



Development of a Framework for Chatbots

by

João Carlos de Almeida Valente

Supervisor: Pedro Miguel Mestre Alves da Silva, PhD

Co-supervisor: Rui Zhang, PhD

A Thesis submitted to the
UNIVERSITY OF TRÁS-OS-MONTES AND ALTO DOURO
for the degree of Master of Science-Philosophiae Doctor
in Electrical and Computer Engineering, according to the
Regulamento Geral dos Ciclos de Estudos Conducentes ao Grau de Mestre na UTAD
DR, 2^o série-N.º133- Regulamento no 658/2016 de 13 de julho de 2016

Development of a Framework for Chatbots

by

João Carlos de Almeida Valente

Supervisor: Pedro Miguel Mestre Alves da Silva, PhD

Co-supervisor: Rui Zhang, PhD

A Thesis submitted to the
UNIVERSITY OF TRÁS-OS-MONTES AND ALTO DOURO
for the degree of Master of Science-Philosophiae Doctor
in Electrical and Computer Engineering, according to the
Regulamento Geral dos Ciclos de Estudos Conducentes ao Grau de Mestre na UTAD
DR, 2^o série-N.º133- Regulamento no 658/2016 de 13 de julho de 2016

Scientific Supervision:

Pedro Miguel Mestre Alves da Silva, PhD

Assistant Professor of the
Engineering Department
School of Science and Technology
of the University of Trás-os-Montes and Alto Douro, Portugal

Rui Zhang, PhD

Assistant Professor of the
College of Computer Science & Technology
of the Jilin University, China

”Have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary.”

Steve Jobs (1955 – 2011)

- *À minha mãe, ao meu pai e irmão.*

Desenvolvimento de uma Framework para *Chatbots*

João Carlos de Almeida Valente

Submetido na Universidade de Trás-os-Montes e Alto Douro
para o preenchimento dos requisitos parciais para obtenção do grau de
Mestre em Engenharia Electrotécnica e de Computadores

Resumo — O crescente interesse pela utilização de *chatbots* tem levado muitas empresas a investirem neste tipo de soluções para aprimorar o seu serviço de apoio ao clientes. Não só nesse campo como muitos outros, os *chatbots* têm vindo a ganhar fama e o facto de estarem instalados nos dispositivos que usamos no dia a dia, como os nossos smartphones, contribuiu favoravelmente para o seu crescimento. O problema que se enfrenta agora, não passa pela interpretação da informação, mas sim pelo aumento da rentabilização da aplicação. Isto deve-se ao facto de existir um número cada vez maior de serviços de comunicação onde os utilizadores escolhem passar o seu tempo. Além disso, tendo tecnologias como o Node.js, ganhado uma enorme fama no departamento da web, são reduzidas as ferramentas que existem para o desenvolvimento de *chatbots* em Java. Não obstante, esta linguagem de programação continua a ser uma das mais utilizadas a nível mundial, sobretudo no desenvolvimento de servidores web, necessários para o tipo de aplicações como é o caso dos *chatbots*. Nesse sentido, sente-se a falta de uma solução para o desenvolvimento de *chatbots*, em ambiente Java, para um vasto número de serviços de mensagens existentes. Esta dissertação procura solucionar este problema, propondo uma framework em Java para o desenvolvimento universal de *chatbots*. Assim, com um único código passará a ser possível implementar um *chatbot* em diversas plataformas de mensagem, aumentando o número de pessoas abrangidas pelo mesmo e reduzindo substancialmente o tempo de produção e desenvolvimento da aplicação. Ainda, permitindo que desenvolvedores consigam implementar novas plataformas que poderão emergir no futuro, proporcionando assim uma solução duradoura e flexível.

Palavras Chave: *Chatbot*, Framework, Java.

Development of a Framework for Chatbots

João Carlos de Almeida Valente

Submitted to the University of Trás-os-Montes and Alto Douro
in partial fulfillment of the requirements for the degree of
Master of Science-Philosophiae Doctor in Electrical and Computer Engineering

Abstract — The growing interest in the use of chatbots has led many companies to invest in these solutions to improve their customer relations services. In this field and many other fields, chatbots have been gaining fame, and the fact that they are installed in the devices we use daily, such as our smartphones, contributed favorably to their growth. The problem now faced is not the interpretation of the information but the increase the profitability of the application. It is because there is an increasing number of communication services where users prefer to spend their time. Also, having technologies such as Node.js gained a considerable reputation in the web department, the tools that exist for the development of chatbots in Java are reduced. Nevertheless, this programming language remains one of the most used worldwide, especially in web server development, necessary for the type of applications such as chatbots. In this sense, there is a lack of a solution for developing chatbots in the Java environment for many messaging services. This dissertation seeks to solve this problem, proposing a Java framework for the universal development of chatbots. Thus, with a single code, it will be possible to implement a chatbot on multiple platforms, increasing the number of people covered by it and substantially reducing the time of production and development of the application. Also, allowing developers to implement new platforms that may emerge in the future, thus providing a durable and flexible solution.

Key Words: Chatbot, Framework, Java.

Agradecimentos

Ao longo do meu percurso académico, que inclui o desenvolvimento desta dissertação de mestrado, muitos contribuíram para a minha formação académica e pessoal. Tive o enorme privilégio em poder contar com o apoio de várias pessoas a quem passo a agradecer individualmente.

Aos meus pais, Paula Almeida e Carlos Valente, e ao meu irmão, Nuno Valente, por todo o apoio, por nunca me deixarem vacilar e por sempre me fazerem acreditar em mim e nunca desistir do que quero para mim. Um agradecimento especial aos meus pais por todos os sacrifícios que fizeram para me poderem proporcionar os estudos que eu sempre quis e tudo o que a vida universitária sempre implicou. Ainda, por todos os ensinamentos e educação que me transmitiram. Não consigo expressar o quanto vos agradeço. A vós dedico este trabalho.

Aos meus orientadores, Professor Pedro Mestre e Professor Rui Zhang por toda a ajuda prestada durante o desenvolvimento deste trabalho. Em particular ao Professor Pedro Mestre, por todos os bons momentos, paciência, orientação e, acima de tudo, toda a ajuda, tanto a nível pessoal como académico, prestada ao longo do meu percurso académico.

À pessoa mais importante que a universidade trouxe para a minha vida, Jorge Sousa, por, à sua maneira, sempre me fazer seguir os meus sonhos e ajudar a cumprir os

meus objetivos. Sem ti, muitos deles não teriam sido possíveis.

À mais antiga de todas as amigas, Joana Cabral, porque sem ti nada disto faria sentido. Por me encorajares, por me dizeres quando estou errado e quando não estou a dar tudo de mim. Por me ensinares tanto, tanto, mas, acima de tudo, por fazeres de mim uma pessoa melhor.

À irmã que a UTAD me deu, Sara Crespo, agradeço por toda a camaradagem, noites de estudo ininterrupto e trabalho conjunto. Por todas as gargalhadas e conselhos que partilhámos juntos. E, claro, por todas as noites de diversão que tornaram os dias de estudo mais apetecíveis.

Aos meus amigos e colegas Bruno Novo, Davide Machado e Tiago Barbosa. Esta jornada começou com vocês e com vocês termina. Obrigado por toda amizade, lealdade e companheirismo que sempre tiveram comigo e que marcou a minha vida universitária.

À Sofia Teixeira Guedes e Inês Silva, por estarem sempre disponíveis para me proporcionarem com a vossa ajuda e amizade quando mais precisei.

Por fim, um enorme agradecimento a toda a universidade de Trás-os-Montes e Alto Douro, incluindo professores e funcionários que fizeram desta universidade uma segunda casa para mim.

Muito obrigado a todos!

UTAD,

João Valente

Vila Real, 20 de julho de 2021

Acknowledgments

Throughout my academic journey, which includes developing this master's thesis, many contributed to my good academic and personal training. I had the enormous privilege of counting on the support of several people I would like to thank individually.

To my parents, Paula Almeida and Carlos Valente, and to my brother, Nuno Valente, for all the support, never letting me falter, and always making me believe in myself, and never giving up what I want for myself. Special thanks to my parents for all the sacrifices they made to provide me with the studies that I wanted for myself and everything that university life has always entailed. Still, for all the teachings and education that they transmitted to me. I cannot express how much I thank you. I dedicate this work to you.

To my advisors, Professor Pedro Mestre, and Professor Rui Zhang, for all the help provided during this work's development. In particular to Professor Pedro Mestre, for all the good times, patience, guidance, and, above all, all the help, both personally and academically, provided throughout my university life.

To the most important person that the university brought into my life, Jorge Sousa, for, in his own way, always making me follow my dreams and helping to fulfill my goals. Without you, many of them would not have been possible.

To the oldest of all friends, Joana Cabral, because without you, none of this would make sense. For encouraging me, for telling me when I'm wrong and when I'm not giving it my all. For teaching me so much, so much, but, above all, for making me a better person.

To the sister UTAD gave me, Sara Crespo, I thank you for all the camaraderie, nights of uninterrupted study, and working together. For all the laughs and advice we shared together. And, of course, for all the fun nights that made the study days more desirable.

To my friends and colleagues Bruno Novo, Davide Machado and Tiago Barbosa. This journey started with you and ends with you. Thank you for all the friendship, loyalty, and companionship you have always had with me, which marked my academic path.

To Sofia Teixeira Guedes and Inês Silva, for always being available to provide me with your help and friendship when I needed it most.

Finally, a huge thank you to the entire university in Trás-os-Montes and Alto Douro, including professors and staff who made this university a second home for me.

Thank you all!

UTAD,
Vila Real, 20 July 2021

João Valente

Contents

<i>Resumo</i>	ix
Abstract	xi
<i>Agradecimentos</i>	xiii
Acknowledgments	xv
List of Tables	xxi
List of Figures	xxi
Abbreviations	xxv
1 Introduction	1
1.1 Background	1
1.2 Motivation and Objectives	2
1.3 Document Structure	4
2 Chatbots	5
2.1 Chatbot Definition	5
2.2 History of Chatbots	6
2.3 Chatbots Usage	7
2.4 Advantages and Limitations	9
2.4.1 Advantages	9

2.4.2	Limitations	9
2.5	Real-world Existing Applications	10
2.5.1	Cleverbot	10
2.5.2	Instalocate	12
2.5.3	HealthTap	13
2.5.4	Domino's Pizza	14
2.5.5	Personal Assistants	15
2.6	Related Work	16
2.6.1	Chatbot Frameworks	16
2.6.2	Conclusion	20
3	Messaging platforms	23
3.1	Usage	23
3.2	Facebook Messenger	25
3.2.1	Structure	26
3.2.2	Parameters	28
3.2.3	Handling HTTP Requests	32
3.2.4	Webhook	33
3.3	Telegram	34
3.3.1	Structure	34
3.3.2	Parameters	35
3.3.3	Handling HTTP Requests	38
3.3.4	Webhook	39
3.4	WeChat	40
3.4.1	Structure	40
3.4.2	Parameters	41
3.4.3	Handling HTTP Requests	42
3.4.4	Webhook	43
3.5	Other Platforms	45
4	Software Concepts	47
4.1	Development Methodology	47
4.1.1	The Waterfall Model	47
4.2	Design Patterns	49
4.2.1	Factory Design Pattern	50
4.2.2	Observer Design Pattern	51
4.3	HTTP Protocol	53
4.3.1	Methods	53
4.3.2	URI	55
4.3.3	HTTP versions	55

4.3.4	HTTP Message Structure	56
4.3.5	HTTPS Protocol	58
4.4	Data formats	59
4.4.1	XML vs JSON	59
4.5	JSON Web Token (JWT)	63
5	Framework Design and Implementation	67
5.1	Used Technologies	68
5.1.1	Development Platform	68
5.1.2	Server	70
5.2	Requirements	72
5.3	Architecture	73
5.3.1	Framework Components	76
5.3.2	Message and User Objects	77
5.4	Developed System	78
5.4.1	Framework	78
5.4.2	Application	84
6	Tests & results	95
6.1	Putting Everything Online	96
6.1.1	Ngrok	96
6.2	Test Scenarios	98
6.2.1	Scenario 1: Only one chatbot working with a single service	98
6.2.2	Scenario 2: Two chatbots working with a single service	103
6.2.3	Scenario 3: Two services working with one chatbot each	104
6.2.4	Scenario 4: One single chatbot working with two distinct services	106
6.2.5	Scenario 5: Adding support for a new messaging service	108
7	Conclusion	113
7.1	Conclusions	113
7.2	Future Work	114
	References	115

List of Figures

2.1	Screenshot of a conversation with Cleverbot	11
2.2	Screenshot of the Instalocate working	12
2.3	Screenshot of a conversation with HealthTap chatbot	13
2.4	Screenshot of a conversation with Domino's Pizza chatbot	14
2.5	Google Assistant (left), Bixby by Samsung (center) and Siri by Apple (right)	15
3.1	Time spent on social media APPs in July 2020	24
3.2	Messaging platforms usage statistics	25
3.3	Facebook Messenger basic structure	26
3.4	Facebook Messenger setting webhook interface.	33
3.5	WeChat original setting webhook interface.	43
3.6	WeChat translated setting webhook interface.	44
4.1	Waterfall Model of software development	48
4.2	A Factory Design Pattern UML diagram.	50
4.3	The complete Factory Design Pattern implementation UML diagram	51
4.4	Observer Design Pattern UML diagram	52
4.5	OSI Model	54

4.6	HTTP message structure.	56
4.7	JSON Web Token online debugger screenshot	65
5.1	Java Servlet API diagram	71
5.2	Working process of a webhook	72
5.3	Factory Design Pattern UML diagram implementation.	74
5.4	General concept of the framework developed	75
5.5	Factory Design Pattern UML diagram implementation.	81
5.6	Observer Design Pattern UML implementation.	83
6.1	ngrok running on macOS terminal	97
6.2	Example diagram of one single chatbot working with only one service.	98
6.3	Text shown on the application debug terminal.	100
6.4	Testing Facebook Messenger chatbot.	101
6.5	Text shown on the application debug terminal after setting the web- hook on telegram.	102
6.6	Testing Telegram chatbot.	103
6.7	Example diagram of two chatbots working with one service only.	104
6.8	Example diagram of two chatbots working with one service each.	105
6.9	Testing Facebook Messenger and Telegram chatbots at the same time.	105
6.10	Example diagram of one single chatbot working with one distinct services.	106
6.11	Testing Facebook Messenger and Telegram with only one chatbot for both services.	107
6.12	Example diagram of one single chatbot working with three distinct services, being one of them not implemented in the framework.	108
6.13	Text shown on the application debug terminal when setting WeChat's webhook.	110
6.14	Results of the tests performed with three messaging platforms being one of them not included in the Framework.	110

Acronyms

AIML Artificial Intelligence Markup Language. 6, 19

ALICE Artificial Linguistic Internet Computer Entity. 6

API Application Programming Interface. xxii, 26, 28, 31, 32, 39, 69–71

CRUD Create, Read, Update and Delete. 54

FAQ Frequently Asked Questions. 5

HMAC Hash-based Message Authentication Code. 64

HTML HyperText Markup Language. 59, 64

HTTP Hypertext Transfer Protocol. xviii, xix, 26, 32–34, 37, 40, 42, 44, 53, 56, 58, 59, 64, 71, 72, 74, 79, 89, 97

HTTPS Hypertext Transfer Protocol Secure. xix, 53, 58, 59, 80, 97

Java EE Java Enterprise Edition. 68–70

Java SE Java Standard Edition. 69

JSON JavaScript Object Notation. 28, 29, 31, 32, 37–39, 41, 42, 59, 61–64, 70

JWT JSON Web Token. xxii, 32–34, 63–65

OSI Open System Interconnection. 53, 58

REST Representational State Transfer. 70

RSA Rivest-Shamir-Adleman. 64

SDK Software Development Kit. 17

SGML Standard Generalized Markup Language. 60

SHA Secure Hash Algorithm. 64

SSL Secure Sockets Layer. 59

TCP Transmission Control Protocol. 56

TLS Transport Layer Security. 56, 59

TT Turing Test. 6, 11

UML Unified Modeling Language. xxii, 50–52, 74, 81, 83

URI Uniform Resource Identifier. 32, 39, 53–55

URL Uniform Resource Locator. 26, 33, 34, 38, 39, 43, 44, 72, 82, 84, 85, 91, 98, 100

XML eXtended Markup Language. 42, 59–61, 63, 90

Abbreviations

Abbreviation	Meaning(s)
etc.	etecetera, others
et. al.	and others (authors)
i.e.	that is



Introduction

This dissertation aims the development solution for chatbot development that allows the developers to build one chatbot application and deploy it on multiple messaging services. Building chatbots for multiple platforms can be both exhausting and unrealistic because spending hours and hours coding for the different services is not a good investment for any company. For that matter, this framework intends to allow developers to create chatbots for multiple messaging platforms, reduce the amount of code needed and time spent in development, and also make the chatbot more easy to grow in the future and expand to new messaging platforms.

1.1 Background

Communicating is intrinsic to the human being. We do it every day. Nowadays, we can even keep communicating at a considerable distance thanks to the web services available and all the devices we carry with us daily. The evolution of mobile phones, currently better known as smartphones, came to meet this same need for communication. As time goes by and as technology evolves, much of what we do in our daily lives go through a call or telephone conversation. Events include making an

appointment with the dentist, filing a complaint to the telecommunications operator, ordering a hamburger for dinner, or even asking random questions to our personal voice assistant, [Kongaut and Bohlin \(2016\)](#). Chatbots emerged as a technology to mimic the human behaviour and to fool human to think that they are really talking with another human. Later, they evolved as a solution to help customers of a business in order to reduce costs. More than just being pleasant for users, chatbots can help them be more agile and more effective. Also, for those who implemented them, chatbots are a less expensive solution than any other that requires a human to perform it, for example, customer service, [Lindvall and Ljungström \(2018\)](#).

Although chatbots can be implemented on a variety of different platforms, it is common to see them on messaging platforms and services like Facebook Messenger and Telegram. However, despite the desire to make the chatbot more accessible, any administrator is forced to choose between some of those platforms, mainly because having a chatbot running on a vast number of messaging services requires much time on the part of the development team, giving that they must have to code one chatbot for each messaging platform. Furthermore, for companies, reducing the time needed for achieving a goal, means saving more money, thereby increasing profits. To fill this gap in the chatbots market, and in order to make it possible to expand this type of solutions to the largest number of platforms, this dissertation proposes a framework for the development of chatbots that allows programming one chatbot and making it available to more than just one instant messaging service.

1.2 Motivation and Objectives

Messaging platforms are becoming more popular each year. People have been choosing to type on their smartphones over the traditional phone calls, whether it is to talk to a friend or family member or to solve an issue, let us say, with the water company receipt, [Spectrm \(2020\)](#). Between 2007 and 2015, a third to half of the online interactions are due to chatbots usage. We realize that most companies, especially the big ones that most times need some kind of customer service, for example, a call

center, adopt chatbots to perform some screening and reduce flow to their customer help center, whether it is physical or over a call center, [Radziwill and Benton \(2017\)](#).

The mentioned above reasons are only the tip of the iceberg that could lead a company to implement a chatbot service.

If we consider the development teams' perspective, if a company decides to have their chatbot service available in a wide number of platforms as possible, that could mean a lot of effort, money, and time to create a chatbot specifically for each platform, [Chung et al. \(2020\)](#).

For that reason, the main focus of this dissertation is to develop a universal framework for chatbots that would give the developers the possibility to have one chatbot available in multiple platforms, in the same server application, thereby reducing the amount of code and resources needed.

The framework has to simultaneously support more than one messaging application in order to fix the problem of coding for multiple platforms, allowing to reduce the amount of code and time needed to create a working chatbot. The framework has to fulfill the aspects listed below:

- If possible, it should be completely standalone, i.e., should not require the developer to acquire, download or install any other software, dependency, or library to use the framework;
- Perform connections with multiple messaging services at the same time;
- Allow the developer that implements it to be able to customize the chatbot actions;
- When the framework does not have the implementation for a specific messaging service, the subject using it should be able to add support for the required service.

As a case of study, there reference implementation will be made using Java programming language.

1.3 Document Structure

This document is divided into seven chapters as described below.

The present chapter introduces the main subject of this dissertation and summarizes its main objectives.

Chapter 2 refers to state of the art. It explains a chatbot's definition, followed by a brief history of how the concept was born and how it has evolved until now. How people use chatbots and its functionalities and limitations are also topics within this section. It additionally makes a quick approach to some real applications for chatbots and how they can help us. Finally, some related works on this dissertation's topic are presented.

In chapter 3 it is presented the usage of messaging platforms and their growth along time. Additionally, it shows an analysis of Facebook Messenger, Telegram, and WeChat platforms structure which is a necessary step before the framework implementation.

In chapter 4 it can be found some of the core technological concepts relevant for this framework's development, without which its implementation would not be possible.

Chapter 5 explains the entire framework design and implementation, together with all the details necessary to understand the development process and its structure.

In chapter 6 are presented all the tests made to the framework as well as the obtained results.

Finally, chapter 7 makes a final analysis of the results and overall conclusions about future work that can improve the framework to another level.

Chatbots

This chapter makes a full description of a chatbot and provides useful information to understand this document, such as chatbot history, usage, and critical functionalities and limitations. Also, it presents some real-world applications of chatbot services and shows a list of all the current most well known frameworks for chatbot development.

2.1 Chatbot Definition

The Oxford Dictionary describes a chatbot as “*a computer program that can hold a conversation with a person, usually over the internet*”, [Dictionary \(2020\)](#). Chatbots can have various purposes, such as information seeking, site guidance, and FAQ (Frequently Asked Questions). Not only that, but they can help the user in many different subjects like customer service, education, website support, and entertainment, [Chen \(2019\)](#). A chatbot can help or assist users in performing tasks or giving them some information, according to the user requests and the purpose of the chatbot, [Arnaud Gellens \(2019\)](#).

It is important not to confuse a chatbot with a bot. A bot is a “*computer program*

that performs tasks repeatedly”, which is not the purpose of a chatbot.

2.2 History of Chatbots

The concept of a chatbot is as old as the computer itself, that was introduced in 1950 by Alan Turing. At that time, he also introduced a test that became very famous, known as the “Turing Test” (TT), [Sannikova \(2018\)](#). This test’s starting point was seeking an alternative to the question, “*Can machines think?*”. Since that time, when Turing released its article about the subject, the TT has become the ultimate goal that a chatbot should achieve to be considered a good chatbot, [Redstone \(2019\)](#).

For this project’s matter, the Turing Test is not a point of interest, giving that it evaluates the level of “intelligence” of a chatbot, and the goal of this dissertation is not to create a functional chatbot. Nevertheless, it is crucial to understand how chatbots have grown in time, how important they became to people in the present, and what their future may be.

Since the first chatbot creation, we have been watching a constant evolution over the years, leading to the chatbots we know nowadays. Even during this evolution, not only have chatbots evolved, but they have also contributed to the birth of some technologies that we still use today.

ELIZA, developed in 1966 by Joseph Weizenbaum, is considered the first functional chatbot, created to prove how superficial the interactions between humans and computers are, [Anna and Weißensteiner \(2018\)](#). Despite not being endowed with intelligence, ELIZA inspired another chatbot called ALICE (Artificial Linguistic Internet Computer Entity). Created in 1995 by Dr. Richard Wallace, ALICE promoted a new markup language still used today, AIML (Artificial Intelligence Markup Language). ALICE kept information collected from conversations in AIML files, guaranteeing some learning skills and, consequently, a certain level of intelligence, [Bhagwat \(2018\)](#).

These are just two examples of how chatbots came about. Since then, many others have been developed, and even the way to interact with them has changed. For example, nowadays, we can use chatbots over voice, the ones we call voice assistants, such as Apple's Siri or Alexa from Amazon – more about them in later sections, [Merisalo \(2018\)](#).

In conclusion, the way we interact with chatbots has also changed over the years since their first appearance, and many applications and services make use of them today for a lot more than just fake a human to human conversation.

Later in [Section 2.5](#), are detailed some examples of chatbots that make use of the modern messaging platforms.

2.3 Chatbots Usage

As the years go by, we see an increment in the number of people using instant messaging APPs in their daily life. Smartphones lay in our pockets every day, and we take them wherever we go. We have applications for many different ends inside our smartphones, from productivity in college to procrastination on a social media while scrolling through content shared by our friends and family. Thousands of applications that stand for a variety of different things. However, the most used and the most important for us in our daily lives are messaging APPs. As humans, we cannot live without communicating with each other, and for that reason, the majority of the time we spend on our smartphones is inside a messaging APP. Nowadays, those messaging APPs do a lot more than allow us to keep in touch with each other. Those applications enable us to share instant pictures with all our followers of what we are doing at the moment, play games, do business meetings, share our location with someone we are going to meet, and some let us pay for our purchases at any store. At some point in time, we started thinking about the possibilities of having a “robot” inside of those messaging applications that people could use to chat when they need help with any subject. It was when the concept of chatbot arrived in the APPs that we use everyday. Not only there, but also in

websites and other web services.

A chatbot's popularity increases as its convenience also increase. In other words, the more people tend to need that service, or the more people like to use that service, the more popular that chatbot will be.

A very convenient advantage of chatbots is that, in most cases, they “live” inside an instant messaging APP, and for that reason, people do not have to download additional applications for their smartphones and do not have to learn how to use them. They can search for the chatbot on the contact list of their favorite messaging APP and simply start a conversation, which we already know how to do, and we love to do.

Nowadays, most chatbots use machine learning. They can have personalized service and conversations and learn from every experience they go through. They can also provide feedback to developers with their users' questions and issues that the chatbot could not answer. By doing that, the developer can later update the service to meet its users needs, [Ojapuska \(2018\)](#).

As with everything in this world, we cannot say that chatbots have unique and specific use. People tend to adopt technologies to their own needs, and for that reason, there are always several applications for every piece of tech, and chatbots are no different.

Businesses have been taking advantage of chatbots by using them to chat with users and solve the most common problems or questions instead of having one or more people dedicated to answer the same problem repeatedly. Their conversation focuses on the user's needs and questions. Let us take, for example, a restaurant chatbot. Its objective should be answering questions about the kind of food the restaurant has, what the prices are, book a table, and so on.

Conversations with business chatbots are usually short and take less than 15 minutes, knowing that the user typically only wants to have answered a couple of questions, [Amondarain \(2018\)](#).

2.4 Advantages and Limitations

As with every system, chatbots also have some advantages and features that make them great and some limitations that reduce their implementation as a perfect solution. The next sections make an overview of the main advantages and limitations found within chatbots.

2.4.1 Advantages

The first significant advantage is that a chatbot allows a business improvement. By implementing a chatbot as a feature to provide to its customers, a company can improve its services. Moreover, a chatbot can be available 24x7, which means it is always available to serve customers at any time and without adding more salaries for multiple employees that would be needed if the chatbot was not implemented.

Chatbots can also provide faster customer service and save time to any company by handling with the customers simple and more common questions and letting a specific team focus on the more complex issues.

Companies can also track the user interests and behaviours in order to improve its marketing strategies or its own investments priorities, [Arsenijevic and Jovic \(2019\)](#).

If we expand the chatbot implementation to the personal usage, we could say that a chatbot can also make us more productive, for example, by having a chatbot on one messaging platform and use it to control smart house accessories. Another example is using a chatbot to constantly check a specific service and send a message notifying when the service status changed or it is not working correctly.

2.4.2 Limitations

Although there are a great variety of applications making use of chatbots, they still have some limitations. The most remarkable one relates to its ability to keep

an efficient and effective conversation with the human [Tiha \(2018a\)](#). Some other drawback of chatbots are:

- Not having the ability to recognize grammatical errors;
- Being limited to a closed-domain or being based on what is predefined in a database;
- Not being able to ask questions based on the previous answers;
- Being ambiguous and having some sentences with no context and unclear meaning;
- Having bad accuracy and sometimes suddenly change subject leading to unpredictable responses;
- Begin limited by language structure;
- Not being able to detect the emotion of the human chatting with;
- Poor documentation.

2.5 Real-world Existing Applications

The best way to understand how chatbots are essential in our lives and how they can help us be more productive and efficient is to show examples of real-world applications. The following examples refer to specific and already existing applications, and their objectives are exposed separately.

2.5.1 Cleverbot

Cleverbot was developed by Rollo Carpenter in 1988 and its goal is to perform a human-like conversation with another human. In other words, users can enter

the application and start chatting with the chatbot with no singular purpose. The chatbot will try to maintain a rational conversation in each answer, [Orin \(2017\)](#).

It communicates by implementing rule-based AI techniques and collecting all the data from every conversation with the human. Cleverbot does not have any programmed reply, which means it builds its responses to the human subject based on every knowledge it collected in previous conversations. The chatbot was classified with 59% score by the Turing Test, [Nuruzzaman and Hussain \(2018\)](#). Figure 2.1 shows a screenshot of a conversation with Cleverbot.

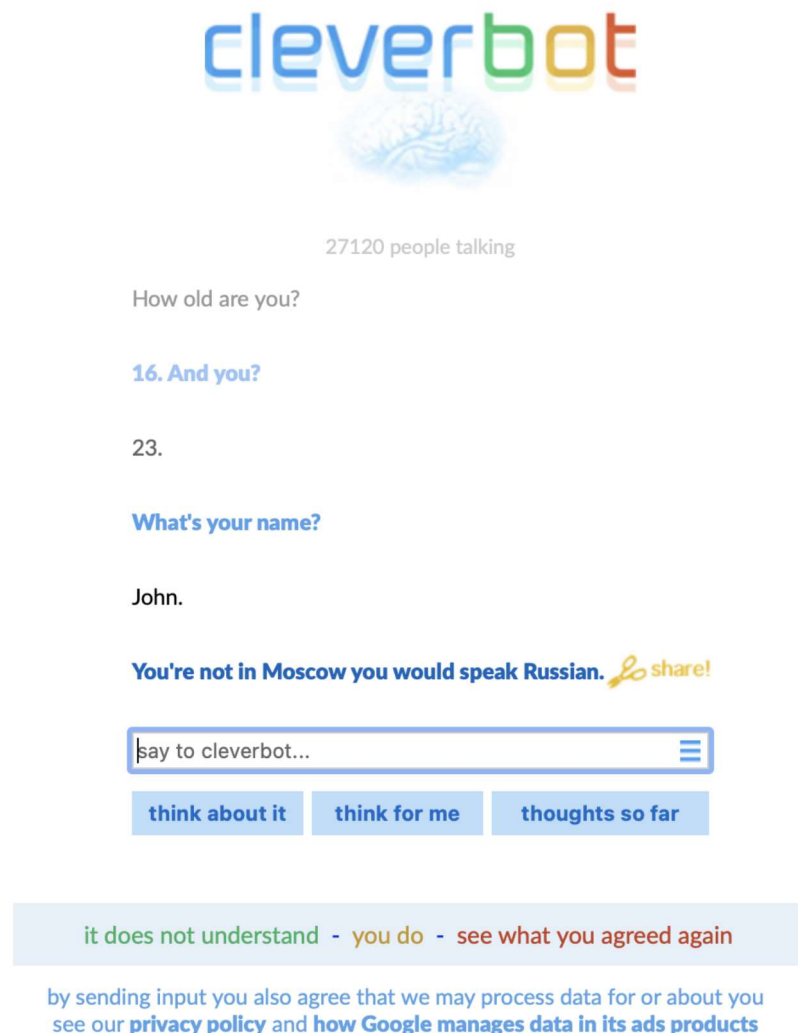


Figure 2.1 – Screenshot of a conversation with Cleverbot, [Cleverbot \(2006\)](#).

This chatbot's most significant purpose is to prove that good algorithms applied in the right chatbots can trick humans and make them think that they are actually talking with another human, [Arnaud Gellens \(2019\)](#).

2.5.2 Instalocate

Instalocate is a chatbot that works on Facebook Messenger and was built as an AI travel assistant to provide the user with a worry-free travel experience. The service combines the power of machine learning with the Internet of Moving Things and voice to help reduce travel anxiety.

Its operation is simple: the service locates flights in real-time, predicts possible problems that may occur, and translates them into delay times, [Orin \(2017\)](#). Figure 2.2 shows a screenshot of a conversation with Instalocate.

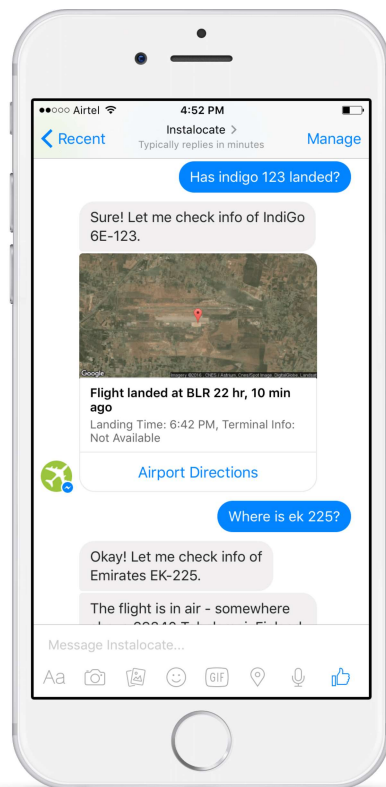


Figure 2.2 – Screenshot of the Instalocate working, [Instalocate \(2019\)](#).

2.5.3 HealthTap

The HealthTap, Inc. is the company that owns Dr. A.I, which is a chatbot dedicated to health care. Dr. A.I. is available not only through text but also via personal assistant with Alexa, for example. This chatbot's main objective is to help people with health issues by analyzing the symptoms and diagnosing it, [Ma et al. \(2018\)](#).

Massive implementation of this kind of chatbots could be the solution for world-wide crises, such as the one we are living right now with the COVID-19 pandemic. A chatbot similar to this one could be the national health system's savior by making medical screening and reducing contacts to the services phone line.



Figure 2.3 – Screenshot of a conversation with HealthTap chatbot, [Robeznikes \(2016\)](#).

Figure 2.3 shows two screenshots of a conversation with the HealthTap chatbot on Facebook Messenger APP.

2.5.4 Domino's Pizza

Domino's Pizza chatbot is the perfect example of how this kind of solutions can free up employees for other tasks. Having a chatbot that handles all the online orders, that helps and advises the customers without the need of human interaction allows to improve customer satisfaction. More than that, the chatbot can learn the user preferences and make suggestions based on that knowledge. Also, it helps reduce the time needed in every order because customers do not need to wait for an employee to be available.

The chatbot allows customers to perform three starting actions: Order, Track Order or Customer Support. Everything a user could need in a basic experience, [AdEspresso \(2020\)](#). Figure 2.4 shows a screenshot of a conversation with Domino's Pizza chatbot on Facebook Messenger APP.

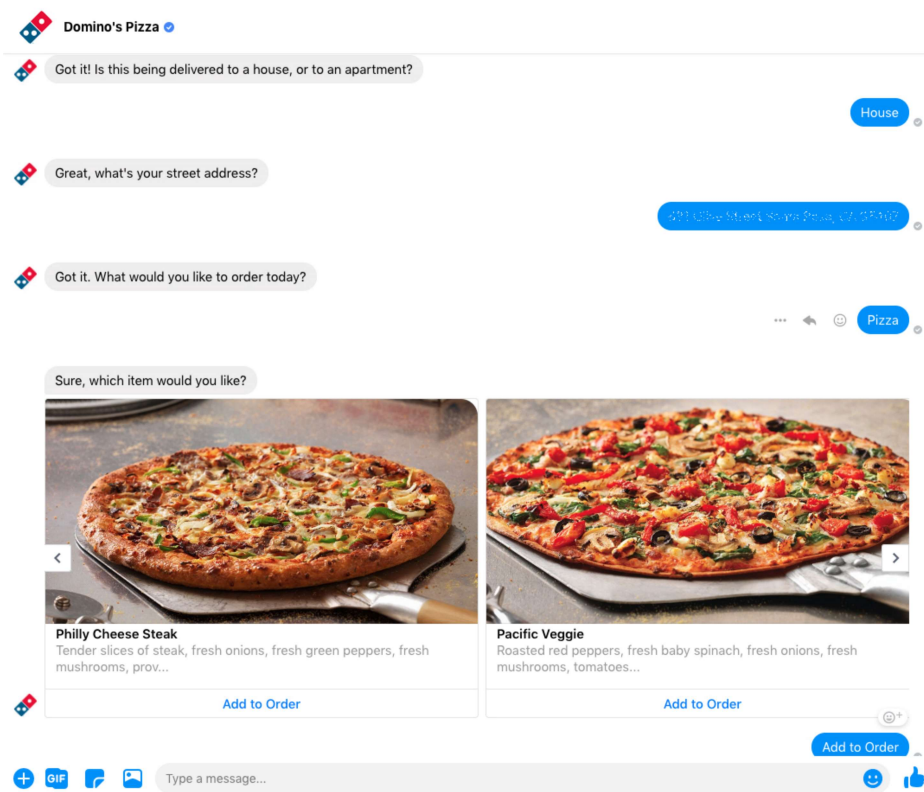


Figure 2.4 – Screenshot of a conversation with Domino's Pizza chatbot, [AdEspresso \(2020\)](#).

2.5.5 Personal Assistants

Personal assistants have been growing in the last years and can help us with our daily tasks. People tend to interact with this kind of chatbots with their voices over their thumbs. Even though personal assistants are a category of chatbots, they're built to interact with the user with their voices over text.

These type of chatbots are probably the most used by the everyday user. That is because they are intuitive, and people do not even need to touch their devices to interact with them. A voice command like "*Hey Siri*", "*Ok Google*" or, only, "*Alexa*" wake up these personal assistants that are ready to listen to us and answer our questions. Personal assistants can tell us the weather, search for news or facts like, "*what is the highest building in the world*", send messages through messaging applications, turn on/off the lights of the house, call a friend, answer an email from work and a lot other more tasks, [Wei et al. \(2018\)](#).

Figure 2.5 shows three different personal assistants from three different companies: Google Assistant, Bixby and Siri.

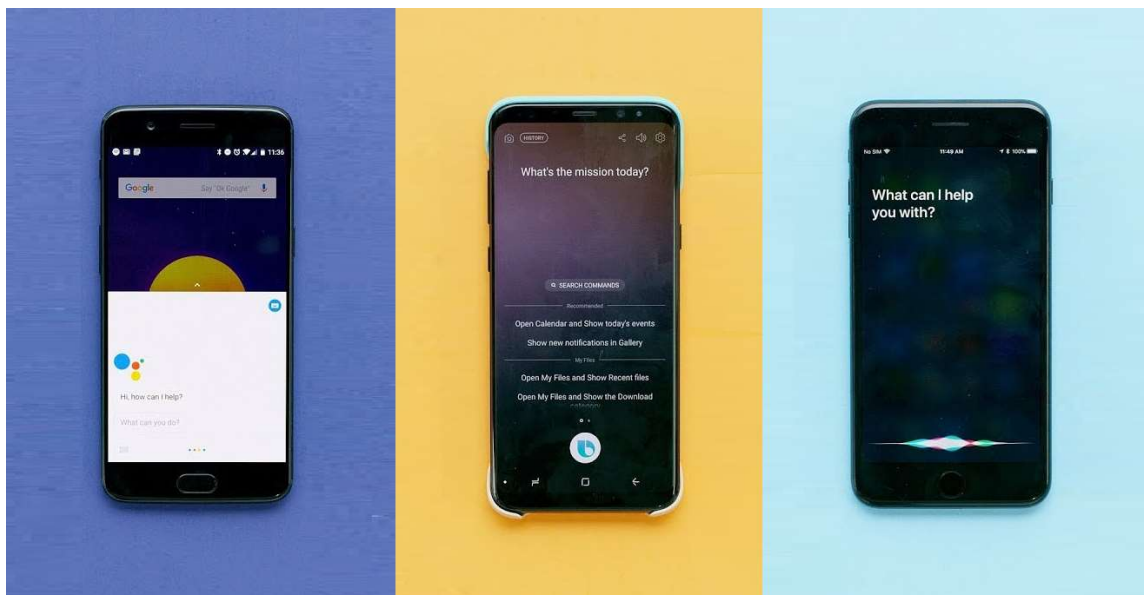


Figure 2.5 – Google Assistant (left), Bixby by Samsung (center) and Siri by Apple (right), [Jackson \(2018\)](#).

2.6 Related Work

After going through some examples of good applications of chatbots and its usage, let us take a closer look on some already existing solutions for the chatbot development.

The objective of this dissertation is to develop a framework for the development of chatbots. For that reason, it is essential to be familiar with other frameworks related to the subject. Chatbots are very popular nowadays, and there is an excellent variety of solutions in this field. This section lists some of the most used frameworks in chatbot development.

2.6.1 Chatbot Frameworks

A critical concept to attempt when developing a program is the reuse of code. The intent of code reuse is simple and obvious: instead of repeatedly coding the same thing from scratch, reuse in new projects what we have already done in other projects [Wang et al. \(2013\)](#).

When we think more in-depth about this matter, we can define three levels of reuse. The lowest level is when the programmer reuses classes like libraries or containers in its code. Frameworks sit at the highest level and make the code the more abstract possible it can get for solving the problem pretended and represent the problems by classes and define relationships between them. The Hollywood principle of “*don’t call us, we’ll call you*” is the basis behind frameworks. For that reason, it lets us define the behavior we desire, and it will call us when it is our time to do something. Between classes and frameworks, it still is the middle level. This level covers the concept of patterns since they are smaller and even more abstract than frameworks, [Shvets \(2019\)](#).

From the above definition of a framework, we can easily describe the concept of a chatbot framework. It is a set of methods and classes that are predefined and can be used by chatbot developers, making the programming process more logical, quick,

and straightforward.

The next sections present some of the popular frameworks for chatbot development at the writing time of this dissertation.

Microsoft Bot Framework

Microsoft Bot Framework is an open-source SDK (Software Development Kit) that provides everything a developer might need for a chatbot. It comes with support for C#, JavaScript, Node.js, Python, Ruby and Java programming languages. Besides, this framework allows the implementation for multiple platforms such as email, Facebook Messenger, Kik, Line, Skype, Slack, Telegram, WeChat, and others more, [Developers \(2019\)](#).

It is widely integrated with Microsoft's Azure cloud service, which is a great benefit for developers aiming to build a chatbot with most of the work simplified and saved on the cloud. Besides that it has free and paid plans that offer some advantages from one to another, [Patil et al. \(2017\)](#).

Wit.ai

Unlike Microsoft Bot Framework, which covers a wide range of the most used messaging platforms, Wit.ai focuses on Facebook services only. Additionally, it is not just for chatbots, and its objective is to provide an easy implementation for natural language. Like most of the frameworks, Wit.ai is also free, but its SDK is only available for Node.js, Python, and Ruby programming languages, [Siegert et al. \(2020\)](#).

This framework is not open-source, but is completely free to use.

Dialogflow

Dialogflow is an online service owned by Google, based on natural language conversation, [Keszocze and Harris \(2019\)](#). This service's fundamental purpose is to provide developers with tools to build chatbots with more natural and rich conversations with the users, providing it with some level of AI, [Muhammad et al. \(2020\)](#).

Although Dialogflow works on many different platforms like Google Assistant, Alexa, Cortana, Facebook Messenger, Skype and Telegram, its focus resides on providing an excellent natural language conversation solution to the developer and not the support for multiple platforms, [Google \(2019\)](#).

This framework requires to be implemented in Node.js and provides both free and paid plans for different approaches and implementations.

Watson Assistant

Similar to Google's Dialogflow, IBM's Watson Assistant powers chatbots with AI technology natural language on various platforms. IBM Watson uses machine learning built over a neural network of one billion Wikipedia words to accomplish this kind of response, [Watson \(2019a\)](#).

The Watson API contains many different packages that make it a good choice for a variety of solutions. It works over webchat, voice, and messaging platforms like Facebook, Slack, and Intercom, [Watson \(2019b\)](#). It is not open-source and, like Dialogflow, also has free and paid plans.

Pandorabots

Pandorabots has integration with the most popular messaging platforms and services, and it has AI support. Besides being a development platform, Pandorabots is also a hosting platform and allows building chatbots on an online web service. One of the most significant advantages of this platform is the support for the open standard

scripting language – AIML (Artificial Intelligence Markup Language), [Pandorabots \(2019\)](#).

Its SDKs are available for the following programming languages:

- Java;
- Node.js;
- Python;
- Ruby;
- PHP;
- Go.

This framework is open-source and has a free and paid plans.

Botpress

Botpress has a modular architecture and is open-source. It allows us to build local chatbots and deploy them to any cloud hosting service. It has natural language understanding, an editor, and is multi-channel, i.e., its chatbots can be used on various platforms, [Botpress \(2019a\)](#).

Its platforms support expands to the following services, [Botpress \(2019b\)](#):

- Facebook Messenger;
- Whatsapp (available through Smooch.io);
- Slack;
- Microsoft Teams.

ChatterBot

Chatterbot is a Python library that adds machine learning algorithms to a chatbot, allowing it to be more efficient and useful. It is easy to implement and can make a software engage in a conversation.

The software starts with zero knowledge of how to communicate and learns with each statement that the user inputs. That means that as more input the Chatterbot receives, the more accurate and smart its answers will be, [Chatterbot \(2019\)](#).

2.6.2 Conclusion

Although there are already many frameworks out there, some do not support Java programming language and services. Besides that, the base core of all of them is machine learning and natural language, which might not be the primary focus of every chatbot developer. Also, some are paid and cannot scale to new or not supported messaging services. Chatbots have become so popular that they are being used for more purposes than just chatting. A good example could be a chatbot used to control the smart devices inside a house. It does not need to have natural language processing, only needing to recognize some specific keywords related to the name and type of devices, the room where each device is, or the action to perform.

To summarize, Table [2.1](#) makes a comparison between all the referenced chatbot frameworks.

Framework	Open Source	Programming Language	Messaging Platforms	AI	Free or Paid?
Microsoft Bot Framework	Yes	Node.js C# JavaScript Python Ruby Java	Facebook Messenger Kik Line Skype Slack Telegram WeChat	No	Both
Wit.ai	No	Python Ruby Node.js	Facebook Messenger	Yes	Free
Dialogflow	No	Node.js	Facebook Messenger Skype Telegram Kik Line Viber Slack	Yes	Both
Watson Assistant	No	Java C++	Facebook Messenger Slack Intercom	Yes	Both
Pandorabots	Yes	Java Node.js Python Ruby PHP Go	Facebook Messenger Viber WeChat	Yes	Both
Botpress	Yes	JavaScript	Facebook Messenger WhatsApp Slack Microsoft Teams	Yes	Free
ChatterBot	No	Python Node.js	Facebook Messenger	Yes	Paid

Table 2.1 – Comparison between all the mentioned chatbots frameworks.



Messaging platforms

Before getting our hands into the Framework's design and implementation, it is crucial to analyze all the messaging platforms used in the development and testing of the Framework.

The development and testing of the Framework requires the usage of different messaging platforms in order to confirm its purpose. For that matter, two messaging platforms were used during the development stage (Facebook Messenger and Telegram) and a third messaging service (WeChat) was later used during the testing of the Framework in order to prove that the Framework can be used in the future with another messaging platforms. The structure of all three used messaging platforms used will still be detailed in the current chapter.

3.1 Usage

Over the years, the use of messaging APPs has been growing considerably, and it does not seem it is even near to stabilize, [MindSea \(2020\)](#). As seen in Figure 3.1, half of the time that people spend on their mobile devices is on social and communication APPs, [Kemp \(2020\)](#). With that in consideration, a chatbot is an excellent tool for

companies that want to save money and make their customers happier. When we want to use a chatbot, we do not even need to exit the APP we are already using. We can simply search for the chatbot as if it were part of our contact list and start chatting with it.

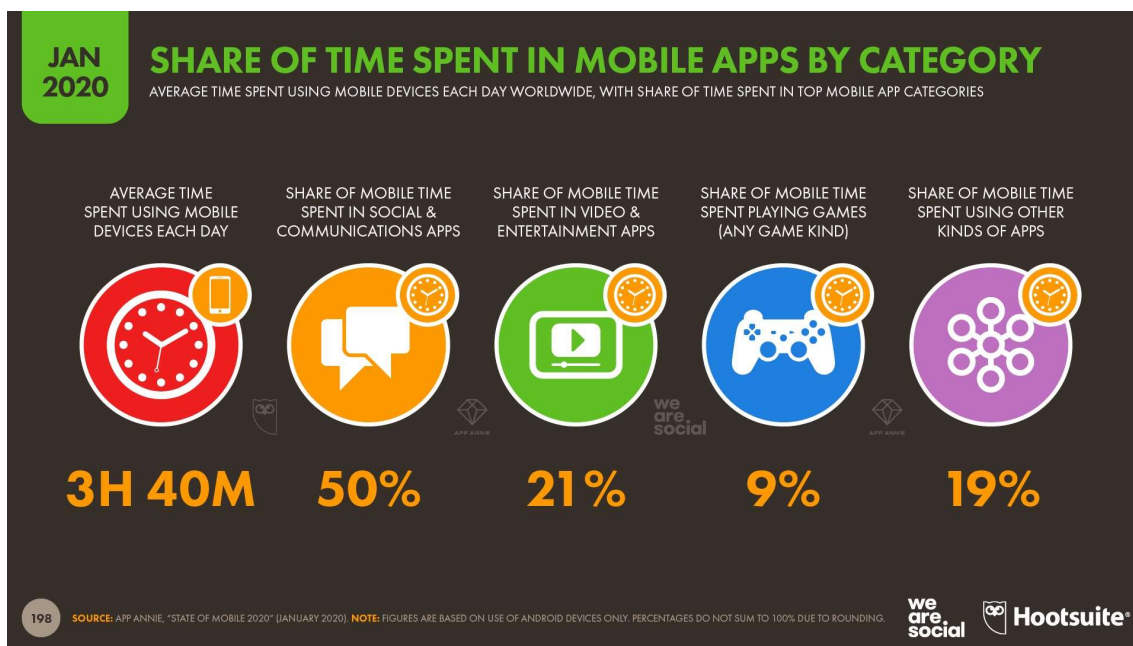


Figure 3.1 – Time spent on social media APPs in July 2020, from Kemp (2020).

In a recent research of the world's most used social platforms, we can take a closer look at the preferred messaging APPs in the whole world. Figure 3.2 shows that the most used messaging service in July of 2019 was WhatsApp, with 1.600 million users, followed by Facebook Messenger, with 1.300 million users. At last, WeChat has over 1.151 million active users, Bucher (2020).

It is to notice that Tencent owns WeChat, one of the biggest Chinese companies, and, for that and other reasons, the Chinese market is its primary focus. In China, WeChat is the most used messaging service and it is integrated with pretty much everything Chinese people use, from trains to buying things in a supermarket. Everything can be done inside WeChat. In contrast, outside of China, WeChat is not very common among users, and WhatsApp and Facebook Messenger are the most famous. However, although those are the most used globally, they are not used in

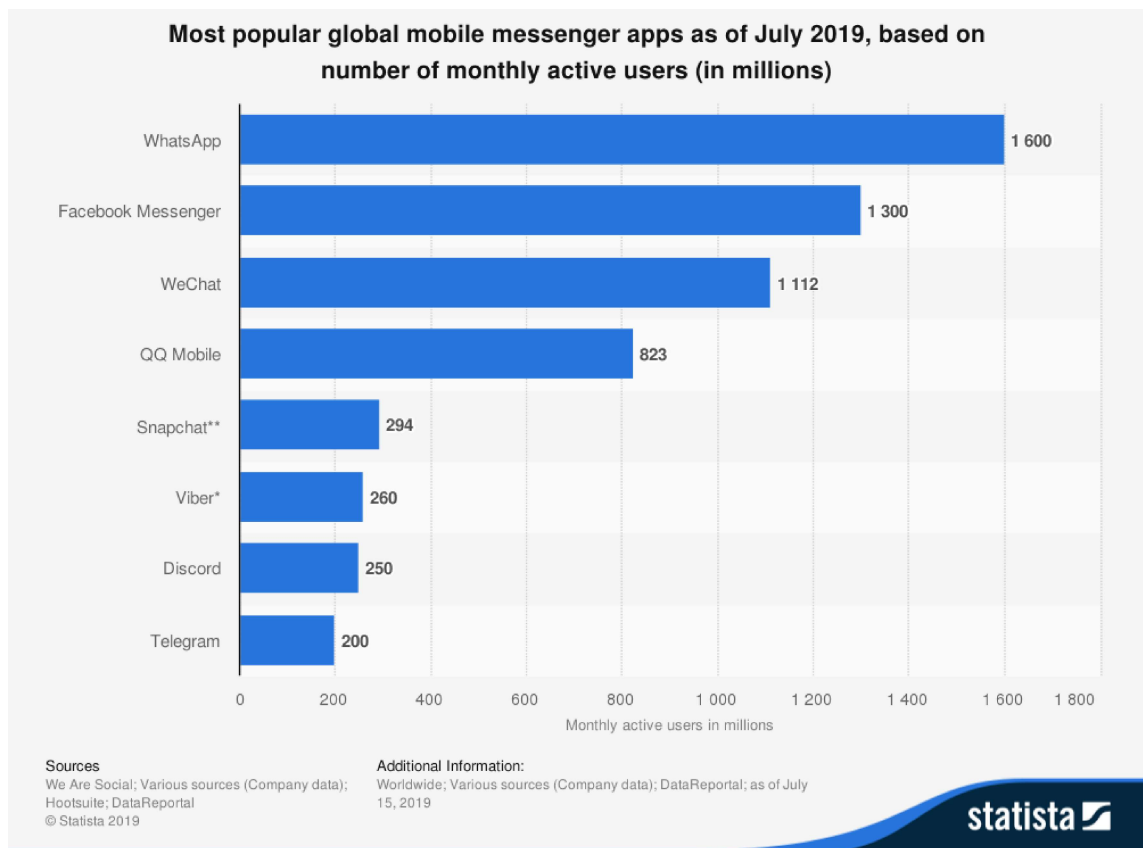


Figure 3.2 – Messaging platforms usage statistics, from [Bucher \(2020\)](#).

China.

Knowing that preferred messaging APPs change from country to country and that companies may want to cover different markets with their chatbot systems, a framework with the ability to develop one code to fit all different platforms could be a game-changer for many companies.

3.2 Facebook Messenger

Being Facebook Messenger, one of the most used messaging APP in July of 2019, and it has support for chatbot implementations, makes it the perfect choice to be one of the first messaging services to be supported by the Framework.

Far from being just a messaging service, Facebook Messenger has an integration with the Facebook company services. That means that businesses with pages on that social media service can expand their services or customer support over to Messenger service and improve its customer satisfaction, reduce the waiting time, and maybe the costs.

3.2.1 Structure

When analyzing the Facebook Messenger platform documentation, we can see the diagram shown in Figure 3.3. It represents how a chatbot works with Facebook Messenger platform and how the communication between the user and the chatbot itself is made.

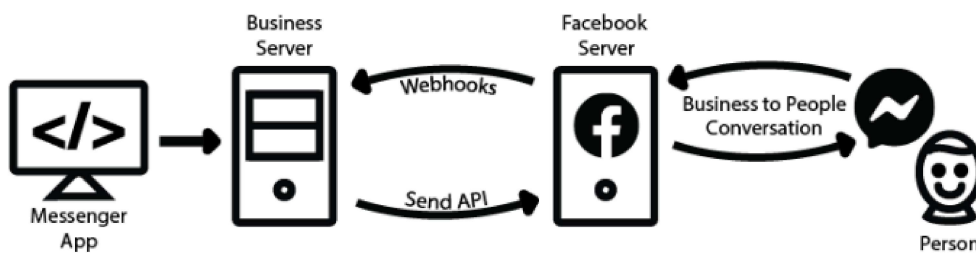


Figure 3.3 – Facebook Messenger basic structure, from [Facebook \(2021\)](#).

We can decompose the diagram in Figure 3.3 to better understand its functionality. When someone sends a message using the Facebook Messenger APP (Person), the message's content is sent directly to a Facebook Server. This server will then use the callback URL defined by the developer for the webhook to send that content to the Business Server. The chatbot will be running as an application inside the Business Server and will process the message and send a reply back to the Facebook Server through the Send API. Finally, the Facebook Server will send the message back to the user (Person) that will receive it on its Facebook Messenger APP.

All the communications between the chatbot application and the Person chatting with the chatbot follow the HTTP protocol and use its methods to make changes

happen in both sides.

With this description of the Facebook Messenger structure, we now understand the steps required for a message to be exchanged between a user and chatbot service.

Components

Facebook Messenger platform offers support for different conversation styles through a variety of components, namely:

- Text messages
- Assets and Attachments
 - Audio
 - Video
 - Images
 - Files
- Message Templates
- Quick Replies
- Sender Actions
- Welcome Screen
- Persistent Menu

These components bring versatility to the service and allow different chatbots implementations. Despite its versatility, this dissertation will mainly focus on sending and receiving text messages only.

3.2.2 Parameters

This platform bases its workflow on Facebook's Send API. This API is used to send messages to users in JSON format that the both parties of this process have to be able to understand and parse. With that said, the Messenger APP that a person uses to chat with a chatbot has to understand the content of that JSON file, as well as any chatbot that wants to make use of that same platform. An example of a JSON payload sent to from a chatbot to a Facebook Messenger server following the Send API is shown in Listing 1.

```
1 {  
2   "messaging_type": "UPDATE",  
3   "recipient": {  
4     "id": "417583215476"  
5   },  
6   "message": {  
7     "text": "Hello, World!"  
8   }  
9 }
```

Listing 1: Example of a JSON message sent by a chatbot to Facebook Messenger server, from Facebook (2021).

Although the JSON file sent from a chatbot to a Facebook Messenger server follows the Send API, the content of the JSON messages received by the chatbot from the Facebook Messenger server are quite different. Listing 2 shows an example of a JSON message received by a chatbot from a Facebook Messenger server.

```
1 {
2     "object": "page",
3     "entry": [{
4         "id": "100172858287426",
5         "time": 1617959435680,
6         "messaging": [{
7             "sender": {
8                 "id": "2035175412698512"
9             },
10            "recipient": {
11                "id": "100172858287426"
12            },
13            "timestamp": 1617959435070,
14            "message": {
15                "mid": "m_8K5HwUvDmFA5fqDtH",
16                "text": "Mensagem."
17            }
18        }]
19    }]
20 }
```

Listing 2: Example of a JSON request sent by Facebook Messenger to a chatbot, from Facebook (2021).

As it can be seen in Listing 1, Facebook Messenger requires the chatbot to send some basic parameters for the message to reach the user. The required parameters change depending on a variety of factors, such as the type of message sent. Table 3.1 lists the basic objects needed in a JSON payload to successfully send a message from the chatbot to the user.

Content	Type	Required	Value
messaging_type	String	Yes	RESPONSE UPDATE MESSAGE_TAG
recipient	Recipient	Yes	-
message	Message	Yes	-
sender_action	String	No	typing_on typing_off mark_seen
notification_type	String	No	REGULAR SILENT_PUSH NO_PUSH
tag	String	No	CONFIRMED_EVENT_UPDATE POST_PURCHASE_UPDATE ACCOUNT_UPDATE HUMAN_AGENT

Table 3.1 – Facebook Messenger payload parameters, from [Facebook \(2021\)](#).

All the non-required parameters can be used in simple requests to show actions or events in the user messaging APP. For example, the payload value “typing_on” assigned to the parameters “sender_action” can be used by the chatbot to show the user that its message is being processed. Or, the “mark_seen” can be used to show the user that the chatbot is running and has received its message.

The “recipient” and “message” objects, shown in Table 3.1 are detailed in tables 3.2 and 3.3, respectively.

Content	Type
id	String
user_ref	String
post_id	String
comment_id	String

Table 3.2 – Facebook Messenger Recipient object parameters, from [Facebook \(2021\)](#).

The Recipient object is mandatory and the platform will not accept the message if this parameter is not set. That is because it is this object that tells the platform to whom to send the message to. For that reason, at least one of the Recipient parameter has to be present.

Content	Type
text	String
attachment	Object
quick_replies	Array <quick_reply>
metadata	String

Table 3.3 – Facebook Messenger Message object parameters, from [Facebook \(2021\)](#).

In the same way, the Message object cannot be skipped and represents the type of content and actually contains the content that the chatbot is sending to the user. In this object, either text or attachment have to be present, and both quick_replies and metadata are optional.

All these values will be important during the development of the framework in order to create objects and classes that represent them in a universal way and allow to communicate with other messaging platforms.

On the other hand, as mentioned earlier, the events sent by the Messenger platform to the chatbot application are quite different than the ones requested by the Send API. Listing 2, written in JSON format, is an example of a text message event received from Facebook Messenger API.

All events received from the Messenger platform follow the same basic structure as the one seen in Listing 2.

Now that we already know how the platform structures its content, we have to understand how to establish a communication between the chatbot application and the Facebook Messenger platform.

3.2.3 Handling HTTP Requests

Messenger's API sends and receives data using the HTTP protocol and uses mostly GET and POST requests. It performs a GET request to set or verify if the webhook is correctly working and POST requests to send/receive messages.

When the API performs a GET request it expects the application to return a **200 OK** HTTP response after receiving the event. If this step is not successful, the Messenger platform will send the webhook event again every 20 seconds until it receives the application server's right response. After the webhook is set, messages are sent and received using the HTTP POST method.

If the chatbot wants to send a message to a user, it must perform a POST request to a specific URI with a JSON message on its payload. The request is sent to a URI similar to the following example: *<https://graph.facebook.com/v8.0/me/messages?6NJDUri6lZBI78QZAmqDsvZAsOZCjthVEd=3ASgqKoKo3Fpam3Whgg0Nj74shCq4e>*.

An example of a JSON payload sent to the Facebook server is shown in Listing 1 and 2 above mentioned.

When handling the HTTP requests from and to the Facebook Messenger platform, is important to notice that some of them might use JSON Web Token (JWT) to increase data security. This technology will be detailed later in Chapter 4.

3.2.4 Webhook

Giving that the communication is made using the HTTP protocol, it is required a callback URL to be set inside the service's platform. Therefore, it is essential to understand how to set the callback URL for the webhook events. Besides, this is also a required step to validate that the developer controls or owns the domain hosting the chatbot application. To do that, we first have to create an APP inside the Facebook Messenger Developers Platform and then access the following URL in order to access the APP created:

<https://developers.facebook.com/apps/>

We can then select the application that will be our chatbot and change its settings inside the “products” menu. In the “Webhooks” section, we can enter the callback URL together with the Verify Token. This token is a random string generated by the developer. Figure 3.4 highlights the interface where we have to set all the webhook properties used to authenticate every attempt of communication from the chatbot.



Edit Callback URL

Callback URL

<https://www.facebook.com/messenger>

Verify Token

EAAB53C6WkbEBABO7Gkvm5RVRL7ZBah97wAHJuYkdZCZAFmlhZALyChvVno5dYL4UmZAuldzmpGwlvExiJf

[Learn more](#) Cancel Verify and Save

Figure 3.4 – Facebook Messenger setting webhook interface, from Facebook (2021).

After these settings are applied and with the chatbot application running, the application will receive an HTTP GET request with a JSON Web Token explicit in Table 3.4.

Parameter	Description
hub.mode	Set to subscribe
hub.verify_token	The custom verify token that the developer provided
hub.challenge	Generated by the Messenger Platform. Contains the expected response.

Table 3.4 – JWT received from the Facebook Messenger platform when setting a callback URL, from [Facebook \(2021\)](#).

By analyzing the GET request, those parameters have to be set according to the provided documentation. In other words, the **mode** has to be set as “subscribe”, the **verify_token** needs to contain a string with the same token provided inside the Messenger platform and, at last, the **challenge** has to be returned in a GET response to the platform with the HTTP status code 200 OK. With all this done, the callback URL is defined, and a connection can be effectively established between our chatbot and Messenger platform, ensuring that the messages can be exchanged between our chatbot and any person using the Messenger service, [Facebook \(2021\)](#).

3.3 Telegram

Famous thanks to its end-to-end encryption and the ability to subscribe to channels inside the APP, Telegram is also very popular among young users or freelancers that want to share their work. In addition, Telegram chatbots are very versatile and allow for multiple configurations.

Also, remembering [Figure 3.2](#), in July of 2019, Telegram was the eighth most used messaging service.

3.3.1 Structure

Telegram’s platform supports two exclusive ways of receiving updates: Webhooks and Long Polling. the key differences between them two is that while webhooks just

wait for an update message, the long polling method is constantly “asking” to the server if there is new information available. Between these two, webhooks represent a better option mainly because it is the same method already used by the Facebook Messenger platform.

Knowing that the webhook usage is barely the same, we can consider that the diagram relative to Facebook Messenger shown in Figure 3.3 also applies to the Telegram platform and its chatbots, Telegram (2020).

3.3.2 Parameters

Telegram structure is very different from the Facebook Messenger structure. Table 3.5 represents the Telegram API update object. This is the base object of every event, either it represents a text message, and image or a poll.

Content	Type	Required
update_id	Integer	Yes
message	Message	No
edited_message	Message	No
channel_post	Message	No
...

Table 3.5 – Telegram Update object parameters, Telegram (2020).

The above is truncated because the rest of the parameters are not of interest for the development of this dissertation and can be consulted in Telegram API documentation.

Observing the composition of the update object, for the development of this dissertation, we have special interest only in the Message object.

The Message object covers all the types of messages that can be sent using the platform. For that reason, it contains a great variety of parameters that do not have interest for the case of study. Table 3.6 shows some parameters of interest of

the Message object.

Content	Type	Required
message_id	Integer	Yes
from	User	No
sender_chat	Chat	No
date	Integer	Yes
text	String	Yes
chat	Chat	Yes
...

Table 3.6 – Telegram Message object parameters, [Telegram \(2020\)](#).

The User object represents the user sending and receiving messages using the Telegram messaging service. In Telegram API, chatbots are also considered users, and for that reason, they have to follow this object. A chatbot also has a user ID, and all the parameters that any other user has. Table 3.7 lists all the parameters of this object.

Content	Type	Required
id	Integer	Yes
is_bot	Boolean	Yes
first_name	String	Yes
last_name	String	No
username	String	No
language_code	String	No
can_join_groups	Boolean	No
can_read_all_group_messages	Boolean	No
supports_inline_queries	Boolean	No

Table 3.7 – Telegram User object parameters, [Telegram \(2020\)](#).

At last, the Chat object. This object represents a chat and it is important to identify a conversation although its main parameters are identical to the User parameters.

Content	Type	Required
id	Integer	Yes
type	String	Yes
title	String	No
username	String	No
first_name	String	No
last_name	String	No
photo	ChatPhoto	No
bio	String	No
...

Table 3.8 – Telegram Chat object parameters, [Telegram \(2020\)](#).

All the objects mentioned above are parsed into a JSON formatted payload that is sent over an HTTP request. Listing 3 shows a JSON formatted file containing those objects.

```
{
  "update_id": 80611948,
  "message": {
    "message_id": 312,
    "from": {
      "id": 813501482,
      "is_bot": false,
      "first_name": "Jo\u00e3o",
      "last_name": "Valente",
      "username": "joaovalentee",
      "language_code": "pt-br"
    },
    "chat": {
      "id": 813501482,
      "first_name": "Jo\u00e3o",
      "last_name": "Valente",
      "username": "joaovalentee",
      "type": "private"
    },
    "date": 1603727532,
    "text": "Testing Telegram"
  }
}
```

Listing 3: JSON payload received from the Telegram platform.

3.3.3 Handling HTTP Requests

Similarly to the Messenger platform, Telegram also exchanges its messages using HTTP POST requests. In this case, the request has to be sent to the following URL: *<https://api.telegram.org/bot123456:ABC-DEF1234ghIk23ew11/sendMessage>*.

The content of the payload for text messages has to be similar to the example in Listing 4.

```
{  
  "chat_id": "3546875132",  
  "text": "Testing Telegram chatbot."  
}
```

Listing 4: Example of a JSON formatted file for the Telegram text messages.

Although the example in Listing 4 is quite simple, Telegram has an extensive list of parameters that make its chatbots very diverse.

3.3.4 Webhook

We need a webhook in order to establish a communication between the chatbot and the Telegram platform, that will then communicate with the users.

The process to declare the callback URL to the Telegram platform is different than the one used by the Messenger platform specified in Section 3.2.4. In this case, we do not need to submit the parameters in the service platform. Instead, our chatbot has to perform a POST request to the Telegram API containing a JSON payload with the callback URL directly to a URI like this: *<https://api.telegram.org/bot123456:ABC-DEF1234ghIk23ew11/setWebhook>*.

Listing 5 shows the JSON file sent to the Telegram platform in order to define the callback URL for the webhook events.

```
{  
  "url": "https://3b8b44fb0afd.ngrok.io/telegram"  
}
```

Listing 5: Telegram JSON text to the the webhook.

More parameters can be sent inside this message but the URL is the only one required to make a chatbot work. Other parameters needed can be found inside the Telegram documentation.

After submitting the POST request to set the webhook, a response will be received inside the chatbot application. If the response is a 200 HTTP status code then the webhook is successfully set, otherwise, something went wrong with this process.

3.4 WeChat

WeChat is mainly used by the population of China, being the most famous messaging application in China, right next to QQ, also from Tencent. In there, WeChat is used to do a lot more than just chatting with friends and family. Chinese people understood pretty soon the power of the messaging applications and have put every major services inside of it. They can chat, order items from stores, pay in every shop, restaurant or even the subway, play games, share moments of their lives in the feed of moments, and a lot more things. One of those things is, of course, talk to chatbots. Chatbots are everywhere, whether it is a store or a service, and being WeChat on the third place of the statistics of the most used messaging APPs in the world, as mentioned in Figure 3.2, it would be the APP of choice if any company wants to expand its business to China.

Being more focused on the Chinese market, WeChat has its documentation mainly written in Chinese, which can represent a barrier to understand how its chatbots can be implemented. A translator was used in order to surpass this obstacle.

3.4.1 Structure

Exactly as both Facebook Messenger and Telegram platforms, WeChat is based on the same concept shown in Figure 3.3. It means that WeChat communication also works using HTTP protocol with the help of webhooks, and the process to send a message from a user to the chatbot application and the reply sent by the chatbot back to user is identical to the process in that figure, [WeChat \(2020\)](#).

3.4.2 Parameters

Unlike Facebook Messenger and Telegram, WeChat's platform does not send its data to a chatbot in a JSON format. That means that the content is not structured as an object and, because of that, in its documentation, WeChat has the content of each type of message individually detailed.

Considering that all the parameters inside the API documentation may not be of interest for this implementation, only the basic parameters together with the parameters to send a text message and an image attachment will be taken in consideration for the analysis, [WeChat \(2020\)](#).

Table 3.9 lists all the common parameters used in every message sent by the API to the chatbot application. These parameters are all required and have to be present in every message sent.

Content	Type
ToUserName	Integer
FromUserName	String
CreatTime	Integer
MsgType	String
MsgId	Integer

Table 3.9 – WeChat payload basic parameters, from [WeChat \(2020\)](#).

Depending on the type of message sent, different parameters are added to the ones present in Table 3.9. With that said, to send a text message, the parameter in Table 3.10 has to be added to the the previous list of parameters in 3.9.

Content	Type
Content	String

Table 3.10 – WeChat text message parameters, from [WeChat \(2020\)](#).

In the same way, when an image is meant to be sent, the parameters in Table 3.11 have to be added to the ones on Table 3.9.

Content	Type
PicUrl	String
MediaId	String

Table 3.11 – WeChat image object parameters, from [WeChat \(2020\)](#).

3.4.3 Handling HTTP Requests

WeChat requires the chatbot application to send its content in JSON format in the body of an HTTP POST request. But, when the platform redirects the messages sent by the users to the chatbot server, it uses an XML structure. For that reason, the payload the chatbot receives is different than the payload it sends. Listing 6 shows an example of a payload received by a WeChat chatbot application in XML format.

```
<xml>
  <ToUserName>
    <![CDATA[gh_1450e947aefa]]>
  </ToUserName>
  <FromUserName>
    <![CDATA[o5dd36M20fwn3f9A-k0fe5vVTI]]>
  </FromUserName>
  <CreateTime>1615302952</CreateTime>
  <MsgType><![CDATA[text]]></MsgType>
  <Content><![CDATA[Hello World!]]></Content>
  <MsgId>2312531644507951182</MsgId>
</xml>
```

Listing 6: Example of a WeChat payload message sent by the WeChat server.

Listing 7 shows an example of a message sent for the chatbot to the WeChat server in order to reply to a user. This time, the content is JSON-formatted.

```
{
  "touser": "o5dd36M20fwn3f9A-k0fe5vVTI",
  "msgtype": "text",
  "text":
  {
    "content": "WeChat: Hello World!"
  }
}
```

Listing 7: Example of a WeChat payload message sent by the chatbot application.

3.4.4 Webhook

As Facebook Messenger and Telegram, WeChat also works with webhooks to send the messages received from the users to the chatbot application.

Setting the callback URL for the webhook events for the WeChat chatbot is identical to the process performed in the Facebook Messenger platform. With that said, we can find the interface shown in Figure 3.5 when accessing the WeChat developer page the URL:

<https://mp.weixin.qq.com/debug/cgi-bin/sandboxinfo?action=showinfo&t=sandbox/index>

接口配置信息

请填写接口配置信息，此信息需要你有自己的服务器资源，填写的URL需要正确响应微信发送的Token验证，请阅读[消息接口使用指南](#)。

URL

https://api.weixin.qq.com/cgi-bin/

wechat

Token

EWABE17C7E0E42BF7C257E0BFCAABF

提交

Figure 3.5 – WeChat original setting webhook interface.

Giving that this page has only a Chinese version, the interface in Figure 3.6 was obtained using a built-in translator inside the browser.

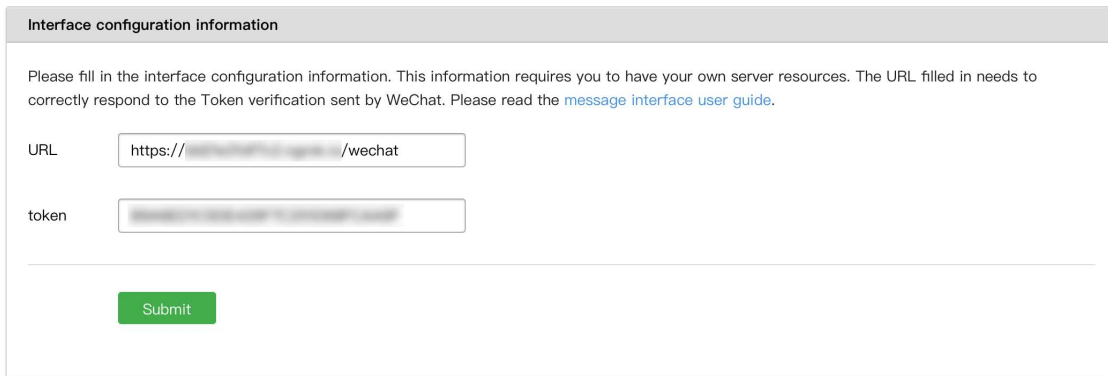


Figure 3.6 – WeChat translated setting webhook interface.

After setting the URL and the token values, clicking the “Submit” button makes the platform perform a GET HTTP request to the chatbot application with four important parameters, shown in Table 3.12.

Parameter	Description
signature	Weixin encrypted signature. The signature combines the token parameter entered by the developer and the timestamp and nonce parameters in the request.
echostr	Random number.
timestamp	Timestamp.
nonce	Random String.

Table 3.12 – GET request received by the application from the WeChat platform when setting a callback URL, from WeChat (2020).

The application has to verify the content of the payload received from the WeChat platform and reply back with a 200 status code response, containing the *echostr* value from the above table value in its body.

3.5 Other Platforms

As said previously, the framework has to be prepared to receive new platforms to come in the future. For that matter, Facebook Messenger and Telegram will be integrated into the framework and later, when testing it, WeChat will be implemented as an external messaging service, to simulate a future new platform that could emerge and be added to the framework by a chatbot developer.



Software Concepts

Before presenting the development of the Framework proposed in this dissertation, it is essential to understand the technologies and methodologies that were used in its development. Therefore, these concepts and methodologies are going to be presented in this chapter, to better understand all the framework development choices.

4.1 Development Methodology

All work that involves software development must follow a specific model with its own software development methodologies. In this sense, the Waterfall model was the base behind all software work during this project.

4.1.1 The Waterfall Model

When coding, the developer should always follow a good model of software development. Over the years, different software development approaches have emerged, and some of them have become more popular than others. The Waterfall model is the first and most traditional of them all. However, despite its age, it is still used

in the current days, [Dooley \(2017\)](#).

Another widely used model is Agile. This plan-driven process model is based on the simple concept that everything always changes. The Waterfall model assumes that what is set during the requirements analysis, which is the model's first step, never changes. On the other hand, the Agile model wants frequent software deliveries. Regular deliveries lead to frequent and fast feedback from the customers using the software in development and, consequently, changes in the requirements that improve the future software, [Filipova and Vilao \(2018\)](#).

The Waterfall model is the basis behind the development of this Framework. In comparison with Agile, the Waterfall model is more set to the type of projects with a limit date and not so frequent and regular feedback. This does not invalidate the fact that the project can be improved in the future or that it receives feedback from the users, only that the feedback is not quick and regular.

With that said, and before detailing the Framework's design, let us have a closer look at what the Waterfall model requires. Figure 4.1 shows all the steps taken into account during the software development process.

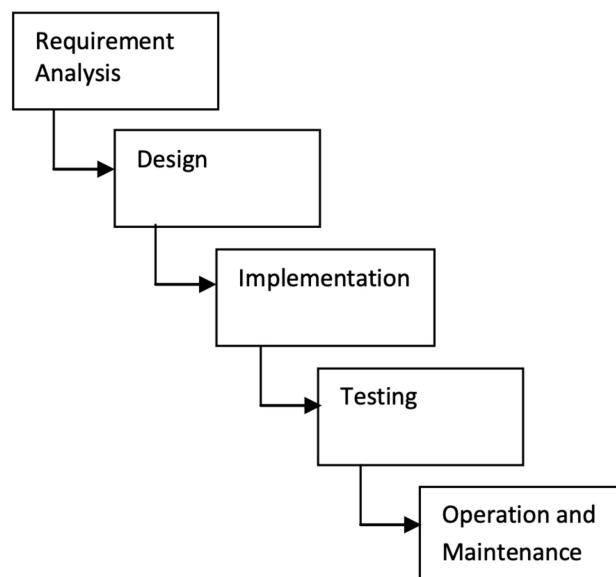


Figure 4.1 – Waterfall model of software development, from [Adenowo and Adenowo \(2020\)](#).

The first step to take is to analyze all the requirements for the project. In other words, the final result of this dissertation has to fulfill all the requirements settled in its beginning. After having all the requirements properly listed, the Waterfall model suggests the design of the code. During this step, it is analyzed the data collected in the requirements stage, [Petersen et al. \(2009\)](#). The system's design is made independently of the hardware and software, so it can be considered an abstract implementation study of the future code to be made. This stage can be taken to the lower-level of the design only after the higher-level logical design is completed. When reaching that step, the hardware and software can be taken in place for the design, [Eason \(2016\)](#). The third stage reflects the actual implementation that, in this case, represents the writing of all the code that will make the framework. The verification, and fourth stage, is relative to the testing made to the code before declaring it finalized and will be explained in Chapter 6 of this document, [Gallagher et al. \(2019\)](#). The last, but not least critical stage, the maintenance phase, is covered in this dissertation's last chapter.

4.2 Design Patterns

Design patterns are solutions to common problems in software development. They work as a blueprint that can solve a particular problem by adapting them to the problem in hands, [Nahar and Sakib \(2016\)](#). There is a big variety of different patterns, but for the development of this Framework, only two seem to be important: the Factory and the Observer Design Patterns. While the Factory Design Pattern is related to creating objects, the Observer Design Pattern is a behavioral design pattern that, as its name implies, handles the behavior of a specific part of the software.

The reason why these two are important and how these concepts will help to accomplish a working framework will be detailed in the next sections.

4.2.1 Factory Design Pattern

The Factory Design Pattern is a software model that provides an interface for creating objects through a superclass, ensuring that its subclasses can change the type of object that will be created, [Shvets \(2019\)](#).

This concept will allow the developer using the framework to simply “ask” the framework to create an object that will allow the communication with a specific messaging service, and get an object back from the factory method inside the framework. By applying this pattern, the usage of the framework becomes a lot more simplified.

To better understand the concept of the Factory Design Pattern, Figure 4.2 shows a simple UML (Unified Modeling Language) diagram of this pattern.

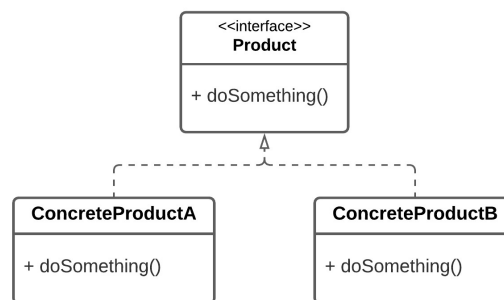


Figure 4.2 – A Factory Design Pattern UML diagram, adapted from [Shvets \(2019\)](#).

The **Product** declares an interface representing the basic methods of the concrete product that will be produced. The **ConcreteProduct** is the actual product that the developer wants to build, and it is a specific implementation of the **Product** interface. Even knowing that the **ConcreteProduct** implements the **Product** interface, each implementation can have its own specific methods. In the example of the diagram represented in Figure 4.2, the implementation of the method **doSomething()** will define what the **ConcreteProduct** is supposed to do. But, for example, **ConcreteProductA** can have more methods that are unique to it that the **ConcreteProductB** does not need to have, and vice-versa.

As it is, this concept is already handy in this case scenario. However, there is still missing a vital feature of this design pattern. Like in a real factory, the example in Figure 4.2 also needs a creator. The **Creator** represents the method that can create something specific by returning new product objects. Figure 4.3 shows a complete UML diagram implementation of this concept.

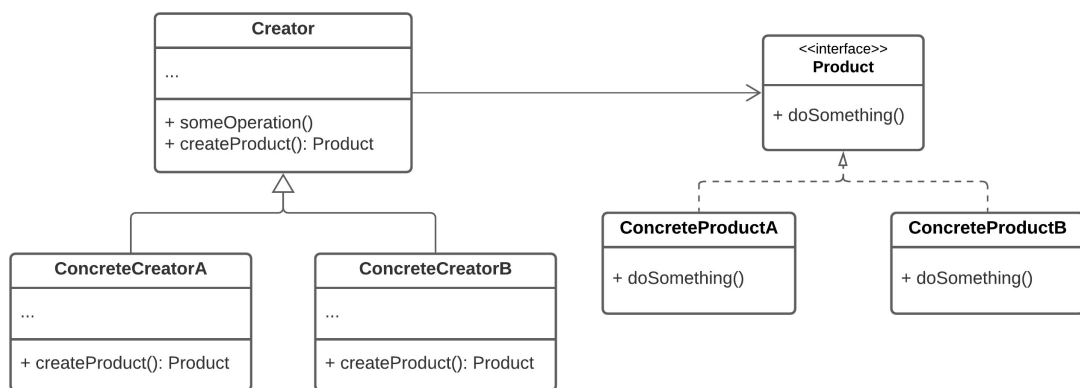


Figure 4.3 – The complete Factory Design Pattern implementation UML diagram, adapted from Shvets (2019).

The **Creator** usually is an interface or an abstract class that forces its implementations to override and specify its methods. For that reason, the **ConcreteCreator** is responsible to create a specific **ConcreteProduct**. For example, the ConcreteCreatorA will return an object of the kind of ConcreteProductA.

The implementation of this pattern in the Framework is later described in Chapter 5.

4.2.2 Observer Design Pattern

The basic operation of the Observer Design Pattern is simple: it allows the establishment of a subscription system that will notify multiple objects when an event occurs in the object being observed, Shvets (2019).

In the framework, this pattern will allow the developer implementing it to subscribe to a notification system for each chatbot messages. Therefore, the object subscribing

to the notification system will be notified every time a new event occurs in that specific messaging platform.

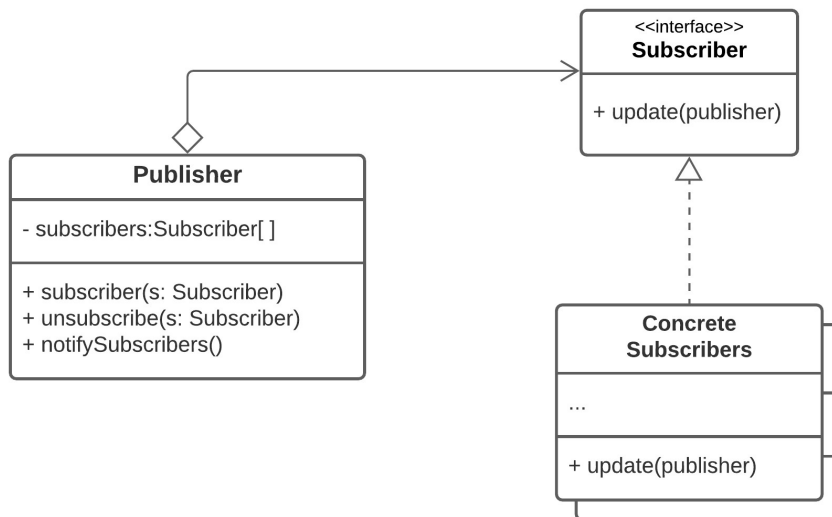


Figure 4.4 – Observer Design Pattern UML diagram, adapted from Shvets (2019).

An abstract implementation of this pattern is detailed in the UML diagram in Figure 4.4. In there, the **Concrete Subscribers** implement the **Subscriber** interface. They are interested in knowing when some event occurs in the **Publisher** object. For that reason, they have to subscribe to the Publisher to be notified when that event happens.

On the other hand, the Publisher has to have a list of all its subscribers. When a change happens on its states or an event occurs, the Publisher will run through its subscribers' list and call each notify method. If it is necessary, it can send data through that same method.

Now, when the Publisher calls the notify method, each Concrete Subscriber will have its own and personal update method that will produce a unique reply.

4.3 HTTP Protocol

The chatbot application has to communicate with a platform unique to each messaging service when creating a chatbot solution. The process of exchanging messages between a chatbot and each platform is made through HTTP (Hypertext Transfer Protocol), or HTTPS (detailed later in this chapter). HTTP is a request/response protocol, where the client sends a request to the server using one of the methods provided by the protocol with all the necessary information. After processing the request, the server sends a response to the client with a response code, Stanivuk et al. (2017).

According to the OSI (Open System Interconnection), the HTTP protocol sits on the last layer – the Application layer.

Every HTTP request must contain the following fields, RFC2616 (1999):

- **Method** – responsible to identify the type of request performed by the client to the HTTP server, and vice-versa;
- **Uniform Resource Identifier (URI)** – identifies the resource that the client desires to access via name, location, or any other characteristic;
- **HTTP version** – represents the version of the HTTP protocol in use and it is required to inform the client how the message is structured.

Since all of these fields are required for the HTTP protocol to work correctly, the next sections describe each of them in more detail.

4.3.1 Methods

The HTTP protocol relies on multiple methods to be functional. Although all of them are important for the protocol to work, we will focus on those necessary to

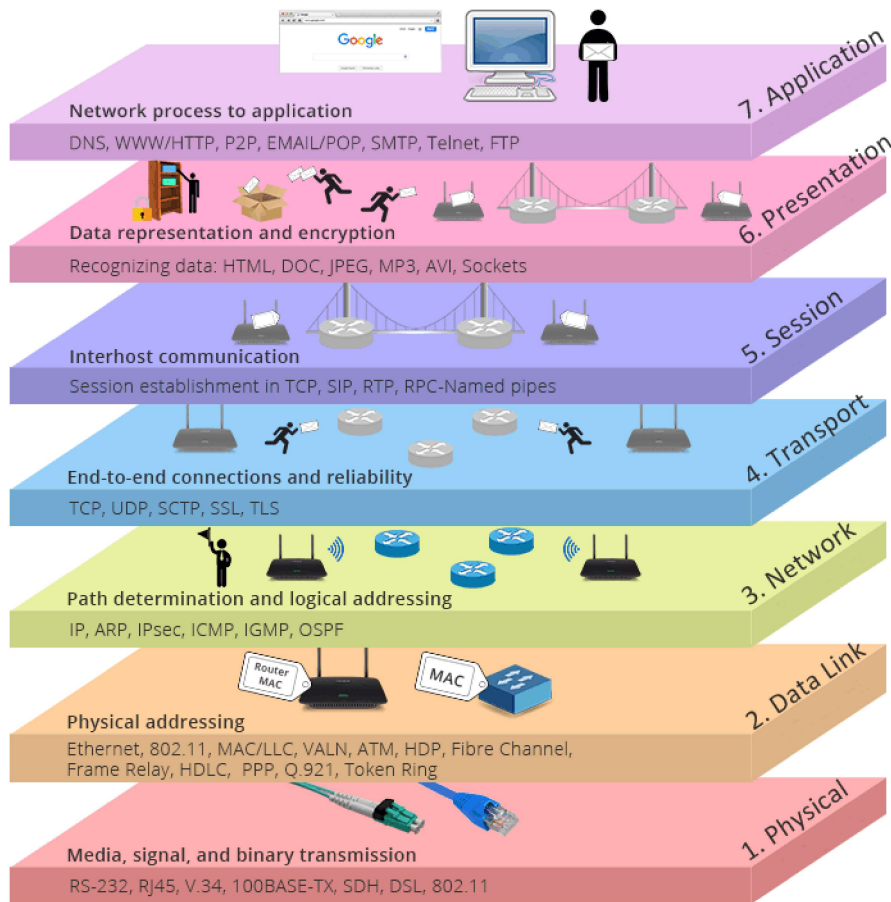


Figure 4.5 – OSI Model, from Suite (2019).

manipulate data. These methods are responsible for performing CRUD (Create, Read, Update and Delete) operations, RFC2616 (1999):

- **GET** – this method requires the data of a specific resource;
- **POST** – used to submit data to a specific resource, usually causing changes on the server;
- **PUT** – requests that the supplied Request-URI stores the enclosed entity;
- **DELETE** – requests that the server delete the resource identified by the Request-URI.

4.3.2 URI

The URI is what allows the client to access a specific resource. In practice, this is what enables the messaging platform to reach a chatbot application. An example of a URI is present in the Listing 8, [RFC3986 \(1999\)](#).

```
https://3b8b44fb0afd.ngrok.io/messenger
```

Listing 8: Example of an URI

The usage of URIs makes possible the use of multiple platforms by the same application without having different chatbots answering for the wrong service because it allows to identify each different messaging platform.

4.3.3 HTTP versions

HTTP has been used as the most common worldwide web protocol since 1990 and, because of that, it makes it being in constant development. Since the 1990s, HTTP has improved that has gone through several versions.

Although version 1.1 is the most adopted version, HTTP has already moved to version 2.0 and has some important differences compared to the HTTP/1.1, namely, [AB-COM \(2019\)](#):

- it is a binary protocol rather than text. This allows for faster transfers, and also, it unleashes new techniques that may be implemented;
- it is a multiplexed protocol which means that multiple requests can be handled at the same time and over the same connection;
- reduces the number of headers when it detects that there are duplicates;
- allows the server to push data into the client cache, instead of being the client to require that information.

The protocol continues to evolve, and the version HTTP/3 is already in development and will bring changes in the transport layer by replacing TCP/TLS (Transmission Control Protocol/Transport Layer Security) in advance for QUIC when a secure connection is needed, [Mozilla \(2020a\)](#).

4.3.4 HTTP Message Structure

Each message send or received through the HTTP protocol has a specific structure. Both the requests and responses share a same structure and are composed by, [Mozilla \(2020b\)](#):

- **Start-line** – it is the first line of the HTTP messages and describes the method for the request or the status code in case of a response;
- **Headers** – a set of HTTP optional headers that can specify a request or describe the body of the response message;
- **Empty-line** – indicates that all the meta-information needed has been sent;
- **Body** – contains the actual information that is meant to be sent, either in requests or responses.

Figure 4.6 shows an example of the structure of an HTTP request and response message where are explicit all the parameters and concepts previously detailed.

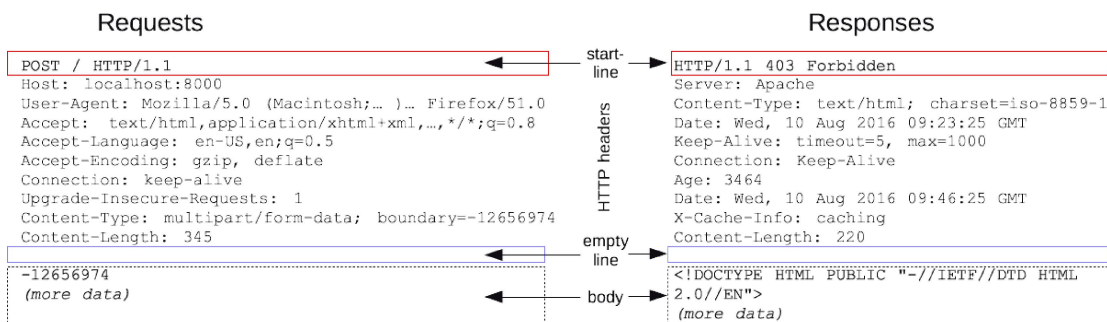


Figure 4.6 – HTTP message structure, from [Mozilla \(2020b\)](#).

As above mentioned, this protocol messages have status codes that inform the client about the result and state of the sent request, [Salvadori \(2015\)](#). The codes are a three-digit integer that can represent multiple situation. Giving that there are a several number of different codes, they are divided into five main categories of states that are listed bellow, [RFC7231 \(2014\)](#):

- **1xx** (informational): the request has been received and the process is flowing;
 - 100: Continue;
 - 101: Switching protocols.
- **2xx** (successful): the request was successfully received and accepted;
 - 200: OK;
 - 202: Accepted;
 - 203: Non-Authoritative Information.
- **3xx** (redirection): the request needs more actions to be performed in order to be successful;
 - 300: Multiple Choices;
 - 302: Found;
 - 307: Temporary Redirect.
- **4xx** (client error): request has a bad syntax or cannot be fulfilled;
 - 400: Bad Request;
 - 401: Unauthorized;
 - 403: Forbidden;
 - 404: Not Found.
- **5xx** (server error): the server failed to fulfill an apparently valid request.
 - 500: Internal Server Error;

- 501: Not Implemented;
- 503: Service Unavailable;
- 505: HTTP Version Not Supported.

4.3.5 HTTPS Protocol

HTTPS stands for Hypertext Transfer Protocol Secure and it is used to guarantee a secure connection between the two parties. That layer of security is accomplished by using the HTTP protocol together with an SSL/TLS connection. Doing that guarantees that data exchanged using this protocol is safe and secure, and only the recipient will understand the content of that data, [RFC2818 \(2000\)](#).

HTTPS provides three main characteristics:

- **Confidentiality** – ensuring that the data sent is received without any stranger watching it;
- **Integrity** – making sure that no one adulterates a message sent over the internet;
- **Authentication** – guaranteeing that the website or service in use is what it claims to be. In other words, ensures that a website that says it is *example.com* is actually *example.com*.

How the Security is Accomplished

Considering the above mentioned OSI Model, the HTTP protocol belongs to the layer 7 which is the application layer. This layer is the last of this model and is responsible to promote the interaction between the user and the machine, where the resources for the communication also belong.

In order to secure the data transmitted with the HTTP protocol over the internet, is is needed the usage of a protocol of the layer 4 (Transport layer) in the OSI model

– the TLS. TLS is now on its version 1.3 and replaces the SSL protocol used in the first versions of this technique.

In resume, HTTPS security level is achieved using the HTTP protocol over an SSL/TLS (Secure Sockets Layer/Transport Layer Security) connection, meaning that the structure of an HTTPS message is the same as an HTTP message, [Shbair et al. \(2017\)](#).

4.4 Data formats

Data needs to be exchanged between the messaging platform servers and the chatbot application. We have already seen that the messages are transmitted using an HTTP secure connection, which means that its content should be safe and secure.

Now that we know how the data is transmitted, we have to know how it is formatted in order to understand that data in question. Only by understanding the data we will be able to manipulate it and analyze it to properly answer each message that the chatbot receives.

4.4.1 XML vs JSON

Services can transmit data using a variety of different formats. Considering that, the most used data formats on the web are JSON and XML. Although they share the same purpose, they have a different composition and structure, and that can lead one of them to be preferred over the other. Next sections detail both XML and JSON formats.

XML

XML (eXtended Markup Language) purpose was to quickly and simply exchange documents through the world wide web. Although HTML is very used, it is oriented

to the presentation and not to documents' structure. Given that, XML inherited some base concepts from the SGML (Standard Generalized Markup Language), a more complex meta-language to define document annotations languages, Martins (2002). An example of this type of data format is explicit in Listing 9.

```
<note>
  <from>Jani</from>
  <to>Tove</to>
  <message>Remember me this weekend</message>
</note>
```

Listing 9: Example of a XML formatted file, from W3schools (2020).

Some of the XML advantages are:

- Makes documents transportable across systems and applications.;
- XML separates the data from HTML;
- XML simplifies platform change process.

And, its disadvantages:

- XML requires a processing application;
- The XML syntax is very similar to other alternatives 'text-based' data transmission formats which is sometimes confusing;
- No intrinsic data type support;
- The XML syntax is redundant;
- Does not allow the user to create tags.

JSON

JSON (JavaScript Object Notation) is a data-interchange format derived from the object literal of the ECMAScript programming language standard. JSON is lightweight, text-based, and language-independent, but, more important than that, it is easy for humans to read and write, making it an excellent choice for a data structure. Its objects are analyzed as string arrays, making it easy for machines to parse and generate. Listing 10 is an example of a JSON formatted object.

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated" : false,
    "IDs": [116, 943, 234, 38793]
  }
}
```

Listing 10: Example of a JSON formatted file, from [RFC7159](#) (2014)

JSON implementation brings some advantages when compared to the XML annotation:

- Provide support for all browsers;
- Easy to read and write;
- Straightforward syntax;

- Easy to create and manipulate;
- Supported by most back-end technologies;
- It allows you to transmit and serialize structured data using a network connection;
- JSON is text which can be converted to any object of Java into JSON and send this JSON to the server.

Nothing is perfect, and JSON also has some disadvantages:

- No namespace support, hence poor extensibility;
- Limited development tools support;
- It offers support for formal grammar definition.

Comparing XML and JSON

Table [4.1](#) shows the main differences between these two formats.

JSON	XML
JSON object has a type	XML data is typeless
JSON support multiple types: string, number, array and Boolean	All XML data should be string
JSON has no display capabilities	XML offers the capability to display data because it is a markup language
Retrieving value is easy	Retrieving value is difficult
Native support for object	The object has to be express by conventions
It supports only UTF-8 encoding	It supports various encoding
It does not support comments	It supports comments
JSON files are easy to read as compared to XML	XML documents are relatively more difficult to read and interpret
It does not provide namespaces support	It supports namespaces
It is less secured	It is more secure than JSON

Table 4.1 – Comparison between JSON and XML formats.

4.5 JSON Web Token (JWT)

JSON Web Token are an open industry standard method for representing claims to be transferred between two parties. Also, it is compact and URL-safe making it a good choice for internet communication, [RFC7519 \(2015\)](#).

This token-based authentication format is widely adopted in many different services and purposes due to the flexibility of the JSON data that it carries, [Alkhulaifi and El-Alfy \(2020\)](#).

JSON Web Token is composed of three fields that, in its encoded simplified way, look like the example in Listing 11.

xxxxx.yyyyyy.zzzzz

Listing 11: Example of an encoded JWT, from [Mozilla \(2020b\)](#).

After it is decoded, the JWT has a more pleasant and readable format. It consists on the following three parts, [Haekal and Eliyani \(2016\)](#):

- **Header** – is typically divided in two parts:
 - **Type** – represents the type of the token. In this case is JWT ;
 - **Hashing algorithm used** – for example, HMAC (Hash-based Message Authentication Code), SHA256 (Secure Hash Algorithm 256) or RSA (Rivest-Shamir-Adleman).
- **Payload** – which contains the claims. Claims are statements that can be related to the user to whom the token belongs to, and it can also supply supplementary metadata;
- **Signature** – this is what secures the data transmitted and what guarantees that is trustworthy. A signature is created by concatenating the header and the payload and then converting it to BASE64, adding a secret key and, last but not least, sign it with a cryptographic algorithm. It is then used to verify the message was not changed along the way and it also allows to verify if the sender of the JWT is who it says it is.

After all the three parameters are set, the JSON is Base64Url encoded and each one is concatenated with a dot separating each parameter to form the encoded JWT. The encoded JWT is easily passed in HTML and HTTP environments and is more compact than the XML-based formats, [Auth0](#).

Figure 4.7 shows an example of a JSON Web Token accessible via an official debugger online.

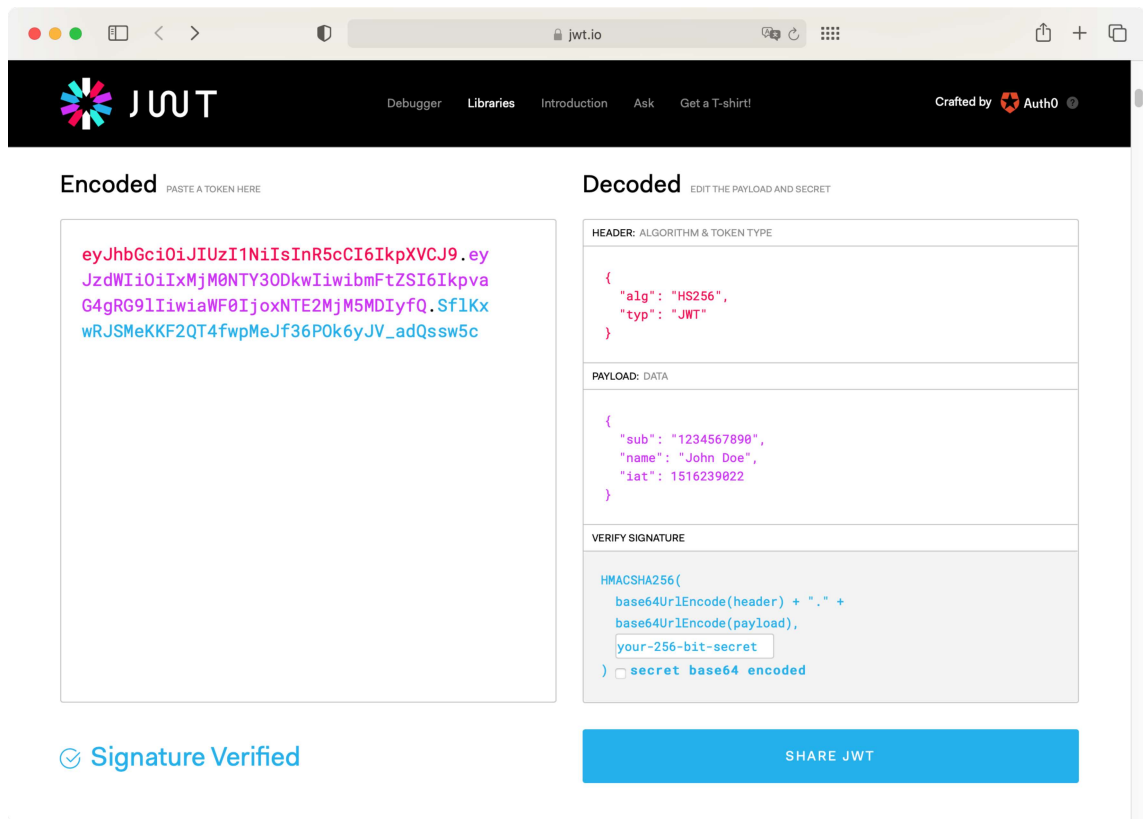


Figure 4.7 – JSON Web Token online debugger screenshot from Auth0.

This concept is very important to understand how to communicate with some of the messaging platforms used while building the Framework.



Framework Design and Implementation

This projects' goal is to create a universal solution for the development of chatbots for multiple messaging platforms, so that one chatbot application can work with more than one messaging platform at the same time. This chapter explains all the steps taken during the development process of the Framework. The next sections of this chapter also detail the components and technologies in use during the development process.

As we become more and more dependent on technology and in being connected with each other, the use of messaging applications and services related with communications has increased, [MindSea \(2020\)](#). Knowing that most of the time a regular user spends on its smartphone is talking with friends and family inside a chat APP, companies started to wonder if using those messaging APPs could make their business more profitable, [Lin \(2020\)](#). With that mindset, chatbots popularity has been growing over time, and the interest to invest in this solution has started to grow among companies. Its applications are limitless, [EQUITY \(2019\)](#). Nevertheless, one big problem still insists not to disappear: the multiple applications and services people use worldwide to communicate with each other. Companies want to reach the most significant amount of people, but spend less money possible. That is how business work, and if there are multiple paths to reach the same destination, the best will be

the one that saves more time and, consequently, money. This Framework aims to help those interested in producing a chatbot for multiple messaging platforms using Java technology.

That is possible by presenting a solution to create and maintain a chatbot to establish communication with multiple messaging services simultaneously. One code for multiple platforms is the goal that every chatbot developer wants to work with.

For that matter, this dissertation's primary focus is developing a framework that can provide the developer the necessary tools to build and maintain a chatbot for multiple messaging services and applications without having to code individual solutions for each one. Also, it should provide the tools needed to add support for messaging platforms, other than those implemented as a case study.

5.1 Used Technologies

During the process of development of the Framework, different technologies were chosen for its implementation. This section makes a brief resume and presentation of the various technologies applied to the framework.

5.1.1 Development Platform

As a case of study, the Framework development is based on the Java Enterprise Edition and Java Standard Edition. Java was introduced in 1995 by Sun Microsystems and has been evolving since then. It has been an enormous success and has become one of the most used programming languages in the entire world, [TIOBE \(2021\)](#). Its technology is a combination of the Java programming language and a Java platform. Java programming language consists on four platforms:

- Java Platform, Standard Edition (Java SE);
- Java Platform, Enterprise Edition (Java EE);

- Java Platform, Micro Edition (Java ME);
- JavaFX.

All Java platforms consist of two main components that make its applications run on any compatible system and being platform-independent, stable, easy to develop and secure [Oracle \(2012\)](#):

- a Java Virtual Machine (JVM): a program for a particular hardware and software that runs Java technology applications;
- an Application Programming Interface (API): a collection of software components that can be used to create other software components or applications.

Let us take a closer look on Java Standard Edition and Java Enterprise Edition, giving that are the both used in the development of the Framework.

Java Standard Edition

This platform is the base of all the other Java platforms, and the technology itself. The core of the Java programming language lays on the Java SE API, once it defines everything this languages provides, from basic types and objects, to high-level classes that allow networking, security, database access and more.

More than that, Java SE has the most common tools and class libraries used in Java applications.

Java Enterprise Edition

Java Enterprise Edition is widely adopted and is developed with the contributions of individuals, industry experts, commercials and even open source organizations, [Oracle \(2021\)](#).

This platform is built on top of the Java EE platform and provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable and secure network applications.

Giving that it allows to build reliable and secure network applications, it makes it a perfect base to build our chatbot framework.

Besides that, the platform is constantly evolving trying to meet the industry needs and requests, making it an easy solution for a variety of different purposes. Also, allows the use of multiple design patterns that make the code reusable and more efficient, [Liu and Chen \(2009\)](#).

The last version, Java EE 8, release in 2017, brought huge improvements such as an updated Java Servlet API with HTTP/2 support, enhanced JSON support including a new JSON binding API, a new REST Reactive Client API, a new portable Security API, and a lot more, [Oracle \(2021\)](#).

The Framework core runs all over the Java Servlet API and this piece of the Java EE platform is one of the most important in this dissertation. The Server is detailed in the next section, together with the technologies that make it up.

5.1.2 Server

The framework requires an application server to be functional, in order to establish a connection with the servers of the messaging services implemented. Java supports various web servers, however the framework uses the Apache Tomcat server which is an open source implementation of some Java EE APIs.

Giving that it implements the Java EE APIs, it allows the usage of the Java's servlet technology.

Java Servlet Technology

Servlet is a class of the Java EE platform that extends the capabilities of servers that host applications that implement a request-response protocol, as HTTP. When using HTTP protocol, the Java Servlet technology defines specific servlet classes with methods such as `doGet` and `doPost`, in order to handle GET and POST requests, respectively.

The technology runs on a Java web server or an application server providing services of request and response on the web. The web server can load servlets automatically on server startup or when a client performs a request for the first time. The process is described in Figure 5.1, Ferreira (2008).

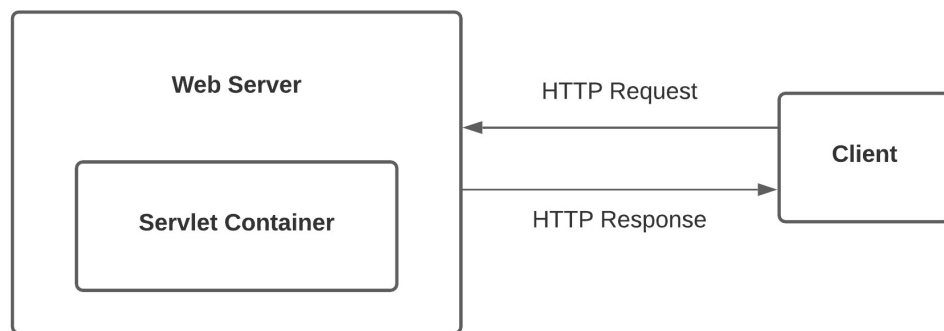


Figure 5.1 – Java Servlet API diagram, adapted from Oracle (2014).

The Framework needs to make use the servlet methods to handle the GET and POST requests from each messaging platform. From the concept of servlet, we realise that one servlet has to be unique, which means that each connection to a messaging platform server requires its unique servlet.

The servlet can be accessed when an HTTP request is performed to the server. For that, the messaging platforms need to have defined the callback URL that will allow the usage of webhook concept.

Webhooks

A webhook is a transmission method that is used to send real-time events to external listeners. Webhooks enable external applications to subscribe to events and receive their updates when they occur. When a change occurs in the subscribed service, an HTTP request is sent to the specified callback URL, [Nafis and Setiawan \(2019\)](#).

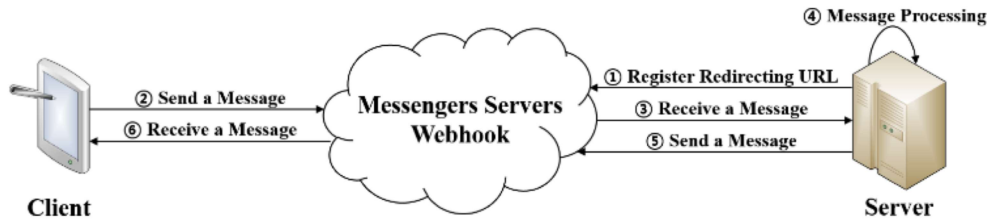


Figure 5.2 – Working process of a webhook.

Figure 5.2 illustrates how an webhook works in a messaging service. First of all, for the system to work, the server needs to register its URL in the Messengers Server, from where it wants to receive updates (1). Then, when the client sends a message it is first sent to the Messengers Server (2). That server will then use a webhook to redirect the message received to the message processing server (3). In there, the Server will interpret the content of the message received (4), and send a reply back to the messaging service server (5) that, in the end, will send it back to the correspondent client (6).

Our Framework will implement this concept giving that all the messaging service platforms selected for this project make use of it, [Lee et al. \(2020\)](#).

5.2 Requirements

Following the Waterfall methodology, every software development project's first stage should start with a set of requirements. This stage is where all the final results' requirements have to be explicitly defined.

We can divide the requirements into two categories: functional and non-functional requirements. The functional requirements are related to specific features of the software, for example, receive text messages, and help define what the software has to do, whereas the non-functional requirements define how the software will do what it has to do. An example of a non-functional requirement is the minimum number of supported chatbots being two.

Applying the concept to the real scenario, we can define the functional requirements to be:

- Receive and send messages to users;
- The chatbot actions are programmed by who is implementing the framework;
- Have multiple chatbots and multiple services simultaneously;
- Have a way to add new messaging services not yet supported by the framework.

The non-functional requirements are:

- Developed in Java programming language.

5.3 Architecture

The Framework needs to be developed with only one thing in mind: be flexible and universal, i.e., the developer should be able to choose which platforms the chatbot should work on, access its messages and process its replies. It is equally important to be easy to use and implement, so anyone that has the needed software development skills, and is interested in building chatbots can use it. Therefore, in this section, the concepts behind the Framework are explained and clarified.

Before getting our hands into the Framework itself, is important to understand how the messages travel from the client that uses a messaging service to have a

conversation with the chatbot, to the chatbot application server that will process it and send a response back. Figure 5.3 shows an Unified Modeling Language diagram to better understand this process.

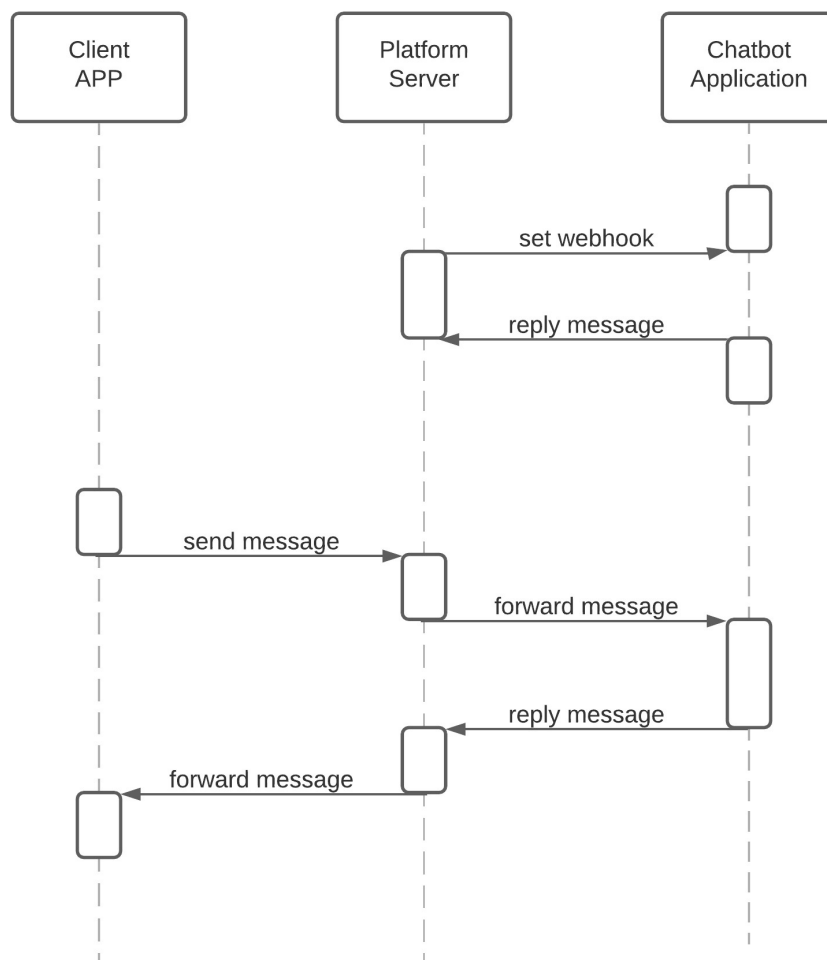


Figure 5.3 – Factory Design Pattern UML diagram implementation.

In the above figure, the platform server sends an HTTP request to the chatbot application to verify the callback URL (1). Then, the chatbot application send a reply back with a token, in order for the platform to verify its integrity (2). This first two steps, depending on the messaging platform may be in a different order. After the callback URL for the webhook events is correctly set, the chatbot is ready to work. When client using an APP of the messaging service sends a message to

the chatbot. First, that message is sent to the messaging platform server (3) that will redirect it to the chatbot application server (4). The application processes the message and sends a reply back to the platform server (5) that will then redirect it to the client (6).

Knowing that base concept of how chatbot applications communicate with each client or user, we can now start building the framework.

Figure 5.4 illustrates the main diagram of the structure of the Framework.

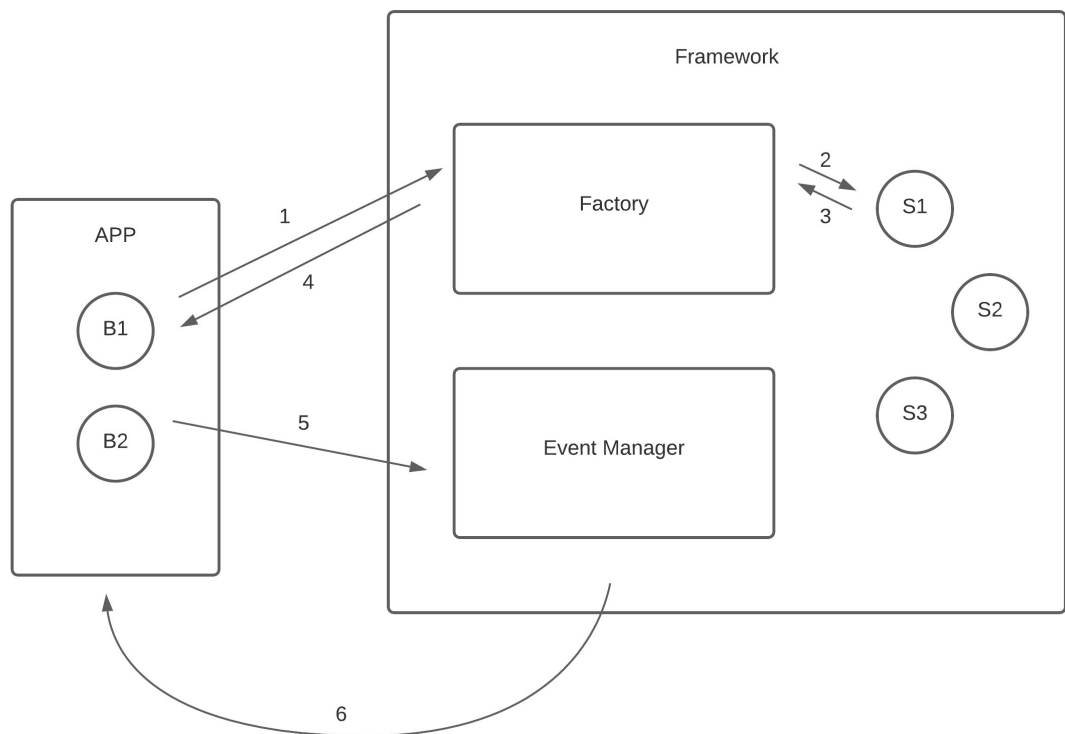


Figure 5.4 – General concept of the framework developed

Explaining the concept illustrated in Figure 5.4, when the application starts, it requests the Factory to create and return a specific service representation (1). The Factory will then instantiate (2/3) and return the required service object (4) that will be responsible to manage the connection between the application and the messaging service platform. After returning the object to the application, the event manager will use it to know how many messaging services the chatbot will subscribe to

(5). Last, the event manager is also responsible to notify each chatbot that a new incoming message arrived. The application will be responsible to analyze and reply to each message received.

5.3.1 Framework Components

Giving that the Framework was developed in Java programming language, and its usage requires a Java application, it is essential to understand how the application have to be built and also, how it will work together with the Framework in order to allow the chatbot service to work on multiple platforms. Nevertheless, the Framework will be built in a way that the developer using it will not have to worry about how the Framework works, but only on how to use it.

With that in mind, the framework will be divided into two modules, and a third one that consists on the application implementing the framework.

- **Factory** — as its name implies, this module is responsible for generating a new chatbot and return it to the developer. A creational design pattern is used to achieve this result. Inside this subcategory of software design patterns reside the Factory Design Pattern detailed back in [Chapter 4](#).
- **Event Manager** — this module will be the brain of the Framework. This block of the framework has to handle the redirects of the request received. In other words, every time a chatbot receives an HTTP request with a message, the event manager has to notify the chatbot that it has received a message and forward it to the respective chatbot. Then, the chatbot can process the message and send a response back to the user.
- **Application** — is the program built by the developer implementing the Framework to create a chatbot on multiple platforms. It represents the program that the developer will code to build a chatbot using the Framework. In this dissertation's case, an application was developed to simulate a real-life usage of the framework. Also, it was used to test the Framework in [Chapter 6](#).

5.3.2 Message and User Objects

As seen in Chapter 3, there is no such thing as a standard or universal chatbot message structure. Every provider applies a different message and payload structure to its chatbot service. The payload is all the content of interest sent in one HTTP message. Inside the payload is the actual message being exchanged between a user and the chatbot server that can be text, images, audios, etc. Besides the message itself, there is more information, namely the user name, user ID, date when the message was sent, and other parameters of interest. Giving that those values change from platform to platform, in order to make the Framework compatible with a wide variety of messaging platforms, the solution found was to create a Message object that will be common to every chatbot. Table 5.1 has a full representation of all the parameters that make up the Message object.

Name	Type	Description
id	String	Message unique identifier.
chatId	String	Chat unique identifier.
date	String	Date the message was sent.
text	String	Text sent by the user.
messagingType	String	Type of message.
isEcho	boolean	Flag to mark if a message is an echo of a previous message. Required for Facebook Messenger.
user	Object (User)	User object that represents an actual user. This object has multiple parameters needed to represent the user.

Table 5.1 – Definition of a Message object.

Each service connector is responsible for converting the framework Message object and its platform's payload.

Exactly as the Message object, a User object has to be created to specify the user parameters. The User object is represented in Table 5.2.

Name	Type	Description
userId	String	User unique identifier.
firstName	String	User first name.
lastName	String	User last name.
username	String	User unique username.
isBot	boolean	Flag that represents if the user sending a message is a bot or not.

Table 5.2 – Definition of a User object.

5.4 Developed System

The developed system is a combination of the Framework that any developer will be able to implement, and a Java application that makes use of that same Framework. This last one was implemented as a case of study to test the Framework.

To better explain how the Framework was built and the pillars that make it up, as well as to provide a good description on the application that uses the Framework, the two of them were separated in the next two sections. In the Section 5.4.1 is described how the Framework was implemented in an high level code design, and in the Section 5.4.2 is made a detailed explanation on how the Framework can be used to build a chatbot application.

5.4.1 Framework

As described in Section 5.3, the Framework is build over two main blocks: the Factory and the Event Manager. The application refereed in the same section is later detailed in Section 5.4.2 once it is meant only to represent the application that each developer will build and that will implement the Framework in question.

Creating a Working Server

Knowing that the chatbot communicates via HTTP, we first need to create a web server to allow our server to receive and send messages to the messaging platforms servers. To make it easier for the developer implementing the Framework, it already implements a server that can be managed by the application implementing it. Like that, the developer is responsible for creating, initializing, and setting the server to a await state – state where the server is waiting for incoming requests.

Later, when creating the service connector, the server needs to be sent as parameter to its creator method, in order to create a bridge between the chatbots class and the servlet that manages the communication with its messaging platform. A simple implementation of the server usage is presented in Listing 12.

```
1 public static void main(String[] args) {  
2     // Creating a server  
3     Server server = new Server();  
4  
5     /*  
6         Chatbot definition and initialization goes here  
7     */  
8  
9     // Initializing the server  
10    try {  
11        server.init();  
12    } catch (LifecycleException e) {  
13        e.printStackTrace();  
14    }  
15  
16    // Setting the server to await state  
17    server.await();  
18 }
```

Listing 12: Usage example of the Server class.

After the server is up and running, a servlet is responsible for receiving and sending messages to those trying to communicate with the chatbot. The communication is made using HTTPS protocol set by default on port 443. The port for the communication can be changed by the developer before calling the *init* method of the server class.

This design shows a problem: the framework has a servlet inside, accountable for receiving the user's answer, and the developer does not have a way to access to that answer. The solution for this problem will later be solved by implementing the event manager that will notify every chatbot of new incoming messages. But, before that, we need a way to connect with each messaging service.

Generating the Services

Since the main purpose of this peice of software is to be as more universal and flexible as possible, i.e. be compatible with the greatest number of messaging services possible, the code has to be designed from the beginning with the mindset that the Framework needs to be compatible with the majority of the messaging platforms available at the moment. Not only that, but also the Framework should be able to implement new messaging services to come in the future.

With that in mind, the Framework needs a way to connect to each different platform of interest. Moreover, a method to insert new messaging services in the framework, so that, for example, a company could expand its business to another messaging service that is not yet implemented in the framework and re-utilise the code already made.

A Factory Design Pattern, detailed in Section 4.2.1, is a creational design pattern used to create objects in a subclass from a common interface. Those sub-classes should be able to change the type of objects that are meant to be created, [Mu and Jiang \(2011\)](#).

Back in Section 4.2.1, the Factory Design Pattern was explained with an UML diagram. In that diagram was mentioned the concept of Product and ConcreteProduct. The same concept will be adopted to the actual case.

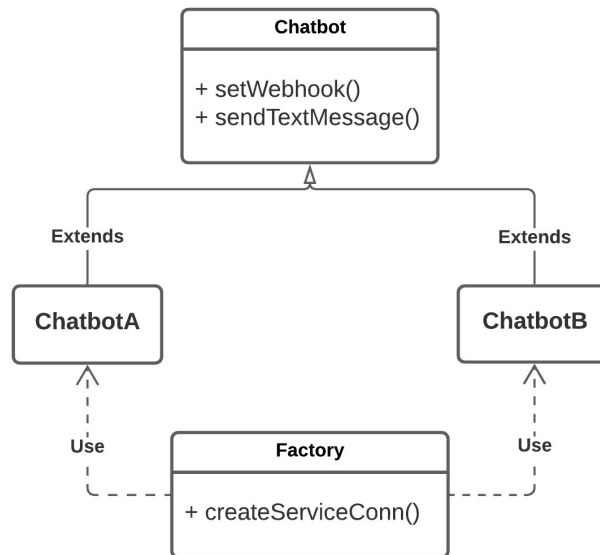


Figure 5.5 – Factory Design Pattern UML diagram implementation.

Figure 5.5 illustrates how the Factory method can be implemented in the framework. In there, the **Chatbot** class represents the Product. That is an abstract class that defines the overall behaviour of every chatbot, and it can be instantiated when needed. It also implements the GET and POST requests required to communicate with the messaging platforms. **ChatbotA** and **ChatbotB** represent two different specific services. Each chatbot has to extend the abstract class Chatbot. Last but not least, the **Factory** class is the one responsible to instantiate the chatbots in question. It will return a specific object of the type Bot.

With that said, using the Factory is easy as creating an object and using a special method to create a chatbot. The method to create the chatbot connector takes the following values as input parameters:

- **name**: The name of the chatbot. Used only to recognize the chatbot;
- **id**: The ID of the chatbot from the messaging platform;

- **token**: The token of the chatbot from the messaging platform;
- **webhookUrl**: The callback URL to where the chatbot needs to address the messages.

An example of usage of the Factory class is shown in the Listing 13.

```
1 Factory factory = new Factory();  
2  
3 Chatbot example = factory.createChatbot("Messenger",  
4 "gq8jikd7bevjcs3z", "3ASgqKoKo3Fpam3Whgg0Nj74shCq4e",  
5 "https://dd21e31df7c2.ngrok.io");
```

Listing 13: Example of an implementation of the Factory method to create a chatbot using the framework.

At this stage, there is already a way of generating multiple service connectors, so we can now implement a way to analyze and reply to messages received from the messaging platform. To do that, we have to create a class to perform that analysis.

This implementation requires a class to provide to the framework the ability to have multiple ‘listeners’ for the same chatbot. In other words, it allows to have more than one chatbot replying for the same messaging service.

But, now there is still a problem to solve: a way to notify each chatbot that a new incoming message arrived.

Notifying Every Listener

The concept behind this idea is simple: the developer implementing the Framework has to create a class that will handle and process every message sent to its chatbot, and that class has to subscribe to a service of notifications. The Framework will notify that class when a new message arrives and sends the necessary data to it.

After that, it is up to the developer to choose what the chatbot will do with that information.

This pattern is described as the Observer Design Pattern and was detailed in Section 4.2.2. Now, Figure 5.6 shows how the Framework can apply the Observer Design Pattern to the specific use case.

Even though this pattern is more complex than the Factory Design Pattern, a UML diagram such as the one shown in Figure 5.6 makes it a lot easier to understand and implement.

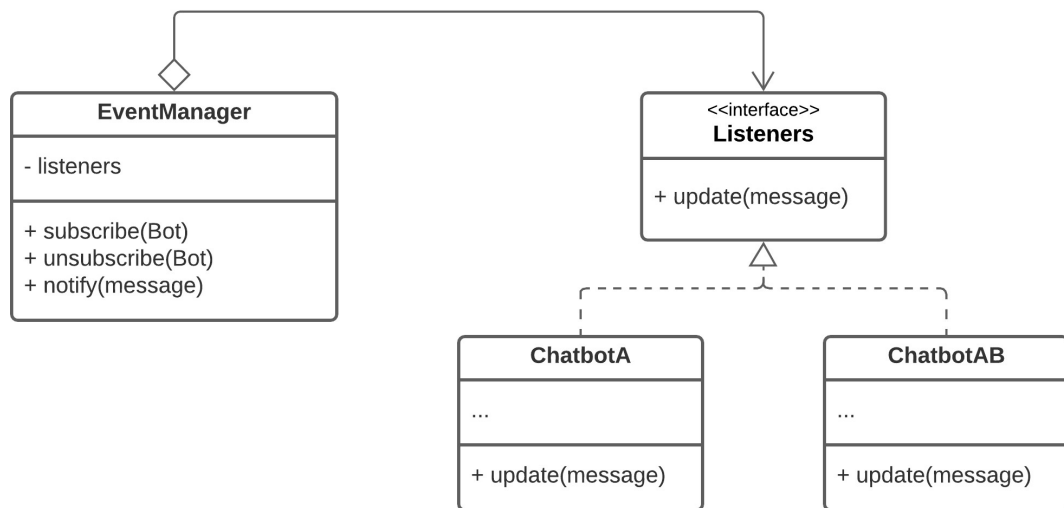


Figure 5.6 – Observer Design Pattern UML implementation.

By observing Figure 5.6, we can see the three main components that make up this design pattern. The first one, the **EventManager**, is responsible, as its name says, to manage every ‘listener’ for a specific chatbot. The **subscribe** method allows a listener to subscribe to that chatbot, and the **unsubscribe** method does the opposite, removes a listener from the chatbot listeners list. For that matter, both subscribe and unsubscribe methods take a Bot object as a parameter.

To subscribe to the EventManager, the listener created by the developer has to implement the **Bot** interface, represented in the same image. Then, each listener

has to have an **update** method. This method will be called when a new message arrives at the chatbot and is responsible for processing the information and sending a reply back to user.

Finally, the notify method present in the EventManager is where the magic happens and will be responsible for notifying every registered listener by calling the update method.

At this point, we already have a way to generate connectors for multiple services and, at the same time, a way to notify all chatbot listeners. Now that we all the concepts of the framework are clarified, we can proceed to an explanation of a general Java application to implement the Framework.

5.4.2 Application

A simple proof of concept and testing application was built to implement a chatbot for three distinct messaging platforms – two already supported by the Framework and a new one not yet supported. Besides that, to prove that the Framework is functional and working correctly, the only thing that this application needs to do is echo back the user's message. By doing that it is proved that the Framework can establish a communication with the messaging platform server and send/receive messages.

After showing that the application can communicate with the user, the next thing to prove is that it can be used to have a conversation with multiple messaging services and even have numerous chatbots to the same platform. Finally, the application was able to add to the framework, in run-time, support for a messaging platform that is not supported by default by the Framework.

The first thing to do in our application is to define the URL of the server where the application is running in order to allow a communication between each messaging platform server and our application server. Other parameters required to establish a connection with each messaging platform server also need to be set.

Summarizing, each service needs defined the following parameters:

- ID - defines the chatbot ID for the specific messaging platform;
- Token - defines the token for the communication with the specific messaging platform;
- URL - identifies a unique service inside the server.

Knowing that, Listing 14 shows an example of the parameters definition in Java code. All the values are only an example and were randomly generated.

```
1  // Server URL
2  String url = "https://dd21e31df7c2.ngrok.io";
3
4  // Facebook Messenger parameters
5  String messengerId = "gq8jikd7bevics3z";
6  String messengerToken = "3ASgqKoKo3Fpam3Whgg0Nj74shCq4e";
7  String messengerUrl = url+"/messenger";
8
9  // Telegram parameters
10 String telegramId = "5730906289";
11 String telegramToken = "NT3RmP2f4FG57tgtZV2MQes2NNQcsD";
12 String telegramUrl = url+"/telegram";
```

Listing 14: Defining basic required parameters for the Framework to work.

The process to acquire these two parameters depends on each platform and can be found in Chapter 3 in a simplified way or inside its documentation for a more detailed version.

Now that every essential parameter is well defined, the next logical step is to create a server that will allow the communication with all the platforms. For that, we created a Server object based on a class provided by the Framework, as seen in Listing 15.

```
1 // Web Server
2 Server server = new Server();
```

Listing 15: Creating a Server object.

We also need a factory an object responsible to establish a connection to the platform server. In this case, the factory has to make two service connectors: one for Telegram and another for Facebook Messenger. Later in this section, our Chatbot object will be created and make use of these two classes. Listing 16 shows an example of how to create a factory class.

```
1 // Services Factory
2 Factory factory = new Factory(server);
```

Listing 16: Creating a Factory object.

The Factory class has a method defined to return a Chatbot object. It is also responsible for saving a list of all the connectors created and in use. The code in Listing 17 is used to create a connector for the two messaging platforms to be used.

```
1 // Creating a Telegram service connector
2 ChatbotConn telegram = serviceFactory.createConnector("telegram",
3     "Telegram", telegramId, telegramToken, telegramUrl);
4
5 // Creating a Facebook Messenger service connector
6 ChatbotConn messenger = serviceFactory.createConnector("messenger",
7     "Messenger", messengerId, messengerToken, messengerUrl);
```

Listing 17: Creating the connector for each service.

At this stage, *telegram* and *messenger* objects are capable of establishing a connection with the respective platforms and, consequently, receive messages from the respective servers. However, they still do not have specified method to reply to

the messages they receive. For that, we created a class that implements the Bot interface, subscribes to the event manager and overrides the update method that is called every time there is a new incoming message.

An example of a chatbot class implementation is seen in Listing 18 where is made an override of the update method to reply to the messages received with the same text received.

```
1 public class TelegramChatbot implements Bot {
2
3     private final Telegram telegram;
4
5     // Class constructor
6     public TelegramChatbot(ChatbotConn telegram) {
7         this.telegram = (Telegram) telegram;
8         this.telegram.getEventManager().subscribe("telegram", this);
9     }
10
11     // Method to handle the received messages
12     @Override
13     public void update(String service, Message message) {
14         String response = "{\"chat_id\":\"813501482\", \"text\":\""
15             +message.getText()+"\"}";
16         telegram.sendTextMessage(response);
17     }
18 }
```

Listing 18: Example of a chatbot object implementing the Bot interface.

Now that we already have the class the handles messages received by that chatbot, we have to subscribe to the event manager in order to be notified by the framework when a new message arrives in our server.

The Event Manager is what manages all the chatbots and notifies everyone when a new incoming message arrives. For that reason, an EventManager object has to be

created and, depending on the chatbot implementation, adding the service connector to the manager. Following that, the service also has to be added to each chatbot for the event manager to work.

Listing 19 shows a chatbot that can handle two messaging services at the same time.

```
1  // Creating an event manager
2  EventManager manager = new EventManager();
3
4  // Adding the services to the event manager
5  manager.addService(telegram);
6  manager.addService(messenger);
7
8  // Creating a chatbot that works with two messaging services
9  MultipleBot multiple = new MultipleBot();
10 multiple.addChatbot("telegram", telegram);
11 multiple.addChatbot("messenger", messenger);
```

Listing 19: Adding the services to the Event Manager and creating the chatbot object.

With all the code done, our application implementing the framework is able to run for the two messaging services (Facebook Messenger and Telegram) and is also able to expand for new messaging services in the future. To do that, we only need to create another class that implements the GET and POST requests for the required messaging service, and a class that actually represents the chatbot, like the class shown in Listing 18.

Adding an Unsupported Service

The application implementing the Framework also adds support for a messaging service that is not supported by the Framework, and as been mentioned along this document, WeChat service was the chosen to be implemented.

The first thing to code is the class responsible to establish the communication between the application and the messaging service's server. This class, has to extend the Chatbot class from the framework. By extending the Chatbot class, our class will implement the servlet API from Java Enterprise Edition. Giving that this class will handle the HTTP GET and POST requests, it has to override the **doGet** and the **doPost** methods from the servlet API. Listing 20 shows the constructor of this class together with the main required parameters.

```
1 public class WeChat extends Chatbot {
2     private final String token, webhookurl, messageRequestUrl,
3     appId, name;
4     private static final String wechatUrlMessages =
5     "https://api.weixin.qq.com/cgi-bin/message/custom/
6     send?access_token=%s";
7     private String payload = "";
8     private Message message;
9     private String appSecret;
10
11     public void setAppSecret(String appSecret) {
12         this.appSecret = appSecret;
13     }
14
15     public WeChat(String name, String id, String token,
16     String webhookUrl) {
17         super();
18         this.appId = id;
19         this.token = token;
20         this.name = name;
21         this.webhookurl = webhookUrl;
22     }
23 }
```

Listing 20: WeChat class constructor.

This class has to override four main methods: `doPost`, `doGet`, `initialise`, and `send-TextMessage`. Starting with the **`doPost`** method. The platform sends all the messages in a POST request to the chatbot application, and for that reason, the **`doPost`** method will handle all the messages received from the users. It creates a buffer to save all the content in the HTTP payload, and extracts the content of the XML formatted payload to a `Message` object. This is also the method responsible to notify every listener of this connector, and send it the message received.

```
1  @Override
2  protected void doPost(HttpServletRequest req,
3  HttpServletResponse resp) throws ServletException, IOException {
4
5      req.setCharacterEncoding("UTF-8");
6      StringBuffer stringBuffer = new StringBuffer();
7      String line = null;
8
9      try {
10         BufferedReader reader = req.getReader();
11         while ((line = reader.readLine()) != null) {
12             stringBuffer.append(line);
13         }
14         this.payload = stringBuffer.toString();
15
16         this.message = xmlToMessage(this.payload);
17         events.notify("wechat", this.message);
18
19     } catch (Exception e) {
20         System.err.println("ERROR: " + e);
21     }
22 }
```

Listing 21: Override of the `doPost` method.

Listing 22 shows a simple implementation the GET request. Based on WeChat API,

this method separates the parameters needed to verify the callback URL when it is submitted in the WeChat platform.

```
1  @Override
2  protected void doGet(HttpServletRequest req,
3  HttpServletResponse resp) throws ServletException, IOException {
4
5      Enumeration<String> headerNames = req.getHeaderNames();
6      while(headerNames.hasMoreElements()) {
7          String headerName = headerNames.nextElement();
8      }
9
10     Enumeration<String> params = req.getParameterNames();
11     while(params.hasMoreElements()){
12         String paramName = params.nextElement();
13     }
14
15     String signature = req.getParameter("signature");
16     String timestamp = req.getParameter("timestamp");
17     String nonce = req.getParameter("nonce");
18     String echostr = req.getParameter("echostr");
19     String[] parameters = {token,timestamp,nonce};
20
21     Arrays.sort(parameters);
22     StringJoiner joiner = new StringJoiner(",","","");
23     for(int i =0; i<parameters.length; i++){
24         joiner.add(parameters[i]);
25     }
26
27     String sha1params = DigestUtils.sha1Hex(String.valueOf(joiner));
28     resp.setStatus(HttpServletResponse.SC_OK);
29     resp.getWriter().write(echostr);
30 }
```

Listing 22: Override of the doGet method.

Another method from the Chatbot abstract class that also requires the application to Override it is the **initialise**. This method adds the this service connector to the Framework server and sets itself as a servlet to handle the HTTP requests.

```
1  @Override
2  public void initialise(Server server) {
3      this.messageRequestUrl = String.format(wechatUrlMessages,
4      getAccessToken());
5      if(!this.getEventManager().getListenerMap()
6      .containsKey("wechat")){
7          server.addChatbot(name,this,webhookUrl);
8      }
9  }
```

Listing 23: Adding an unsupported messaging service to the Framework.

The fourth but not less important method, the `sendTextMessage`, is responsible to send messages to the server. It uses a method that returns the response from the server after it receives the messages the application sent.

```
1  public void sendTextMessage(String msg){
2      String response = send(this.messageRequestUrl, msg);
3      System.out.println(">sendTextMessage: " + response);
4  }
```

Listing 24: Implementation of a simple `sendTextMessage` method.

Finally, the class that really implements the chatbot itself, responsible to process the message received from the messaging service and to send a response back. This class has to implement the Bot interface and override the update method.

```
1 public class WeChatBot implements Bot {
2
3     private final WeChat wechat;
4
5     public WeChatBot(Chatbot wechat){
6         this.wechat =(WeChat) wechat;
7         this.wechat.getEventManager().subscribe("wechat", this);
8     }
9
10    @Override
11    public void update(Message message) {
12        String response = "{\n" +
13            "    \"touser\": \"" + message.getUser().getUserId()
14            + "\",\n" +
15            "    \"msgtype\": \"" + message.getMessagingType()
16            + "\",\n" +
17            "    \"text\":\n" +
18            "        {\n" +
19            "            \"content\": \"" + message.getText()
20            + "\"\n" +
21            "        }\n" +
22            "    }";
23        wechat.sendTextMessage(response);
24    }
25 }
```

Listing 25: Adding an unsupported messaging service to the Framework.



Tests & results

Testing software is a crucial step to guarantee its success. There are always missing pieces and there are some details that were not properly planned during development. For that reason, this chapter details all the tests performed on the Framework and the application aimed at this work. Achieving good test results is a consequence of multiple tests that can lead to the improvement of the work already done. However, knowing that, in good practice, tests performed to a software should never be done by whoever developed that same software, and even though there are several software testing techniques, like Unit tests, Integration tests, etc. the ones performed to the framework are pretty simple and just prove that it works. With that said, the Java application created to test the Framework was coded in five different ways in order to create multiple scenarios:

- Only one chatbot working with a single service;
- Two chatbots working with a single service;
- Two services working with a chatbot each;
- One single chatbot working with two distinct services;
- Adding support for a new messaging service not included in the Framework.

Each stage requires testing the Framework to send and receive text messages from users over the two different messaging services included in the Framework. The last stage aims to prove that the Framework is future proof and new platforms can be added to the Framework.

Every chatbot was programmed to echo back the same message received from the user. This way guarantees that the message is received, processed, and then an answer is sent back to the user.

The developer coding the chatbot that is implementing the Framework is responsible for processing the user's message. The result of that processing stage will then define the reply message that the user will receive.

6.1 Putting Everything Online

Before starting any test, an essential step is to provide a way to establish a connection between the application hosting the chatbot and the messaging platform servers. Like said in previous chapters, the application needs a web server to communicate with the users, and, consequently, an open connection to the machine running the application.

In this tests, giving that a personal computer was used to run the application, we found that a good solution to allow a communication with outside the network was using an third-party service called ngrok.

6.1.1 Ngrok

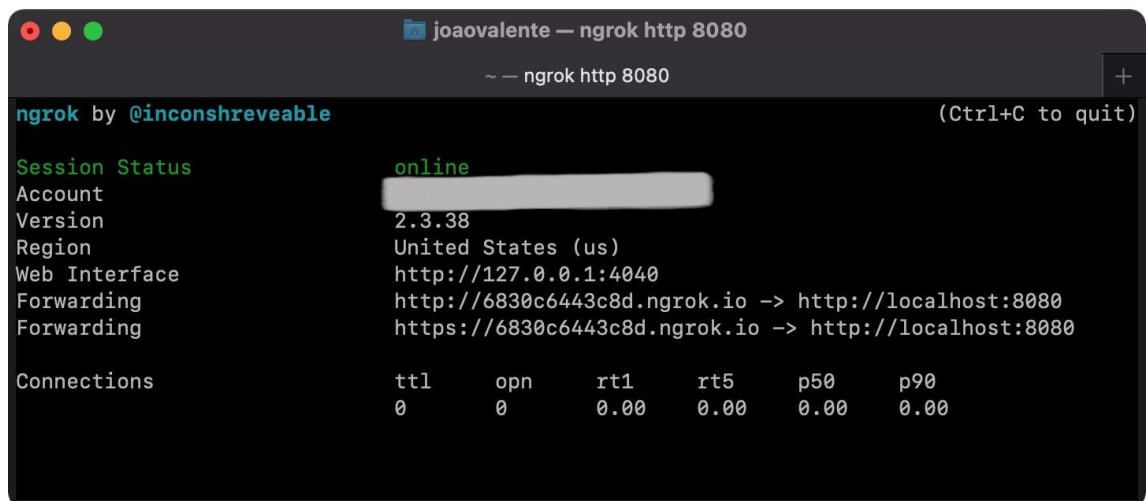
Ngrok is a solution that generates public URLs helping to expose our local server to the world. It is an easy to use tool that can help test our chatbot without spending unnecessary time configuring a server to be accessed outside the local network [Ngrok \(2020\)](#).

Using this service is simple and easy. We only need to open the terminal and use the command shown in Listing 26. The software will then automatically generate and generate both HTTP and HTTPS URLs.

```
./ngrok http 8080
```

Listing 26: Command used in terminal to start ngrok service.

The command in Listing 26 specifies the HTTP protocol together with the port 8080. That port can be changed depending on user preferences.



```
ngrok by @inconshreveable (Ctrl+C to quit)

Session Status      online
Account             [REDACTED]
Version             2.3.38
Region              United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          http://6830c6443c8d.ngrok.io -> http://localhost:8080
Forwarding          https://6830c6443c8d.ngrok.io -> http://localhost:8080

Connections
  ttl    opn    rt1    rt5    p50    p90
    0     0    0.00   0.00   0.00   0.00
```

Figure 6.1 – ngrok running on macOS terminal

Figure 6.1 shows the result of the command shown in Listing 26 written in the terminal. The URL used to configure the chatbot was the HTTPS version, not only because it is more secure and safe, but also because every platform demands that version of the protocol: *https://6830c6443c8d.ngrok.io*. Considering that the testing process took more than one day, this dissertation is likely to have different URLs because when shutting down the computer and starting the service again, Ngrok always provides a different URL.

Having the chatbot application accessible outside the personal network where it is running, we are now ready to go through every framework test.

6.2 Test Scenarios

The test scenarios' choice depends on the practical use cases a developer would encounter when implementing the chatbot development framework. At the beginning of the current chapter were presented the five scenarios to consider during the development stage.

Before starting detailing the tests and its results, a quick note about the webhook service and the setting of the callback URLs. The messaging platforms need the callback URL for the webhook service, in order to send the messages and events to our chatbot application. All tests scenarios bellow consider the process of setting the callback URL detailed in Chapter 3.

6.2.1 Scenario 1: Only one chatbot working with a single service

This experiment represents the Framework's basic usage when a developer simply wants to make the chatbot available in only one messaging service. In this case, the Framework's use may not make much sense because it can be accomplished without its implementation. Figure 6.2 shows a diagram with this implementation.



Figure 6.2 – Example diagram of one single chatbot working with only one service.

We can contradict the idea that the Framework does not make sense in this case scenario if we think about software development. Most companies want their software to be able to evolve and change in time. In other words, the software should adapt in the future and follow the technological evolution. For that reason, by implementing the Framework to build a chatbot to work in just one messaging service, it is

guaranteed that in the future, it can work in more services if the developer decides to expand the chatbot support.

Listing 27 shows an example code used to perform this test. Although this code is used to create a chatbot for Facebook Messenger, with a few changes on the parameters and class names, a similar code can create a chatbot for Telegram.

```
1 String messengerId = "e52f127617624ac692126b0220e86177";
2 String messengerToken = "EAAB53C6WkbEBAB07Gkvm5RVRL7ZB";
3 String messengerAccessToken = "EAAB53C6WkbEBANXZCftda9ZBcb5wS1";
4 String messengerUrl = "https://6830c6443c8d.ngrok.io/messenger";
5
6 //Server, Factory and EventManager init
7 Server server = new Server();
8 ServiceFactory serviceFactory = new ServiceFactory(server);
9 EventManager manager = new EventManager();
10
11 // Creating the connector for Facebook Messenger
12 Chatbot messenger = serviceFactory.createConnector("messenger",
13 "Messenger", messengerId,messengerToken,messengerUrl);
14
15 // Adding messenger and telegram services to the event manager
16 manager.addService(messenger);
17
18 // Server init and setting to await state
19 try {
20     server.init();
21 } catch (LifecycleException e) {
22     e.printStackTrace();
23 }
24
25 server.await();
```

Listing 27: Command used in terminal to start ngrok service.

Following the above code, this test was performed individually on Facebook Messenger and Telegram services to prove that it works to more than one alternative. With that said, sections 6.2.1 and 6.2.1 show the needed steps performed to implement the chatbot into the Framework.

Facebook Messenger

The first test performed on the Framework used an application implementing a chatbot to work only with Facebook Messenger messaging service shown back in Listing 27.

Before starting a conversation, and with the chatbot application already up and running, Messenger platform requires the developer to access its page and submit the callback URL and the Verify Token. While Ngrok provided the callback URL, the Verify Token can be any key that the developer wants to use.

After submitting the callback URL and the respective token in the platform, the Framework shows a message to the application terminal as shown in Figure 6.3.

```
>Messenger GET request received!  
Everything is correct! Webhook has been set! Enjoy ;)
```

Figure 6.3 – Text shown on the application debug terminal.

With the webhook set, it was time to send some messages. When sending a message to the chatbot it is expected that it sends the same message back to the user. For that reason, we sent two messages to the chatbot using the Facebook Messenger APP on a smartphone. The first message content was “Testing Facebook Messenger chatbot with the framework application supporting just this service.”, and the second message was “Testing Facebook Messenger chatbot with the framework application supporting just this service a second time”. Figure 6.4 provides screenshots of the result of the test performed.

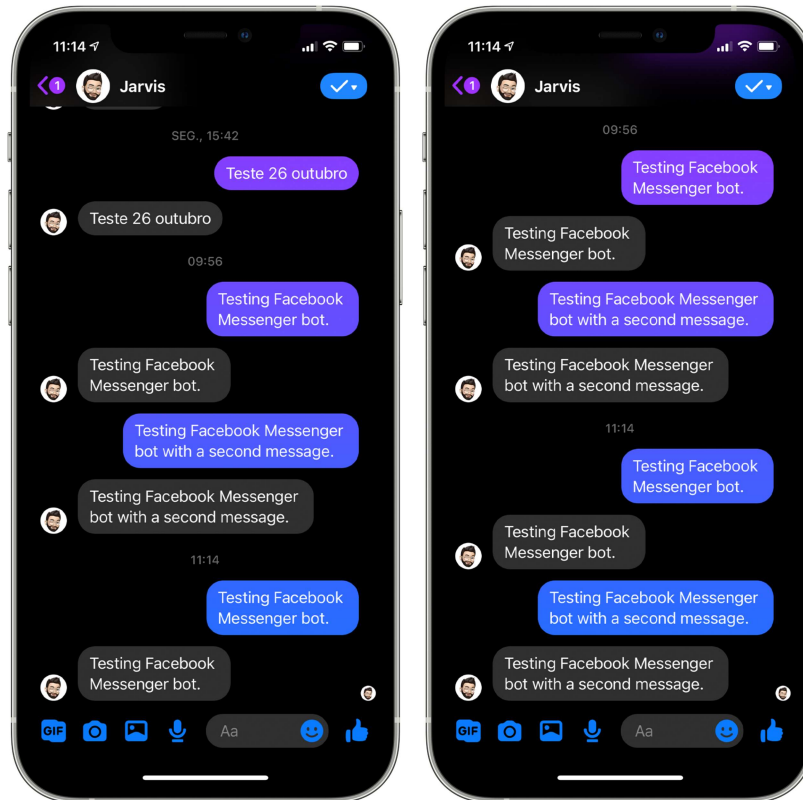


Figure 6.4 – Testing Facebook Messenger chatbot.

As seen in the Figure 6.4, the chatbot answers the message sent with the same text that it receives.

This test proves the framework supports the Facebook Messenger service, and it can answer its text messages.

Telegram

With Telegram, the webhook callback URL setting is quite different from the process performed for Facebook Messenger in the previous section. For this service, when initializing the application, it is only needed to perform a POST request to the URL shown in Listing 28.

```
{  
  https://api.telegram.org/bot23f283ncb8b34b95bn5345/setWebhook  
}
```

Listing 28: URL to set the webhook on the Telegram platform.

The content of the message sent in the POST request to the URL above must have a JSON formatted text with the callback URL for the webhook service. In this case, the POST request is performed as soon as the chatbot application starts running, so the process for setting the webhook is automatic. If the webhook is already set, the platform will ignore the request.

If the response to the POST request is a 200 OK status, the message shown in Figure 6.5 will appear in the application debug terminal.

```
Telegram servlet initialized  
Telegram webhook has been configured! Enjoy ;)
```

Figure 6.5 – Text shown on the application debug terminal after setting the webhook on telegram.

Now that we successfully set the webhook, we should correctly exchange messages between the Telegram APP and the chatbot application. Like the Messenger application, this one also answers back with the user’s same text message.

The test performed on the Telegram application was identical to the one performed on Facebook Messenger. We sent two messages from the Telegram smartphone APP. The first one with the text “Testing Telegram chatbot with the framework application supporting just this service.”, and the second message saying “Testing Telegram chatbot with the framework application supporting just this service for the second time.”.

Figure 6.6 shows screenshots of the messages successfully exchanged between a user and the chatbot application. Both messages received the text sent as an answer

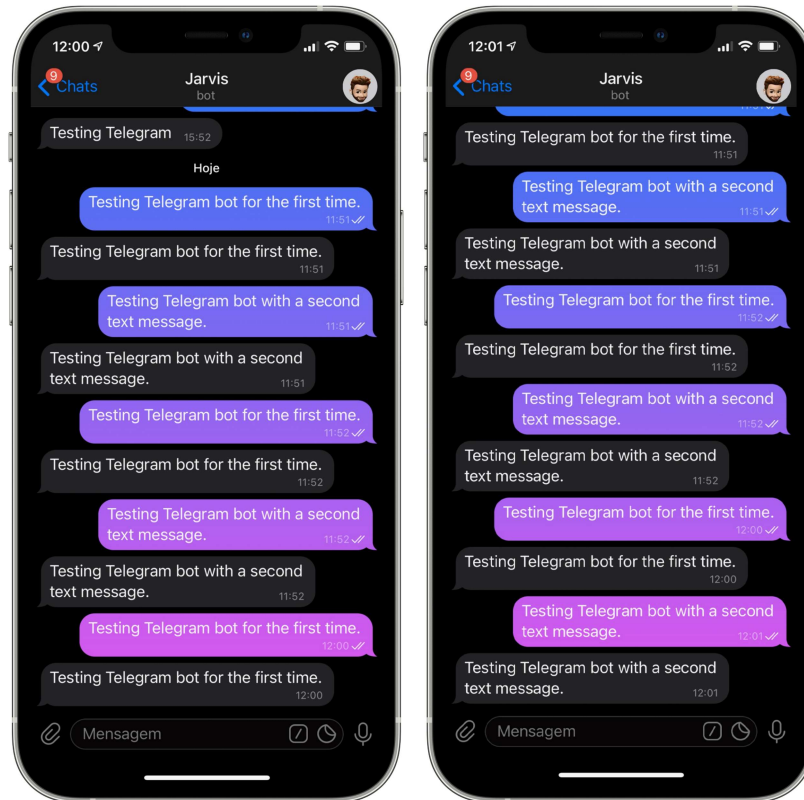


Figure 6.6 – Testing Telegram chatbot.

from the chatbot application.

Testing two platforms independently prove that the framework can handle different messaging services with no problems. Now, this could be the starting point of many developers trying to embrace the world of chatbots. However, as things evolve, the next logical step could be to improve the service for a single platform by increasing the number of chatbots to the same messaging service. That is what the section [6.2.2](#) means to test.

6.2.2 Scenario 2: Two chatbots working with a single service

Two chatbots working with a single service is the case where, for example, a developer wants to provide a better service for the same platform. A good example

can be to increase the number of chatbots that assist the same messaging service. In that case, more than one chatbot in the application is receiving the messages the user is sending. Another useful implementation could be if the developer wants the chatbot service to support more than one language. By adding two chatbots to the service, each one could be responsible for a specific language. Figure 6.7 has a diagram that represents this scenario.

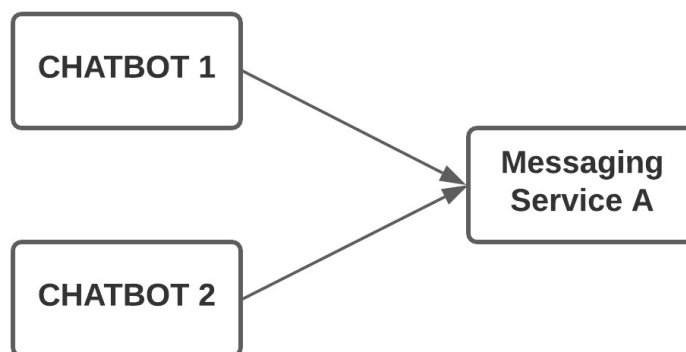


Figure 6.7 – Example diagram of two chatbots working with one service only.

The tests to this scenario are positive, and the user received only one answer to its message, although more than one chatbot is listening to its messages. The screenshots of this test are visually the same as the previous case scenario.

Now that we already have two chatbots working for the same service, what if we want to expand support for another messaging service that is already implemented in the framework? Subsection 6.2.3 tests if this scenario is possible or not.

6.2.3 Scenario 3: Two services working with one chatbot each

In this scenario, things start to get interesting. We want to see if the Framework can handle two chatbots, each with its own messaging service. Figure 6.8 shows a diagram with this implementations.

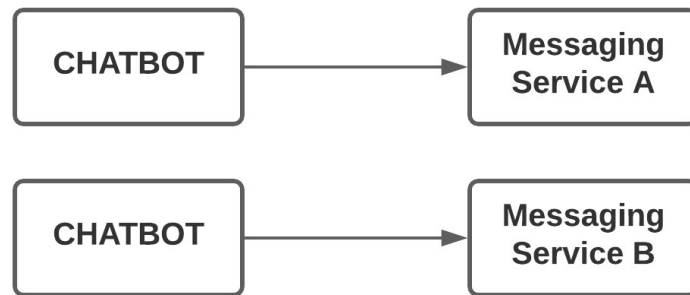


Figure 6.8 – Example diagram of two chatbots working with one service each.

Using Facebook Messenger and Telegram, we sent a message from each of them to the correspondent chatbot. The results, shown in Figure 6.9, were as expected, and the chatbot application implementing the Framework answered back with the echoed message.



Figure 6.9 – Testing Facebook Messenger and Telegram chatbots at the same time.

With that said, the framework can effectively handle two different messaging services, each with its own chatbot. However, what if the chatbot can be the same for both messaging services? Can the framework provide support for more than one platform with just one chatbot coded? That is what the 6.2.4 wants to prove.

6.2.4 Scenario 4: One single chatbot working with two distinct services

The most useful case scenario might be when the developer wants to code only one chatbot and make it available in many different services. To test that case scenario, we added support for Facebook Messenger and Telegram and created a single chatbot that replies to the user's messages with the same text. The diagram presented in Figure 6.12 represents the tested scenario.

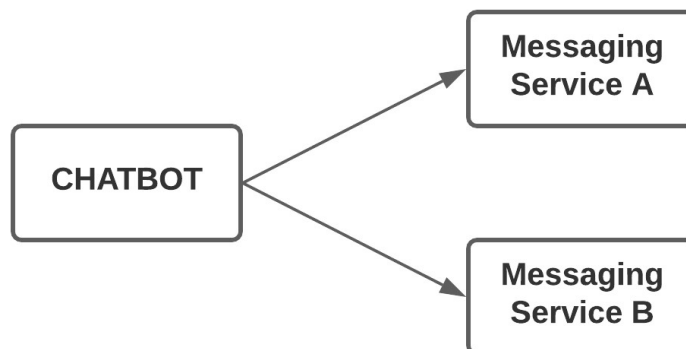


Figure 6.10 – Example diagram of one single chatbot working with one distinct services.

Although we are using the same chatbot for the different platforms, it is still possible to apply different algorithms or replies depending on the messaging service we are talking. To understand which chatbot is receiving and replying to the messages, we added the platform's name at the beginning of the text sent in the replies.

The screenshots in Figure 6.9 show the different messages being exchanged between the two messaging services.

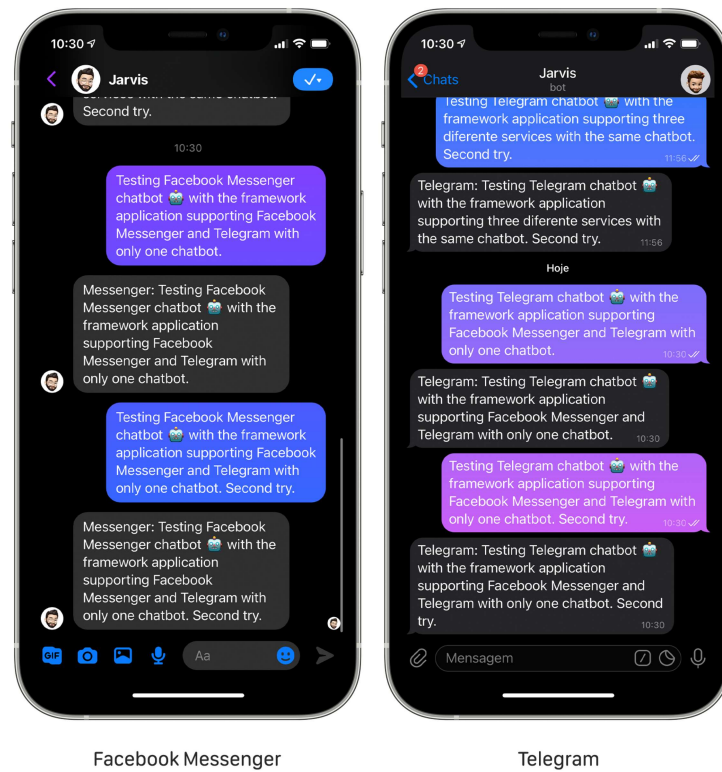


Figure 6.11 – Testing Facebook Messenger and Telegram with only one chatbot for both services.

The results of this test are positive, and the Framework allows one single chatbot for multiple platforms. In this test, we separated the two replies in order to understand if that was possible. However, the response can be coded to be the same for every platform if that is the chatbot's purpose.

Although we are already covering multiple platforms simultaneously with only one chatbot, which was this dissertation's first goal, it is still missing the addition of messaging services that are not yet implemented in the Framework and that any developer might want to see added to its chatbots. This last case scenario is tested in Section 6.2.5.

6.2.5 Scenario 5: Adding support for a new messaging service

Aiming to create a future proof Framework, it has to ensure support for upcoming messaging platforms. Under those circumstances, the application coded to test the framework's viability was used to add support for a new messaging service that is not already supported by the framework.

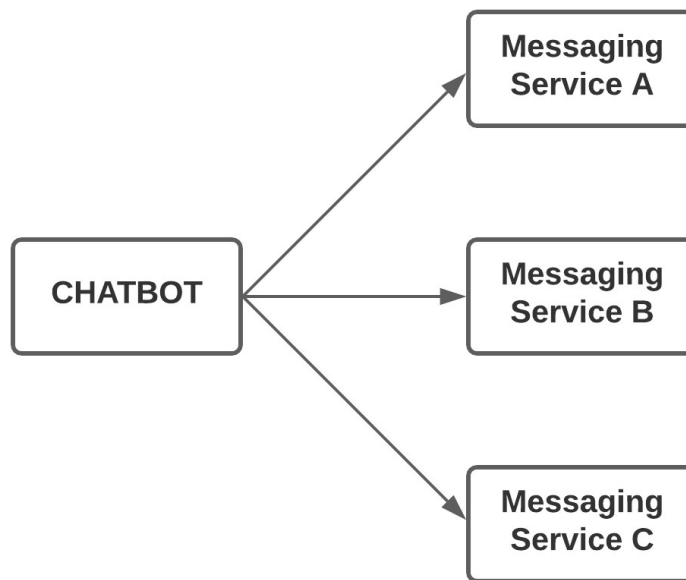


Figure 6.12 – Example diagram of one single chatbot working with three distinct services, being one of them not implemented in the framework.

We chose WeChat to test if the Framework was capable of receiving and support upcoming messaging services. In order to add this messaging service, we created two classes:

- **WechatBot** - implements the Bot interface and is responsible for defining what to do when the bot receives a new incoming message.

- **Wechat** - extends the Chatbot class and defines every detail about the communication to the server such as webhook setting and HTTPS GET and POST requests.

The main methods of these two classes were detailed back in Chapter 5, when the application implementing the Framework was detailed.

After having the two classes created, they can be added to the rest of the application with simple lines of codes, shown in Listing 29.

```
1 String wechatId = "e52f127617624ac692126b0220e86177";
2 String wechatToken = "EAAB53C6WkbEBAB07Gkvm5RVRL7ZB";
3 String wechatUrl = "https://6830c6443c8d.ngrok.io/wechat";
4 String wechatAppSecret = "3364tv45e42295520d23f3f6973cbd"
5
6 Chatbot weChat = new WeChat("WeChat", wechatId, wechatToken, url,
7 "/wechat");
8 weChat.setAppSecret(wechatAppSecret);
9
10 weChat.initialise(server);
11
12 // Adding messenger and telegram services to the event manager
13 manager.addService(wechat);
14
15 new WeChatBot(weChat);
16
17 // Server init and setting to await state
18 // ...
```

Listing 29: Command used in terminal to start ngrok service.

Again, with the application running, setting the callback URL is mandatory and the process to do it was already detailed in Chapter 3. After submitting in the parameters by the platform, the application's terminal shows the message represented in Figure 6.13.

```
>WeChat GET request received!
```

```
>WeChat webhook has been set!
```

Figure 6.13 – Text shown on the application debug terminal when setting WeChat’s webhook.

After setting the webhook, it is time to test the messaging service. This scenario means to test if it is possible to add new platforms that are not supported by the Framework. However, the tests were performed with the other two services working, meaning that, in this test, the application will offer support for three different messaging services: Facebook Messenger, Telegram, and WeChat.

We sent a simple text message from each APP on a smartphone and successfully received the same text back from the chatbot. Figure 6.14 proves the messages where sent and received.

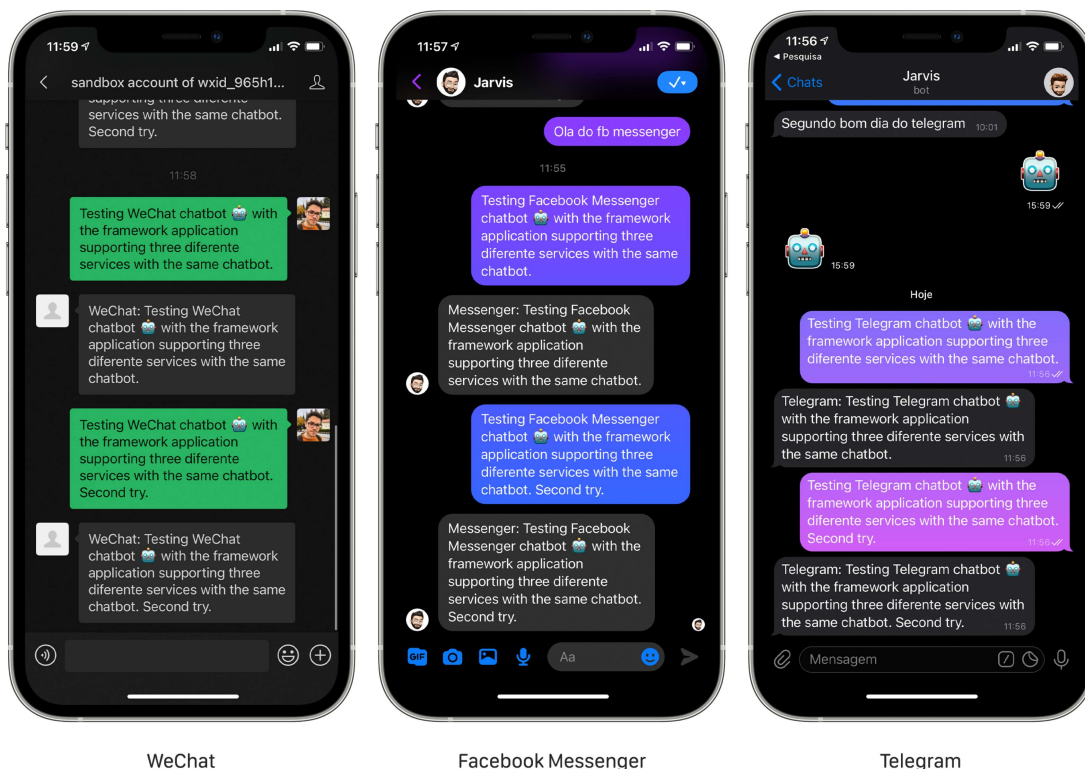


Figure 6.14 – Results of the tests performed with three messaging platforms being one of them not included in the Framework.

Giving that this test is similar to the one shown in Section [6.2.4](#), the chatbots were also coded in order to reply with its respective messaging service name in the beginning of each reply.

With this last fifth test working as expected, the Framework is in reality ready to a real-world implementation and is, in fact, future proof.



Conclusions and Future work

This chapter considers all the systems developed during this dissertation and the technologies used during this process. The framework has a lot to grow, and some future work can improve the framework making it more reliable and stable. Section [7.2](#) details some of that future work.

7.1 Conclusions

This dissertation focused on developing a framework capable of creating a chatbot for multiple platforms intuitively and simply. Although there are many solutions in chatbot development, most of them are complex and focus on questions like natural language and AI, which might not be a point of interest for some developers. Moreover, with the current popularity of languages like PHP and Node.js, most existing solutions forgot that Java servers are still a significant percentage of the market, same with the Java language itself.

Chatbots have been growing over time and, every day, we see more businesses investing in this technology as a solution to their problems. However, chatbot's limitations are endless, and people can use them for various means.

Considering that developing a framework for creating and developing chatbots in a Java environment was this dissertation's primary goal, the final result is satisfactory. The framework can create a chatbot that can establish communication with multiple messaging services simultaneously and do all this while being standalone and not requiring the developer to implement other third-party software.

Each chatbot's particular implementation of every messaging platform complicated the development process to achieve a universal chatbot development framework. Nevertheless, in the end, the structure of the framework makes it a universal solution and future proof by adding a possibility of implementing future messaging services that may emerge in the market.

7.2 Future Work

After finishing the work that led to the development of this dissertation and analyzing the final result, it is possible to conclude that there are still some improvement points in the developed system. Some of those improvements are:

- Perform specific software tests to the Framework, like Unit tests, in order to guarantee its success.
- Implement the support for other message formats rather than text, such as attachments and quick replies. It is important because it is available on most platforms and is a point of interest for many chatbots developers.
- Study the interest in integrating an AI service to help new developers build a chatbot more capable without the need for third-party software. Its addition could increment the popularity of the framework.
- Study the possibility of different ways to communicate with a messaging service platform rather than webhooks. A very known solution adopted, for example, by Telegram, is long polling.
- Add support for other messaging platforms like Signal and QQ, for example.

References

- ABCOM (2019). HTTP/1.1 vs HTTP/2: What's the difference? <https://www.digitalocean.com/community/tutorials/http-1-1-vs-http-2-what-s-the-difference>. [Online, consulted on January 7, 2021]. 55
- Adenowo, A. and Adenowo, B. (2020). Software engineering methodologies: A review of the waterfall model and object- oriented approach. *International Journal of Scientific and Engineering Research*, 4:427–434. 48
- AdEspresso (2020). The most inspiring facebook messenger chatbots how to build your own. <https://adespresso.com/blog/5-inspiring-chatbots-facebook-messenger/>. [Online, consulted on January 25, 2021]. 14
- Alkhulaifi, A. and El-Alfy, E. M. (2020). Exploring lattice-based post-quantum signature for jwt authentication: Review and case study. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5. 63
- Amondarain, M. F. (2018). Indubot. Master's thesis, School of Industrial Engineering of Barcelona. 8

- Anna, A. and Weißensteiner, A. (2018). *Chatbots as an approach for a faster enquiry handling process in the service industry*. PhD thesis, Modul University Vienna. 6
- Arnaud Gellens, S. G. (2019). *JAQ : A Chatbot for Foreign Students*. PhD thesis, École polytechnique de Louvain. 5, 12
- Arsenijevic, U. and Jovic, M. (2019). Artificial intelligence marketing: Chatbots. pages 19–193. 9
- Auth0. Jwt. <https://jwt.io>. [Online, consulted on November 19, 2020]. 64, 65
- Bhagwat, V. A. (2018). *Deep Learning for ChatBots*. PhD thesis. 6
- Botpress (2019a). Botpress Documentation. <https://botpress.com/docs/introduction>. [Online, consulted on December 5, 2020]. 19
- Botpress (2019b). Channels available for enterprise chatbot platform - Botpress. <https://botpress.com/features/supported-platforms>. [Online, consulted on December 12, 2020]. 19
- Bucher, B. (2020). WhatsApp, WeChat and Facebook Messenger Apps – Global usage of Messaging Apps, Penetration and Statistics. <https://www.messengerpeople.com/global-messenger-usage-statistics/>. [Online, consulted on January 9, 2021]. 24, 25
- Chatterbot (2019). Chatterbot Documentation. <https://chatterbot.readthedocs.io/en/stable/>. [Online, consulted on December 5, 2020]. 20
- Chen, Z. (2019). *Co-designing a Chatbot for and with Refugees and Migrants*. PhD thesis, Aalto University. 5
- Chung, M., Ko, E., Joung, H., and Kim, S. J. (2020). Chatbot e-service and customer satisfaction regarding luxury brands. *Journal of Business Research*, 117:587 – 595. 3
- Cleverbot (2006). Cleverbot. <https://www.cleverbot.com>. [Online, consulted on November 9, 2020]. 11

- Developers, M. (2019). Microsoft Bot Framework. <https://dev.botframework.com/>. [Online, consulted on December 5, 2020]. 17
- Dictionary, C. (2020). Meaning of chatbot in english. <https://dictionary.cambridge.org/dictionary/english/chatbot>. [Online, consulted on October 15, 2020]. 5
- Dooley, J. F. (2017). *Software Development, Design and Coding*. Apress. 48
- Eason, O. K. (2016). Information Systems Development Methodologies Transitions: An Analysis of Waterfall to Agile Methodology. *University of New Hampshire*, pages 1–23. 49
- EQUITY, B. B. C. F. (2019). Chatbot report 2019: Global trends and analysis. <https://chatbotsmagazine.com/chatbot-report-2019-global-trends-and-analysis-a487afec05b>. [Online, consulted on November 27, 2020]. 67
- Facebook, f. D. (2021). Messenger platform. <https://developers.facebook.com/docs/messenger-platform/>. [Online, consulted on January 10, 2021]. 26, 28, 29, 30, 31, 33, 34
- Ferreira, J. A. O. (2008). *Interface Homem-Máquina para Domótica baseado em tecnologias Web*. PhD thesis, Faculdade de Engenharia da Universidade do Porto. 71
- Filipova, O. and Vilao, R. (2018). *Software Development From A to Z: Deep Dive into all the Roles Involved in the Creation of Software*. Apress. 48
- Gallagher, A., Dunleavy, J., and Reeves, P. (2019). The Waterfall Model: Advantages, disadvantages, and when you should use it. <https://developer.ibm.com/technologies/devops/articles/waterfall-model-advantages-disadvantages/>. [Online, consulted on November 27, 2020]. 49
- Google (2019). Dialogflow. <https://cloud.google.com/dialogflow/docs>. [Online, consulted on December 5, 2020]. 18

- Haekal, M. and Eliyani (2016). Token-based authentication using json web token on sikasir restful web service. In *2016 International Conference on Informatics and Computing (ICIC)*, pages 175–179. 64
- Instalocate (2019). Instalocate. <https://www.instalocate.com>. [Online, consulted on January 5, 2021]. 12
- Jackson, A. (2018). Siri vs google assistant vs bixby. <https://medium.com/@alice.jackson/siri-vs-google-assistant-vs-bixby-bde8573b03f6>. [Online, consulted on February 5, 2021]. 15
- Kemp, S. (2020). Digital 2020: Global Digital Overview. <https://datareportal.com/reports/digital-2020-global-digital-overview>. [Online, consulted on January 9, 2021]. 23, 24
- Keszocze, O. and Harris, I. G. (2019). Chatbot-based assertion generation from natural language specifications. In *2019 Forum for Specification and Design Languages (FDL)*, pages 1–6. 18
- Kongaut, C. and Bohlin, E. (2016). Investigating mobile broadband adoption and usage: A case of smartphones in sweden. *Telematics and Informatics*, 33(3):742 – 752. 2
- Lee, S., Lee, J., Lee, W., Lee, S., Kim, S., and Kim, E. T. (2020). Design of integrated messenger anti-virus system using chatbot service. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1613–1615. 72
- Lin, Y. (2020). 10 mobile usage statistics you should know in 2020 [infographic]. <https://www.oberlo.com/blog/mobile-usage-statistics>. [Online, consulted on November 27, 2020]. 67
- Lindvall, N. and Ljungström, R. (2018). Chatbot for configuration. Master’s thesis, Lund University. Student Paper. 2

- Liu, S. and Chen, P. (2009). Developing java ee applications based on utilizing design patterns. In *2009 WASE International Conference on Information Engineering*, volume 2, pages 398–401. 70
- Ma, J., Che, C., and Zhang, Q. (2018). Medical answer selection based on two attention mechanisms with birnn. *MATEC Web of Conferences*, 176:01024. 13
- Maia, R. F. (2012). *Desenvolvimento de uma Aplicação Web para Apoio de Cálculo de Estruturas Metálicas*. PhD thesis, Faculdade de Engenharia da Universidade do Porto.
- Martins, A. M. P. (2002). *Especificação XML de aplicações para WWW*. PhD thesis, Faculdade de Engenharia da Universidade do Porto. 60
- Merisalo, S. (2018). *Developing a Chatbot for Customer Service to Metropolia UAS Student Affairs Office*. PhD thesis. 7
- MindSea (2020). 28 Mobile App Statistics To Know In 2020. <https://mindsea.com/app-stats/>. [Online, consulted on January 7, 2021]. 23, 67
- Mozilla (2020a). Evolution of http. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP. [Online, consulted on November 14, 2020]. 56
- Mozilla (2020b). Http messages. <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Mensagens>. [Online, consulted on November 14, 2020]. 56, 63
- Mu, H. and Jiang, S. (2011). Design patterns in software development. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pages 322–325. 80
- Muhammad, A. F., Susanto, D., Alimudin, A., Adila, F., Assidiqi, M. H., and Nabhan, S. (2020). Developing english conversation chatbot using dialogflow. In *2020 International Electronics Symposium (IES)*, pages 468–475. 18
- Nafis, R. M. and Setiawan, E. B. (2019). Application for Booking Handyman Services Using Webhook and Google Event Calendar Technology. 72

- Nahar, N. and Sakib, K. (2016). Acdpr: A recommendation system for the creational design patterns using anti-patterns. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 4, pages 4–7. 49
- Ngrok (2020). What is ngrok? <https://ngrok.com/product>. [Online, consulted on November 23, 2020]. 96
- Nuruzzaman, M. and Hussain, O. K. (2018). A survey on chatbot implementation in customer service industry through deep neural networks. pages 54–61. 11
- Ojapuska, E. (2018). The impact of chatbots in customer engagement. *Vaasa University of Applied Sciences*, pages 1 – 40. 8
- Oracle (2012). Differences between java ee and java se. <https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html>. [Online, consulted on January 22, 2021]. 69
- Oracle (2014). Java ee 7 apis. <https://docs.oracle.com/javaee/7/tutorial/overview007.htm#BNACJ>. [Online, consulted on January 23, 2021]. 71
- Oracle (2021). Java ee at a glance. <https://www.oracle.com/java/technologies/java-ee-glance.html>. [Online, consulted on January 7, 2021]. 69, 70
- Orin, T. D. (2017). Implementation of a Bangla Chatbot. *SpringerBriefs in Applied Sciences and Technology*, (April):49–61. 11, 12
- Pandorabots (2019). Pandorabots. <https://home.pandorabots.com/home.html>. [Online, consulted on December 5, 2020]. 19
- Patil, A., Karuppiah, M., A, N., and Niranchana, R. (2017). Comparative study of cloud platforms to develop a chatbot. *International Journal of Engineering & Technology*, 6:57. 17
- Petersen, K., Wohlin, C., and Baca, D. (2009). The waterfall model in large-scale development. In *The Waterfall Model in Large-Scale Development*. 49

- Radziwill, N. and Benton, M. (2017). Evaluating quality of chatbots and intelligent conversational agents. 3
- Redstone, J. D. B. (2019). *Turing Tests as Reflexive Experimental Apparatus*. PhD thesis, Carleton Univertisty. 6
- RFC2616 (1999). Hypertext transfer protocol–HTTP/1.1. <https://tools.ietf.org/html/rfc2616>. [Online, consulted on November 19, 2020]. 53, 54
- RFC2818 (2000). HTTP Over TLS. <https://tools.ietf.org/html/rfc2818>. [Online, consulted on November 19, 2020]. 58
- RFC3986 (1999). Uniform Resource Identifier (URI): Generic Syntax. <https://tools.ietf.org/html/rfc3986>. [Online, consulted on November 19, 2020]. 55
- RFC7159 (2014). The JavaScript Object Notation (JSON) Data Interchange Format. <https://tools.ietf.org/html/rfc7159>. [Online, consulted on January 19, 2021]. 61
- RFC7231 (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. <https://tools.ietf.org/html/rfc7231#page-47>. [Online, consulted on January 19, 2021]. 57
- RFC7519 (2015). JSON Web Token (JWT). <https://tools.ietf.org/html/rfc7519>. [Online, consulted on January 19, 2021]. 63
- Robeznikes, A. (2016). Healthtap integrates research summary provider docphin after quiet acquisition. <https://medcitynews.com/2016/07/healthtap-docphin-acquisition/>. [Online, consulted on February 5, 2021]. 13
- Salvadori, I. (2015). *Desenvolvimento de Web APIs RESTful Semânticas Baseadas em JSON*. PhD thesis, Universidade Federal de Santa Catarina. 57
- Sannikova, S. (2018). *Chatbot implementation with Microsoft Bot Framework*. PhD thesis, Metropolia University of Applied Sciences. 6
- Shbair, W., Cholez, T., and Chrisment, I. (2017). *Service-Level Monitoring of HTTPS Traffic*. PhD thesis, University of Lorraine. 59

- Shvets, A. (2019). *Dive Into Design Patterns*. 16, 50, 51, 52
- Siebert, I., Sinha, Y., Jokisch, O., and Wendemuth, A. (2020). *Recognition Performance of Selected Speech Recognition APIs – A Longitudinal Study*. Springer. 17
- Spectrm (2020). Messaging Apps Have Taken Over — Usage & Growth Statistics. <https://spectrm.io/insights/blog/messaging-app-statistics-most-popular-communication-method-2020/>. [Online, consulted on December 3, 2020]. 2
- Stanivuk, I., Bjelić, V., Samardžić, T., and Simić, (2017). Expanding lua interface to support http/https protocol. In *2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)*, pages 407–410. 53
- Suite, I. (2019). Entendendo os conceitos entre os modelos tcp/ip e osi. <https://www.iperiusbackup.net/pt-br/entendendo-os-conceitos-entre-os-modelos-tcpip-e-osi/>. [Online, consulted on November 14, 2020]. 54
- Telegram (2020). Telegram bot api. <https://core.telegram.org/bots/api>. [Online, consulted on March 10, 2020]. 35, 36, 37
- Tiha, A. (2018a). *Intelligent Chatbot using Deep Learning*. PhD thesis. 10
- Tiha, A. (2018b). Intelligent chatbot using deep learning. Master's thesis.
- TIOBE (2021). Python is tiobe's programming language of 2020! <https://www.tiobe.com/tiobe-index/>. [Online, consulted on January 7, 2021]. 68
- W3schools (2020). XML on the Server. https://www.w3schools.com/xml/xml_server.asp. [Online, consulted on January 19, 2021]. 60
- Wang, X., Xu, B., and Gu, R. (2013). The application of code reuse technology based on the mvc framework. In *2013 International Conference on Computer Sciences and Applications*, pages 534–537. 16

- Watson, I. (2019a). About Watson. <https://www.ibm.com/watson/about>. [Online, consulted on December 5, 2020]. 18
- Watson, I. (2019b). IBM Documentation. <https://cloud.ibm.com/docs/assistant?topic=assistant-index>. [Online, consulted on December 12, 2020]. 18
- WeChat (2020). Official accounts. https://developers.weixin.qq.com/doc/offiaccount/en/Basic_Information/Access_Overview.html. [Online, consulted on September 17, 2020]. 40, 41, 42, 44
- Wei, C., Yu, Z., and Fong, S. (2018). How to build a chatbot: Chatbot framework and its capabilities. In *Proceedings of the 2018 10th International Conference on Machine Learning and Computing*, ICMLC 2018, page 369–373, New York, NY, USA. Association for Computing Machinery. 15

