards they will define what the application must answer to the user in a response

JSON and send it back to the digital assistant. The assistant will then tell the application's answer back to the user. The handlers can also perform the following operations:

- Read or write data to persistency;
- Make requests to external APIs, to improve the application's answer to the user's request;
- Change the conversational state of the application, if there is the need for it.

However, there are some Intents that are more complex and that should be dealt with by more than one handler (Fig.6). The secondary handlers will provide the rest of the functionality that the main handler doesn't. An example of this is when the developer is coding an Intent that needs to obtain a set of Inputs from the user, if he doesn't provide them outright.

In this case, the main handler will have to verify if any of the Inputs is missing and if there are indeed Inputs missing, it will send a question to the digital assistant, so it can ask the user for them. When the user provides the missing Inputs, a secondary handler will be activated and it will verify if they are correct and proceed to the fulfillment of the user's request. If there are still Inputs missing, the secondary handler will ask for them and wait for the user to provide them. This will happen until all the necessary Inputs are provided.

Furthermore, by separating a functionality in various handlers, the testing and debugging phase of the development process will be simpler, given that each handler will only deal with a certain part of the functionality so detecting a bug and fixing it will be easier and altering other related handlers might not be necessary. However, the developers should be careful to not separate a functionality over too many handlers.

Having a handler that can be part of the execution flow of more than one functionality is also possible. For instance, if there is a handler that deals with requests to an API and those requests can be different depending on the available Inputs, then it can be used by more than one functionality, if necessary. The answer this shared handler would send to the digital assistant would be different depending on the previous handler in order to be coherent with the current user's request.

Regarding the use of Inputs, it should be mentioned that not all Intents need them to correctly fulfill the user's request (Fig.6). The need for Inputs will depend on what the developer wants its application to provide to the user. For instance, a voice application that provides a random quote won't expect the user to provide the type of quote (Input) he wants to hear. Yet, if the developer wants to add an extra functionality, that allows the user to specify what type of quote he wants, there will be the need for an Input, that allows the specification of the type of quote.

In regards to the use of conversational states, the states in Fig.6, these are used to maintain the current context of the conversation between the user and the application. By knowing in which state the conversation is in, the application has more information about which handler should be used to answer the user's request.

An example for the use of states is when there is the need to develop handlers that are specifications of base handlers such as the Yes Intent handler, that has the business logic for when the user says "yes"

to something. Saying ”yes” to ”Do you want to know more about a product?” and saying it to ”Do you want to add three vintage frames to your cart? has different meanings to the application, due to the fact that they imply the execution of different actions and, therefore, handlers. These handlers will handle the same type of answer (”Yes”) however they handle it in different states of the conversation, one when the user is searching for a product and other when the user is adding items to its shopping cart.

With the use of conversational states, the application can handle receiving a request for the same Intent, in this case the Yes Intent, and then check which state is active in order to know which Yes Intent handler should be executed. This means that the base Yes Intent handler will never be used when a specific Yes Intent handler is needed and vice-versa.

It should also be mentioned, that when states are used there is, normally, a default state to which the application goes back to every time the more specific ones are no longer necessary. Additionally, the developer can also use states to guarantee that an Intent is only executed if a previous one was also executed. For instance, the aforementioned Yes Intent handler will only be executed after the Search Intent handler or Add product to cart Intent handler where executed and activated their corresponding states.

In conclusion, the studies conducted upon the necessary components of the backend of a voice application led to the conclusion that there is one main component that must always be defined (Fig.6). The main component are the handlers, that can either implement the business logic (Intents) or deal with the HTTP POST requests that the digital assistant sends to initiate the execution of a certain functionality (request handler). The other components that can be used are the Inputs and states.

Additionally, this study and the one presented in section 2.6.2 lead to the conclusion that both the digital assistants targeted in the construction process have the need for the same components in the backend, despite the fact that they structure it differently.

When it comes to the definition of the backend components, the specification of the frontend of the approached assistants provides information that can help in their development. The language model, that defines the frontend, has information regarding the Intents, which can be used to specify the methods, that will fulfill them, on, for example, a boilerplate code file.

Given that the construction process will use a generic language model template, that will allow the specification and generation of the frontend, it's possible to also use the information present in it to generate the boilerplate code for the initial development of the backend of a voice application. The generation of the boilerplate code will consist, for instance, in the definition of the possible conversational states that the application can be in, if the developer defined any, and in the declaration of the Intents handlers. How the boilerplate code is going to be generated will be presented and explained in detail in a following section.

4.2.3 Specification of the platform-independent construction process

The construction process will be composed by a generic language model template, which will allow the specification and generation of the frontend and the generation of the boilerplate code for the initial development of the backend of a voice application. In the specification of the frontend, more precisely, in the definition of the sample phrases of an Intent the developers will be able to use the language that they see

fit to write them, however the construction process was defined with the English (US) language in mind due to the fact that it's the one most commonly used by the assistants and their users.

It should also be mentioned that it was decided to name the language models, that the process will use and that the developers will specify, language model templates because they will serve as the starting point to the generation of the platform-specific language models and the boilerplate code.

In section 3.3, it was decided that an architecture with emphasis on separation of concerns should be used in order to minimize the repercussions that future modifications to the proposed construction process might bring due to changes on the digital assistants application models. To make the future modifications to the generic language model template more straightforward when faced with changes in the application models of the assistants, some decisions had to be taken. The first decision was related to how the template should be represented in order to achieve separation of concerns when it comes to its components. This decision is important due to the fact that we should be able to change or perfect a component of the template, without having to rewrite all the other components if there is no necessity for it.

The class diagram (Fig.7) depicts the components of the language model template and their relationships. The frameworks Jovo and Violet, weren't taken into consideration in the definition of the template components, due to the fact that Jovo's language model is inspired in the model of Alexa and that Violet doesn't have a proprietary language model and uses the models of Alexa and Assistant. The platform BotTalk, wasn't also taken into consideration due to the fact that its proprietary language model has the same components as the models of the digital assistants (e.g Intents, Inputs, etc).

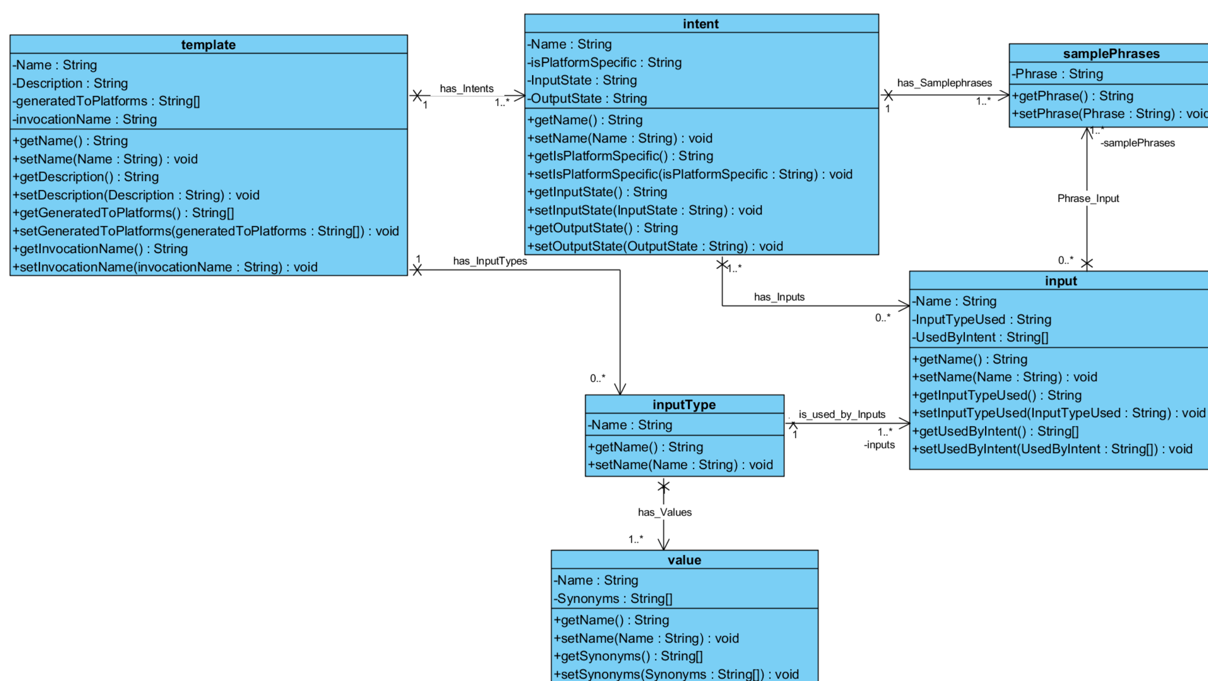


Fig. 7.: Components of the language model template

The second decision was related to how the template would be transformed into the platform-specific language models, due to the fact that, even though the assistants have the same components in their

language models, the structural rules for their output in JSON files differs among them (Table.9). It should also be mentioned, that the definition of the Jovo's proprietary language model requires one JSON file, similarly to Amazon Alexa, and that BotTalk proprietary model requires the definition of two YAML files, one to specify the Intents and other the Inputs.

In order to surpass this difference, it was decided that four different language model generators should be developed, so that the structural rules of both digital assistants, Jovo and BotTalk would be respected. This decision allows the future updates to the construction process to be less complex when faced with changes in the language models of the approached systems, given that if one model changes only its respective generator will be modified. Additionally, separating the generation of the language models in four generators improves the testing and debugging phases, because if there is a problem in the generation of a certain model it's centralized in only one area of code.

The third and last decision was related to how the template information would be used to generate the boilerplate code for the initial definition of the backend functionality. In order to generate the boilerplate code, four code generators will be implemented. There is the need for four different generators because, like in the dilemma regarding the output of the platform-specific language models, each digital assistant, framework and platform has a different approach to the codification of the backend. A code generator for Violet wasn't developed as this framework was still in beta (version 0.16) when the generators were being developed. The development of a code generator for Violet will be left as future work.

The boilerplate code will be generated for JavaScript (Node.JS) given that it's supported by both assistants via a SDK and is the programming language used by Jovo.

Regarding BotTalk, this platform doesn't require the coding of the backend using a programming language but instead uses a proprietary markup language (section 2.4) to define two YAML files, that specify the business logic and how it should be tested, respectively. Therefore, the boilerplate code, in this case, will be generated for the BotTalk markup language and not for JavaScript (Node.JS).

Through the development of a generation process where the same generic template can be fed to different generators and those can be executed separately (Separation of concerns), it's possible to extend the number of supported systems in the future. This extension would happen by adding more generators as long as they could work with the template or the template could be altered without forcing major changes to the existing generators.

After having defined how the future modifications to the construction process could be made less complex when faced with changes in the digital assistants application models, additional decisions were taken before moving on to the implementation phase.

One of the motivations of this dissertation is that the construction process should be easy to use for both technical and non-technical users. Consequently, it was decided that a graphical user interface should be provided in order to abstract the definition process of the language model template and the generation of the frontend and of the backend boilerplate code.

Therefore, to accomplish this decision, a platform, with a visual editor incorporated, will be developed. This platform was denominated VoicePrint, given that the language model templates will serve as blueprints to the definition of voice applications. The requirements of VoicePrint are presented in annex A.1.

Knowing now what must be implemented, the construction process and the platform that will abstract it's use, the programming language that would be used had to be decided. The programming language chosen was Java, due to the fact that it's a solid and mature language and the familiarity with it will also ease the development process.

Additionally, this language is also widely used in the technological market and it allows the development of VoicePrint for more than one operating system. It should also be mentioned, that another programming language could have been chosen as long as it was suitable for the development of the construction process and allowed the development of the VoicePrint for more than one OS.

There was also the need to decide how the generic template was going to be represented in persistency. Regarding this decision, it was decided to use a File database with JSON files to represent the templates in persistency, due to JSON high expressiveness and due to the fact that it can be efficiently parsed to code and vice-versa.

In Java, there are a plethora of libraries that can help with the serialization and deserialization of JSON files (e.g Javax) and they will be useful to implement the persistency of the templates and the generation of the JSON files that will contain the platform-specific language models for the digital assistants and Jovo.

Lastly, it was necessary to decide how the code generators were going to be developed. Given that the code generators only had to generate boilerplate code, a template for the developers to initiate the development of the backend, whose structure was already known, it was decided to use Apache Velocity to generate it.

Apache Velocity is a Java-based template engine that allows the generation of source code, disregarding which programming language the code is targeted to, and the use of Java statements and methods to, for instance, extract the information necessary for the execution of the generation process.

Therefore, Velocity allows the use of the generic language model template to generate the platform-specific boilerplate code in an easy and powerful manner, due to the possibility of using statements such as "for each" to ease the coding of equally structured information on the generated code.

It should also be mentioned that Velocity will be used to generate the BotTalk language model. Such is due to the fact that the BotTalk model needs to be specified in YAML, with a somewhat rigid structure, and using this template-engine to specify it proved to be more swifter than using a JSON to SnakeYAML library.

4.3 Implementation

The decisions that were made led to the definition of a tool that will help in the definition of the frontend, the high-level activity diagrams, and to the definition of the necessary components for the development of the proposed platform-independent construction process. Furthermore, those decisions also led to the definition of a platform, with a visual editor incorporated, that will abstract the use of the construction process.

In this section, the general implementation of the decisions will be presented.

4.3.1 High-level activity diagrams

The purpose of the UML activity diagrams is to allow the developers to model the control flow of an application, which means that they can be seen as flowcharts that represent the flow from one activity to another one. Such purpose made the activity diagrams adequate candidates for the development of a tool, that allows the definition of the conversation model of a voice application. These diagrams will be high-level due to the fact that they will abstract the overall flow and logic of the conversation between the user and the application.

In the high-level UML activity diagrams the developers will specify: the Intents, that the application will offer to the user; a sample phrase, that leads to the fulfillment of the Intent; and the way that the application allows the user to go from one Intent to another (the main flow of conversation). For instance, the Intent that welcomes the user into the application will, normally, be the first Intent that the user will encounter when he opens the application. In terms of conversation flow, this Intent will be the starting point for all the other Intents that the user can invoke, given that, usually, only after its execution will the user express his request.

Additionally, the activity diagrams will also allow the developer to specify the interactions between the user and the application during the fulfillment of an intent in a high-level way. This will allow the developer to specify what he wants his application to do when faced with a certain user's request.

In this more detailed activity diagram, the focus of the developers shouldn't be on detailing all the users possible next steps or, for instance, the ways that they can answer a certain question. The developers should be more focused on defining what the application will say to the user throughout the fulfillment of its request, for instance, in what regards to the questions or responses that the application will say to the user.

Furthermore, the developer should also be focused on the possible interruptions that can happen while both the user and the application are engaged in the execution of a functionality. One of those interruptions can be when the user says something that needs to "trigger" the application to direct him to the error handling functionality or to cancel the current user's request.

In order to demonstrate how the activity diagrams should be used in order to achieve their full potential, it was decided to show the high-level activity diagrams that were developed for two voice applications, that will be presented in detail in chapter 5. The first voice application will have a more conversational aspect and will allow the user to shop for items on the Etsy e-commerce website. The second voice application will have a more request-response aspect and will allow the user to ask for the daily news or headlines about a certain topic, among other options.

The activity diagram (Fig.8) presents the main menu of the aforementioned e-commerce voice application that, for instance, lets the user search for a product on the Etsy website, manage its cart and calculate its total price, and also check the user's current orders status. By developing this activity diagram, what the user can do in the application was specified without having to detail all its possible steps and the main flow of conversation between the user and the application was also shown. For instance, an user can go to the Search product Intent after being welcomed into the application and to demonstrate it, it wasn't

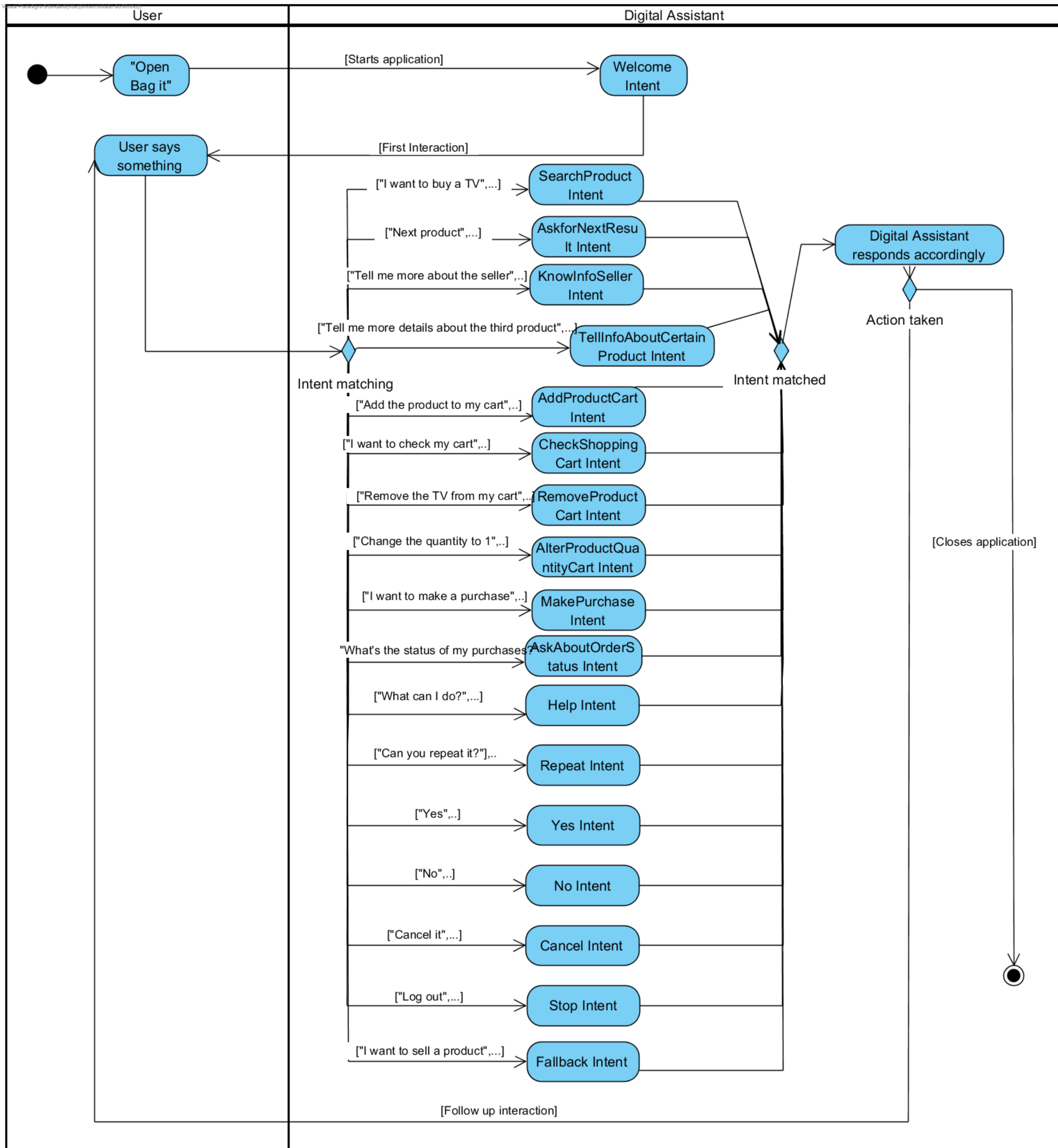


Fig. 8.: Main conversation flow between the user and e-commerce application

necessary to specify that before he might, or not, have asked for another Intent (e.g the Ask about order status Intent).

Therefore, this activity diagram allowed the specification of the functionalities that the application will offer to the user, without having to make prior commitments of which paths must necessarily be followed in order for a user's request to be fulfilled.

The inner path of the Search product Intent, that corresponds to the flow of conversation between the user and the application during its fulfillment, was specified by developing a more detailed activity diagram (Fig.9).

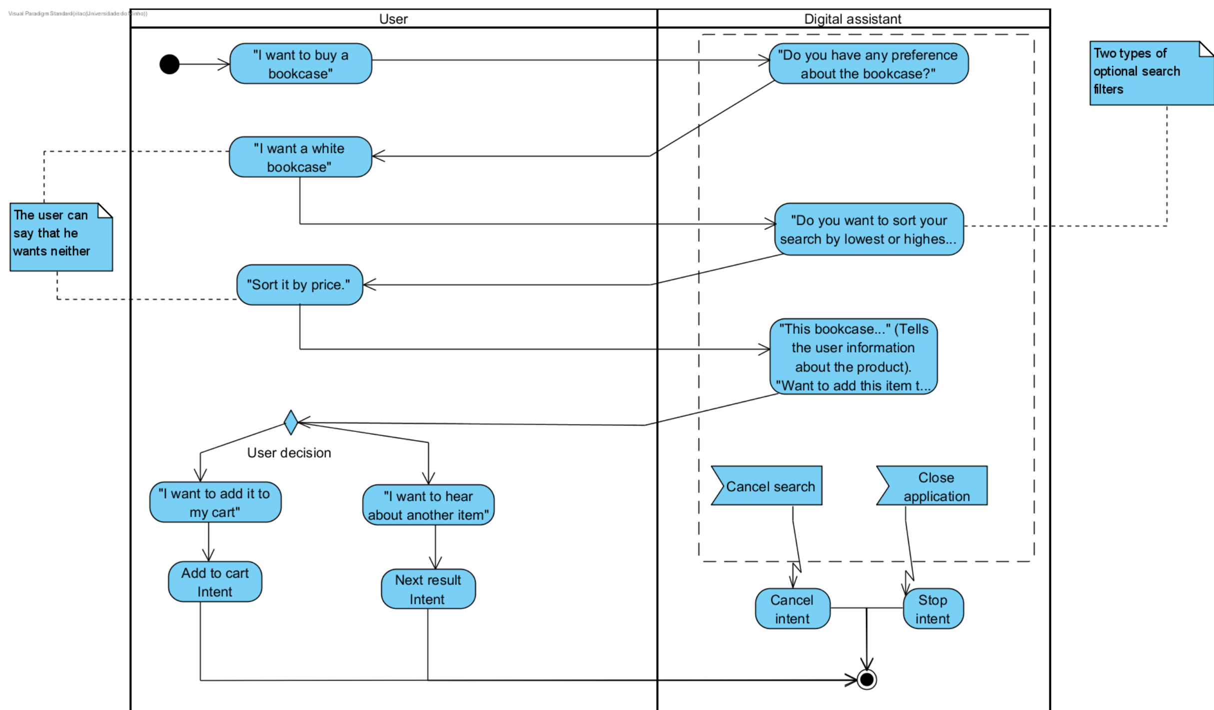


Fig. 9.: Search product Intent flow

This application will have a conversational aspect given that in order to fulfill the user's request to search for a certain product, a conversation will have to be established between them. For example, the application needs to talk with the user in order to know if he wants to sort its search with a filter and, after the search process ends, if he wants to add one of the presented items to the cart or if he wants to hear the next results.

This conversational aspect is visually explained by the activity diagram in Fig.9, where it was specified the logical steps of the functionality, what must be asked to the user, the possible ways the user can change functionality (e.g add item to cart or get the next result of the search process) and that the user can cancel its request or close the application at anytime (the two possible interruptions that are represented by signals).

The activity diagrams related with the user's request of adding the item to the cart and of asking the digital assistant for the next result of the search process are also presented in Fig.10 (Add product cart

Intent) and in Fig.11 (Ask for next result intent), respectively. It was decided to present those two diagrams in order to show what can happen after the user makes a decision at the end of the Search product Intent.

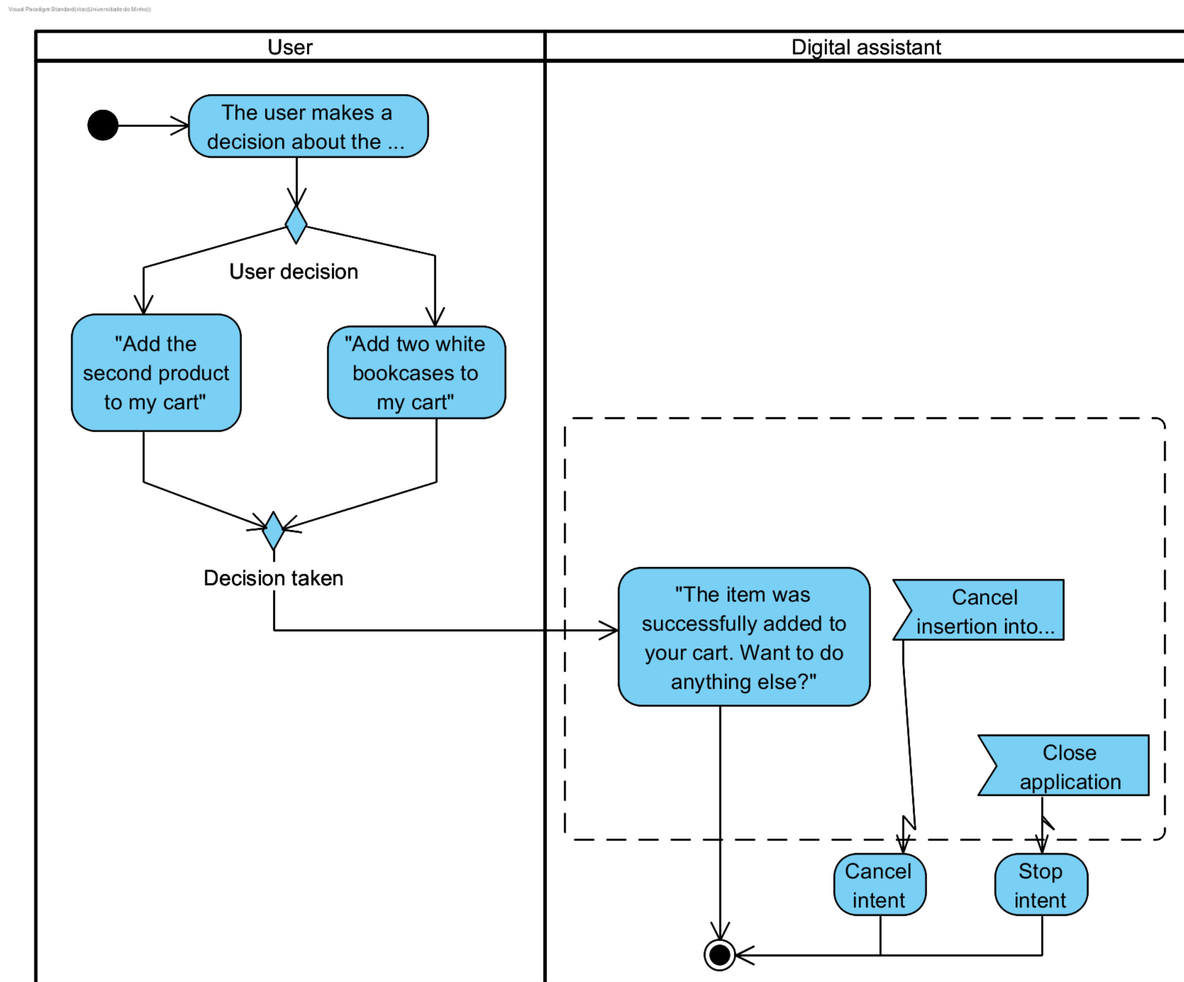


Fig. 10.: Add product cart Intent flow

Through the definition of activity diagrams such as the one presented in Fig.10, the developer already knows, for instance, if the application funcionalities are going to need user arguments to fulfill its request, and, therefore, if the user responses are going to contain arguments (Inputs) or not (e.g. the minimum or maximum price that the user is willing to spend). As such, the specification process of the language model template will be more straightforward.

The activity diagram of Fig.12 presents the main menu of the news voice application, This application will, for instance, let the user ask for news about a certain topic or the headlines of today's news.

Even though, this voice application is conceptually different from the e-commerce application presented above, the main conversation flow of both of them can be represented by akin high-level activity diagrams. Such is due to the fact that the fundamental idea behind their diagrams is the same, which is to abstract the conversation flow, show the application functionalities and, for example, how the user can go from one functionality to another.

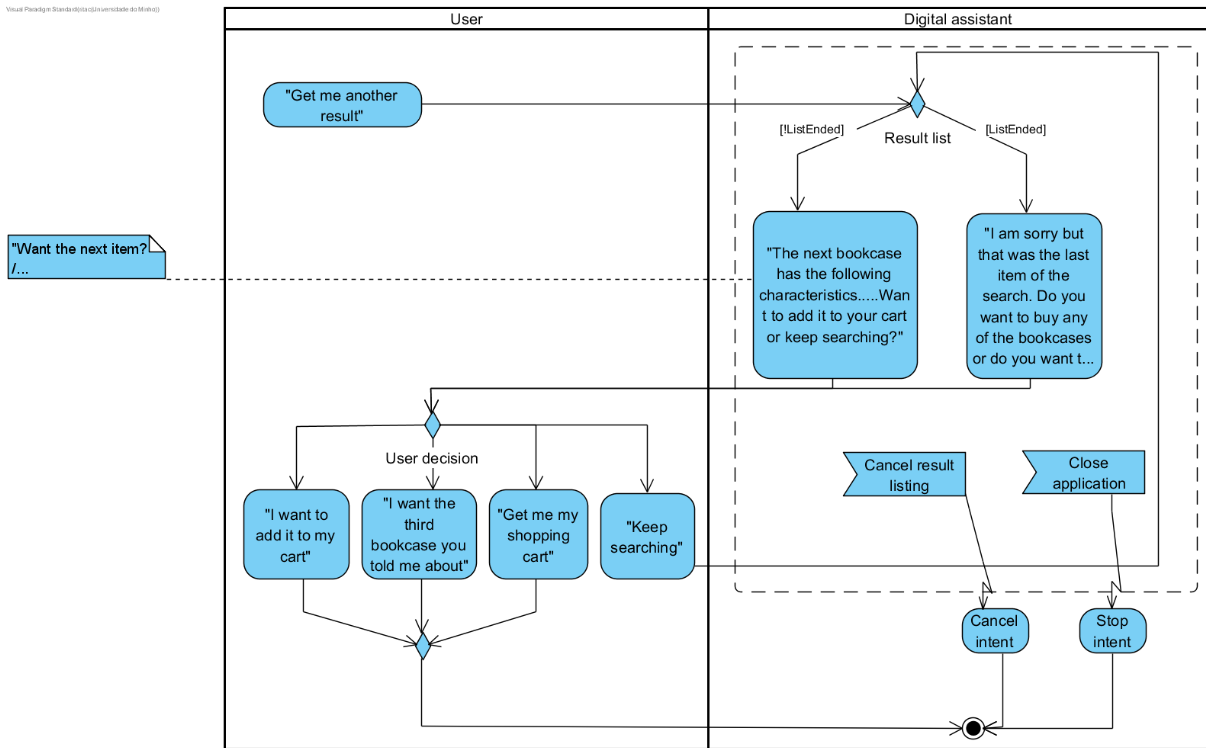


Fig. 11.: Ask for next result Intent flow

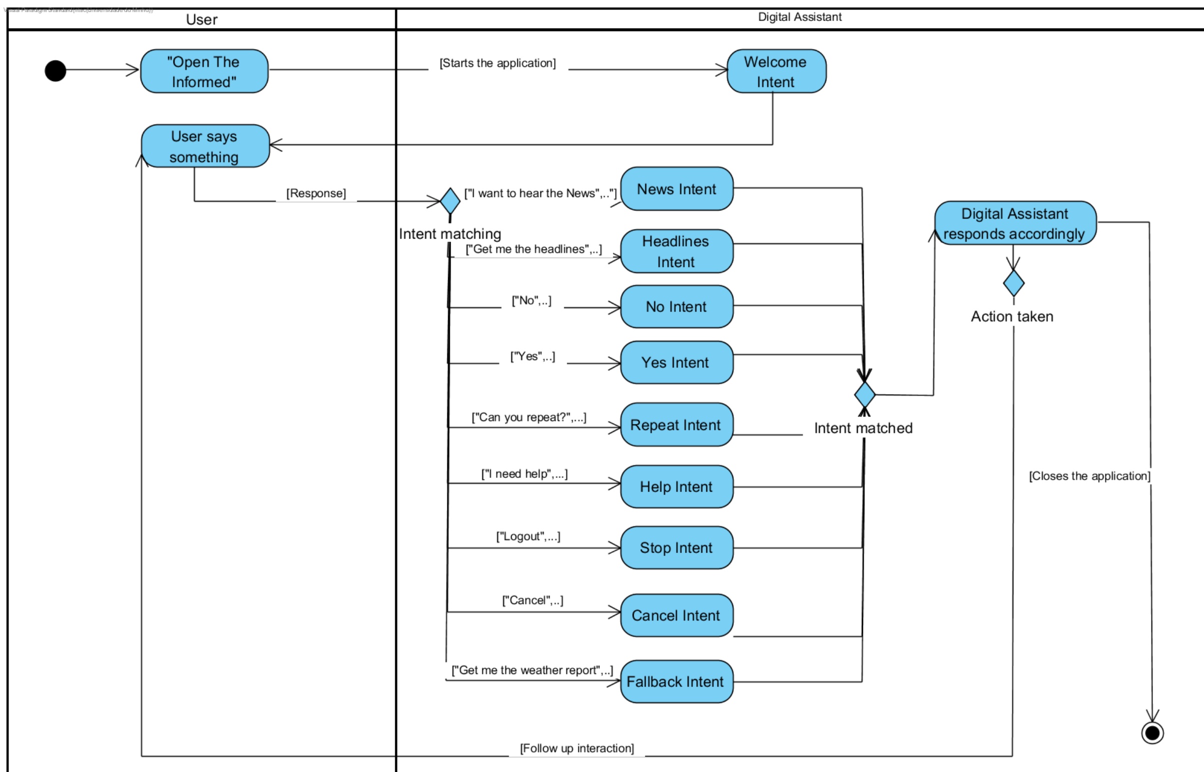


Fig. 12.: Main conversation flow between the user and news application

The differences between the diagrams of both applications consists in the specification of their main functionalities, for instance, the e-commerce application will have the search functionality (Fig.8) while the news application will have the news functionality (Fig.12). Their remaining functionalities can be seen as the base ones, that should always be present in every voice application. The base functionalities are the following:

- Yes Intent;
- No Intent;
- Help Intent;
- Repeat Intent;
- Welcome/Launch Intent;
- Cancel Intent;
- Stop Intent;
- Fallback Intent;

The activity diagram (Fig.13), that presents the inner path of the News Intent, can be considered more succinct when compared to the one presented in Fig.11. Such is due to the fact that this application is of the type request-response, which means that its interactions with the user will be straightforward and short-lived, given that the user will request something of the application and the later will provide him with an adequate answer.

The news voice application interactions with the user will consist on him either asking for today's news/-headlines or for news/headlines about a certain topic or from a certain day. The only other required intervention from the user is when he needs to accept or decline the application offer to hear the remaining news/headlines. This happens due to the fact that only three pieces of information, in this case news or headlines, are presented, in each iteration, in order to preserve the user's short-term memory.

In conclusion, the high-level activity diagrams, proposed in this master's dissertation, are divided in two types:

- Activity diagrams that present the main flow of conversation between the user and the application and the application functionalities;
- Activity diagrams that present the inner flow of conversation between the user and the application when a user's request is being fulfilled.

Regarding the development of the first type of diagrams, as it can be verified in Fig.8 and in Fig.12 and as it was mentioned previously, this diagram is similar from one voice application to another. This is due to the fact that, normally, the workflow of voice applications is the same despite them offering different functionalities.

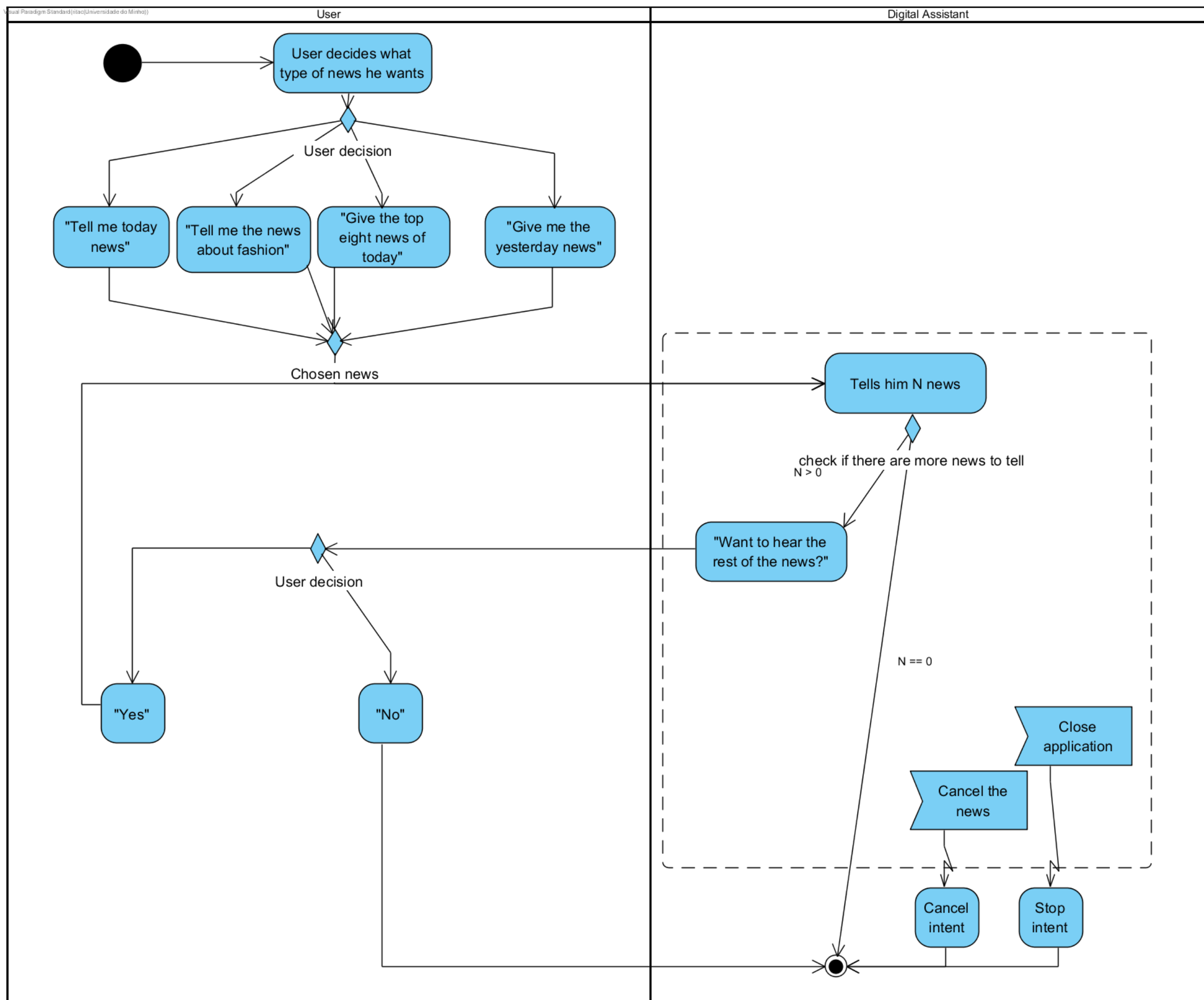


Fig. 13.: News Intent flow

The general workflow of a voice application consists in the user opening the application, the welcome Intent executes and then the user can state its request(s) and the application will fulfill it. This workflow will be interrupted when the user invokes the Stop Intent, that doesn't have a follow up interaction but instead leads to the shutdown of the application.

Regarding the developer's perspective, when he is developing this type of activity diagrams, he will know that there is a flow "template" that he can use, in result of the general workflow of voice applications, and that in that "template" he only needs to change the functionalities (Intents) region of the diagram to his application functionalities.

In what regards the development of the second type of diagrams, as it can be verified in Fig.9 and in Fig.13, this diagram is very distinct from one voice application to another. This is due to the fact that, usually, every functionality needs to perform and say different things in order to fulfill the user's request. The type of the voice application also influences the inner flow of the functionalities, given that the complexity of a request-response application is usually lower than that of a conversational application.

In regard to the developer's perspective, he will have more freedom in the specification of this type of activity diagrams, without disregarding their structural UML rules. However, there are some guidelines that he can use, such as, using signals to show that the Cancel and Stop Intent can be invoked at anytime by the user in order for him to be able cancel its request or to stop the application execution, respectively.

4.3.2 Language model template

The generic language model template is the core component of the proposed construction process. This template is going to be represented as a Java object, whose components are going to be Java objects that each correspond to a necessary component of the platform-specific language models of the approached digital assistants. How the template and its components are interconnected is explained in Fig.7.

During the development of VoicePrint, the platform that is going to abstract the specification of this generic template, it was decided to not only allow the users to create and specify their language model templates in it, but also provide them with examples of those templates in the platform. The examples will help the users learn how to use VoicePrint for the specification of a language model template and also to understand what can be done with the platform.

Due to the fact that there will be two types of templates, one that refers to the templates that the user will define and other that refers to the example templates that are going to be provided by default, the UML class diagram presented in Fig.7 had to be expanded in order to distinguish them.

As it can be verified in the class diagram (Fig.14), an inheritance relationship between the templates was defined, given that the only main difference between the default and the custom templates is going to be that the users won't be able to edit the first ones.

All the remaining core information and how the templates are composed (e.g Intents, Input types, etc) is going to be identical for both of them. However, the default templates are going to have a field that indicates their version while the custom templates will have two fields, one that indicates their creation date and other that indicates their modification date.

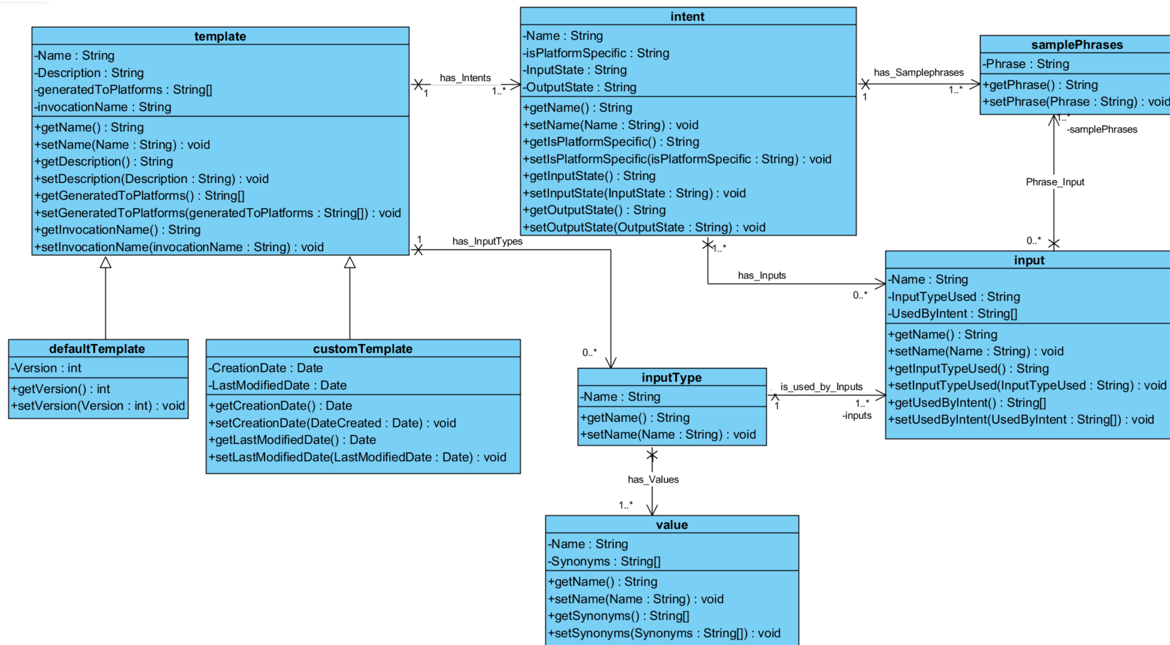


Fig. 14.: Components of the generic language model template - Two types of template

It should also be mentioned that even though the default templates will be read-only, it will be possible to use them to generate platform-specific language models and/or boilerplate code and also deploy them.

Regarding the persistency of the generic template in JSON files, the difference between the custom templates and the default templates resides on where they are going to be stored, given that they will be persisted in separate directories.

Listing 4.1 shows a default template regarding a voice application that provides the user with information about the INforum symposium [16] and the listing 4.2 shows a custom template regarding a voice application that allows the users to shop at the e-commerce website Etsy (Chapter 5). It should also be mentioned that due to the length of both listings it was chosen to only show them partially, however snippets of all the components of the templates are presented.

```
"_Version": 1.0
"_Name": "INForum2019",
"_Description": "Voice application for the Inforum symposium",
"_GeneratedToPlatforms": ["All"],
"_InvocationName": "inforum guide",
"_Has_Intents": [{
    "_Name": "ScheduleIntent",
    "IsPlatformSpecific": "All",
    "InputState": "",
    "OutputState": "Schedule",
    "_Has_Samplephrases": [{"_Phrase": "tell me what is going to be presented at {time} on
        the {number} day"},(...)
    ],
    "_Has_Inputs": [
      {
        "_Name": "number",
        "_InputTypeUsed": "Ordinal",
        "_UsedByIntent": [
          "ScheduleIntent"
        ],
        "_Phrase_Input": [{"_Phrase": "what is going to happen on the {number} day"
          },(...)
        ]
```

```
        },
        (...),
         "_Has_InputTypes": [{
              "_Name": "LocationsINForum",
              "_Has_Values": [{
                      "_Name": "Auditorium",
                      "_Synonyms":[]
                    },
                    {
                      "_Name": "Classroom 3.7",
                      "_Synonyms": ["Room 3.7"]
                    }
                  ],
              "_Inputs": [{
                  "_Name": "location",
                  "_InputTypeUsed": "LocationsINForum",
                  "_UsedByIntent": ["ScheduleIntent"],
                  "_Phrase_Input": [{"_Phrase": "what is the schedule for {location}"
                    },(...)
                  ]
                }
              ]
       },(...)
}
```

Listing 4.1: Default language model template

```
{ (...)
  "_Name": "BagIt",
  "_Description": "Template for the bag it voice application",
  "_GeneratedToPlatforms": ["All"],
  "_InvocationName": "bag it",
  "_Has_Intents": [{
      "_Name": "AskAboutOrderStatusIntent",
      "IsPlatformSpecific": "All",
      "InputState": "",
      "OutputState": "Order",
      "_Has_Samplephrases": [{"_Phrase": "Give me a status update on my open orders"},(...)],
      "_Has_Inputs": []
    }, (...)
    "_Has_InputTypes": ["_Inputs": [{
          "_Name": "numberOfItems",
          "_InputTypeUsed": "Number",
          "_UsedByIntent": ["AddProductQuantityCartIntent"],
          "_Phrase_Input": [{"_Phrase": "I want to buy {numberOfItems} items"
            },(...)
          ]
        },(...)
    ]
}
```

Listing 4.2: Custom language model template

How these listings are used by the generators to generate the platform-specific language models and the boilerplate code for the initial development of the backend will be explained in the following section.

One of the goals and challenges of this dissertation is allowing the specification of platform-specific functionalities and the use of built-in Intents and Input types, that the digital assistants provide, in the generic template and, therefore, in the construction process. For instance, if a functionality is going to be platform-specific, this will have to be specified in the generic template and the list of built-in Intents and Input types will have to be kept in persistency in order for them to be used in VoicePrint and, if the developer chooses so, defined in the template.

Regarding the platform-specific functionalities, it was decided to define the IsPlatformSpecific field in the Intent component of the template (Fig.14) for their specification. This field will allow the generators of the platform-specific models to know if the Intent must be present in the model that they are generating or not.

In regard to the Intents, a study concerning the built-in Intents that the targeted digital assistants make available to the developers and the Intents whose specification is mandatory in the voice applications of the targeted assistants was conducted. It should be mentioned that the built-in Intents that are going to be provided are for the English (US) language given that, as it was stated in section 4.2.3, this construction process was developed for the most common language used by the digital assistants and their users. Providing built-In Intents for other languages will be left as future work. The Intents gathered during the conducted study are presented in table 11 and table 12.

Concluded the study concerning the Intents, it was decided that the proposed construction process should provide the mandatory and the general built-in Intents of each digital assistant. Choosing which Intents to provide consisted in first checking the mandatory Intents of each assistant and then verifying if they could be supported by both of them. The Fallback Intent was the only one that was mandatory in both Alexa and Google Assistant.

Regarding the remaining mandatory Intents, the ones that are mandatory in Alexa are also supported by Google Assistant. However, there are two exceptions, being that the first one concerns the Welcome Intent, that is required only by Google Assistant and that enters in conflict with the way Alexa designed the launch of its voice applications. For Amazon Alexa, the developers don't have the need to specify launching phrases in order for the users to be able open the application, due to the fact that Alexa already does that for them, being that the only thing that they need to provide is an invocation name.

In order to provide the Welcome Intent for Google Assistant without creating a conflict with Amazon Alexa, it was decided that this Intent would be platform-specific by default, which means that the users will only be able to specify it for Google Assistant. Additionally, it was also decided to provide this Intent automatically when the users create a custom template in VoicePrint.

The second and last exception concerns the Navigate home Intent, that is only required by Amazon Alexa. This Intent is only available to voice applications that are running on devices that have screen support and its execution is automatically handled by Alexa. Due to the fact that this Intent is rather specific to Alexa and to a certain type of device, it was decided to apply the same strategy that was taken with the Welcome Intent and provide this Intent as platform-specific by default and automatically when a custom template is created in VoicePrint.

The Help, Cancel and Stop Intents are only mandatory in Alexa yet they are supported by both digital assistants. Therefore, it was decided that, unlike with the Welcome and Navigate home Intents, these Intents didn't needed to be platform-specific by default.

The last decision taken regarding the mandatory Intents was that they must always be present in the language model templates that the users create. Consequently, when the user decides to create a custom template the aforementioned Intents will be automatically created with it. The Welcome and Navigate

home Intent will already be marked as platform-specific to the correct assistants and the user won't be able to change it, in order to conform to the aforementioned decisions regarding these two Intents.

Regarding the general built-in Intents, they refer to Intents whose specification is not mandatory in the approached digital assistants voice applications. As it can be verified in tables 11 and 12, the built-in Intents of Google Assistant were designed towards more specific situations and voice applications. For instance, the built-in Intent Get Horoscope will most likely only be present in a voice application whose focus is astrology.

The built-in Intents of Alexa were designed to be more generic and are well suited to be in a myriad of voice applications despite their type (e.g news, education, etc). Thus, it was decided to only provide the general built-in Intents of Alexa as platform-independent functionalities, along side the aforementioned mandatory Intents, in the proposed construction process. Such is due to the fact that those built-in Intents are useful to a more variety of voice applications. It should also be mentioned that, at the time this master's dissertation was developed, only the Google Assistant built-in Intent Play Game was available, given that the rest of these assistant built-in Intents were still in developer preview.

Concerning the built-in Input types, a similar study to the one performed upon the built-in Intents, provided by the digital assistants, was conducted. Contrary to the built-in Intents, the digital assistants don't obligate the developers to use any specific Input Type in their voice applications, when they are defining the type of an Input. It should be mentioned that the provided built-in Input types, similar to the built-in Intents, are for the English (US) language. Providing built-in Input types for other languages will be left as future work.

Due to the fact that the list of Input Types gathered during the conducted study was quite lengthy it was decided to present a few samples in the tables 14 and 15. For the complete list of built-in Input types for Amazon Alexa consult [7] and for Google consult [32].

It should also be mentioned that, when the support for the platform BotTalk was introduced in the proposed construction process, it was found that this platform also provided some built-in Input Types. Those built-in Input types are presented in table 13.

Finished the study concerning the built-in Input Types, it was concluded that some of them were either provided by both digital assistants or could be cross-referenced, which, normally, meant that although they had different denominations in the assistants they referred to the same subject. However, the number of correspondences between the digital assistants built-in Input types was low.

In order to provide the users with more options and, therefore, reduce the number of potential Input Types that they would have to define by hand, it was decided that the built-in Input Types that didn't have a correspondence in one of the assistants would be corresponded with the generic type of the other assistant. The generic Input Type is a type that can capture any less-predictable non-empty input. In Alexa this type is AMAZON.SearchQuery while in Google Assistant this type is @sys.any.

By making this decision, the users will have more available Input Types to chose from, in the VoicePrint platform, when they are creating an Input. Additionally, the users won't have to worry about having to define those Input types values and synonyms because the digital assistants already have that information.

Nonetheless, there was a setback regarding the use of the Input type AMAZON.SearchQuery due to the fact that this type, contrary to the @sys.any type, cannot be used in sample phrases combined with other Input Types and each Intent can only use one Input with this type. These rules would make the use of the Google Assistant built-in Input types that were corresponded with this type, very rigid, because they would have to obey to those same rules. For these reasons, it was decided to only perform the aforementioned correspondence process unilaterally, which resulted in corresponding the Amazon built-In Input Types, that aren't present in Google Assistant, with @sys.any.

In regard to the built-in Input types provided by BotTalk, given that they are supported by both digital assistants, it was decided that they would also be supported by the proposed construction process.

A future improvement concerning the previously presented correspondence process would be to find a way to also provide the built-in Input Types of Google Assistant, that aren't supported by Alexa, without having to correspond them with the AMAZON.SearchQuery Input type, which is not an option at the moment.

Lastly, another goal of this dissertation is allowing the users to define input and output conversational states for an Intent. By developing this feature, the users will be able to define beforehand, in the template, if there is a certain state of the conversation that needs to have been reached (input state) for the application to be able to fulfill the user's request or if the fulfillment of a request leads to the activation of a certain state (output state) in the application.

This feature is related with the generation of boilerplate code, given that the skeleton code will include the list of defined conversational states. These states will also appear in the skeleton code of the Intents methods that use them, whether they are used as input or output states. The way that the conversational states and the boilerplate code of the Intents methods are related varies between the systems approached in this dissertation and it will be presented in section 4.3.4.

A future improvement regarding the use of input and output conversational states is allowing the users to define a list of input/output states for an Intent. Such would allow the definition of nested states, which translates in only being able to execute an Intent if all its input states are active, in the boilerplate code.

4.3.3 Platform-specific language model generators

When the developer decides to generate a language model template, whether it's a custom or default one, the general steps of the generation process will be the same. First, its necessary to see for which platform(s) the developer wants the template to be generated to. The developer will be able to chose if he wants the template to be generated for an individual platform or for all of them.

Afterwards, the template will be passed to each required generator(s), where it will be processed accordingly to the structural rules of each platform, the existency of platform-specific functionalities and the use of built-in Intents or Input Types. When the processment of the template by the generator(s) reaches its final step, the platform-specific language model will be written into one or more JSON/YAMLL files, depending on what is required by the chosen platform(s).

It should also be noted that, due to the hierarchy established between the templates (Fig.14) it was possible to use polymorphism during the development of the generators. The application of polymorphism materialized in passing the parent object Template to the generators instead of its children, Custom or Default templates. This approach was chosen because it was more advantageous to use the parent class (Template) in the coding of the generators than using the child classes (the Custom and Default template), given that the way they were developed wouldn't influence the way the generators had to be coded. Furthermore, using polymorphism also allowed the improvement of the code readability and maintainability.

Regarding the technologies chosen for the development of the platform-specific language model generators, the ones that were chosen were Java, Javax Json and Apache velocity. The generators were coded in Java due to the fact that it's the programming language that is being used to develop the construction process and VoicePrint (section 4.2.3).

Apache velocity had to be used to generate the language models for the BotTalk platform , due to their rigid structure in YAML (section 4.2.3). Javax Json was chosen to generate the JSON files due to the integration that Javax has with Java and the straightforward way that it allowed the specification of the language model template (a Java Object) to a JSON file, that will contain the generated platform-specific language model.

Alexa language model generator

In regard to the generation of the Amazon Alexa language model, as it was previously mentioned, this digital assistant only requires one JSON file for the specification of the model. The development of this generator can be considered straightforward given that, mostly, it only required a meticulous specification of the model components in their correct order in the Javax Json Object.

Additionally, the generator also had to verify if any platform-specific functionalities had been defined (Listing 4.3) or if built-in Intents or Input Types had been used (Listing 4.4 and Listing 4.5 respectively). In regard to the platform-specific functionalities, if there are any functionalities that are specific to Google Assistant they won't be specified in Alexa language model.

Regarding the built-in Intents, the generator will have to add to them the prefix "Amazon." in order for them to be recognized by Alexa and also verify if the developer didn't defined any Inputs in their sample phrases, given that these Intents cannot have them.

In regard to the built-in Input Types, the generator will verify which type is being used, in order to get the corresponding Amazon Input Type, and will also prevent their definition, in terms of values and synonyms, in the language model. Such is due to the fact that Alexa built-in Types cannot be redefined, only extended.

Allowing the developer to define extensions for the Alexa built-in Input Types will be considered as future work, due to the fact that it that wasn't considered a fundamental feature for the proposed construction process at the moment.

```
@Override
public void templateGenerator(template temp, String filePath) throws Exception{(...)
//Json array that contains all the intents
JsonArrayBuilder intents = Json.createArrayBuilder();

(...)
```

```
for (intent its : temp.get_has_Intents()) {
    try {
        //if the intent is not google specific then we can write it
        if (its.getIsPlatformSpecific().toLowerCase().contains("all") || its.
            getIsPlatformSpecific().toLowerCase().contains("amazon")){(...)}
    }(...)
}
```

Listing 4.3: Alexa - Checking for the existence of platform-specific functionalities example

```
@Override
public void templateGenerator(template temp, String filePath) throws Exception{
(...)
//If the intent is a builtIn intent we need to add the "Amazon." prefix
if(builtInInformation.builtInIntents.contains(its.getName())){ it.add("name","AMAZON."+its.
    getName());}
else{it.add("name", its.getName());}

//The builtIn intents cannot contain slots (and their sample phrases if they exist)
if (!builtInInformation.builtInIntents.contains(its.getName())) {
    JsonArrayBuilder slots = Json.createArrayBuilder();
    (...)
    it.add("slots", slots);
}
(...)}
```

Listing 4.4: Alexa - Checking for the use of built-in Intents example

,

```
@Override
public void templateGenerator(template temp, String filePath) throws Exception{
(...)
JsonArrayBuilder slots = Json.createArrayBuilder();

//If this intent uses slots
if (!its.get_has_Inputs().isEmpty()) {
    //For each slot we need to create a JSON object that represents it
    for (input ipts : its.get_has_Inputs()) {
        JsonObjectBuilder slot = Json.createObjectBuilder();
        slot.add("name", ipts.getName());

        // If the input is using a built_in input type we need to get the corresponding amazon
             type
        if (builtInInformation.builtInInputTypes.containsKey(ipts.getInputTypeUsed())) {  slot
            .add("type", builtInInformation.builtInInputTypes.get(ipts.getInputTypeUsed()).get
            (0));
        } else {slot.add("type", ipts.getInputTypeUsed());}

        (...)}
}
(...)}
```

Listing 4.5: Alexa - Checking for the use of built-in Input Types example

Google Assistant language model generator

Regarding the generation of the Google Assistant language model, as it was previously mentioned, this digital assistant requires two JSON files for each Intent and Input Type that the developer defined. This assistant also demands that the JSON files are stored in a rigid directory hierarchy (Fig.15) and that the

main directory must be zipped, when the creation of the model ends. The zip of the language model is called Agent by Google Assistant and will be what the developer will provide, or by hand or by deploying the voice application via VoicePrint, to the DialogFlow console in order to create and specify the frontend of the application.

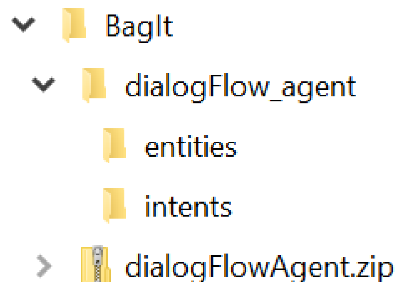


Fig. 15.: Google Assistant directory hierarchy

The development of this generator wasn't as straightforward as the development of the previously presented generator, due to the structural rules that Google Assistant imposes in terms of the hierarchy of the directory, where the language model will reside, the specification of the Intents and Input Types and the required JSON configuration files. Once more, Javax JSON Objects were used to create and specify the required JSON files.

The first step taken towards the development of this generator consisted in defining a method that will create the main directory ("dialogFlow_agent") followed by the creation of the agent and package JSON configuration files and then the creation of the two directories that will contain the Intents ("intents") and the Input Types ("entities").

The second step consisted in developing a method that will create the two JSON files that are necessary to specify an Intent. The first JSON file consists on the global description of the Intent, where informations such as its name, if it needs to call an webhook to be fulfilled, if it uses parameters (Inputs), among others, are defined. The second JSON file consists on the definition of the sample phrases, that lead to the Intent activation, and the Inputs that are required from the user for its request to be fulfilled and their type.

The third step consisted in developing a method similar to the one presented above but that is focused on creating the two JSON files that are necessary to specify the developer defined Input Types (entities). The first JSON file consists on the global description of the Input type, where informations such as its name or if it can be overridden by a built-in Input type, among others, are defined. The second JSON file consists on the values, that the Inputs that use this type can take, and their synonyms, if the developer defined any.

The fourth and last step consisted in developing a method that will zip all the aforementioned directories and files so the developer will have everything ready for submission to the DialogFlow console, once the generation process is completed.

Analogous to what was done in the Alexa language model generator, the methods, that constitute this generator, also had to take into consideration the definition of platform-specific functionalities (Listing 4.6) and the use of built-in Intents and Input types (Lisitng 4.7 and Listing 4.8 respectively). Concerning the

platform-specific functionalities, in case that there are any functionalities that are specific to Amazon Alexa, they won't be specified in Google Assistant language model.

Regarding the built-in Intents, due to the fact that Google Assistant doesn't has any special denomination for them like Amazon does nor it requires the specification of a particular set of information, that labels the Intents as built-in, it was only needed to verify if the Intent was built-in or not. If the Intent is a built-in one there won't be the need to specify any Inputs, given that these type of Intents don't use them.

In regard to the built-In Input Types, the generator will verify if they are being used in the custom Intents in order to specify the Input, in the general description of the Intent and in its sample phrases, with the correct Google Assistant type. Additionally, the generator will also prevent the definition of the built-in Input Types in these language models, given that Google, at the time this master's dissertation was developed, didn't allow them to be redefined.

```
(...)
// If the intent is a custom intent and not specific to amazon then we can write it
if(it.getIsPlatformSpecific().toLowerCase().contains("all") ||  it.getIsPlatformSpecific().
    toLowerCase().contains("google")) {
    try{ templateIntentsGenerator(it, filePath + "intents"); }catch (Exception e){(...)}
}(...)
```

Listing 4.6: Assistant - Checking for the existence of platform-specific functionalities example

```
(...)
JsonArrayBuilder param = Json.createArrayBuilder();

//if the intent is a built_in intent or a fallback intent then he won't have any inputs
if (!builtInInformation.builtInIntents.contains(it.getName()) && !it.getName().toLowerCase().
    contains("fallback")) {
        if (!it.get_has_Inputs().isEmpty()){(...)}
}
resp_params.add("parameters", param);
(...)
```

Listing 4.7: Assistant - Checking for the use of built-in Intents example

```
private void templateIntentsGenerator(intent it, String filePath) throws Exception{
(...)
 for (input i : it.get_has_Inputs()) {
    JsonObjectBuilder entity = Json.createObjectBuilder();
    entity.add("isList", false);
    entity.add("name", i.getName());
    entity.add("value", "$" + i.getName());

    //If the input type is built_in we need to go get it's google defined type
 if (builtInInformation.builtInInputTypes.containsKey(i.getInputTypeUsed())) {
        String tp = builtInInformation.builtInInputTypes
                    .get(i.getInputTypeUsed()).get(1);
        entity.add("dataType", tp);
    }else {(...)}
    param.add(entity);
}
(...)
}

(...)

@Override
public void templateGenerator(template temp, String filePath) throws Exception{
(...)
```

```
    for(inputType inpT : temp.get_has_InputTypes()){
        if (!inpT.getName().isEmpty()){
            // If the input type is a custom input type then we can write it
            // built_in input types don't need to be specified
            if(!builtInInformation.builtInInputTypes.containsKey(inpT.getName())) {(...)}
        }
    }
}
(...)
```

Listing 4.8: Assistant - Checking for the use of built-in Input Types example

Jovo language model generator

In regard to the development of the Jovo language model generator, as it was aforementioned, this framework only requires one JSON file in order to specifiy the model. This file can be divided in three separate sections: the platform-independent section, the Alexa specific section, and the Google Assistant specific section. The last two sections only have to be defined if the developer specified platform-specific functionalities. Once more, Javax JSON Objects were used to create and specify the JSON file.

The first step taken towards the development of this generator, consisted in developing a method that began by generating the Jovo proprietary components of the model and then invoked other two methods to generate the platform-specific components. The components that will be specified on this step are the invocation name, the platform-independent Intents and their Sample phrases and Inputs, and, lastly, the custom Input Types.

The second and third steps consisted in developing two methods, one for Alexa and other for Google Assistant, for the generation of the platform-specific components, if necessary. The approach taken in these two methods is very similar to the ones taken in the development of the two previously presented generators. Such is due to the fact that Jovo requires its users to append to their proprietary model the language models of Alexa and/or Google Assistant, if they want to be able to define platform-specific information (e.g Intents or Input types).

This generator, alike the Alexa and Google Assistant generators, also checks for the definition of platform-specific functionalities (Listing 4.9) and the use of built-in Intents or Input Types (Listing 4.10 and Listing 4.11, respectively). In regard to the platform-specific functionalities, it's their existence that leads to the specification of the Alexa or Google Assistant language model or of both of them. In this language model, due to its structure it's possible to define functionalities that only belong either to Alexa or to Google Assistant.

Regarding the built-in Intents, the generator will verify if there are Intents that are built-in or platform-specific in the methods that deal with the Alexa and Google Assistant language models, so the Intents are defined in those models and not on the Jovo proprietary model. Additionally, in the definition of the Jovo proprietary components, the generator will verify if there are built-in Intents that are supposed to appear in both assistants, so they can be defined, and also if these don't contain Inputs, because these types of Intents cannot contain them. Lastly, the generator will also specify the Alexa name of the built-in Intents, because Jovo requires the definition of a field that specifies the Amazon denomination of the Intent.

In regard to the built-in Input Types, similar to the built-in Intents, the generator will verify if the developer used any Input type that is built-in in the template definition. This verification will happen in the methods responsible for the specification of the Alexa and Google Assistant language model parts, respectively, in order to define the Inputs with the correct type denomination.

In the Jovo proprietary model definition, the generator will also verify if the Inputs are using a built-in Input Type, in order to specify the correct type denominations for both digital assistants. This will translates in a field for Alexa and another one for Google Assistant, each with the correct type name. Furthermore, the generator will also prevent the definition of these Input Types in the whole model, given that they can't be redefined.

```
private JsonObjectBuilder generateAlexaTemplatePart(template temp, JsonObjectBuilder jsonTemp,
     boolean navigateHomeIntentFlag) throws Exception{
        JsonObjectBuilder alexaPart = Json.createObjectBuilder();

        //Create JSON components
        (...)

        for(intent its : temp.get_has_Intents()) {
            try {
                // the intent is specific to amazon alexa so we need to
                // add it to the alexa part of the jovo language model
                if(its.getIsPlatformSpecific().toLowerCase().contains("amazon")) { (...)
                    JsonObjectBuilder it = Json.createObjectBuilder();
                    it.add("name", its.getName());
                    (...)
                }
            }catch {(...)}
        (...)}
    (...)}
(...)}
```

Listing 4.9: Jovo - Checking for the existence of platform-specific functionalities example

```
private void generateJovotemplate(template temp, String filePath) throws Exception{
     //It's a built_in intent we need to specify it for amazon
    if (builtInInformation.builtInIntents.contains(its.getName())) {
        JsonObjectBuilder alexa = Json.createObjectBuilder();
        alexa.add("name", "AMAZON." + its.getName());
        it.add("alexa", alexa);
    }
    (...)
    //Built_in intents don't require inputs
    // However specialized built_in intents can have inputs
    if (!builtInInformation.builtInIntents.contains(its.getName())){
        if (!its.get_has_Inputs().isEmpty()) {
            for (input inp : its.get_has_Inputs()) {
                if (!inp.getName().isEmpty()){(...)}
        }
    (...}}
}
```

Listing 4.10: Jovo - Checking for the use of built-in Intents example

```
private JsonObjectBuilder generateAssistantTemplatePart(template temp, JsonObjectBuilder
    jsonTemp, boolean fallbackFlag) throws Exception{
(...)

//If the input type is built_in we need to go get it's google defined name
if (builtInInformation.builtInInputTypes.containsKey(i.getInputTypeUsed())) {
    String tp = builtInInformation.builtInInputTypes
```

```
                .get(i.getInputTypeUsed()).get(1);

    entity.add("dataType", tp);
    error_pos++;
} else {(...)}
        (...)
    if (builtInInformation.builtInInputTypes.containsKey(inp.getInputTypeUsed())) {
        String tp = builtInInformation.builtInInputTypes.get(inp.getInputTypeUsed()).get(1);

        spE.add("meta", tp);
    }else {(...)}
}
(...)

private void generateJovotemplate(template temp, String filePath) throws Exception{
(...)
//If the input uses a built_in input type
if (builtInInformation.builtInInputTypes.containsKey(inp.getInputTypeUsed())) {
    ArrayList<String> aux = builtInInformation.builtInInputTypes.get(inp.getInputTypeUsed());
    JsonObjectBuilder types = Json.createObjectBuilder();
    types.add("alexa", aux.get(0));
    types.add("dialogflow", aux.get(1));
    inpt.add("type", types);
} else {(...)}
(...)

if (!inpTs.getName().isEmpty()){
// The inputType is only going to be written if it's a custom input type as
// the built_in inputType don't need to be defined because Jovo knows how to handle them
// when the template is generated for the digital assistants

if (!builtInInformation.builtInInputTypes.containsKey(inpTs.getName())) {
    JsonObjectBuilder inpT = Json.createObjectBuilder();
    inpT.add("name", inpTs.getName());
    }
}
```

Listing 4.11: Jovo - Checking for the use of built-in Input Types example

BotTalk language model generator

Regarding the development of the BotTalk language model generator, contrary to the previously presented generators, this one wasn't defined with the help of Javax JSON but with the help of Apache velocity (section 4.2.3). It was chosen to use a template-engine due to the fact that BotTalk model is defined through YAML files, one for the Intents and other for the Inputs, which have a set of rigid structural rules. By using Apache Velocity, the generation process proved itself more straightforward, in terms of following the structural rules of the model, than using a JSON to SnakeYAML library.

The first step in the development of this generator, consisted in creating two velocity templates, one for the Intents file (Listing 4.12) and other for the Inputs file (Listing 4.13), where the skeleton of the components of the YAML files was specified. Java statements for the automatization of the writing of the components information (e.g. a forEach to write the sample phrases of an Intent), were used in these templates.

The second and last step consisted in developing the methods that will use the velocity templates in order to generate the YAML files. The general functioning of these methods consists in reading the templates

to an Apache Velocity Template Object, create and specify the context, that carries the data between the Java layer and the template layer, and then render and write the template with the data in a file.

In this generator, it's also checked for the definition of platform-specific funtionalities and the use of built-in Intents and Input types. In regard to the platform-specific functionalities due to the fact that BotTalk language model, at the time this master's dissertation was developed, wasn't prepared for their specification, it was decided not to generate them in the model (Listing 4.14).

Regarding the built-in Intents, they are specified in the model as if they were custom Intents (Listing 4.15), in order to follow the model's rules. In regard to the built-in Input Types, they are written in the YAML Inputs file in order to correctly specify the type of the Inputs that use them. This translates in having the generator map each Input, that uses a built-in Input type, to its correspondent type, given that the type denomination that is used is either the one defined by BotTalk, if it exists, or the one defined by the assistants (Listing 4.16). Lastly, the mapping is sent to the template layer via the context.

However, it should be mentioned that BotTalk doesn't allow the definition of two types, one for each digital assistant, for one Input, so it was decided that, for the time being, the built-in Input Types of Amazon Alexa would be the ones used when there isn't a BotTalk defined one. Updating the generator to also use the built-in Input Types of Google Assistant will be left as future work given that this change will depend on the future updates of BotTalk.

```
intents:
#foreach($intent in $intents)
    $intent.getName():
#foreach($samplePhrase in $intent.get_has_Samplephrases())
        - '$samplePhrase.getPhrase().replaceAll("'","")'
#end
#end
```

Listing 4.12: BotTalk - Apache velocity template for the YAML Intent file

```
slots:
    #Custom Slots
#foreach($input in $inputTypes)
    $input.getName():
#foreach($value in $input.get_has_Values())
        -'$value.getName()'
#foreach($syn in $value.getSynonyms())
        -'$syn'
#end
#end
#end
    #Built-in Slots
#foreach($mapEntry in $builtInInputTypes.entrySet())
    $mapEntry.getKey(): '$mapEntry.getValue()'
#end
#end
```

Listing 4.13: BotTalk - Apache velocity template for the YAML Input file

```
private ArrayList<intent> getAllNonPlatformSpecificIntents(Template template){
    ArrayList<intent> intents = new ArrayList<>();
    for (intent intt : template.get_has_Intents()) {
        if (intt.getIsPlatformSpecific().toLowerCase().equals("all")) {
            intents.add(intt);
        }
    }
```

```
    return intents;
}(...)
```

Listing 4.14: BotTalk - Platform-specific Intents

```
private void intentTemplateGenerator(VelocityEngine ve, template temp, String filePath) throws
     Exception{
    Template t = ve.getTemplate("/botTalkIntentsTemplate.vm");

    // Create the context and setup the data
    VelocityContext context = new VelocityContext();
    context.put("intent", intent.class);
    context.put("intents", getAllNonPlatformSpecificIntents(temp));

    // Render the template
    StringWriter writer = new StringWriter();
    t.merge(context, writer);
    (...)
}(...)
```

Listing 4.15: BotTalk - Built-In Intents

```
private HashMap<String, String> getAllBuiltInInputTypes(Template template) {
    HashMap<String, String> inputTypes = new HashMap<>();

    for (intent it : template.get_has_Intents()) {
        for (input inpt : it.get_has_Inputs()) {
            String inptype = inpt.getInputTypeUsed();

            if (builtInInformation.builtInInputTypes.containsKey(inpt.getInputTypeUsed())) {
                String btString = getBotTalkSlotName(inptype.toLowerCase());
                if (!btString.isEmpty()) {
                    inputTypes.put(inptype, btString);
                } else if (it.getIsPlatformSpecific().toLowerCase().contains("google")) {
                    inputTypes.put(inptype, builtInInformation.builtInInputTypes.get(inptype).
                        get(1));
                } else {
                    inputTypes.put(inptype, builtInInformation.builtInInputTypes.get(inptype).
                        get(0));
                }
            }
        }
    }
    return inputTypes;
}(...)
```

Listing 4.16: BotTalk - Built-in Input Types

4.3.4 Platform-specific boilerplate code generators

The general steps for the generation of the boilerplate code for the initial development of the backend functionality are akin to the ones presented in the previous section 4.3.3.

First, the developer will choose if he wants to generate the code for a single platform or to all the available ones and also if he wants a database to be configured or not. The databases available will depend on the target platform(s).

After choosing the platform(s), the template is sent to the required generator(s), where it will be processed taken into account the coding rules of the chosen platform(s) and if platform-specific functionalities were defined or not. The skeleton code of these types of functionalities will only be defined in the boilerplate code file, that corresponds to the specified platform. Furthermore, the generator(s) will also check for the definition of conversational states and which functionalities are going to use them.

Concluded the realization of the generation process, the boilerplate code will be written to one or more JavaScript (Node.JS) files, depending on what is required by the platform(s).

Regarding the technologies chosen for the development of the boilerplate code generators, as it was aforementioned in section 4.2.3, Java and Apache velocity were used. These generators, similar to the ones presented in the previous section, were also coded in Java. In regard to the boilerplate code structure, it will have all the necessary configurations (e.g routing handler, etc) and the skeleton code of each one of the functionalites that the developer defined. If the developer chose a database and/or defined conversational states, they will also be configured in the boilerplate code.

The definition of the four code generators followed a similar approach among them. The approach taken consisted in reading the Velocity template(s), creating and setting up the context, that would pass the necessary data to the template layer, render the template and then, finally, write the generated code in the JavaScript file(s). The code correspondent to this approach is presented in Listing 4.17.

```
public boolean generateCodeSkeleton(Template template, ArrayList<String> databases, String
    filePath) throws Exception {
        VelocityEngine ve = new VelocityEngine();

        //Set properties
        Properties p = new Properties();
        (...)
        ve.init(p);

        Template tas = ve.getTemplate(Template name);

        // Create the context and setup the data
        VelocityContext ctx = new VelocityContext();

        ctx.put("intents",Intents list);
        ctx.put("states",Conversational states list);
        //Other pertinent data (e.g database,...)

        //Render the code skeleton template
        StringWriter writer = new StringWriter();
        tas.merge(ctx,writer);
        (...)

        // Write the generated code skeleton file
        (...)
}
```

Listing 4.17: Boilerplate code generation

Alexa boilerplate code generator

Regarding the Alexa boilerplate code generator, in addition to the presented approach, this generator will also gather the conversational states and the non platform-specific funcionalities, that were defined in the template, as information to be passed to the template layer via context.

In regard to the definition of the Velocity template for the Alexa boilerplate code generation (Listing A.1), the functionalities skeleton code will be specified by creating a handler, that will be divided in two functions. The first function, denominated canHandle, will deal with the activation rules of the functionality, for instance, if the request JSON that was received is of the type Intent Request and if the current conversational state of the application is the correct one for the functionality activation. It should also be mentioned that not all functionalities will have the need for a conversational state, given that such decision is up to the developer. The second function, denominated Handle, will deal with the realization of the functionality and the creation and transmission of the response JSON to the digital assistant.

Concerning the remaining boilerplate code, the routing handler, that allows the backend to able to receive the digital assistants HTTP POST requests, will be defined after the functionalities skeletons are specified. It's also in this routing handler that the database. that the developer chose, will be specified. If the developer didn't chose any database a default one won't be specified. Lastly, the conversational states will also be defined, prior to the functionalities skeletons, as a global variable, that will map a key for each state.

Google Assistant boilerplate code generator

Regarding the Google Assistant boilerplate code generator, similar to the previously presented generator, this generator will also gather the conversational states and the non platform-specific funcionalities, that were defined in the template, and write them to the context, that will be passed to the template layer.

In regard to the definition of the Velocity template of the Google Assistant boilerplate code generator (Listing A.2), the functionalities skeleton code will be defined by creating a handler, that will declare that it will only deal with requests referring to a certain functionality and also that expects the reception of a request JSON, whose data is materialized in the conv parameter. Additionally, some handlers may also declare that they are expecting Inputs from the user in order to be able to fulfill its request.

The remaining boilerplate code will consist, prior to the definition of the funcitionalities skeleton, in specifying the necessary configurations to the correct functioning of the database, in the cases where the developer chose to use one, and of the backend (e.g initialization of the DialogFlow client). Following the configurations, the conversational states will be defined in a similar way to the one presented in the previous generator, which means that the states will be defined as a global variable.

Lastly, after the functionalities skeletons are specified, the routing handler is defined and in this case it will only deal with the creation of the backend entry point for the digital assistant HTTP POST requests.

Jovo boilerplate code generator

Regarding the Jovo boilerplate code generator, this generator will only require the gathering of the conversational states, that were defined in the template. There wasn't the need to filter the functionalities for the ones that were platform-specific due to the fact that this framework is cross-platform and can handle functionalities for one or both platforms. It should also be mentioned that, this generator will create two JavaScript files, one that holds the configuration of the backend and other that defines the boilerplate code

for the voice application functionalities. The approach taken to the creation of those two files is the same as the one presented in the beginning of this section.

In regard to the definition of the Velocity template for the Jovo backend configuration file (Listing A.3), this file will contain configurations such as the mapping of the built-in functionalities to their Amazon denomination or of the Stop Intent to the END function, being that this last configuration is a Jovo requirement for the use of the Stop Intent. Furthermore, this file can also contain the configuration of a database, if one was chosen, and the metadata that should be stored in it, among other configurations.

Regarding the definition of the Velocity template for the Jovo boilerplate code (Listing A.4), Jovo requires that the functionalities, whose execution depends on the application being in a certain conversational state, to be defined inside a wrapper handler. This wrapper handler has the name of the conversational state and is where all the functionalities that depend on it will be defined. In order to respect this requirement, it was decided to filter the functionalities by their need of an activation state or not. After having performed this filtering process, first the skeleton code of the functionalities that don't need an activation state will be defined. Afterwards, the skeleton code of the functionalities that do need an activation state will be defined.

This order for the definition of the functionalities was a personal decision sustained by the fact that often the functionalities, that require an activation state, are follow-ups for other functionalities, that activated the conversational state and, usually, didn't had an activation state themselves.

The remaining boilerplate code will consist, prior to the functionalities skeleton code definition, in specifying the necessary configurations that will initialize the backend of the voice application. These configurations will be, for instance, the specification of the Jovo modules that are going to be needed and the database, if the developer chose one, that is going to be used. Lastly, after the definition of the functionalities skeleton code, the routing handler will be defined.

BotTalk boilerplate code generator

Regarding the BotTalk code generator, this generator didn't required the gathering of the non platform-specific functionalities, given that it's cross-platform so it can deal with functionalities that target either one or both platforms. Furthermore, the conversational states, that were defined in the template, won't be gathered due to the fact that, currently, BotTalk doesn't provide support to their use.

It should also be mentioned that, this generator will create two YAML files, one for the boilerplate code and other for the tests to the code that the developer might want to define. The approach taken to create those two files is the same as the one presented in the beginning of this section.

Concerning the definition of the Velocity template for the BotTalk boilerplate code (Listing A.5), the functionalities skeleton code will be defined inside a scenario, more specifically in the steps parameter. It's in the scenario that the application invocation name, its category and locale, and its functionalities, among other informations, will be defined.

The functionalities skeleton code will consist in their name and in the actions, that they must execute in order for the user's request to be fulfilled. Two additional parameters were added to the aforementioned skeleton code and they consist in an entry point, that is necessary if the functionality is the one that is

executed when the user opens the application, and in a "next" parameter, that can be used to specify the functionality that will be executed next and that won't have to wait for user input in order to be triggered.

In the boilerplate code, it will also be defined the necessary includes for the correct execution of the voice application. Regarding the definition of the Velocity template for the BotTalk test file (Listing A.6), this template only consists in defining the section were the developer will be able to define its tests.

4.3.5 Deployment

In order to provide more functionalities to the developers, the proposed construction process will also allow them to deploy their voice applications to Alexa and Google Assistant via the VoicePrint platform.

Regarding the frontend components of the voice application, the deployment will trigger the generation of the necessary platform-specific language model(s) and after it will deploy them to the Alexa Developer Console and/or DialogFlow, depending on the platform(s) that were chosen.

In regards to the backend components of the voice application, the deployment will trigger the generation of the necessary boilerplate code. Afterwards, if the developer chose to host its application on AWS Lambda and provided a Lambda endpoint, the generated code will be deployed to this cloud-server service. It should also be mentioned that, in case that the developer did not provide an AWS Lambda endpoint only the generated platform-specific language models will be deployed.

A possible future improvement to the deployment of the backend components is allowing the developers to also be able to deploy their boilerplate code to Google Cloud.

Furthermore, if the developers are going to use Jovo to develop their voice applications, they can also deploy them through that framework, via VoicePrint.

Regarding the BotTalk platform, due to the way it was developed it wasn't possible to define an approach that allowed the developers to upload their voice application to this platform and from then deploy it to the digital assistants consoles.

The general steps of the deployment process consist in first choosing for which platform the voice application is going to be deployed. Afterwards, the developer will have to provide its credentials to the platform, which can be, for instance, its Ask-profile name or the DialogFlow projectID. If the developer also wants to deploy the boilerplate code, he will also have to provide an AWS Lambda endpoint of a Lambda function, that he previously created.

Granted that every credential was provided correctly, the deployment process will start executing in the background in order to allow the developer to be able to do other tasks in VoicePrint. In the background, the deployment process will first generate the necessary platform-specific language models and boilerplate code, so they can be used in the following step, the deployment itself. The deployment of the generated components differs between the approached systems.

Deployment for Amazon Alexa

Regarding the deployment of the voice application components to Amazon Alexa, the process will start by verifying if the Ask Cli (Alexa skills kit Command line interface), that provides the means for the deployment process to happen, is installed. In case that the Ask Cli isn't installed the developer will be warned that this tool is necessary for the deployment process to happen and then the deployment will be cancelled.

After checking for the Ask Cli installation, the process will verify if the developer has the necessary AWS credentials. In case that the developer doesn't have them, a warning will be issued so he knows that first he needs to gather those credentials, and then the deployment process is cancelled. The next steps of the deployment process are presented in Fig.16.

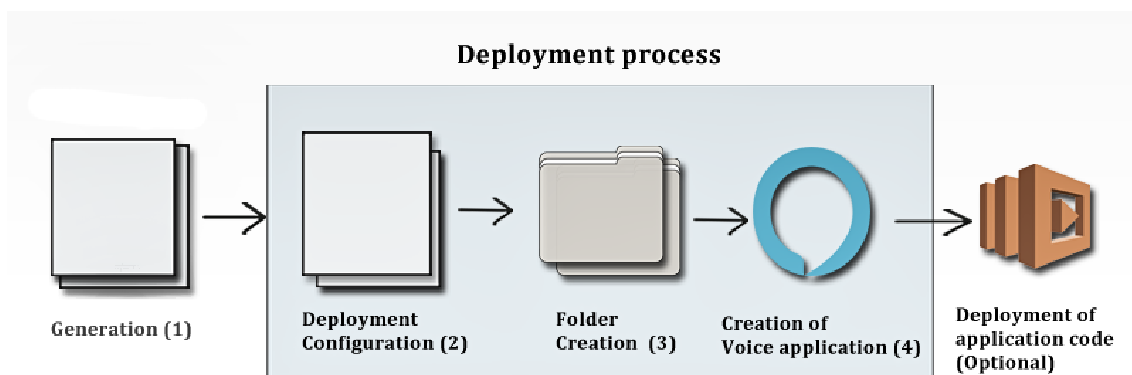


Fig. 16.: Deployment process for Amazon Alexa

Firstly, the Alexa language model and the boilerplate code will be generated. Afterwards, the deployment related tasks (Listing 4.18) will start with the creation of two configuration files. The first file will describe the configuration of the voice application (e.g its name, locales, etc) and is necessary for its creation in the Alexa Developer Console. The second file will contain the deployment settings configuration, which will allow the application, in the developer console, to be updated later on with the generated language model.

The next step consists in creating two folders, one for the language model and other for the boilerplate code, and in transferring the generated platform-specific model and boilerplate code to them, respectively. Lastly, the application will be created in the developer console and its default language model will be updated to the generated model.

It should also be mentioned that, if the developer provided an AWS Lambda endpoint, there is going to be an extra step, that will consist in calling another deployment method, whose only task is dealing with the deployment of the boilerplate code to the AWS Lambda.

```
public void initDeploy(String filePath, ArrayList<String> paths, String tempName,String
    askProfileName, String desc, String endpoint) throws Exception {
    //Create the folder for the Alexa Skill
    File skillFolder = new File(filePath);

    if(!skillFolder.exists()){skillFolder.mkdir();}

    try {
        createSkillJson(filePath, tempName, desc);
        createAskSettings(filePath, tempName,"");
```

```
        createModelAndCode(filePath, paths, tempName);
        deploySkillAsk(filePath, tempName, askProfileName);

        if(!endpoint.isEmpty()){
            DeployLambdaCode deployLambdaCode = new DeployLambdaCode();
            deployLambdaCode.deployCode(filePath,tempName,endpoint,askProfileName);
        }
    }catch (Exception e){(...)}
}
```

Listing 4.18: Alexa - Steps for deployment

Deployment for Google Assistant

The deployment of the voice application components to Google Assistant starts by verifying if Google Cloud SDK is installed. In case that the SDK isn't installed the developer will be warned that to deploy its voice application to DialogFlow he needs to install Google Cloud, and then the deployment process will be cancelled. The next steps of the deployment process are presented in Fig.17.

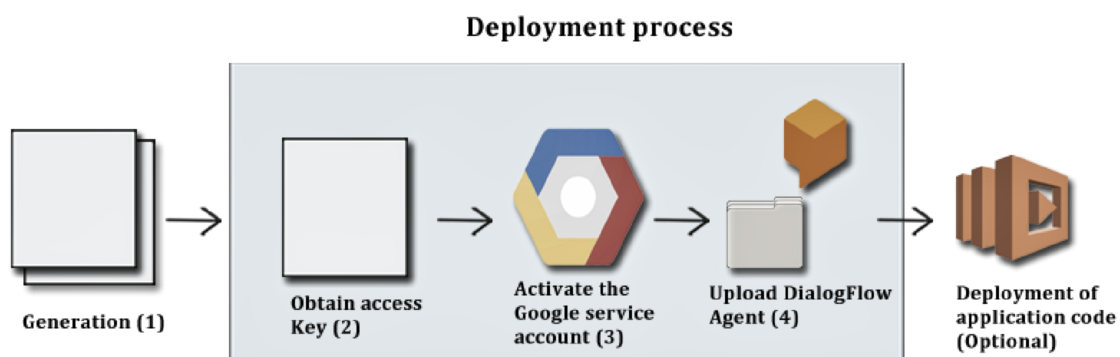


Fig. 17.: Deployment process for Google Assistant

Firstly, the Google Assistant language model and the boilerplate code are going to be generated. Afterwards, the deployment related tasks (Listing 4.19) will start by copying the developer key file, that contains the key to access the developer Google Cloud service account, to the project folder. This file will then be used in the next step, which consists in activating the Google service account in order to later be able to retrieve an access token.

Lastly, an access token will be retrieved and used to upload the DialogFlow agent zip file, that contains the Google Assistant language model (section 4.3.3), to the DialogFlow console. The training of the uploaded agent in the console is also triggered. The creation of the agent in the DialogFlow console will have to be done by the developer before he starts this deployment process. This process cannot create the DialogFlow agent due to a current limitation of Dialogflow, that was present at the time this master's dissertation was developed.

It should also be noted that, if the developer provided an AWS Lambda endpoint, there is going to be an extra step, that will consist in the deployment of the boilerplate code to AWS Lambda, by calling another deployment method.

```
public void initDeploy(String pt, String tempName, String keyFilePt, String projectID, String
    endpoint, String askProfileName) throws Exception {
    try{
        moveKeyFile(pt,keyFilePt);
        activateGoogleAccount(pt,keyFilePt);
        deployAgent(path,tempName,projectID);

        if(!endpoint.isEmpty()){
            DeployLambdaCode deployLambdaCode = new DeployLambdaCode();
            deployLambdaCode.deployCode(pt,tempName,endpoint,askProfileName);
        }
    }catch (Exception e){(...)}
}
```

Listing 4.19: Google Assistant - Steps for deployment

Deployment with Jovo

The deployment of the voice application components via Jovo will use a tool named Jovo cli. Jovo cli allows the developers to deploy their voice application via command line and those commands will be used to implement the deployment process.

First a warning will be sent to the developer related to the AWS Lambda to add the Alexa Skill kit trigger and also to disable its use of the Skill ID. Next, it's verified if the developer has the Jovo framework installed. In case that the framework isn't installed the developer will be informed that Jovo must be installed and the deployment process will be cancelled. The next steps of the deployment process are presented in Fig.18.

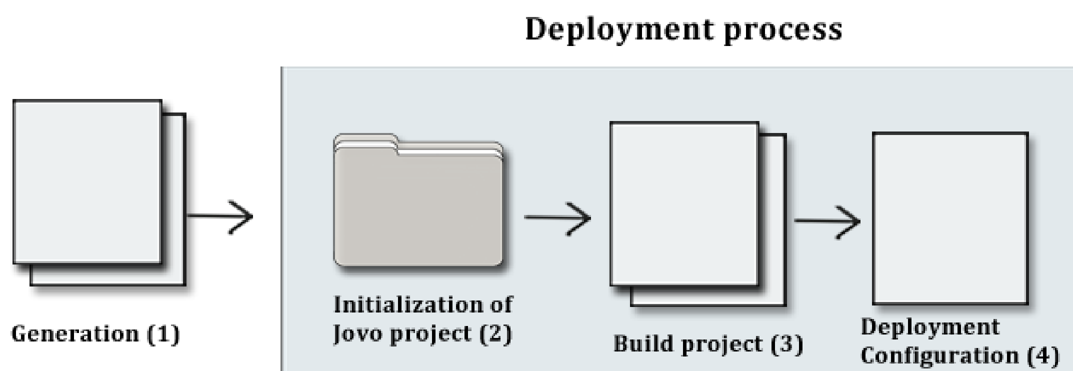


Fig. 18.: Deployment process with Jovo

Firstly, the Jovo language model and the boilerplate code are going to be generated. Afterwards, the deployment related tasks will start with the initialization of a Jovo project and the gathering of the generated language model and boilerplate code (Listing 4.20). These two components will be moved to their respective folders in the newly created project.

Subsequently, the building process of the project, which consists in generating, from the Jovo language model, the platform-specific language models for the chosen digital assistants will start. The last step is the creation of a JSON file, that will contain the project configuration for the deployment phase.

It should also be noted that, contrary to the two previously presented deployment processes, this process doesn't need to use the deployment method that handles the deployment of the boilerplate code to AWS Lambda. Such is due to the fact that the Jovo Cli command for deployment already handles the deployment of the boilerplate code to the Lambda endpoint.

```
public void deployJovo(String filePath, ArrayList<String> paths, String tempName, String plat,
     ArrayList<String> configurations, String endpoint) throws Exception {
    //Initialization of auxiliary vars
    (...)

    //Initialize commands
    if(plat.toLowerCase().equals("all")){(...)}
    else if(plat.toLowerCase().equals("google assistant")){(...)}
    else{cmdBuild = "jovo build -p alexaSkill";
        cmdDeploy = "jovo deploy -p alexaSkill";
    }

    try {
        initJovo(filePath,tempName,paths);
        buildJovo(filePath, tempName, cmdBuild);

        if(flag){
            generateProjectConfiguration(filePath, tempName, plat, configurations, endpoint);
            deploy(filePath, tempName, cmdDeploy);
            deploy(filePath, tempName, cmdDeploy1);
        }else {
            generateProjectConfiguration(filePath, tempName, plat, configurations, endpoint);
            deploy(filePath, tempName, cmdDeploy);
        }
    } catch (Exception e) {(...)}
}
```

Listing 4.20: Jovo - Steps for deployment

Deployment for AWS Lambda

The method (Listing 4.21) that performs the deployment of the boilerplate code to AWS Lambda is the one that is invoked as the last step of the deployment process of Amazon Alexa or Google Assistant, as long as the developer provided a Lambda endpoint. A possible future improvement to this process would be creating the Lambda function in AWS Lambda for the developer and then, in the end of the process, provide him with the endpoint of the created function. The next steps of the deployment process are presented in Fig.19.

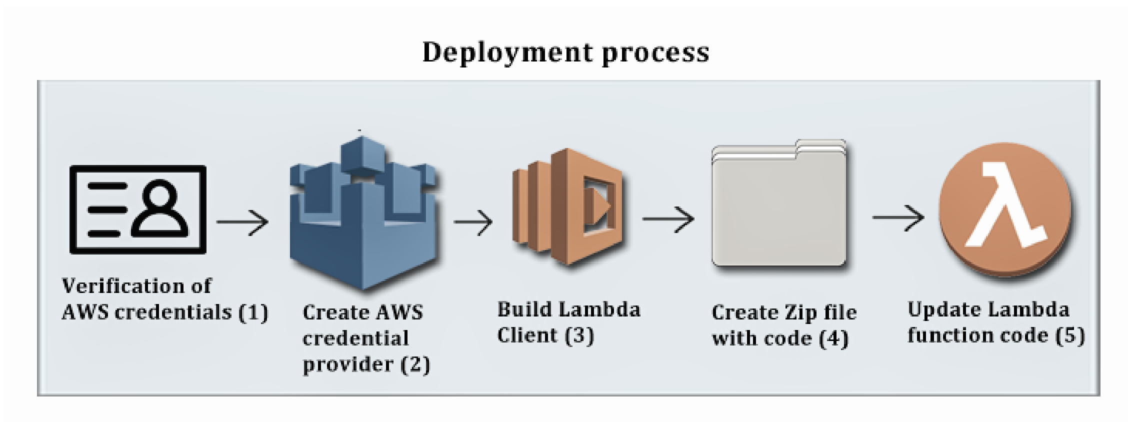


Fig. 19.: Deployment process for AWS Lambda

The first step will consist in verifying if the AWS access credentials aren't defined as environment variables, given that if they are, the deployment process will be interrupted. Supporting the use of the AWS credentials as environment variables was relegated as future work. In case that the credentials aren't defined as environment variables, the next step will consist in gathering and use those credentials to create an AWS credential provider. This provider is necessary to build a Lambda client.

The next step will consist in creating a zip file with the boilerplate code and in making a code request to update the Lambda function, that the endpoint points to. Lastly, the Lambda client will send the update code request and then the developer will be warned if the deployment operation was successful or not. In case it was successful, the Lambda function that the user created was updated and will have the boilerplate code.

```
public void deployCode(String filePath,String tempName,String endpoint,String askProfile)
    throws Exception {
    if(System.getenv("AWS_ACCESS_KEY_ID") != null || System.getenv("AWS_SECRET_ACCESS_KEY") !=
         null){
        (...)
    }else{
        AWSCredentialsProvider credentials =  getAWSCredentials(askProfile);
        (...)
        Pattern pat = Pattern.compile(pattern);
        Matcher m = pat.matcher(endpoint);
        String region;

        if(m.find()) {region = m.group();}
        else{(...)}

        (...)

        AWSLambda LambdaClient = AWSLambdaClientBuilder.standard().withCredentials(credentials
            ).withRegion(region).build();

        //Get zipCode
        String pathZip = createSRCZip(filePath);

        //Update Lambda
        try{
            updateFunction(LambdaClient,pathZip,endpoint,tempName);
        }catch (Exception e){(...)}
    }
}
```

Listing 4.21: Lambda - Steps of the deployment of the boilerplate code

4.3.6 VoicePrint

The last goal presented in section 3.2 referred to the development of a platform, with a visual editor incorporated, whose main function was to provide an user interface for the platform-independent construction process proposed in this dissertation. By providing the construction process with a UI, it will be more user-friendly and the visual editor can be adapted to the needs of both technical and non-technical users. Additionally, the platform will allow the developers to not have the need to manually specify the language model.

Having finished the specification and development of the construction process, which was the foundation for the platform, the first step in the development of the platform was the definition of how it would be named and how it would be composed. In regard to its name, as it was previously stated, this platform was named VoicePrint. The reason behind this name is the fact that the language model templates can be seen as blueprints for the development of voice applications.

Regarding how VoicePrint is going to be composed, its layout is presented in Fig.20. Essentially, as it was aforementioned, VoicePrint is going to provide a graphical UI in order to abstract the construction process components, the language model template, the generators, the deployment modules, and their functioning. Therefore, this platform layout is going to be composed by the construction process components and the necessary components to establish the visual editor, and the persistency of the templates and of the generated voice applications components.

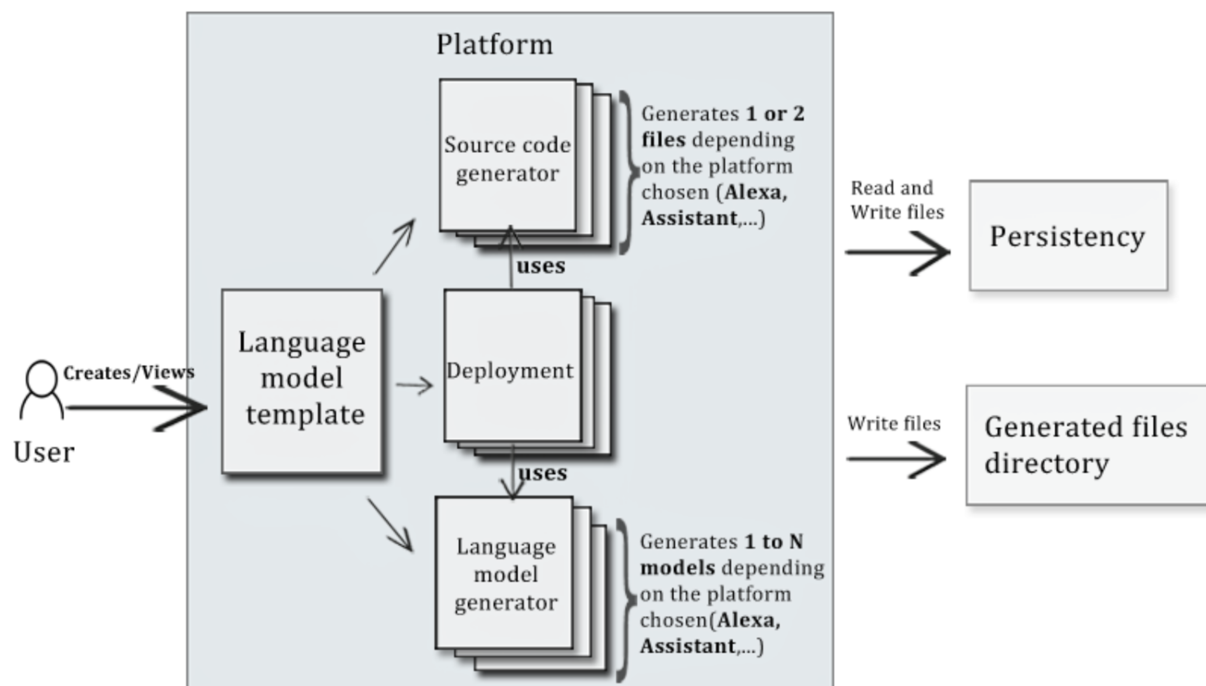


Fig. 20.: VoicePrint Layout

In addition to allowing the developers to create and specify a myriad of language model templates, VoicePrint also allows the deployment of those models to the Alexa Developer Console and DialogFlow and

the generation and deployment of the boilerplate code to AWS Lambda. Furthermore, the possibility of deploying both the language model and the boilerplate code through Jovo is also made available to the developers.

The deployment process of the voice application consists in gathering the necessary credentials from the developer and in generating the corresponding language model and boilerplate code files (Fig.20). These files will then be deployed to the chosen digital assistant(s) and cloud-server service.

Regarding the development of VoicePrint, as it was previously mentioned in section 4.2.3, it was decided to develop it recurring to Java due to the familiarity with this programming language and the possibility of developing this platform from the same codebase, to three distinct operating systems (Windows, Linux and MAC).

In order to develop the graphical UI, it was decided to use JavaFX along with Java, due to its integration with the chosen programming language and because it allowed the use of the JavaFX Scene builder. The Scene builder is a tool that simplifies the designing of the UI due to the fact that it eases the specification of both the FXML files and the CSS files or rules for the definition of the UI components.

The FXML files are defined in the FXML scripting language and allow the specification of UI components, such as buttons and text fields. Additionally, these files can be used along with external CSS files while also having the option to define individual CSS rules in them, for each defined component. By combining the specification in the FXML files and the CSS rules, it was possible to define a clean and coherent aspect for VoicePrint.

The MVC architectural pattern and the Facade pattern were used to design VoicePrint and establish the communication between the components, that defined the proposed construction process, and this platform components, respectively. It was decided to apply the MVC pattern in the development of VoicePrint in order to achieve separation of concerns and to have the ability to reuse components, and the flexibility of changing or adding a component with minimal impact in the rest of the system.

The Models in this scenario correspond to the language model template components, that are POJO's (Plain Old Java Objects), presented in Fig.14. The Views correspond to the developed FXML files, that contain the specification of the UI components and how they are structured together to form the VoicePrint pages, among other information.

Lastly, it was defined a Controller for each View, that the platform will have, and they will control the execution flow of VoicePrint and return a response to the developer's requests. These Controllers will also deal with tasks, such as, saving, or reading, a template to file persistency or triggering the generation of the platform-specific language models while redirecting the user to another Controller. By redirecting the user to another Controller, he will be able to perform other tasks while waiting for the models to be generated.

The Facade pattern was applied in order to define an interface that centralized the communication of the controllers with the generator, deployment and persistency modules. Additionally, this interface will define an API that only provides access to relevant operations, such as generating a language model to Amazon Alexa or boilerplate code to Google Assistant.

By using a Facade, it's possible to "hide" who is providing the referred modules functionalities from the rest of the system, which in turn allows for the possibility of iterating upon the modules components

or completely replacing them, without affecting the whole system as long as the defined API continues to being supported.

As to the passing of information between the Controllers, such as information about the previous Controller and stage, and the template that is currently being created/viewed/modified, the design option was to have an approach that consisted in setting the necessary parameters for the new controller in the FXML loader. This approach consists in defining a constructor in the controllers with the necessary parameters to their correct functioning and then, when there is the need to call a certain controller, the caller will pass the required parameters and then set the created controller on the FXML loader instance, before loading the correct FXML entity (Listing 4.22).

```
(...)
public void redirectToCtxStates(javafx.event.ActionEvent event) throws IOException{
    FXMLLoader loader = new FXMLLoaderr(getClass().getResource("/org.openjfx/ContextStates.\
        acrshort{fxml}"));

    loader.setController(new Contextstatescontroller(defaultTemplates,customTemplates,this,"/
        org.openjfx/IntentDefinition.fxml",customTemp,intent,intentIndex,(Stage) ((Node)event.
        getSource()).getScene().getWindow()));

    Parent root = loader.load();
    (...)
}
(...)
```

Listing 4.22: Example of passing information between controllers

A method that will be used when there is the need to propagate information backwards was also developed to ensure that the controllers are always working with up to date information. This method logic translates in a Controller passing the altered information back to the Controller who called him (its parent Controller). Even though the aforementioned approach was used to pass information between the controllers, there were also other approaches that could have been used, such as dependency injection or event bus mechanisms.

However, due to the simple nature of VoicePrint and the fact that it's part of a proof of concept, created to show that there is feasibility for real-world application of the proposed construction process, and not a product to be launched in the market, it was decided that it was more straightforward to pass the parameters directly rather than using extra frameworks to, for instance, implement the aforementioned mechanisms. The benefits of using frameworks to implement dependency injection or event bus mechanisms would be more mid-to-long term (e.g maintainability), which would only be more visible if this construction process and platform were going to be released on the market and grow from there.

It should also be mentioned that the remaining proof of concept refers to the voice applications that were developed via VoicePrint, and that are going to be presented in Chapter 5.

Lastly, in regard to the demonstration of how the developers will interact with the developed platform, it will also be shown in Chapter 5. In that chapter, the explanation about the general steps that a developer takes to define a language model template and then generate it to platform-specific language models for the definition of the frontend and to boilerplate code for the initial development of the backend functionality will be presented. Additionally, how the templates are deployed via VoicePrint will also be explained.

4.4 Summary

In this chapter, the approach and the conceptual decisions taken before the development of the proposed high-level activity diagrams, the platform-independent construction process, and the VoicePrint platform were presented. Afterwards, the general implementation of the solutions, that arose from the approach and the decisions taken, for each proposed tool was presented.

The first step taken before defining the approach, that was followed in the implementation phase, consisted in taking the knowledge gathered in the study regarding the state of the art and in conducting another, even more thorough, study regarding the components of a voice application for Alexa and Google Assistant and how they are developed. Additionally, this new study was also focused on the existing ways that an application conversation model can be defined. This study lead to conceptual decisions regarding how the proposed tools would be developed and their overall architecture, that in turn lead to the definition of the implementation approach.

For each one of the proposed tools, first it was presented how they would be defined, composed and developed, and also which technologies were going to be used. Given that it was decided to use Java as the programming language for the development of the construction process components and VoicePrint, it will be required that to use this tools the developers will have to have Java installed on their machines.

Afterwards, for each proposed tools, it was presented a general overview of how they were implemented and how they could help the developers in the development of cross-platform voice applications. Furthermore, it was also explained the necessity of developing VoicePrint as a graphical UI for the construction process and how their components communicated with each other.

Regarding the challenges, presented in section 3.3, the decisions that were taken before and throughout the implementation phase made possible to provide solutions to some of the challenges. The remaining challenges for which a full-fledged solution couldn't be found were handled in the best way possible, taking into account the current state of the art.

One of those challenges was making the updates to the construction process architecture less convoluted when faced with innovations in the technological area, that the voice applications belongs to. Such is due to the fact that even though the construction process architecture was defined with the concept of separation of concerns in mind to improve its maintainability and ease its future improvement, at any moment in time the innovations in the development of voice applications might obligate to a deep reconstruction of the process and its components.

To conclude, the approach used to guide the implementation phase and the way that the tools and their components were implemented was presented in detail in this chapter. In the end of the implementation phase, a fully functioning construction process and platform were obtained and will be used to develop a few voice applications as case studies.

5

CASE STUDIES

5.1 Introduction

Concluded the implementation of the proposed construction process, high-level activity diagrams and the platform VoicePrint, it's possible to state that the goals, defined in Chapter 3, were satisfied. Having implemented the construction process and tools, it was necessary to apply them to a few case studies in order to verify their viability and the results that they produced. Thus, the case studies will serve as proof of concept for the proposed platform-independent construction process and will allow the demonstration of how VoicePrint works.

Furthermore, it will also be possible to show that the conceptual decisions taken in Chapter 4.2 allowed the development of a construction process that solves, completely or in the best way possible, the challenges presented in section 3.3.

In this chapter, two distinct case studies, that correspond to two voice applications, being that one will be of the conversational type and the other of the request-response type, will be presented and explained in detail. It should also be mentioned that, there were other two voice applications that were developed throughout this dissertation.

The first one allowed the comparison, that is presented in section 2.4.3, between Jovo and Violet. Due to the fact that this application wasn't developed with the proposed construction process it won't be approached in this chapter. The second one is a voice application of the request-response type, that tells the user information about the Portuguese symposium INForum ([16]). Due to the fact that another voice application of the request-response type is going to be presented, it was decided not to present this one as it wouldn't bring anything relevant to this chapter.

Additionally, in this chapter, the tests performed to the developed Java code, that constitutes the construction process will be presented. Regarding the type of tests that were performed, it was decided to use unit tests instead of integration tests or a fusion of both, given that it proved to be the best fit to implement the tests to the developed code. In regard to the tool chosen, it was decided to use the framework JUnit to develop the automatic unit tests for the construction process given that one of its focus is unit testing and it works well with Java.

5.2 The voice applications market

Preceding the development of the two aforementioned voice applications, a study concerning the voice applications market, more specifically the Amazon Alexa and Google Assistant ones, was conducted in order to find the volume of voice applications, ordered by categories, in each assistant app store. Such decision is based on the fact that, usually, developers tend to develop voice applications for the users preferred applications categories, so they can reach a substantial audience.

Therefore, the voice applications should belong to the applications categories (e.g "News", "Lifestyle",etc) that have the most considerable volume of applications in the app store of both digital assistants. Additionally, they should also be of one of the following types:

- The type where the user and the application maintain a dialogue in order for the user's request to be fulfilled (e.g a transactional application that allows the user to shop online);
- The request-response type, where the user only requests something from the application and gets an answer back.

In regard to information about the volume of voice applications that exist, in each category, in Amazon Alexa app store, VoiceLabs [84] had already performed an analysis concerning that subject. However, since that information is dated from December 2016, it was decided that a scraper, that would perform an analysis similar to the one made by VoiceLabs on Alexa app store, should be developed [1].

Fig. 21.: Alexa App store - Volume of Applications on April 2019

1 This study was conducted on April 2019, so the following data regarding the volume of voice applications in the application store of each digital assistant might be outdated by the time this dissertation is concluded.

The top four categories that contain more voice applications in Alexa app store (Fig.21) are "News", "Games, Trivia & Acc.", "Education & Reference" and "Lifestyle.

Regarding Google Assistant, given that there weren't any available statistics concerning the volume of voice applications that exist, in each category, in it's app store, it was decided that another scrapper should be developed. This scraper will perform an analysis on the Google Assistant app store similar to the one presented above.

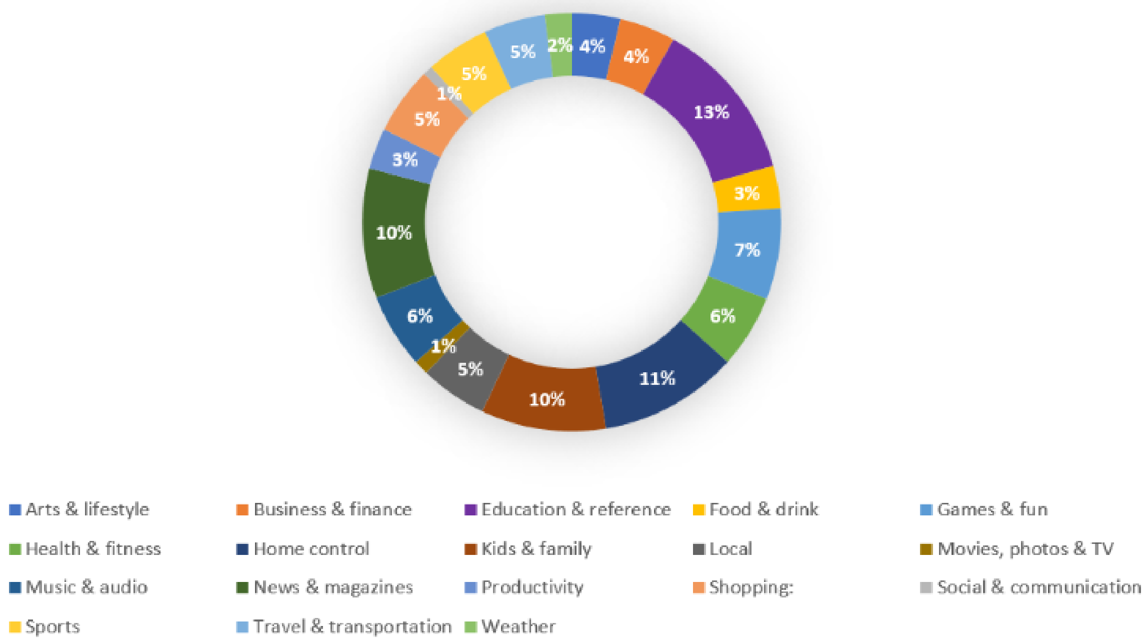


Fig. 22.: Google Actions App store - Volume of Applications on April 2019

The top four categories that contain more voice applications in Google Assistant app store (Fig.22) are "Education & Reference", "Home control", "Kids & family" and "News & magazines".

Knowing the top four applications categories, that have the most volume of applications, in each digital assistant app store, there is one decision left to be made. This decision concerns the choice of which category(ies) will the voice applications, developed in this dissertation, belong to, without disregarding that one must be of the conversational type and the other must be of the request-response type.

In regard to the request-response voice application, it was decided that it would belong to the "News & magazines" category, that is one of the top four categories on both digital assistants. Such is due to the fact that a News application won't require the user to maintain a continuous dialogue with the digital assistant, given that he only needs to express, for instance, a request for the top N news of a certain day or that he wants to hear the news about a certain topic.

In regard to the conversational voice application, it was decided that it would belong to the "Shopping" category, even though it's not present in the top four categories of neither digital assistant. Such is due to the fact that, nowadays, most people prefer to shop online because of the vast product selection that most

e-commerce websites offer and because of the convenience that exists in doing, for example, the weekly grocery shopping with just a few clicks or in this case a few voice commands. Furthermore, this decision also fulfills the requirement of an application that maintains a dialogue with the user given that in order to satisfy the user's request of buying a product from an e-commerce website, the assistant needs to, for instance, ask him if he wants to apply a filter to his search or how many of a product he wants to buy.

In conclusion, chosen the types and categories for the two voice applications, that are going to be developed as case studies, the specification and development phases can commence.

5.3 News voice application - The Informed

The Informed will be a voice application dedicated to telling it's users the most recent news or headlines. The chosen news and headlines source was BBC news. As it was aforementioned, this application will be of the request-response type. Such is due to the fact that its interactions with the users will, normally, consist in the user requesting, for instance, news about economy and in the application providing them, without the need to start a dialogue with him.

However, in order to apply the Information Spreading design pattern (section 2.5.1), so the users short-term memory is respected, it was decided that after the application presents three news/headlines to the user, he will be asked if he still wants to hear more news/headlines or not. This decision adds an aspect of conversation with the user but it was considered small enough to not threaten the claim of this application being of the request-response type.

The first development step consisted in the definition of both the functional and non-functional requirements of the voice application. Those requirements are presented in Annex A.4. The second development step dealt with the employment of the Persona design pattern (section 2.5.1), which allowed the definition of how the voice application will sound to the users. The defined Persona is presented in Annex A.5.

The third development step consisted in the definition of the high-level activity diagrams, which will help to streamline the specification of the conversation model and of the application functionalities. The main flow of conversation between the user and the application and the specification of the news functionality are presented in Fig.12 and Fig.13, respectively. Similar activity diagrams were developed for the remaining functionalities, which are present in the main flow of conversation (Fig.12). However, given that their definition is very akin to the news functionality it was decided that their presentation wouldn't add value to this section.

Having specified and modeled the voice application, the fourth development step consisted in choosing an API, that provided news from BBC news, and in the development of the application frontend and backend. In regard to the API, it was decided to use the News API, due to the fact that it returns easy to parse JSON files with news or headlines from a myriad of blogs or news articles, its free to use and one of the news source that can be used is BBC news.

Regarding the tools used in the development phase, it was decided to use the Jovo framework, given that it would streamline the cross-platform development of the voice application, and because it can be

used together with VoicePrint, given that the construction process supports both the generation of the Jovo language model and of boilerplate code for it.

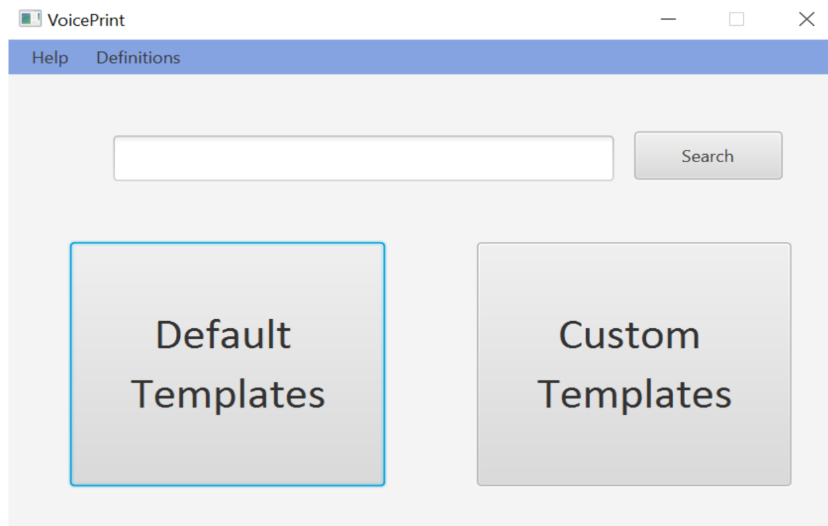


Fig. 23.: VoicePrint - MainPage

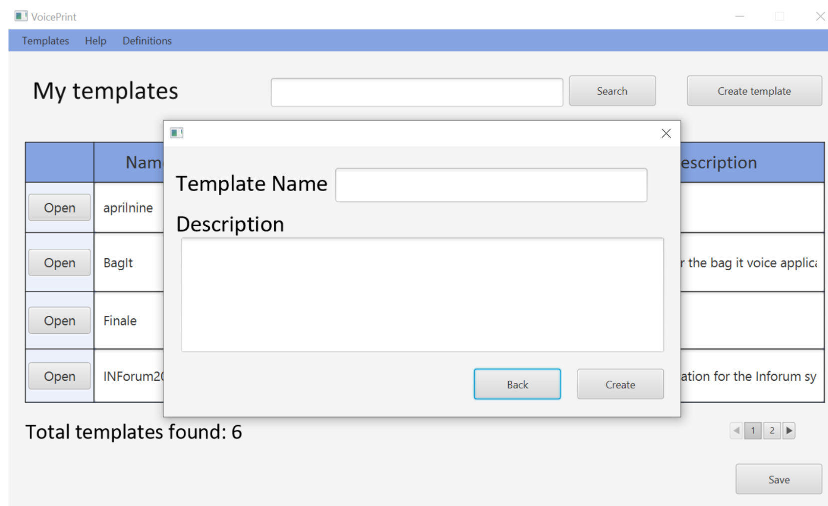


Fig. 24.: VoicePrint - Create custom template

The first phase of development consisted in creating and specifying the language model template, that constitutes the frontend. The creation of this template was influenced by the high-level activity diagrams, that were previously developed, due to the fact they show the main flow of conversation of the application, the funcionalities that needed to be defined, and also allowed us to infer the possible sentences that the users could use to request one of the functionalities.

In regard to the definition of the template, it was defined using VoicePrint (Fig.23). In VoicePrint, the creation and specification of the template began with the definition of the template name and description, being that the description wasn't mandatory (Fig.24). Afterwards, the following components of the template were defined:

- Application invocation name;
- Intents;
  - Sample phrases;
  - Inputs;
  - Context states.
- Input types.

The definition of the invocation name was accomplished in the main page of the template (Fig.25). In regard to the creation of an Intent, there were two options that could have been chosen, which refer to the two types of Intents that can be created, the built-in Intent or the custom Intent (Fig.26). The built-in Intents were already explained in section 4.3.2 and are Intents that are provided by the digital assistants. These Intents, contrary to the custom Intents, won't allow the users to define or use Inputs due to digital assistants rules.

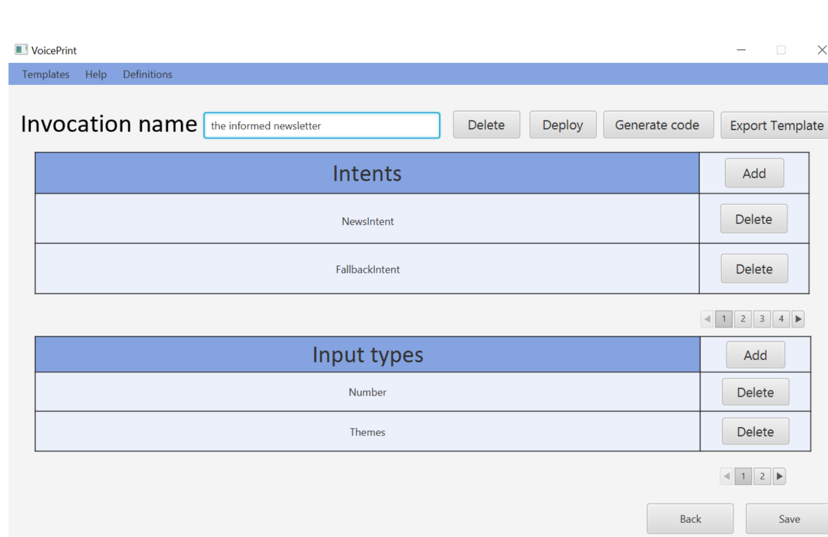


Fig. 25.: VoicePrint - Custom template main page

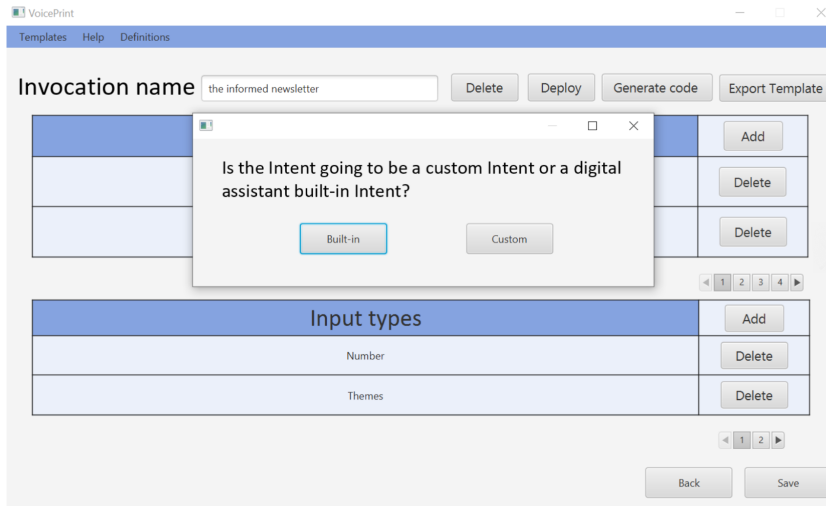


Fig. 26.: VoicePrint - Create Intent

The definition of the built-in Yes Intent can be observed in Fig.27. For this Intent, it was defined its name, if it was going to be platform-specific or not and the sample phrases, that the user can say to invoke it. No context states were defined for this Intent, given that this application doesn't need to be in a certain state to handle an affirmative response by the user.

It should also be mentioned that VoicePrint will check the Intent definition for errors, such as the developer trying to insert an Input in one of the sample phrases, in order to prevent the failure of, for instance, the generation process of the platform-specific language models.

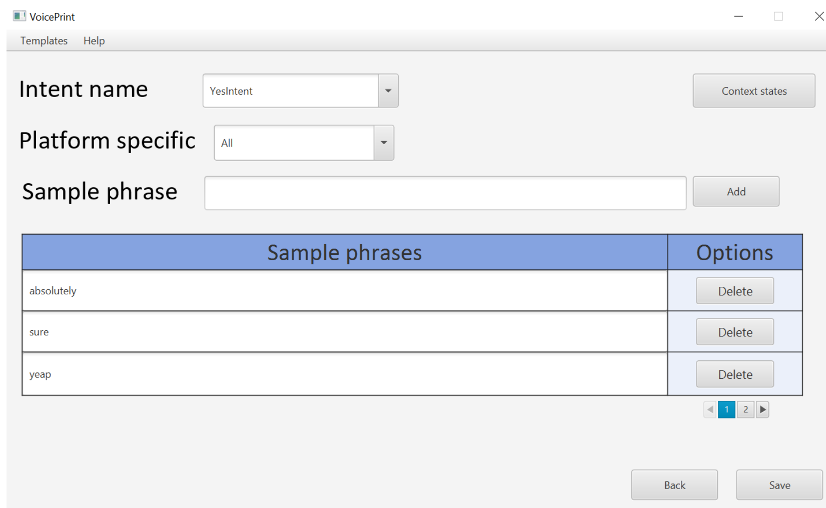


Fig. 27.: VoicePrint - Built-in Intent

The custom Intents that were defined correspond to the application functionalities, such as the News or Headlines Intents. The definition of the News Intent can be observed in Fig.28. It was decided to show the definition of this Intent because its representative of what was defined in the remaining Intents. Additionally, its conversation flow was already presented in the explanation of the high-level activity diagrams (section 4.3.1) in Fig.13, so its possible to show, for instance, how it was decided to handle with the reception of the news theme (an Input) that the user wants to hear.

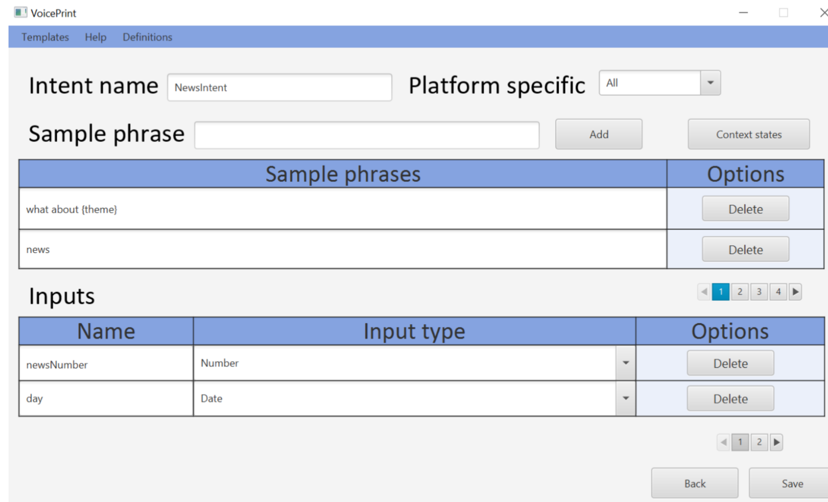


Fig. 28.: VoicePrint - News Intent

The definition of the News Intent (Fig.28), similarly to the definition of the Yes Intent, consisted in defining its name, its sample phrases and if it would be platform-specific or not. Additionally, it was also defined a context state, more specifically an output state, which means that, after the execution of this Intent, the application state will be set to the News state.

The News state is necessary for when the application has to present more than three news to the user and needs to ask him if he wants to hear more news or not. In this situation, the application will be expecting a Yes/No answer, that will need to trigger a particular Yes/No Intent, which will be prepared to answer accordingly to the News state. After the execution of this Yes/No Intent, the application state will be set back to default.

The News Intent will use Inputs in order to allow the user to specify, for instance, if he wants to hear news about a certain theme or from a certain day. These Inputs, if provided by the user, will be used in the creation of the news query in the backend method that implements the News Intent. This query will be used in the request to the News API in order to obtain only news that are pertinent to the user's request.

Regarding the Input types, VoicePrint provides a list of all the available built-in and custom Input types. However, it should be noted that the custom Input types will be specific to each custom template. This means that the custom Input types defined in this template won't be available in other existing templates and vice-versa. Allowing the use of custom Input types across different custom templates will be left as future work.

In regard to the creation of Input types, akin to the creation of Intents, there were two options that could have been chosen, which refer to the two types of Input Types that can be created, the built-in Input Type or the custom Input type. It should also be mentioned that, when the developer uses a built-in Input type VoicePrint automatically creates and adds it to the template.

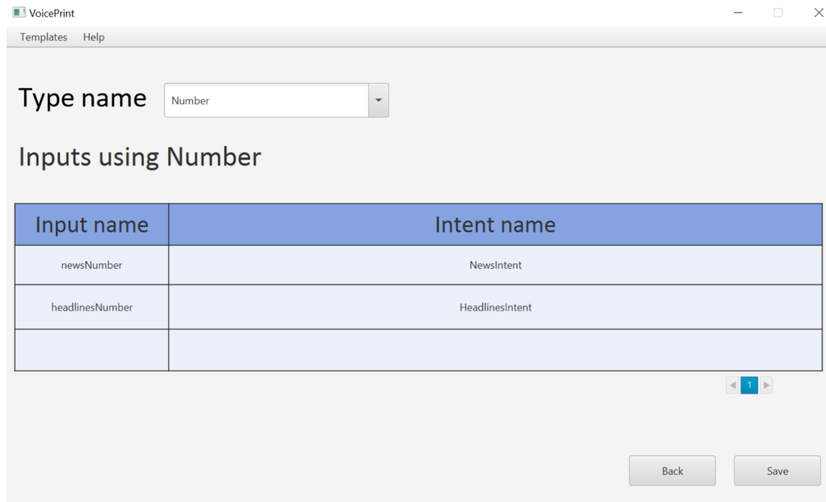


Fig. 29.: VoicePrint - Built-in Input type

As to the built-in Input Types, the developers won't be allowed to specify any information for them due to digital assistants rules as it can be seen in Fig.29, that shows the built-in Number Input type. This type was used in the sample phrases in order for the user to be able to specify the number of news he wants to hear.

Additionally, it was also used the built-In Date Input type in order for the users to be able to specify that they want to hear the news from a certain day. However, it was decided that the users could only ask for news that were one day to a week old, given that the News API can't provide, for instance, news that are one year old. The verification of the date will be done in the backend method that implements the News Intent.

In the case of the custom Input Types, the developers will be able to specify the type name and its values, and synonyms for those values if needed. This can be seen in Fig.30), that shows the custom Themes type. This type was used in the sample phrases in order for the user to be able to specify that he wants to hear news about a certain theme (e.g Economy).

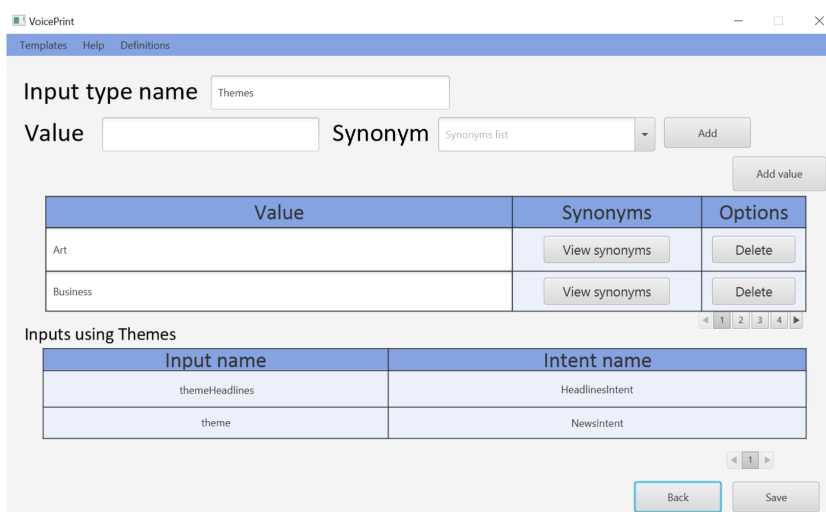


Fig. 30.: VoicePrint - Custom Input Type

The final step in the development of the frontend consisted in the deployment of the voice application components to Alexa and Google Assistant. Firstly, we had to choose to which platform the platform-specific language model and the boilerplate code were going to be generated to. In this case, the platform chosen was Jovo (Fig.31), given that it was the selected framework to develop the application. The deployment will then generate the Jovo language model and boilerplate code.

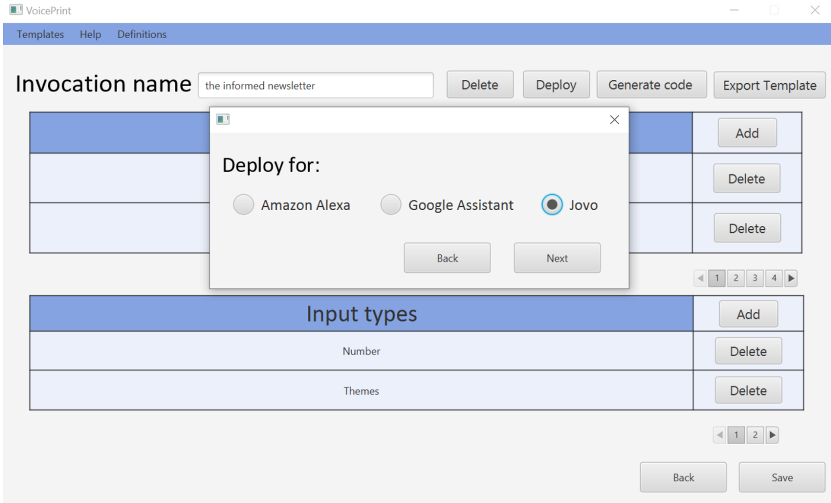


Fig. 31.: VoicePrint - Deploy pt.I

Following the generation of the application components, it was necessary to provide the credentials of both digital assistants and an AWS Lambda endpoint (Fig.32). The platform-specific language models were deployed to the Alexa developer console and to Google DialogFlow and the boilerplate code was deployed to AWS Lambda.

It should also be mentioned that, considering the requirements of the voice application, it was decided that there wasn't the need to select a database (Fig.32) in order to support the persistency of information between sessions. The setup code for the configuration of a database would have been present in the generated boilerplate code.

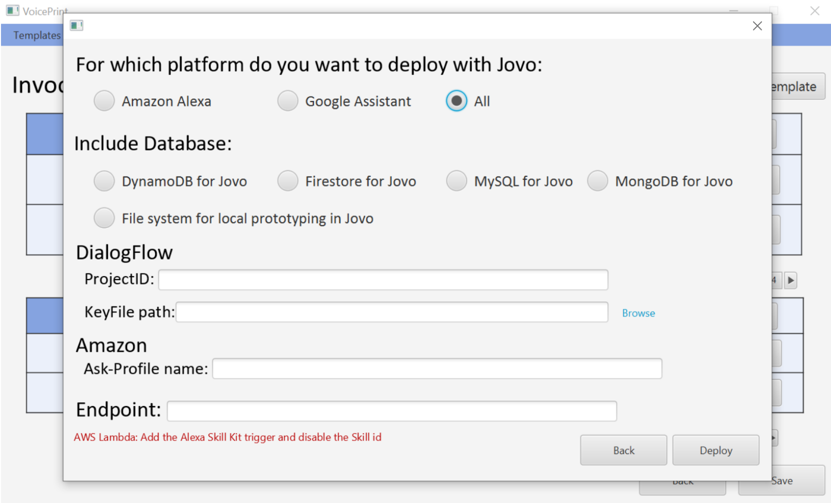


Fig. 32.: VoicePrint - Deploy pt.II

In regard to the development of the backend, it was developed recurring to Jovo and to the generated boilerplate code in Node.JS. Coming from a background in web and mobile development, it can be stated that developing for voice has its challenges and differences, when compared to the development in those two areas (section 2.5.1).

While developing The Informed, the main challenge was making sure that its responses sounded human to the user and that the user STM wasn't overloaded, especially when delivering him the news. In this type of applications, the way that the responses are constructed and delivered to the users can have a more negative impact when compared to mobile or web applications.

Other important aspects that had to be dealt with were, for instance, letting the user know what he could do in each stage of the application and guide him through it without making him feel confined to a certain path in the application, and also develop a way to gather information from the user that didn't add unnecessary complexity and seemed bothersome to him.

These aspects were dealt with the employment of the proper voice design patterns (section 2.5.1), such as, the following ones:

- The Form filling pattern to ask the user to provide once more a previous input, because its previous response didn't match what the application needed from the user to fulfill its request;
- The Structured audio pattern, when it came to structuring how the news and headlines would be presented to the user;
- The Active reference pattern, in terms of always telling the user what he can do next in the application when his request is fulfilled;
- The Index pattern was used to define how help should be provided to the user in terms of telling him what can be done and how (Help Intent);
- The Decoration pattern in the development of the Welcome Intent in order to know if the user is a frequent user or not. If not, the welcome phrase will be decorated with more information.

Posterior to the development of the backend, it was time to test the voice application in the developer console of both digital assistants in order to hear how the defined responses sounded when spoken by them and also to verify if the conversation model had been correctly planned and developed. An example of the tests performed in the developer consoles can be seen in Annex A.11.

Furthermore, the testing phase also consisted in using the AWS Lambda management console, whenever there were errors in the application backend that needed to be fixed. This management console was used due to the possibility of providing the JSON request, that triggered the defective response, in order to mimic the user's request and know more about the error(s) that occurred and where they occurred in the backend code. Additionally, this console also offers access to the CloudWatch logs, which allowed the examination of the full stack trace.

Concluded the development and testing phases, it was verified that The Informed voice application was functioning correctly in both digital assistants and that all the requirements had been met.

5.4 Shopping voice application - Bag it

Bag it will be a voice application dedicated to let the users make purchases in the Etsy e-commerce website. This application will be of the conversational type due to the fact that it will need to engage in a conversation with the user in order to know more about the product he wants to buy, from which seller he wants to buy the product and how many of it, among other details.

The first development step consisted in the definition of both its functional and non-functional requirements. Those requirements are presented in Annex A.3. However, the requirements had to be modified, more specifically, the ones pertaining the purchase aspect of the voice application. Such was due to a conflict between the Etsy API and one of the digital assistants requirements related to the process of account linking. Account linking is the process that will let the user log, via the voice application, into its Etsy account and it's required by the assistants in order for the application to be able to let the user pay for the content in its shopping cart.

Etsy API still uses OAuth 1.0 [26] which enters in conflict with the assistants requirement of their voice applications only allowing account linking to be performed via Oauth 2.0 [8] [22]. This conflict affects the ability of letting the users pay for their shopping cart content because the common way that the digital assistants allow the purchase of physical products is through account linking. Additionally, it was planned to use Paypal to let the users make payments via the application. However, not only it demands the use of OAuth 2.0 but it also didn't work with the allowed payment methods of the assistants.

Therefore, the requirements related to letting the user purchase products via Bag it had to be left as future work. Nonetheless, it was decided that it would still be possible to let the user mimic the purchase of its shopping cart in order for him to be able to know the total cost of the cart's content. Additionally, the user will also be able to ask for the status of its previous purchases.

Having decided which changes needed to be done to the purchase requirements, there was still one decision left to be made, that was caused by the mentioned conflict, regarding the account linking. As it was previously mentioned, the digital assistants require that the account linking process uses OAuth 2.0 yet Etsy still uses OAuth 1.0, a fact that could undermine the use of its API. Faced with this challenge, it was decided that the second development step would consist in the development of an OAuth 2.0 authentication server on top of the one that Etsy provided.

The OAuth 2.0 authentication server was developed recurring to JavaScript (Node.JS) and the Express framework. Express was chosen due its philoshopy of being a fast, robust and easy to use framework, that allowed the creation of the necessary HTTPS API for the OAuth 2.0 authentication process. Furthermore, it was also decided to use the EtsyJS NPM package, that is an asynchronous Node.JS wrapper to Etsy v2.0 API. This wrapper provided the necessary authentication methods and OAuth 1.0 tokens to the development of the server.

Regarding OAuth 2.0, it's a complete rewrite of the previous OAuth 1.0 and 1.1 processes and is not backwards compatible with them, an aspect that lead to the necessity of creating a turn-around with the available OAuth 1.0 tokens. The OAuth 2.0 requires two types of tokens, the access token and the refresh token. However, OAuth 1.0 doesn't use a refresh token, given that, in this version of OAuth, the access

token never expires. Therefore, it was decided to use the access token as both access and refresh token, given that this token in OAuth 1.0 only expires if the user chooses to log out.

In general the authentication process will consist in first gathering a few session informations from the digital assistant and then in redirecting the user to the Etsy login page. When the user inserts its credentials and tries to log into his account, the access and secret tokens will be fetched and, for instance, the access token will be used as an authorization code.

Afterwards, the authorization code will be used to fetch an access and refresh token pair from the developed server. This token pair will be sent back in a JSON to the assistant authorization server. After this process is concluded, the user will be redirected to a page, provided by the digital assistant, that tells him if everything went well in the login process.

It should also be noted that, even though it was managed to mock the existence of OAuth 2.0, it was decided that the resolution of not allowing the user to make purchases in the voice application should be maintained. Such is due to the fact that the application is still using OAuth 1.0 underneath the authentication process and not OAuth 2.0 as the digital assistants require.

Concluded the development of the authentication server, the third development step consisted in employing the Persona design pattern (section 2.5.1). This design pattern helped in the definition of how the voice application responses would be structured, by taking into account how they would sound to the users. The defined Persona is presented in Annex A.6.

The fourth development step consisted in the definition of the application conversation model and in the specification of its functionalities recurring to the development of high-level activity diagrams. The main flow of conversation between the user and Bag it and the specification of the search product, add product to cart and ask for the next result functionalities are presented in Fig.8, Fig.9, Fig.10, and Fig.11, respectively. Akin high-level activity diagrams were also developed for the remaining functionalities, presented in Fig.8, therefore it was decided that showing their definition wouldn't add value to this section, so it was decided not to present them.

Concluded the specification and modeling phases, it was time to choose the tools that were going to be used to develop the backed of the voice application. Once more, it was decided to use the Jovo framework, given that Bag it will be a cross-platform voice application and, as it was mentioned, the construction process can be used together with this framework.

In regard to the API chosen, as it can be deducted by the beginning of this section, it was decided to use the Etsy API along with a Node.JS wrapper for it, denominated EtsyJS. It should also be mentioned, that initially it was planned to use some Ebay APIs to implement this application. However, those APIs were still in beta and the functionalities, that were available to the developers that weren't in the beta program, weren't sufficient for the implementation of Bag it.

The following step consisted in creating and specifying the frontend of the application. The development of the frontend consisted in using VoicePrint (Fig.23) in order to create and define the language model template. This template will later on be used to generate the platform-specific Jovo language model and the boilerplate code.

Overall, the definition of Bag it language model template was akin to the definition of The Informed template, that was presented in the previous section. The general steps consisted in creating the custom template and in defining the following components:

- Application invocation name;
- Intents;
  - Sample phrases;
  - Inputs;
  - Context states.
- Input types.

Such similarity in the definition of the language model shows that the use of the construction process and, therefore, of VoicePrint won't be affected by the differences in the requirements of the voice applications, which means that, in normal use cases, the users workflow with VoicePrint will be consistent.

Given that the process of defining the built-in Intents and Input types is similar to the one used in the development of The Informed frontend, it was decided that in this section it would be best to give more focus to the definition of Bag it custom Intents and Input Types. Therefore, the custom Intents that pertain the search product, add product to cart and ask for another result functionalities will be presented.

These Intents were chosen because they were already presented in the explanation of the high-level activity diagrams (section 4.3.1) so its possible to show, for instance, how it was decided to handle with the reception of the name of the product (an Input) that the user wants to buy. Furthermore, the definition of these Intents is representative of the remaining application functionalities in terms of specification steps and level of difficulty.

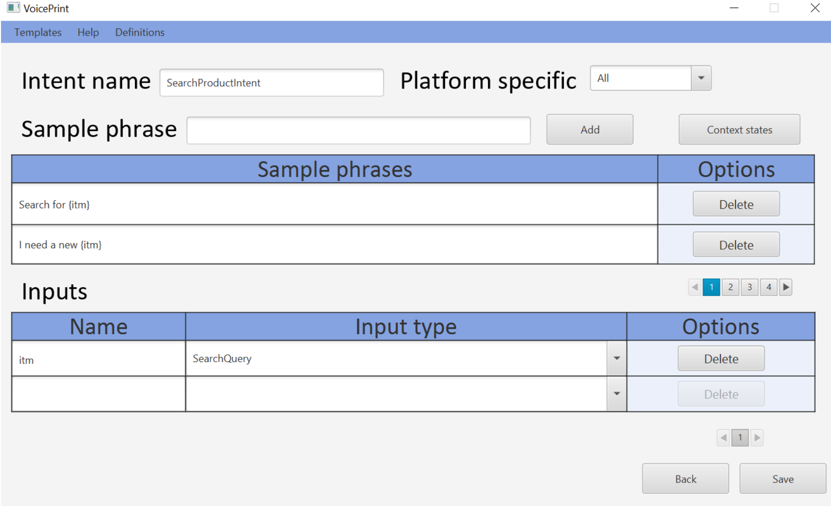


Fig. 33.: VoicePrint - Search product

In the Search product Intent (Fig.33), it was defined its name, the sample phrases, and if it was platform-specific or not. In regard to the use of context states, it was decided that the application didn't needed to

be in a particular state (an input state) in order for the user to be able to search for the product he wants to buy. However, there was the need to define an output state, denominated Searching State, due to the fact that this Intent will be split in two Intents and also because when the application asks the user if he wants to hear the rest of the results from the search process, a particular Yes/No Intent will need to be triggered.

It was decided to separate this Intent in two in order to not focus a somewhat convoluted process in only one function, which could add unnecessary complexity in its future modification and make the maintenance of the code more bothersome.

The first Intent (Fig.33) will deal with the user's request and will activate the necessary context state for the second Intent to be activated. This means that the first Intent will define an output state that will be the input state of the second one.

The second Intent, will check if the user wants to filter its search by minimum or maximum price or if he doesn't want to apply any filter. Additionally, this Intent will deal with the search process on the Etsy catalog and will present to the user four or less products, that fit its request. In case the search process returned more results, the user will be asked if he wants to hear about the remaining products that were found. Contrary to the first Intent, this one won't change the state of the application and the default state will be restored when the user says he wants/doesn't want to hear about the remaining products in the result list, which triggers the execution of a specialized Yes/No Intent.

It should also be mentioned that, in the cases that the search process returned less than four products, they will all be presented to the user and the application will ask him which one he wants to buy and also inform him that he can ask for information regarding the seller of one of the presented products. Furthermore, in this case, the Intent will set the application state back to default, because the search process for the requested product ended.

Regarding the Inputs, the Search Intent will use an Input of the type Search query in order to be able to discover the name of the product that the user wants to buy. The use of Search query, which is a more generic type, imposes two restrictions, the Intent cannot combine Inputs of this type with Inputs of other types in its sample phrases and the Intents can only have one Search query Input.

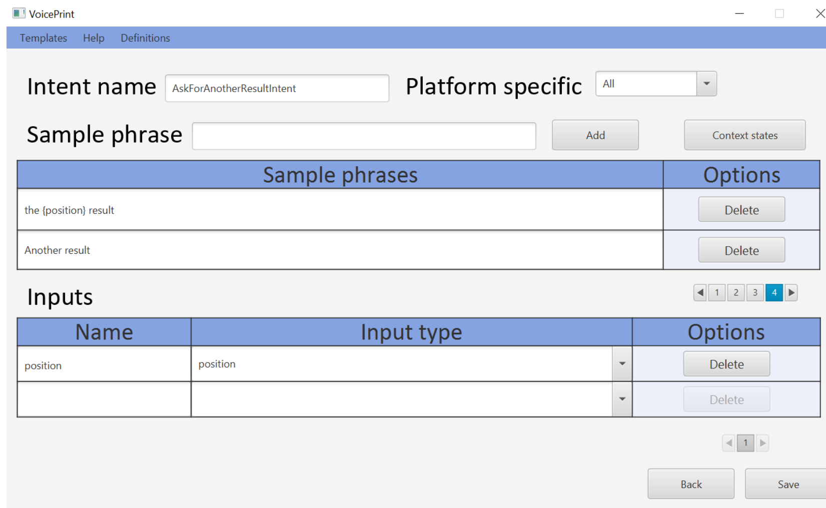


Fig. 34.: VoicePrint - Ask for another result

In the Ask for another result Intent (Fig.34), it was defined its name, the sample phrases, and if it was platform-specific or not. No context states were defined for this Intent. Such is due to the fact that there wasn't the need to use, for instance, an input state to allow this Intent to be triggered, because if the user didn't searched for a product and he asks for another result, the application will simply tell him to search for a product first. This Intent was developed in order to be used, for instance, after the user asked for information about the seller of a certain product but didn't liked its rating and now wants to hear about another product. Therefore, this Intent will allow the user to traverse the result list one product at the time and hear about the product characteristics (e.g its price).

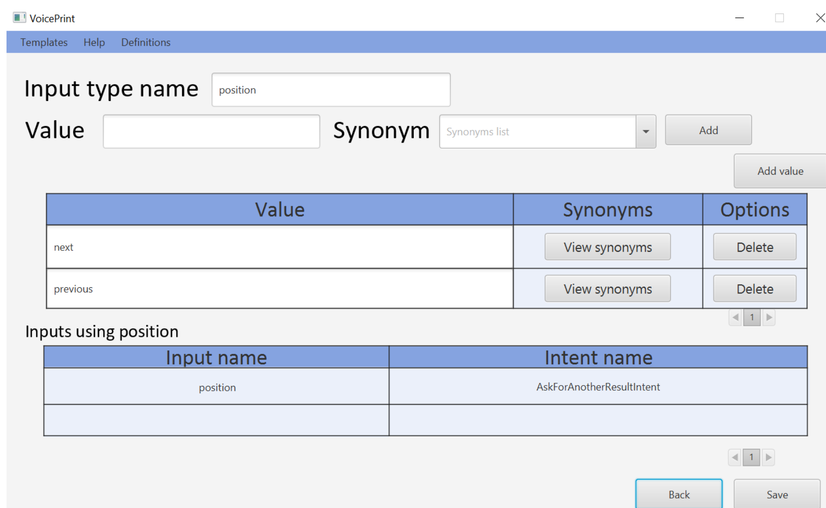


Fig. 35.: VoicePrint - Position Input type

The Ask for another result Intent will also use an Input, denominated position (Fig.35), in order to allow the user to voice its request using the words next and previous, which is a more practical way to ask for another result. It was necessary to create an Input for these two words due to the fact that Amazon Alexa reserves them for Intents that allow the user to ask Alexa to play music, for instance, of a playlist. Given

that this application won't have those music related Intents, the custom Input position had to be created in order for the user to be able to request the previous or the next result.

Fig. 36.: VoicePrint - Add product to cart

Fig. 37.: VoicePrint - Specify quantity of product to add to the cart

In the Add product to cart Intent (Fig.36), akin to the definition of the other Intents, it was defined its name, its sample phrases, and if it was going to be platform-specific or not. Regarding the use of context states, this Intent won't need an input state but will activate the output state Add product. Such is due to the fact that when the user says he wants to add a product to its cart, the application will tell him that there is only a certain amount of that product available and that he needs to specify how many of it he wants to buy. The specification of the quantity of the product to be bought will trigger an auxiliar Intent denominated Add product quantity cart (Fig.37), whose input state will be the Add product state and not the application default state.

It was decided to define the handling of the specification of the quantity of the product to be bought in another Intent and not in the Add product to Cart Intent, due to the fact that it would add unnecessary

complexity to the code of the functionality and it would make future peer revisions and/or maintenance of the developed code a more challenging process than what it needs to be. It should also be mentioned that, the Add product quantity cart Intent will set the application state back to default when it's done executing.

Regarding the Inputs used by the Add product to Cart Intent, it will use Inputs related to the specification of which one of the presented products the user wants to buy. The user will be able to specify the product or by its position on the results list, which uses an Input of the ordinal type, or by its name, which uses an Input of the type Search query. Additionally, the Add product quantity cart Intent will use an Input of the number type, in order to allow the users to specify the quantity of the product they want to buy.

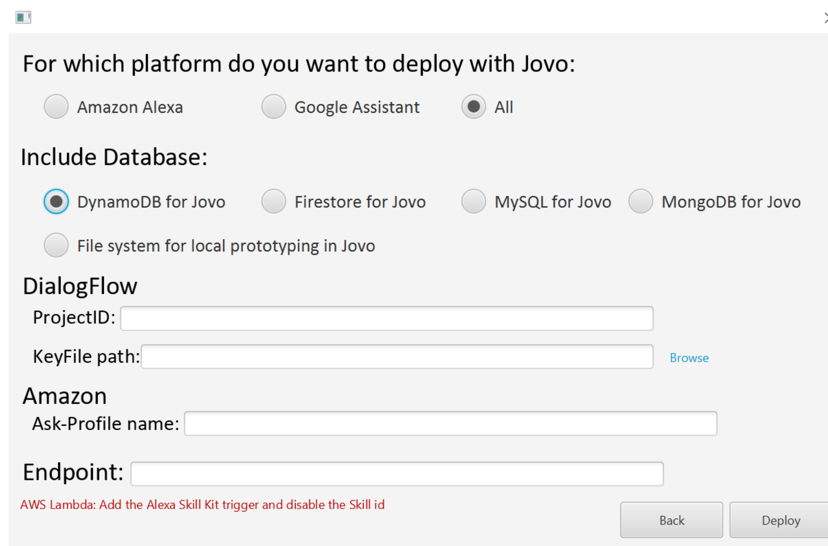


Fig. 38.: VoicePrint - Bag it deploy

The last step consisted in deploying the application components, the platform-specific language models and the boilerplate code, to Alexa and Google Assistant with the help of Jovo, as it was explained in section 4.3.5. First, it was chosen for which platform the application would be deployed, which in this case was Jovo (Fig.38). The deployment will then generate the Jovo language model and boilerplate code.

Furthermore, it was decided to prepare the configuration of a DynamoDB database in the boilerplate code, due to the fact that it was decided to use a database to store information such as the result list of the search process or the position of the product that the user claimed he wanted to know more about, so it's easier to retrieve the product if he ends up wanting to buy it.

Finally, the only information missing is the credentials of both digital assistants and an AWS Lambda endpoint, given that it was decided to host the backend of the application in AWS.

Regarding the development of the backend, it was developed recurring to Jovo and to the generated boilerplate code. In regard to the communication of the backend code with DynamoDB, the supporting methods that Jovo offers were used, which made the process of reading/writing data to the database transparent.

The challenges faced, while developing Bag it, were similar to the ones faced while developing The Informed. Some of the challenges regarded the respect of the user STM, the way that the information was

presented to the user and how it would sound to him, and the gathering of information about the product the user wants to buy without making him exasperated with the amount of possible questions.

Once more, to deal with these challenges it was resorted to the voice design patterns (Section 2.5.1), being that the following ones were applied:

- The Index pattern was used in the development of the Help Intent in terms of how help should be provided to the user;
- The Active reference pattern in terms of telling the user here he is in the application and what he can do next, when his request is fulfilled;
- The Information spreading pattern in order to provide the correct amount of information to the user without overwhelming its STM;
- The Structured audio pattern in the structuring of the application responses, such as the presentation of the products list, that resulted from the search process;
- The Decoration pattern in the development of the Welcome Intent in order to provide more information to an user that doesn't use the application with frequency;
- The Detailed information pattern for when the user asks for more details about a certain product;
- The Form filling pattern to ask the user for the filter he wants to apply to his search, if any.

Posterior to the development of the backend, some tests were performed to the voice application. The testing phase was akin to the one applied to The Informed application due to the fact that the way the tests were conducted allowed for a thorough examination of both the voice application frontend (e.g how the responses sounded) and backend (e.g if there were errors and where).

The tests were then made in the developer console of both digital assistants and in the AWS Lambda management console. An example of the tests that were performed in the developer consoles can be found in Annex A.11.

Concluded the testing phase, it was verified that the voice application, Bag it, was functioning properly in both digital assistants and that almost all the requirements had been met. As it was aforementioned, some of the requirements related to the purchase of products had to be altered due to the lack of support to OAuth 2.0 by the Etsy API.

5.5 Results

In this section, the results obtained from the development of the previously presented voice applications, via the VoicePrint platform, will be discussed in terms of their quality and ease of development. Such is necessary in order to analyze the proposed construction process, the high-level activity diagrams and VoicePrint as tools for the development of cross-platform voice applications.

Regarding the high-level activity diagrams, that were presented in chapter 4.3.1, they successfully helped in the modeling of the conversation model of both voice applications, even though their models were very distinct from one another, given that one application would entail a dialogue with the user while the other would only provide the user with answers to its requests. Additionally, these diagrams were also used to plan the voice applications functionalities.

By defining those activity diagrams, the phase of specification of the language model templates was straightforward due to the fact that it was already known, for instance, what the user would need to say to invoke a certain funtionality (the sample phrases) or if he would need to provide Inputs for its request to be fulfilled and which ones. Furthermore, the diagrams along with the developed personas also helped in the definition of the applications responses to the user on the development of the backend code.

Concluded the development of the activity diagrams for the two case studies, it can be claimed that they can indeed help the developers in the specification of the voice applications conversation models and functionalities. The two possible downsides were already mentioned in section 4.2.1 and they are the following ones:

- Developers that aren't so familiar with modeling in UML may have a higher learning curve than the rest of the developers when defining these diagrams;

- Due to the notation rules of the UML activity diagrams the developers might feel that they don't have enough freedom to express the voice application conversation model.

However, taking into consideration the advantages that using activity diagrams can bring to the developers, it's still considered that, despite the possible downsides, these diagrams are still an appropriate tool that can decrease the complexity of the development of the application conversation model.

In regard to the proposed platform independent construction process, with the development of these case studies it was shown that its components work properly and also that they produce the intended results.

In this case, only the language model generator and boilerplate code generator for Jovo were tested, given that the remaining generators were tested separately using unit testing, and when they were applied to the language model template, they produced the expected platform-specific language model and boilerplate code, respectively.

The deployment of both voice applications used the deployment module for Jovo, given that the remaining deployment modules were tested via unit testing, which interacted with the previously mentioned generators, in order to trigger the generation of the necessary components for the deployment process. The results of using the deployment module were as expected and showed that the interaction between this component and the generators was successful. However, it was noted that some warnings, such as telling the user to disable the Alexa skill trigger in the AWS lambda function, should be displayed to the user so the deployment process would be successful.

A possible future improvement that could be performed upon the proposed construction process would be to add a feature that allowed the developer to provide a platform-specific model, for Alexa or Google

Assistant, that would then be converted to a generic language model template. This would allow the developer to utilize an existing language model, whose generic version could then be edited in VoicePrint and also generated or deployed for other digital assistants. With this feature, the developer would be able to more easily port an existing voice application language model to another digital assistant.

Regarding the VoicePrint platform, that abstracts the construction process by providing it with a visual editor, it was shown, for instance, in Figure 23, Figure 26, Figure 35 and Figure 38. A future improvement that could be performed upon the platform would be redesigning the user interface, while maintaining its overall work flow.

By using this platform to apply the proposed construction process in the development of the aforementioned voice applications, it was concluded that it was indeed a practical and user-friendly tool, given that it provided easy access to the process functionalities and wasn't complex to understand and use, as it was intended (section 1.4). Additionally, this platform eliminates the need to manually write the platform-specific language models in JSON files

The analyze to the proposed tools in this master's dissertation led to the conclusion that they work as it was intended and produce valid responses, for instance, regarding the generation of platform-specific language models from a generic language model template. Therefore, the construction process was able to resolve or handle in the best way possible the challenges presented in section 3.3, and still produce valid results. However, as it was aforementioned in this section and in chapter 4, there are still some improvements that could be made to the construction process and VoicePrint.

5.6 Unit testing

Precedently to the development of the unit tests, a study concerning the different types of testing techniques that exist was performed. The two techniques that seemed more adequate for this dissertation were the Black box and White box testing techniques. It should also be mentioned that there is a testing technique called Gray box testing, which is a combination of the two aforementioned techniques.

Black box testing technique consists in testing the functionalities of the system without knowing how its structured internally and how the code and the communication between modules or external systems was implemented. Essentially, these types of tests are more focused in the system requirements and specifications than on how the system was programmed and can be developed to either test the functional or non-functional (e.g scalability) requirements. It should also be mentioned that these tests are named black box tests due to the fact that an input is provided and an output is produced without the tester knowing what happened in the system.

White box testing technique consists in testing the functionalities of the system with knowledge of how it works internally in terms of structure and implementation. These types of tests will be focused on validating the flow of inputs through the code and if the expected output is produced. It should also be mentioned that due to the fact that these tests are more focused on testing the components functionality, they aren't suited, unlike the black box tests, to test the communication among modules or of the developed system with external systems such as, for example, a database.

Unit and integration tests are used by developers to implement the aforementioned testing techniques. In regards to the unit tests, they are used to implement the White box testing technique by allowing the development of tests that will be applied to a single component of the system and that will assert if its behaviour/state/output is the intended one. This means that for each component of the system, unit tests most be developed and applied.

Regarding the integration tests, they are used to implement the Black box testing technique by mimicking the user interaction with the system. This means that the whole system components and their interaction will be tested in order to assert if the system as a whole is working as intended.

The proposed construction process is composed by independent generators and deployment components, which allows the flexibility to modify each one of those components without affecting the others and to add a new generator or deployment module without compromising the whole process. Therefore, given that the process components are independent of one another and no external dependencies were used, it was decided to apply the White box testing technique. Unit tests were then developed to individually test the construction process components and to ease their maintainability and future improvement.

Every time a component is modified the unit tests, that were developed for that component, are performed in order to check for any problems that may have been introduced with the modification and to verify if everything is still working as it was intended. Regarding the scope of the unit testes, they were developed for each platform-specific language model generators, boilerplate code generators and for the Amazon Alexa, Google Assistant and Jovo voice application deployers.

It should also be mentioned that all the tests are going to use the same language model template as a test fixture in order to ensure that there is an environment in which we can check if the obtained results can be repeated. Furthermore, it was decided to also measure the average time that each component takes to execute its functionalities while performing the unit tests. The time was measured five times and then the average of those times was calculated, being that the averages are presented in Annex A.10.

In regard to the development of the unit tests, it was decided to use the framework JUnit because it allows the use of annotations to write the tests, which made the coding of the tests a straightforward process, and provides different types of assertions to verify the results of the tests execution. Additionally, JUnit also allows the tests to be organized in test suites and provides test runners to automatically run the tests, that were specified in the test suites, and provides the results in a simple way.

5.7 Summary

Throughout this chapter, the development of the case studies, that served as proof of concept for the proposed construction process, were presented and explained in detail. These case studies, one voice application of the conversational type and another one of the request-response type, correspond to possible real scenarios in order to demonstrate what can be done with the proposed construction process via the VoicePrint platform.

Firstly, before the development of the applications, a market study regarding the top four categories that have the highest volume of voice applications in the digital assistants app stores was conducted. Having

concluded the market study and by analyzing and cross-referencing the top four categories of Alexa and Google Assistant, it was decided that the request-response voice application would allow the user to ask for news or headlines and the conversational voice application would allow the user to shop for goods in an e-commerce website.

With a clear vision of what the case studies would be, their development via the platform VoicePrint began. The goal of the development of these case studies was to extensively test the functionalities of the proposed construction process, from the creation, specification, generation and deployment of a custom language model template to the creation or use of built-in Intents and Input types. By testing the various components of the construction process, it was intended to prove that it could indeed be used to develop cross-platform voice applications.

For each one of the development steps, going from the creation of the language model template to its deployment, a description of them along with images of VoicePrint, in order to show how the platform was being used and would be used by regular users, was presented. How the voice applications components were tested in order to verify if they worked accordingly to their functional and non-functional requirements was also presented.

In this chapter, a brief explanation regarding the testing phase that was performed during the implementation phase was also presented. The White box testing technique by way of the development of unit tests was applied in order to verify if the components of the construction process were working as intended. Additionally, by having these unit tests if a component is modified in the future they can be used to assure that it's still producing the expected results.

In conclusion, in the end of this process, both voice applications were working as intended and it was demonstrated that the proposed construction process allowed the successful development of cross-platform voice applications for both Amazon Alexa and Google Assistant. It was also demonstrated that the challenges previously presented in chapter 3.3 weren't an impediment to the development of the process. Furthermore, the development of unit tests to the components of the construction process allowed the verification of their behaviour and the uncovering of possible logical errors in their functioning that would lead to the production of a flawed output.

6

CONCLUSION

Throughout this dissertation it was presented the studies regarding the state of the art in the areas of digital assistants and voice applications, that influenced the decisions that guided the implementation phase, and the way that the proposed construction process was implemented and tested. Now, it's time to evaluate all the work that has been done. Additionally, a list of possible future improvements and updates that can be done to the proposed construction process and platform will be presented.

The first step towards the standardization of the development process of cross-platform voice applications is the creation of platform-independent tools and mechanisms that can increase the efficiency of the process and reduce the time that the developer has to spend performing duplicate work because of platform-specific rules. The absence of standardization makes the development of cross-platform voice applications more complex and time-consuming for the developer due to the current plethora of different digital assistants that have their own application models and development tools.

Regarding the developers perspective, the insuccess and low retention rate of their voice applications [56] may demotivate them from developing more applications of this type. That may lead to a general disinterest for this field and thus to a small developer community and slower advances on this technology.

However, the main issue with the absence of standardization is that the developers don't have a common "language" to use to share ideas or development steps with one another and platform-independent rules to guide them towards the development of voice applications that have a good user experience across digital assistants. Thus, it was proposed, designed and developed in this master's dissertation a set of tools that aims to help mitigate the absence of standardization and ease the development of cross-platform voice applications.

The first tool that was developed was the high-level activity diagrams, that are a tool that allows the developers to more easily model the frontend (e.g helps in the definition of the language model) and communicate ideas with others (e.g client, etc). The other tool that was developed was the platform-independent construction process of voice applications for Amazon Alexa and Google Assistant, whose design and development was this dissertation main goal. The developers will be able to use this construction process via another tool that was developed, which was a platform, with an incorporated visual editor, denominated VoicePrint.

The proposed construction process aims to promote a platform-independent definition of the frontend and to automatize the development of cross-platform voice applications, thus making it easier to develop

and test them and also to maintain a consistent user experience throughout devices. The core component of the process is a cross-platform language model template that was achieved by analyzing the common components of the digital assistants' language models (section 4.2).

This construction process allows the developers to only have to define one language model template in order to specify and generate the frontend and the boilerplate code for the backend functionality, for a cross-platform voice application. Namely, this will lead to less errors, that could appear due to having to define the same language model more than once because of digital assistants' rules, and to a more efficient development process. Lastly, by using this process the developer can be more focused on the requirements and fundamental parts of the voice application (e.g functionalities, user experience, etc) rather than on specific details of the digital assistants application models.

VoicePrint was developed in order to provide both novice and more experienced developers with a practical and user-friendly graphical user interface through which they can use the aforementioned construction process and its provided functionalities. Additionally, VoicePrint will allow the developers to more easily iterate over their language model template components and streamline the use of either built-in Intents or Input Types, and also eliminate their necessity of manually writing the frontend language model specification in JSON. This necessity exists, for instance, when the developers are going to use only the framework Jovo to develop their voice application.

Regarding the way the aforementioned tools work and the way they can be used by the developers, two voice applications, presented in chapter 5, were developed recurring to them. With the development of these applications it was possible to show that the tools indeed work as intended and help streamlining the development of cross-platform voice applications. Additionally, it was also possible to find out and fix what as most visibly missing in VoicePrint in terms of functionalities, warnings, etc, and whose existence could improve the platform user experience.

Therefore, the developers can use the high-level activity diagrams in the definition phase of the voice application to model the frontend and VoicePrint in the development phase to create the language model template and then generate the frontend components and the boilerplate code for the backend or instead deploy the application. Furthermore, by supporting a few frameworks this construction process can be used along side them, most likely in the initial development phase, and by providing a graphical user interface it allows the developers to not have to specify the language model by hand.

Regarding future work, when the development process of the aforementioned tools was finished and after they were used to develop the presented case studies, it was possible to obtain a global assessment concerning the functionalities provided by the construction process and how they could either be improved or extended. The following list contains a set of functionalities that can either be extended in the future or added to the construction process in order to improve it.

- Develop a boilerplate code generator for the frameworks Violet and Voxa. Violet because not only it was decided early in this dissertation that the construction process would provide support for it but also because its recent updates made it reach a stable state, for instance, in terms of the way the developers can code the backend. Voxa because it's another cross-platform framework for Amazon Alexa and Google Assistant, whose support would bring more value and diversity to the process;

- Apply version control to the generation process of boilerplate code. This means that the developers would be able to not only the use the generated code files but also be able to change their language model template and then generate new code files without losing the changes that had already been done in the previous files;

- Allow the developers to provide an existing platform-specific model, for Alexa or Google Assistant, to VoicePrint. This model would then be converted to a generic language model template, that could be edited in the platform.

- Provide built-in Intents and Input Types for other languages aside from English (US) due to the fact that the construction process accepts sample phrases in other languages;

- Allow the developers to define either one or a list of Input or Output states in order to allow, for instance, the definition and use of nested states in the backend (4.3.2);

- If possible in the future, the correspondence process of built-in Input types, presented in section 4.3.2, should be improved. This improvement would consist in providing the built-in Input types of Google Assistant, that aren't already provided, by corresponding them, for instance, with a generic Alexa built-in Input type, that isn't the AMAZON.SearchQuery type;

- In case VoicePrint and, therefore, the construction process ever have the opportunity to be released to the market, usability tests such as SUS (System usability scale) or Cognitive Walkthrough should be performed. These tests would allow the assessment of the usability of VoicePrint and of what should be added or improved (e.g improve the design of the UI).

To conclude, at the end of this master's dissertation it's possible to claim that the proposed goals were achieved and it was indeed possible to define and develop a platform-independent construction process for the development of cross-platform voice applications, despite the differences that existed between the application models of the assistants, that in this case were Amazon Alexa and Google Assistant. Even though this process can be improved, as stated above, it fulfills all the proposed main functionalities and by being provided via a platform, with an incorporated visual editor, makes it easier to define, for instance, the frontend of the voice application. It's hoped that the work developed in this master's dissertation is useful for the future work in the field of voice applications and in the way that these applications are developed for more than one digital assistant.

BIBLIOGRAFIA

[1] Conversion funnel. www.en.wikipedia.org/wiki/Conversion_funnel. Accessed: 2018/10; Online.

[2] The Internet of Things: a movement, not a market Start revolutionizing the competitive landscape.

[3] Webhook. www.en.wikipedia.org/wiki/Webhook. Accessed: 2018/11; Online.

[4] D. A.Coates. Voice Applications for Alexa and Google. Manning, 2018.

[5] R. Agency. Voxa's documentation. https://voxa.readthedocs.io/en/latest/index.html. Accessed: 2019/04; Online.

[6] Amazon. Alexa design guide. www.developer.amazon.com/docs/alexa-design/intro.html. Accessed: 2018/10; Online.

[7] Amazon. Slot type reference. https://developer.amazon.com/docs/custom-skills/slot-type-reference.html. Accessed: 2019/03; Online.

[8] I. Amazon.com. How account linking works. https://developer.amazon.com/docs/account-linking/how-account-linking-works.html. Accessed: 2019/04; Online.

[9] Ashish. 11 things cortana can do for you. www.wpxbox.com/what-can-cortana-do/. Accessed: 2018/12; Online.

[10] C. Baber. Interactive speech technology. chapter Developing Interactive Speech Technology, pages 13–18. Taylor & Francis, Inc., Bristol, PA, USA, 1993.

[11] O. Bahceci. Analysis and Comparison of Intelligent Personal Assistants. 2016.

[12] K. C. Becker and F. I. Parke. Developing a speech-based interface for field data collection. 2016.

[13] Bixby. Moments. https://bixbydevelopers.com/dev/docs/dev-guide/design-guides/service#moments. Accessed: 2019/06; Online.

[14] S. M. Brennan. Conversation as direct manipulation: an iconoclastic view. 1990.

[15] T. M. Brill. Siri , Alexa , and Other Digital Assistants : A Study of Customer Satisfaction With Artificial Intelligence Applications. 2018.

[16] R. Canavarro and A. Ribeiro. Platform for the creation of cross-platform voice applications.

[17] M. Cassinelli. Siri shortcuts: Everything you need to know! https://www.imore.com/siri-shortcuts-faq/. Accessed: 2019/1; Online.

[18] L. Cerejo. Designing voice experiences. www.smashingmagazine.com/2017/05/designing-voice-experiences/. Accessed: 2018/10; Online.

[19] B. R. Cowan, K. Morrissey, N. Pantidi, P. Clarke, D. Coyle, S. Al-shehri, D. Earley, and N. Bandeira. " What Can I Help You With ?": Infrequent Users ' Experiences of Intelligent Personal Assistants. 2017.

[20] J. Daub. Sirikit part 1: Hey siri, how do i get started? www.bignerdranch.com/blog/sirikit-part-1-hey-siri-how-do-i-get-started/. Accessed: 2018/10; Online.

[21] R. De Renesse. Virtual digital assistants to overtake world population by 2021. https://ovum.informa.com/resources/product-content/virtual-digital-assistants-to-overtake-world-population-by-2021. Accessed: 2019/6; Online.

[22] G. developers. Account linking. https://developers.google.com/assistant/identity. Accessed: 2019/04; Online.

[23] A. Esaulov. Bottalk developer documentation. https://docs.bottalk.de/. Accessed: 2019/06; Online.

[24] T. W. O. R. Etail, S. T. L. Aboratory, and V. Venkatesh. Design and evaluation of auto-id enabled shopping assistance in customers' mobile phones. 41(X):1–32, 2017.

[25] D. Etherington. Amazon echo is a $199 connected speaker packing an always-on siri-style assistant. www.techcrunch.com/2014/11/06/amazon-echo/. Accessed: 2018/12; Online.

[26] I. Etsy. Oauth authentication. https://www.etsy.com/developers/documentation/getting_started/oauth. Accessed: 2019/04; Online.

[27] O. Etzioni, M. Banko, and M. J. Cafarella. Machine Reading. pages 1517–1519, 2006.

[28] M. J. Foley. Microsoft's cortana now has 230 skills (and amazon's alexa, 25,000). www.zdnet.com/article/microsofts-cortana-now-has-230-skills-and-amazons-alexa-25000/. Accessed: 2018/11; Online.

[29] S. Ghosh and J. Pherwani. Designing of a Natural Voice Assistants for Mobile Through User Centered Design Approach, volume HCII 2015,. 2015.

[30] N. Goksel-canbek. On the track of Artificial Intelligence : Learning with Intelligent Personal Assistants 1. (Iclel 2015), 2016.

[31] Google. Conversation design. https://designguidelines.withgoogle.com/conversation/conversation-design/what-is-conversation-design.html. Accessed: 2018/10; Online.

[32] Google. System entities. https://cloud.google.com/dialogflow/docs/reference/system-entities. Accessed: 2019/03; Online.

[33] Google. Templates. www.developers.google.com/actions/templates/. Accessed: 2018/11; Online.

[34] Google. Write sample dialogs. https://designguidelines.withgoogle.com/conversation/conversation-design-process/write-sample-dialogs.html. Accessed: 2018/11; Online.

[35] F. Goossens. Designing a vui. www.uxplanet.org/designing-a-vui-voice-user-interface-c0b3b9b57ace. Accessed: 2018/10; Online.

[36] I. Hargreaves. Voice assistant use to grow 1000% to reach 275 million by 2023, juniper says. www.pcworld.idg.com.au/article/642969/voice-assistants-use-grow-1000-reach-275-million-by-2023-juniper-says/. Accessed: 2018/10; Online.

[37] J. Hirschberg and C. D. Manning. language processing.

[38] M. B. Hoy. Alexa, Siri, Cortana, and More: An Introduction to Voice Assistants. Medical Reference Services Quarterly, 37(1):81–88, 2018.

[39] K. Johnson. Every new google assistant feature announced at i/o 2019. https://venturebeat.com/2019/05/08/every-new-google-assistant-feature-announced-at-i-o-2019/. Accessed: 2019/06; Online.

[40] K. Johnson. Google introduces how-to markup for voice app developers. https://venturebeat.com/2019/05/07/google-introduces-how-to-markup-for-voice-app-developers/. Accessed: 2019/06; Online.

[41] A. Jönsson and N. Dahlbäck. The Role of Spoken Feedback in Experiencing Multimodal Interfaces as Human-like. 2003.

[42] D. Jurafsky and J. H. Martin. Speech and Language Processing 3rd edition draft. 2018.

[43] C. Kamm. User interfaces for voice applications. Proceedings of the National Academy of Sciences, 92(22):10031–10037, 1995.

[44] Kapitall. Why cortana assistant can help microsoft in the smartphone market. www.thestreet.com/story/12534433/1/why-cortana-assistant-can-help-microsoft-in-the-smartphone-market.html. Accessed: 2018/12; Online.

[45] A. A. Karpov and R. M. Yusupov. Review Multimodal Interfaces of Human – Computer Interaction. 88(1):67–74, 2018.

[46] O. Z. Khan, R. Sarikaya, M. Corporation, and R. Wa. Making Personal Digital Assistants Aware of What They Do Not Know. pages 1161–1165, 2016.

[47] A. Kharpal. Samsung to challenge amazon's alexa by letting developers make apps for its bixby voice assistant. www.cnbc.com/2018/09/04/samsung-bixby-software-developers-kit-sdk.html. Accessed: 2018/11; Online.

[48] R. Knote, A. Janson, L. Eigenbrod, and M. Söllner. The What and How of Smart Personal Assistants : Principles and Application Domains for IS Research. 2018.

[49] R. Knote, A. Janson, M. Söllner, and J. M. Leimeister. Classifying Smart Personal Assistants : An Empirical Cluster Analysis. 6:2024–2033, 2019.

[50] S. A. Konig, Jan. Jovo documentation. https://www.jovo.tech/docs/. Accessed: 2018/10; Online.

[51] S. A. Konig, Jan. Building cross-platform voice apps. In NYC Voice Assistant Meetup, December 2017.

[52] M. Linley. Google unveils google assistant, a virtual assistant that's a big upgrade to google now. www.techcrunch.com/2016/05/18/google-unveils-google-assistant-a-big-upgrade-to-google-now/. Accessed: 2018/11; Online.

[53] G. López, G. Quesada, Luís, and L. A. Alexa vs. Siri vs. Cortana vs. Google Assistant: A comparison of Speech-Based Natural User Interfaces. 2017.

[54] F. Lyardet and D. Schnelle. Voice User Interface Design Patterns. Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 2006), (January 2006), 2006.

[55] T. Maiolino. Maximus - Automatizando Tarefas por Voz. 2017.

[56] A. Marchick. The 2017 voice report by alpine. www.medium.com/@marchick/the-2017-voice-report-by-alpine-fka-voicelabs-24c5075a070f. Accessed: 2018/10; Online.

[57] M. Meeker. Internet trends 2016 - code conference. www.slideshare.net/kleinerperkins/2016-internet-trends-report?from_action=save. Accessed: 2018/11; Online.

[58] D. Meyer. Introducing alexa conversations (preview), a new ai-driven approach to natural dialogs through the alexa skills kit. https://developer.amazon.com/blogs/alexa/post/44499221-01ff-460a-a9ee-d4e9198ef98d/introducing-alexa-conversations-preview. Accessed: 2019/06; Online.

[59] P. Milhorat, S. Schlögl, G. Chollet, J. Boudy, G. Pelosi, P. Milhorat, S. Schlögl, G. Chollet, J. Boudy, A. Esposito, and G. Pelosi. Building the next generation of personal digital assistants To cite this version : HAL Id : hal-01263483. 2016.

[60] P. Miller. Google assistant will open up to developers in december with 'actions on google'. www.theverge.com/2016/10/4/13164882/google-assistant-actions-on-google-developer-sdk. Accessed: 2018/10; Online.

[61] C. L. M. C. C. B. R. Murad, Christine. Design Guidelines for Hands-Free Speech Interaction. pages 269–276.

[62] C. Myers, J. Nebolsky, K. Caro, and J. Zhu. Patterns for How Users Overcome Obstacles in Voice User Interfaces. pages 1–7, 2018.

[63] L. J. Newville. Development of the Phonograph at Alexander Graham Bell's Volta Laboratory . 1959.

[64] C. Osman and P. Zalhan. From Natural Language Text to Visual Models: A survey of Issues and Approaches. (July 2017), 2016.

[65] C. Pearl. Designing Voice User Interfaces.

[66] S. Perez. Siri shortcutes comes built-in on ios 13, allows for more powerful shortcuts. https://https://techcrunch.com/2019/06/03/siri-shortcuts-comes-built-in-on-ios-13-allows-for-more-powerful-shortcuts/. Accessed: 2019/06; Online.

[67] M. Pinola. Speech recognition through the decades: How we ended up with siri. www.pcworld.com/article/243060/speech_recognition_through_the_decades_how_we_ended_up_with_siri.html. Accessed: 2018/11; Online.

[68] I. Rhee. Bixby : A new way to interact with your phone. www.news.samsung.com/global/bixby-a-new-way-to-interact-with-your-phone. Accessed: 2018/12; Online.

[69] Salesforce. Violet - build voice apps. https://helloviolet.ai. Accessed: 2018/10; Online.

[70] Samsung. Introduction to modeling. www.bixbydevelopers.com/dev/docs/dev-guide/developers/modeling.intro-modeling. Accessed: 2018/12; Online.

[71] D. Schnelle, F. Lyardet, and T. Wei. Audio navigation patterns. EuroPLoP, (May), 2005.

[72] D. Schnelle-walka and D. Schnelle-walka. NLU vs . Dialog Management : To Whom am I Speaking ? (April), 2016.

[73] A. Sellen. " Like Having a Really bad PA ": The Gulf between User Expectation and Experience of Conversational Agents. 2016.

[74] B. Shneiderman. the Limits of Speech Recognition. 43(9):63–65, 2000.

[75] V. Shynkarenka. Hello, invocable. goodbye, clunky voice design tools. www.medium.com/storyline-blog/hello-invocable-8e8c049bb3d1. Accessed: 2018/12; Online.

[76] V. Shynkarenka. Invocable is shutting down. https://medium.com/storyline-blog/invocable-is-shutting-down-132953509f51. Accessed: 2019/06; Online.

[77] C. Stamford. Gartner says worldwide spending on vpa-enabled wireless speakers will top $2 billion by 2020. www.gartner.com/newsroom/id/3464317. Accessed: 2018/10; Online.

[78] A. S. Team. Amazon unveils novel alexa dialog modeling for natural, cross-skill conversations. https://developer.amazon.com/blogs/alexa/post/9615b190-9c95-452c-b04d-0a29f6a96dd1/amazon-unveils-novel-alexa-dialog-modeling-for-natural-cross-skill-conversations. Accessed: 2019/06; Online.

[79] V. team. Voiceflow documentation. https://docs.voiceflow.com/home/what-can-i-do-with-voiceflow. Accessed: 2019/06; Online.

[80] W. A. Team. Cortana to open up to new devices and developers with cortana skills kit and cortana devices sdk. https://blogs.windows.com/windowsdeveloper/2016/12/13/cortana-skills-kit-cortana-devices-sdk-announcement/. Accessed: 2019/06; Online.

[81] S. Technologies. Bottalk has everything you need to create voice interfaces. https://medium.com/@info_86879/bottalk-has-everything-you-need-to-create-voice-interfaces-a2820ed737a7. Accessed: 2019/06; Online.

[82] A. S. Tulshan and S. N. Dhage. Survey on Virtual Assistant : Google Assistant , Siri, Cortana, Alexa. Springer Singapore, 2019.

[83] J. Vincent. Google's next version of assistant will be dramatically faster, hitting pixel phones first. https://www.theverge.com/2019/5/7/18535637/google-assistant-voice-new-faster-search-alarm-update-io-2019. Accessed: 2019/06; Online.

[84] VoiceLabs.co. The 2017 Voice Report - Executive Summary. page 12, 2017.

[85] T. Warren. Microsoft no longer sees cortana as an alexa or google assistant competitor. https://www.theverge.com/2019/1/18/18187992/microsoft-cortana-satya-nadella-alexa-google-assistant-competitor. Accessed: 2019/06; Online.

[86] T. Warren. Microsoft's new cortana chief outlines the company's digital assistant vision. https://www.theverge.com/2019/5/8/18536088/microsoft-cortana-interview-vision-build-2019. Accessed: 2019/06; Online.

[87] N. Wingfield. Cortana, open alexa,' amazon says. and microsoft agrees. www.www.nytimes.com/2017/08/30/technology/amazon-alexa-microsoft-cortana.html?mcubz=3. Accessed: 2018/11; Online.

[88] L.-b. Work. " Alexa is my new BFF ": Social Roles , User Satisfaction , and Personification. pages 2853–2859, 2017.

[89] P. Zawadzki and K. Żywicki. Smart Product Design and Production Control for Effective Mass customization in the industry 4.0 concept. (June), 2016.

SUPPORT MATERIAL

a.1 VoicePrint requirements

In this section, the functional and nonfunctional requirements of VoicePrint, presented in chapter 4, will be presented.

a.1.1 Functional requirements

Systems

- The editor must support the development of language model templates for the platforms Amazon Alexa and Google Home;
- The editor must support the development of language model templates for the frameworks Jovo and Violet and the BotTalk platform;
- The editor must allow the user to generate platform-specific language models for the supported systems;
- The editor must allow the user to generate platform-specific boilerplate code for the initial development of the backend functionality for the supported systems;
- The editor must allow the deployment of the platform-specific language models to the digital assistants consoles;
- The editor must allow the deployment of the boilerplate code to the AWS Lambda;
- The editor must allow the user to deploy both the platform-specific language model and the boilerplate code at the same time.

Templates

- The editor must provide default templates in order to provide examples of the language model templates to the user;

- The editor must allow the user to define its own language model templates (custom templates);
- The user must have access to his templates and the default templates;
- The editor must support the definition of all the following template components:
  - Invocation name;
  - Intents;
  - Inputs;
  - Sample phrases;
  - Input types.
- The editor must allow the user to delete a template defined by him;
- The user defined templates must have the following descriptors, that can be altered at any time with the exception of the creation and last modified dates and the platforms for which the template has been generated:
  - Name;
  - Template creation date;
  - Template modification date;
  - Platforms for which the template has been generated;
  - General description of the template.
- The default templates must have the following descriptors:
  - Name;
  - Version;
  - Platforms for which the template has made;
  - General description.
- The editor must allow the user to define synonyms for the Input type values;
- The editor must allow the user to search for a template or by its name or by platform;
- The editor must allow the user to search for the template that contains a certain Intent or Input type;
- The editor must allow the user to define, remove or alter the definition of the following components on its templates:
  - Templates:
    * Invocation name;

    * Intents;
    * Inputs.
  - Intents;
    * Name;
    * Sample phrases;
    * Inputs used.
  - Inputs:
    * Name;
    * Type.
  - Input types:
    * Name;
    * Value;
    * Synonyms.

Platform

- The platform must provide help to the user through a "Help" menu. In this menu, it must be explained what the user can do with the platform and how. An explanation about the components of the templates and how can they be used with the digital assistants can too be provided;
- The platform must provide labels so the user can specify if he wants to search in the custom templates or in the default templates.

a.1.2 Nonfunctional requirements

Platform

- The platform must generate the platform-specific language models in JSON and in YAML;
- The platform must be developed in Java in order for it to be used in more than one operating system;
- The platform must provide a clear way to specify a language template model;
- The platform must use file persistency to persist the necessary information for its proper functioning (e.g the default templates);
- The platform must check the invocation name for wake words or launch phrases that the digital assistants use and therefore can't be present in it;

- The platform must update the Inputs that a template has when the user defines a new Input in the Intent definition;
- The platform must update the "Inputs using Input type" list when the user adds/removes an Input using said Input Type to/of an Intent;
- The platform must update the Inputs list on the Intent section every time the user adds a sample phrase with a new Input;
- The platform must check if the sample phrase that is going to be added doesn't already exists on the Intent sample phrases list;
- The platform should have a clean look, in terms of user interface, and not obfuscate any of the functionalities it provides.

a.2 Requirements of the trivia application

In this section, the functional and nonfunctional requirements of the voice application "Master Quiz", that will allow a user to perform general knowledge quizzes, will be presented.

a.2.1 Functional requirements

Quizzes

- The application must allow the user to ask for a quiz of a certain theme;
- The application must provide quizzes regarding the following themes:
  - General Knowledge;
  - Sports;
  - Music;
  - Film;
  - Geography;
  - Science and nature.
- The application must provide multiple choice quizzes and/or true of false ones;
- The application must allow the user to choose which type of quiz he wants to do;
- The application must allow the user to skip a question;
- The application must allow the user to cancel a quiz at any time;
- At the end of the quiz, the application must tell the user how many questions he answered correctly.

Score

- The user must gain points for every correct answer that he gives;
- Each right question will earn the user 5 points;
- The points that a user gains will allow him to level up through the following rank:
  - Apprentice;
  - JourneyMan;
  - Expert;
  - Master;
  - Grand Master.
- The user must be able to ask in which rank he is.

a.2.2 NonFunctional requirements

Quizzes

- The application must set a threshold of five questions per quiz;
- The application must provide both type of quizzes if the user is using Amazon Alexa;
- The application must only provide true or false quizzes if the user is using Google Assistant.

Score

- The application must store the points a user has and its rank in a database in order to be able to retrieve information about it if the user asks for it.

a.3 Requirements of the shopping voice application

The functional and nonfunctional requirements of the voice application “Bag it”, presented in chapter 5, are the following:

a.3.1 Functional requirements

Authentication

- The application must allow the users to login in their Etsy account.

Products

- The application must allow the user to search for product(s);
- The application must allow the user to search for products using the following tags:
  - Lowest price;
  - Highest price.
- The application must sort the search by increasing price tag;
- The application must tell the user the following information about a product:
  - Name;
  - Description;
  - Estimated processing date;
  - Postage price;
  - Product review data (numbers of views and favorites).

a.3.2 Seller

- The application must tell the user the following information about a seller:
  - Shop name;
  - Shop title;
  - Seller's name;
  - Number of favorers;
  - Feedback score and count;
  - Rating and number of ratings.

a.3.3 Shopping cart

- The application must allow the user to add products to its shopping cart;
- The application must ask the user how many items of the same product he wants to add to the shopping cart;
- The application must allow the user to remove products from its shopping cart;
- The application must allow the user to change the quantity of a product that is on the shopping cart;

- The application must allow the user to ask for the content of the shopping cart.
- The application must tell the user the following information about the cart:
  - Cart's full price;
  - Total number of products.
  - Name of each product;
  - Quantity of each product.

Purchase

- The application must allow the user to purchase the products that are on its shopping cart;
- The application should inform the user that he can only pay via paypal;
- The application must tell the user about the following purchase information:
  - Name of each product;
  - Quantity of each product;
  - Shipping information;
  - Total price.
- The application must tell the user if the purchase was successful or not;
- The application must be able to tell the user the status of a purchase he made.

a.3.4 NonFunctional requirements

Authentication

- The system must follow Oauth 2.0 in the implementation of the authentication mechanism;
- The system must be able to "refresh" the user authentication information to prevent the user from having to perform the authentication step again.

Purchase

- The application must only allow the user to pay through paypal;
- The application must always confirm the purchase with the user before he tries to pay it and before submitting the payment;
- The application must cap the purchases price below 100 dollars;
- The application must store the purchases identification in a database in order to be able to retrieve information about it in case that the user asks for it.

a.4 Requirements of the news voice application

The functional and nonfunctional requirements of the voice application “The Informed”, presented in chapter 5, are the following:

a.4.1 Functional requirements

News

- The applications must allow the user to ask for news;
- The application must try to provide the user with the most recent news;
- The application must allow the user to ask for news about a certain topic;
- The application must allow the user to ask for news of a certain day;
- The application must allow the user to ask for the N most recent news about a topic;
- The application must tell the user the following information about a news:
  - The title;
  - The description;
  - The publishing date.

Headlines

- The application must allow the user to ask for the top headlines of the current day;
- The applications must allow the user to ask about the top headlines of a certain topic;
- The application must allow the user to ask for the N most recent top headlines;
- The application must tell the user the following information about a headline:
  - The title;
  - The publishing date.

a.4.2 NonFunctional requirements

News

- The default number of news about a certain topic that will be presented to the user is 5;

- The news will be sorted by published date in order for the user to have access to the most recent ones;
- The news will be presented in set of three news at a time in order to not overload the user short term memory. After the three news are presented the application will ask the user if he wants to hear more;
- The value of N most recent news is capped at 10 and the application must inform the user of it;
- The range of days that the user can ask for news varies between the previous day and the previous week.

Headlines

- The top headlines source will be the BBC news website;
- The default number of top headlines that will be presented to the user is 5;
- The value of N most recent headlines is capped at 10 and the application must inform the user of it.

a.5 Personas for the news voice application

a.5.1 Main persona

The main persona inspiration is going to be a news anchor from BBC.

Key adjectives:

- Serious;
- Calm;
- Nice;
- Straightforward;
- Driven.

Characters who embody those adjectives:

- News anchor;
- News reporter;
- Teacher;
- Leader of a company.

a.5.2 Short description

The anchor from "The Informed" is always on top of the latest news and by being on the news industry for most of its life it knows how a news should be delivered to the audience. A firm believer that the people should always be informed in order to know what is going on around them, they are always in deep research with various other reporters to bring more and important news to the public. As a news anchor they've been honored by the news community for they serious work in that field and are loved by the public because of their calm demeanor even when reporting the most alarming news.

a.5.3 Users personas

Who are the people that will use "The Informed"?

- People who like to stay up to date with the news;
- People who don't have time to read the newspapers or watch the news;
- People who want to know what is happening in a certain area of interest.

What do people expect from "The Informed"?

- To be able to listen to the most recent news;
- To be able to ask for news about a certain area of interest;
- To be able to listen to the current top headlines;
- To be able to know what is happening in the world nowadays.

Users personas description

William Blake

A retired news reporter William never lost the news bug that fired his award-winning career and likes to always know what is up in the world especially in the local and international policy news area. He wants an application that quickly allows him to be able to hear the most recent news about what is happening in the policy world.

Margaret O'Neill

Margaret is an expert historian and a firm believer that if we aren't aware of what is going on in the world and how we can change it for the better we are condemned to repeat the errors of the past. So in order to always keep up with the latest news Margaret uses every technology she can to check the news. She would like an application that delivers her the latest news or global or about a certain area of interest and that allows her to do something else while listening to the news.

Brandon Meyer

Brandon is used to never be more than 10 minutes in on place do to the travelling that it's career as a photo reporter entails. By having to sometimes spend long periods of time without access to a internet connection or a newspaper, he likes to inform himself right away when he gets access to such commodities in order to know what happened and to what place should he fly next so he can photograph what is happening. He would like an application that gives him the latest global news or the news from a previous day.

a.6 Personas for the shopping voice application

a.6.1 Main persona

The main persona inspiration is going to be a shopping assistant from a retail shop.

Key adjectives:

- Patient;
- Nice;
- Helpful;
- Charming;
- Qualified.

Characters who embody those adjectives:

- Shopping assistant;
- Customer support assistant;
- Sales person;
- Employee of a shop;

Short description

The shopping assistant from "Bag it" comes from a family of retail owners that started teaching it how to search for the best product on the middle of a product pile since they could barely form a two-worded phrase. With a passion for shopping and an eye for the best deals they've worked with and in retails shops since ever in various job positions that go from personal shopping assistant to trainer of the shop staff. Nowadays, they are hired to teach the staff of the retails shop how to approach and help the customers with sympathy and efficiency and once in awhile to help design some online shopping websites interactions.

a.6.2 Users personas

Who are the people that will use “Bag it”?

- People who like to shop online;
- People who don’t have time to go shopping;
- People who want the help and commodity of a virtual shopping assistant.

What do people expect from “Bag it”?

- To be able to search for a certain product;
- To be able to get the best deal on a certain product accordingly to the chosen search filter;
- To be able to create and alter their shopping cart;
- To be able to purchase the content of their shopping cart.

Users personas description

Marie Tudor

Being the CEO of an IT company sometimes leaves Marie with little to no time to do some shopping. In order to fix this problem Marie likes to use online shopping websites that allow her to buy everything she needs with the click of a button. She would like an application that allows her to shop while taking care of other matters and that gets her the most quality item with the fastest delivery.

Hank Allen

The idea of a shopping list already makes Hank shiver. An averse to shops and consumerism Hank knows what he wants to buy and wants do it quickly and at the lowest price if possible. He would like an application that avoids the need to go shopping and that gives him the best prices in the products that he wants to buy.

Lorelai Ross

Owner of a countryside second hand store Lorelai is always on the look for unique products to sell in her shop. Every Friday Lorelai spends her lunch hour browsing the internet for unique and not very expensive products to sell in her shop and also keeps an eye on some products that her most loyal customers might like to buy. She would like an application that helps her find some unique items at a decent price and that can be delivered quickly so she can start advertising them right away.

a.7 Apache velocity templates

a.7.1 Alexa Velocity template

```
(...)

#set($handler = "Handler")
#foreach($intent in $intents)
#set($name = $intent.name)
#set($fullName = "$name$handler")
const $fullName = {
    canHandle(handlerInput) {
#if(!$intent.getInputState().equals(""))
    const attributes = handlerInput.attributesManager.getSessionAttributes();
#end
        return handlerInput.requestEnvelope.request.type === 'IntentRequest'
#if(!$intent.getInputState().equals(""))
            && attributes.state === STATES.$intent.getInputState()
#end
            && handlerInput.requestEnvelope.request.intent.name === '$name';
    },
    handle(handlerInput){
#if(!$intent.getOutputState().equals(""))
        const attributes = handlerInput.attributesManager.getSessionAttributes();

        attributes.state = STATES.$intent.getOutputState();
#end}
};
#end

(...)

exports.handler = Alexa.SkillBuilders.standard()
    .addRequestHandlers(
    #foreach($intent in $intents)
        //Write Intent name
    #end
).addErrorHandlers(ErrorHandler)
#if(!$databases.empty)
    //Setup DB
#end
    .lambda();
```

Listing A.1: Alexa boilerplate code template example

a.7.2 Google Assistant Velocity template

```
(...)
#if($states.size() > 0)
const STATES = {
    #foreach($state in $states)
        $state : "$state",
    #end
};
#end

(...)

#foreach($intent in $intents)
#if(!$intent.get_has_Inputs().empty)
app.intent('$intent.name', (conv,params) => {
```

```
//Setup of what this intent must do
});
#else
app.intent('$intent.name', (conv) => {
//Setup of what this intent must do
});
#end
#end

// The entry point to handle https POST requests
exports.$tempName = functions.https.onRequest(app);
```

Listing A.2: Google Assistant boilerplate code template example

a.7.3 Jovo Velocity templates

```
module.exports = {
    logging: true,
// Mapping of the built-in intents
    intentMap: {
#foreach($intent in $intents)
#if($builtInIntents.contains($intent.name))
    #set($name = $intent.name)
    #set($fullName = "$name")
    #set($AmazonName = "AMAZON.$name")
    #if($AmazonName.toLowerCase().contains("stopintent"))
    '$AmazonName' : 'END',
    #else
    '$AmazonName' : '$fullName',
    #end
#end
#end
},
(...)};
```

Listing A.3: Jovo configuration template example

```
(...)
const app = new App();
app.use(new Alexa(),
    new GoogleAssistant(),
#if(!$databases.empty)
    new JovoDebugger(),
// Enable DB after app initialization
#foreach($database in $databases)
// Enable the chosen DBs
#end
#end
#else
    new JovoDebugger()
#end
);

app.setHandler({
(...)
#foreach($key in $states.keySet())
#if($key.toLowerCase().contains("intent_nostate"))

#set($IntentsName = $states.get($key))
#foreach($auxName in $IntentsName)
#if($auxName.toLowerCase().contains("stop"))
    #set($funcName = "END()")
```

```
#else
    #set($funcName = "$auxName()")
#end
    $funcName{ },
#end
#else
#set($name = $key)
#if(!$name.toLowerCase().contains("state"))
    #set($aux = "State")
    #set($stateName = "$name$aux")
#else
    #set($stateName = $name)
#end
    $stateName: {
    #foreach($itsName in $states.get($name))
        #set($funcName = "$itsName()")
    $funcName{ },
    #end
    },
#end
#end
(...)
});

module.exports.app = app;
```

Listing A.4: Jovo boilerplate code template example

a.7.4 BotTalk Velocity template

```
intents:
#foreach($intent in $intents)
    $intent.getName():
#foreach($samplePhrase in $intent.get_has_Samplephrases())
        - '$samplePhrase.getPhrase().replaceAll("'","")'
#end
#end
```

Listing A.5: BotTalk boilerplate code template example

```
tests: #Name of the first test (will appear in the Test Tab)
```

Listing A.6: BotTalk test file template

a.8 Built-In Intents

In this section, the gathered built-in Intents of Alexa and Google Assistant are going to be presented.

Table 11.: Built-in Intents of Amazon Alexa

Amazon Alexa	Mandatory
AMAZON.CancelIntent	Yes
AMAZON.FallbackIntent	Yes
AMAZON.HelpIntent	Yes
AMAZON.LoopOffIntent	No
AMAZON.LoopOnIntent	No
AMAZON.NextIntent	No
AMAZON.NoIntent	No
AMAZON.PauseIntent	No
AMAZON.PreviousIntent	No
AMAZON.RepeatIntent	No
AMAZON.ResumeIntent	No
AMAZON.SelectIntent	No
AMAZON.ShuffleOffIntent	No
AMAZON.StartOverIntent	No
AMAZON.StopIntent	Yes
AMAZON.YesIntent	No
AMAZON.NavigateHomeIntent	Yes

Table 12.: Built-in Intents of Google Assistant

Google Assistant	Mandatory
Welcome	Yes
Fallback	Yes
actions.intent.PERMISSION	No
actions.intent.SIGN_IN	No
actions.intent.DELIVERY_ADDRESS	No
actions.intent.CONFIRMATION	No
actions.intent.DATETIME	No
actions.intent.PLACE	No
actions.intent.OPTION	No
actions.intent.LINK	No
actions.intent.GET_FORTUNE	No
actions.intent.GET_HOROSCOPE	No
actions.intent.GET_JOKE	No
actions.intent.GET_QUOTATION	No
actions.intent.GET_CREDIT_SCORE	No
actions.intent.GET_CRYPTOCURRENCY_PRICE	No
actions.intent.PLAY_GAME	No
actions.intent.CHECK_WATERSPORTS_CONDITIONS	No
actions.intent.CHECK_AIR_QUALITY	No
actions.intent.CHECK_WATER_CONDITIONS	No
actions.intent.START_CALMING_ACTIVITY	No

a.9 Built-In Input Types

In this section, the gathered built-in Input Types of Alexa and Google Assistant are going to be presented.

Table 13.: Built-in Input Types of BotTalk

BotTalk	Supported in assistants
@bottalk.color	Yes
@bottalk.country	Yes
@bottalk.date	Yes
@bottalk.duration	Yes
@bottalk.language	Yes
@bottalk.phone-number	Yes
@bottalk.time	Yes
@bottalk.number	Yes
@bottalk.name	Yes

Table 14.: Built-in Input Types of Amazon Alexa

Amazon Alexa	Supported in both assistants
AMAZON.DATE	Yes
AMAZON.DURATION	Yes
AMAZON.Author	No
AMAZON.NUMBER	Yes
AMAZON.Ordinal	Yes
AMAZON.TIME	Yes
AMAZON.SearchQuery	Yes
AMAZON.Actor	No
AMAZON.AdministrativeArea	No
AMAZON.AggregateRating	No
AMAZON.Airline	No
AMAZON.Book	No
AMAZON.CivicStructure	No
AMAZON.Color	Yes
AMAZON.EventType	No
AMAZON.Festival	No
AMAZON.Game	No
AMAZON.Language	Yes
AMAZON.Month	Yes

Table 15.: Built-in Input Types of Google Assistant

Google Assistant	Supported in both assistants
@sys.date	Yes
@sys.time	Yes
@sys.time-period	No
@sys.number	Yes
@sys.ordinal	Yes
@sys.number-integer	No
@sys.flight-number	No
@sys.unit-area	No
@sys.duration	Yes
@sys.age	No
@sys.currency-name	No
@sys.address	Yes
@sys.location	Yes
@sys.last-name	No
@sys.color	Yes
@sys.language	Yes
@sys.any	Yes
@sys.url	No

a.10 Execution times

In this section, the execution times of the construction process components will be presented.

Table 16.: Execution times

Components	Average Time	Comments
Generate Alexa language model	111,8 ms	
Generate Assistant language model	754,8 ms	Includes the time it took to create the model zip
Generate BotTalk language model	191,2 ms	
Generate Jovo language model	105,4 ms	
Generate Alexa boilerplate code	199,6 ms	
Generate Assistant boilerplate code	133,2 ms	
Generate BotTalk boilerplate code	145,4 ms	Includes the generation of both the boilerplate code and tests
Generate Jovo boilerplate code	235,2 ms	Includes the time it took to generate the configuration file
Saving language model template to persistency	347 ms	
Deploying to Amazon Alexa	26023 ms	Includes the time it took to deploy the boilerplate code to AWS Lambda
Deploying to Google Assistant	14029 ms	Includes the time it took to deploy the boilerplate code to AWS Lambda
Deploying to Jovo	116717 ms	Includes the time it took to deploy the boilerplate code to AWS Lambda

a.11 Case studies - Examples

In this section, examples of how the voice applications, The Informed and Bag it, were tested recurring to the digital assistants consoles, will be presented.

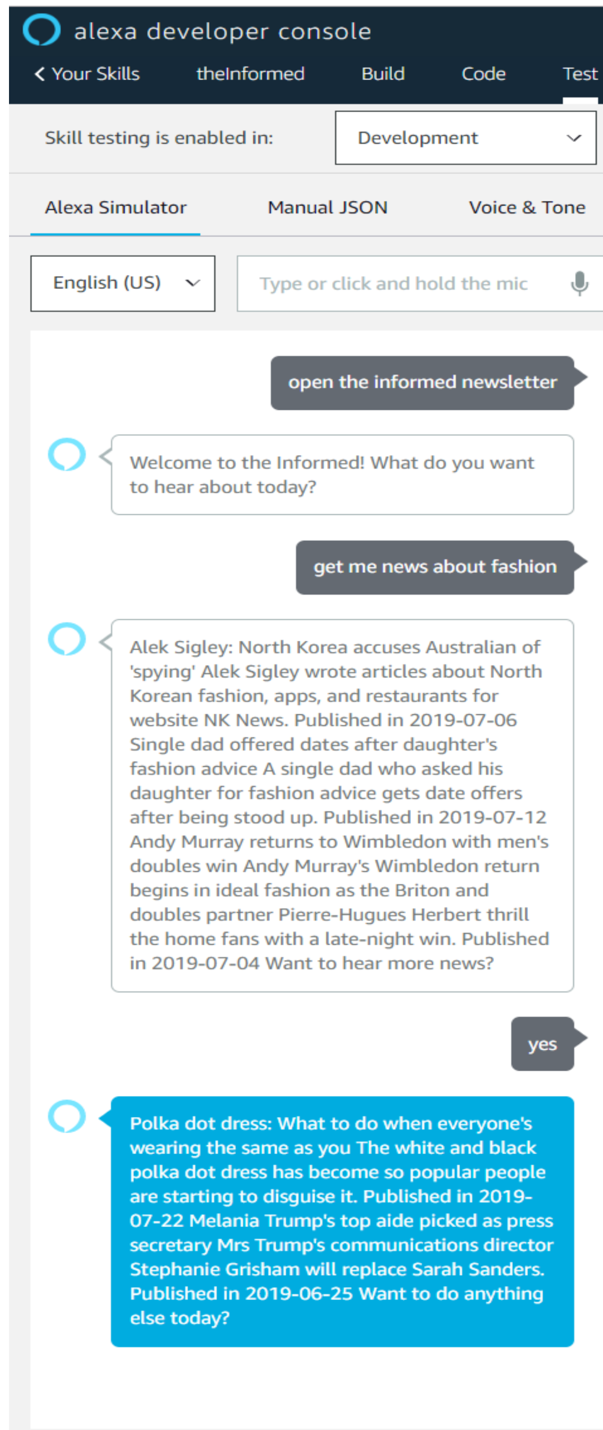


Fig. 39.: The Informed - Alexa developer console test

Actions Console Overview Develop Test Deploy

Simulator

Talk to the Informed newsletter

Alright. Let's get the test version of the Informed newsletter.Welcome to the Informed! What do you want to hear about today?

get me some headlines

'Butcher of Beijing' Li Peng dies at 90 published in 2019-07-23 Police fired for suggesting Ocasio-Cortez be shot published in 2019-07-23 Van filled with drugs crashes into police car published in 2019-07-23 Want to hear more headlines?

no

Want to hear anything else today?

goodbye

Thank you for using the informed. Until a next time

'the Informed newsletter' left the conversation

Suggested input

Talk to the Informed newsletter

Input

Try typing or saying "Talk to the Informed newsletter".

Fig. 40.: The Informed - Actions console test

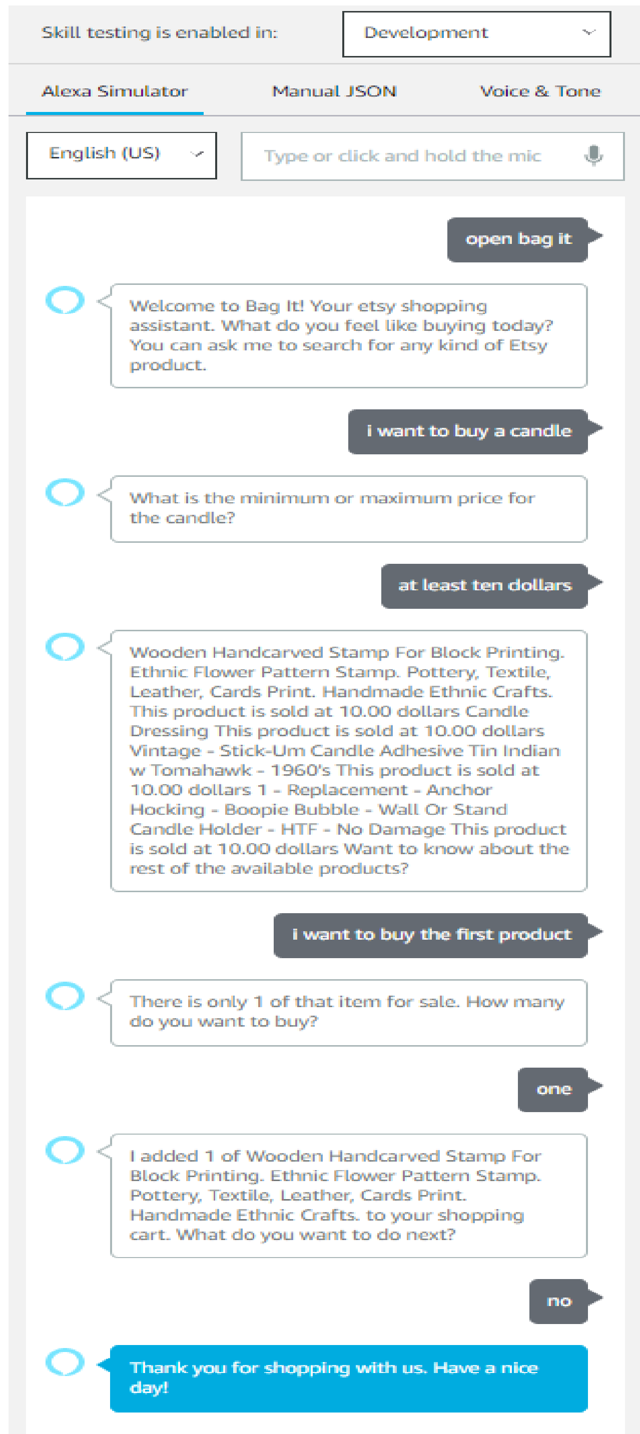


Fig. 41.: Bag it - Alexa developer console test

Actions Console | Overview | Develop | Test | Deploy

Simulator

Talk to bag it

Sure. Getting the test version of bag it.Welcome to Bag It! Your etsy shopping assistant. What do you feel like buying today? You can ask me to search for any kind of Etsy product.

i want to check my cart

You currently have 3 products in your cart for a total of 22.65 dollars with shipping included. You have the following products in your cart. You have 1 Wooden Handcarved Stamp For Block Printing. Ethnic Flower Pattern Stamp. Pottery, Textile, Leather, Cards Print. Handmade Ethnic Crafts. that amounts to 10 dollars. and You have 1 When the world seems darkest LAMINATED BOOKMARK that amounts to 3.11 dollars. and You have 1 Oscar Wilde bookmark that amounts to 1.15 dollars. Want to shop for anything else today or simulate the purchase of your cart?

i want to purchase my cart

For 3 products, your total is 22.65 dollars. To pay go to the etsy web page. Keep in mind that each shop has its own payment methods. If you want to shop for anything else today don't hesitate in asking me!

goodbye

Thank you for shopping with us. Have a nice day!

'bag it' left the conversation

Suggested input

Talk to bag it

Input

Try typing or saying "Talk to bag it".

Fig. 42.: Bag it - Actions console test