

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Miranda de Aguiar

**Framework para desenvolvimento de Chatbots
na plataforma Android**

Uberlândia, Brasil

2019

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Miranda de Aguiar

**Framework para desenvolvimento de Chatbots na
plataforma Android**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau
de Bacharel em Ciência da Computação.

Orientador: Prof. Fabiano Azevedo Dorça

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2019

Gustavo Miranda de Aguiar

Framework para desenvolvimento de Chatbots na plataforma Android

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do grau
de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 24 de novembro de 2012:

Prof. Fabiano Azevedo Dorça
Orientador

Professor

Professor

Uberlândia, Brasil
2019

Em primeiro lugar, à Deus, pela força e coragem durante toda esta longa caminhada.

Aos meus pais, que sempre me deram apoio ao longo dessa jornada. E a minha namorada Lara Carolina, que esteve sempre ao meu lado em todos os momentos.

“O verdadeiro perigo não é que computadores começarão a pensar como homens, mas que homens começarão a pensar como computadores”. Sydney J. Harris

Resumo

Este trabalho consiste na implementação de um *framework* que possibilita o desenvolvimento de *chatbots* que possuam sintetização de fala e reconhecimento de voz. O objetivo é fornecer uma integração direta e simplificada de um aplicativo Android com a plataforma Dialogflow. Para isso, foi desenvolvido um módulo Android que implementa a interface gráfica e todos os métodos necessários a serem sobrescritos da SDK Android disponibilizada pela plataforma. Além disso, é produzida uma documentação, orientando o uso do *framework* através de seus pontos de acesso, e também desenvolvido um estudo de caso de um modelo conversacional que demonstra, na prática, um uso eficaz dos recursos da plataforma, aplicados ao *framework*.

Palavras-chave: *framework*, *chatbot*, Dialogflow, Android, sintetização de fala, reconhecimento de voz.

Lista de abreviaturas e siglas

API	<i>Application Programming Interface</i>
CHATBOT	Robô mensageiro
CORBA	<i>Common Object Request Broker Architecture</i>
HTTP	<i>Hypertext Transfer Protocol</i>
JSON	<i>JavaScript Object Notation</i>
PLN	Processamento de Linguagem Natural
REST	<i>Representational State Transfer</i>
SDK	<i>Software Development Kit</i>
TTS	<i>Text-to-Speech</i>
XML	<i>Extensible Markup Language</i>

Sumário

1	INTRODUÇÃO	9
1.1	Objetivos	10
1.2	Justificativa	10
1.3	Método	11
1.4	Trabalhos Relacionados	11
1.5	Resultados	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	<i>Frameworks</i> Orientados a Objeto	13
2.1.1	Hotspots	13
2.1.2	<i>Coldspots</i>	14
2.1.3	Classificação	14
2.1.3.1	Caixa Branca	14
2.1.3.2	Caixa Preta	16
2.1.3.3	Caixa Cinza	17
2.2	Documentação de <i>frameworks</i> OO	18
2.3	<i>Chatbots</i>	19
2.3.1	<i>Frameworks</i> e APIs	20
2.3.2	Dialogflow	20
2.3.2.1	Agents	21
2.3.2.2	<i>Intents</i> e <i>Contexts</i>	22
2.3.2.3	<i>Fullfilment</i>	23
3	DESENVOLVIMENTO DA APLICAÇÃO	24
3.1	Escolher o tipo de <i>framework</i>	24
3.2	Modelo conversacional	24
3.3	<i>Layout</i> da aplicação	25
3.4	Diagrama de classes	27
3.5	<i>Hotspots</i>	30
3.5.1	setConfig	30
3.5.2	setFalaInicial e BotFala	31
3.5.3	executaAcao	32
3.6	Documentação para instanciação do <i>framework</i>	35
4	RESULTADOS	37
4.1	Validação	37

4.2	Estudo de caso	39
4.2.1	Fluxo conversacional	40
4.2.2	Criação de <i>intents</i> e <i>contexts</i>	41
5	CONCLUSÃO	54
	REFERÊNCIAS	56
	APÊNDICES	58
	APÊNDICE A – DOCUMENTAÇÃO PARA INSTANCIAÇÃO DO FRAMEWORK	59
A.1	Executando o código de exemplo do <i>framework chatbot</i> inteligente	60
A.2	Começando com o seu próprio app	60
A.2.1	Primeira classe	60
A.2.2	Segunda classe	61
A.2.3	Classe AcaoTeste	62
A.2.4	Executando o app	62
	APÊNDICE B – QUESTIONÁRIO USADO PARA VALIDAÇÃO DO FRAMEWORK	64
B.1	Questões abertas	64
B.2	Questões de múltipla escolha	64

1 Introdução

O sistema operacional Android, desenvolvido pela Google em 2008, é uma plataforma completa para dispositivos móveis (*smartphones* e *tablets*). O desenvolvimento de aplicativos para estes dispositivos tem crescido muito nos últimos 4 anos e cada vez mais visado pelas empresas. Segundo levantamento da IDC (2017), o Android deteve 85% do mercado de dispositivos móveis em 2017. O fato de ser *open source*, gratuito e com a possibilidade de desenvolvimento através de qualquer computador contribuiu para seu crescimento.

O reúso de software, uma das técnicas mais importantes da Engenharia de Software, permite ao desenvolvedor economizar recursos e tempo de desenvolvimento para construção de um sistema. Com ele, é possível aproveitar um componente de software já desenvolvido inserindo-o em um outro software diferente. No desenvolvimento para dispositivos móveis, o uso de *Application Programming Interfaces* (APIs) se faz cada vez mais necessário, pois através delas o reúso é visto na prática, assim como seus benefícios.

Neste contexto, surge um importante componente reutilizável: os *chatterbots* ou simplesmente *bots*. *Chatbots* são colaboradores virtuais que simulam uma conversa virtual entre dois seres humanos. Muitos deles têm como função resolver determinado problema que envolve um ou mais passos através de uma interação com o usuário via texto ou voz. Para desenvolver um *chatbot*, genericamente, é necessário desenvolver uma interface gráfica que permita ao usuário escrever o que deseja na conversa e que mostre na tela a resposta do *chatbot* à essa entrada do usuário.

Outro fator importante é alimentar uma base de conhecimento que permita ao *chatbot* ter as informações necessárias para fornecer saídas coerentes com o propósito para o qual o mesmo foi desenvolvido, assim como um cuidado adequado ao modo como o *chatbot* trata essas informações para obter as respostas. Caso o *bot* converse com o usuário através de voz é preciso que na implementação exista um processamento de linguagem natural que reconheça a fala do usuário e contenha uma saída de áudio que simule uma fala humana para o usuário no momento em que ele responde.

Neste trabalho, discutem-se as seguintes problemáticas: é possível desenvolver um *chatbot* de texto e voz para dispositivos móveis na plataforma Android? Como reutilizar um componente de software de um *chatterbot* em outros softwares? Existem bibliotecas ou *frameworks* disponíveis que implementem *bots* prontos com a possibilidade de personalizá-los conforme as necessidades do usuário?

1.1 Objetivos

O objetivo geral deste trabalho é possibilitar o reúso de software através da construção de um *framework* para implementação de *chatbots* na plataforma Android. Com isso, além de promover a reutilização de software no ambiente *mobile*, será possível incentivar desenvolvedores e empresas a construírem o próprio *chatbot* de maneira mais simples e menos onerosa financeiramente.

A fim de atingir o objetivo geral, o objetivo específico consiste em desenvolver um *framework* que possibilita a implementação de *chatbots* com interface de texto e voz, que possa ser usado em aplicações Android.

1.2 Justificativa

No Brasil, segundo dados levantados numa pesquisa de [Paiva \(2017\)](#), existem mais de 8 mil *bots* desenvolvidos por 56 empresas que realizam 500 milhões de interações por mês em diversos canais (Facebook Messenger, sites na *web*, apps móveis, Twitter, etc). Desde a definição original de *chatbot* de [Oliveira \(2017\)](#), associa-se ao papel do assistente virtual resolver determinado problema de forma automatizada através de uma conversa por chat. Assim, os *chatbots* estão revolucionando a forma de atendimento das empresas de maneira rápida e digital, e é a partir do crescimento do desenvolvimento de *bots* no Brasil que, segundo afirma [Martin \(2017\)](#), as empresas que adotarem essa tecnologia terão seus custos com mão-de-obra reduzidos, seu atendimento otimizado e sua produtividade aumentada. No desenvolvimento é muito comum o uso de plataformas de processamento de linguagem natural para criar os *bots*. Segundo a pesquisa de [Paiva \(2017\)](#), 55% dos desenvolvedores utilizam alguma dessas plataformas e apenas 32% dos desenvolvedores de *bots* utilizam suas próprias ferramentas. Cada uma dessas plataformas possuem mecanismos que permitem ao desenvolvedor trabalhar com o fornecimento de uma API que processa e responde as mensagens ou eventos enviados. Essas plataformas disponibilizam SDKs para diferentes tecnologias e linguagens de programação, tanto para *web* quanto para *mobile*. No entanto, ainda é raro encontrar *frameworks* ou bibliotecas que interajam com essas plataformas de modo a serem reusadas em uma aplicação *mobile* de forma simplificada e direta. Assim, ganha cada vez mais relevância o uso de um módulo que automatize esse processo e possa ser incluído em aplicativos como se fosse um componente novo da aplicação. A importância de módulos reutilizáveis de *chatbots* de texto e voz torna-se ainda mais evidente diante do cenário de redução de custos e ganho de produtividade que estes módulos aplicados poderiam ter.

1.3 Método

O presente estudo consiste em pesquisa aplicada, de caráter exploratório, que visa não só discutir os benefícios do reúso de componentes de software, bem como apresentar um módulo que possa servir de base para a construção de um *chatbot* para dispositivos móveis da plataforma Android.

Nesse sentido, os resultados serão apresentados sobre forma quantitativa, a partir da coleta de informações de fontes primárias e secundárias, incluindo revisão bibliográfica. A planificação da pesquisa inclui, em primeiro lugar, o levantamento dos dados primários para avaliar a quantidade de código fonte do módulo a ser desenvolvido e de um programa de teste. Além disso é necessário avaliar a qualidade da documentação produzida do *framework* proposto.

A metodologia que provê suporte a este trabalho tem como base heurísticas de documentação de *frameworks* proposta pelos autores Qinxue Chen e Fangxiao Xu que descrevem em sua obra *Framework Issue: Framework Documentation* que tanto a importância da documentação de *frameworks* quanto os tipos de abordagens podem ser adotadas em sua produção.

1.4 Trabalhos Relacionados

Existem algumas aplicações desenvolvidas com a finalidade de integrar a plataforma Dialogflow em aplicativos Android. A principal maneira de fazer isso é implementando corretamente os métodos da SDK da plataforma e exibir as informações de diálogo entre usuário e *bot* em um layout que siga o modelo padrão de um chat comum.

O projeto feito por [Marcus \(2018\)](#), é uma aplicação Android que implementa um mecanismo de reconhecimento de voz para ser usado como entrada do usuário na interação com o *bot*. Na implementação feita por ele é feita a comunicação do usuário através de fala e a resposta do *bot* é exibida em formato texto na tela. Essa aplicação tem como objetivo retornar o resultado de um jogo de loteria que o usuário solicitar. Outro projeto que visa simplificar a integração entre aplicativos Android e a plataforma é a aplicação feita por [Tyagi \(2018\)](#). Segundo o autor, existem tutoriais e documentos suficientes disponíveis na internet para o aprendizado e criação de *chatbots* da plataforma. No entanto, o suporte da Google para as SDKs Android e iOS não são bem feitas. Seu projeto visa simplificar e facilitar o uso da plataforma através de sua aplicação que consiste em retornar ao usuário as informações de um evento conforme é solicitado. Além dos recursos básicos da plataforma, ele ainda utiliza uma função específica definida na própria plataforma a qual chamamos de *cloud function fulfillment* que é processada no serviço de nuvem do Google. De acordo com o dado específico informado pelo usuário a função é processada e devolve

uma determinada resposta relacionada aquele dado.

Outra abordagem que visa solucionar o problema é o *framework* Kommunicate. Podemos observar seu uso na prática no projeto de Mamgain (2018). Nele é implementado um dos modelos disponíveis do *framework* focado em aplicações comerciais. O objetivo da plataforma é simplificar o processo de codificação para utilizar o Dialogflow. Além disso, fornece uma plataforma amigável bem como modelos práticos para lidar com clientes dos mais diversos tipos de negócio conforme observamos na Figura 1.

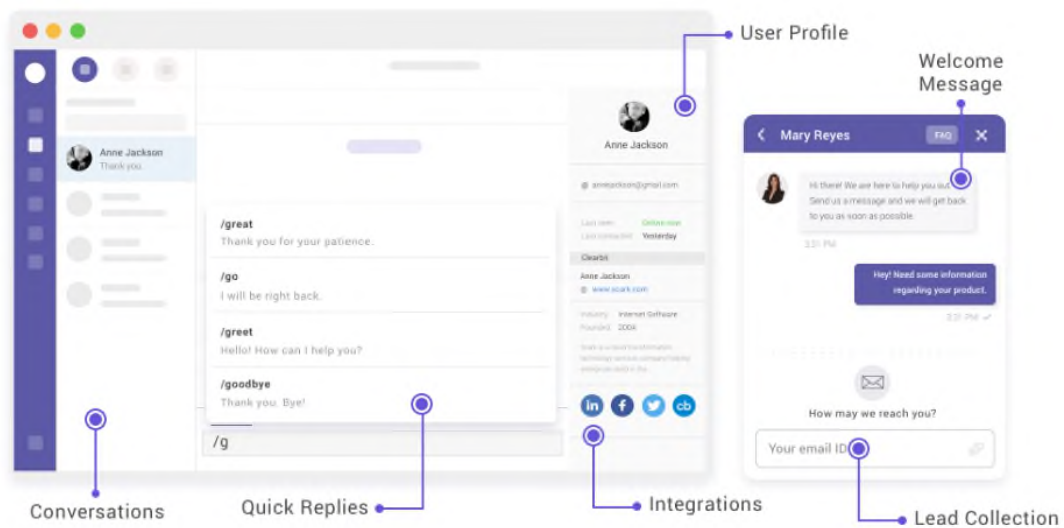


Figura 1 – Ilustração da plataforma Kommunicate (KOMMUNICATE, 2019) .

1.5 Resultados

O resultado esperado deste trabalho será a inserção de um módulo Android com a funcionalidade de *chatbot* de texto e voz em um aplicativo genérico. A apresentação dos resultados quantitativos será acompanhada de análise direcionada ao contexto que configura o módulo Android e o programa de teste, de modo que se cumpra o papel científico deste trabalho, no sentido de alcançar os objetivos propostos.

2 Fundamentação Teórica

Em sistemas que seguem o paradigma de orientação a objetos, o reúso de software baseia-se na construção de novas classes e programas através do uso de componentes já existentes. Uma das principais técnicas de reúso, utilizadas no contexto atual, são os *frameworks*. Neste capítulo, serão abordadas definições e conceitos de *frameworks* orientados a objetos e sua aplicabilidade e contextualização com a API Dialogflow.

2.1 *Frameworks* Orientados a Objeto

Quando se fala em reúso de software é importante destacar que os *frameworks* tornaram-se ferramentas fundamentais no desenvolvimento de sistemas orientados a objetos, auxiliando programadores a resolver problemas usando soluções já existentes. Segundo [Johnson \(1997\)](#), “um *Framework* é um padrão reutilizável para todo ou parte de um sistema, que representa através de conjuntos de classes abstratas como as instâncias destas interagem”.

Já [Fayad e Schmidt \(1997\)](#) define *framework* como “uma aplicação reusável e semi completa que pode ser especializada para produzir aplicações customizadas”. Vale destacar que para [Markiewicz e Lucena \(2001\)](#), “*frameworks* não são executáveis. [...] São geradores de aplicações que estão diretamente relacionados a um domínio específico, isto é, uma família de problemas relacionados”.

Deste modo, diversas aplicações que tenham funcionalidades em comum podem servir de base à construção de um *framework*, incluindo nele a capacidade de personalização de algumas partes do código e automação de tarefas que não sejam simples do ponto de vista do programador.

Segundo [Johnson \(1997\)](#), o desenvolvedor de software não precisa conhecer o código ou detalhes de implementação, uma vez que isso facilita o processo de aprendizagem no uso do *framework*. Assim, o trabalho do desenvolvedor consiste em fornecer, através dos pontos de acesso do *framework*, um comportamento específico que atenda aos seus interesses.

2.1.1 Hotspots

Os pontos de acesso do *framework* são chamados de *hotspots*. Sem eles não seria possível a interação do usuário com o *framework* visto que, para instanciá-lo é necessário implementar um código específico na aplicação usando tais pontos.

Hotspots são classes abstratas, interfaces ou simplesmente métodos que necessitam

ser implementados para o bom funcionamento do *framework*. É através de uma chamada de retorno que um código implementado pelo usuário poderá comunicar-se com o código do *framework*. Desta forma, na ocorrência de determinado evento este código é executado, e assim, posteriormente, com a execução do *framework* será fornecido o serviço esperado pelo usuário (MARKIEWICZ; LUCENA, 2001).

No *framework* proposto, um dos *hotspots* é o método em que o usuário informa a mensagem desejada para posterior execução de sintetização de fala (TTS). Consequentemente, a mensagem é exibida no layout da aplicação em forma de texto, bem como o áudio resultante na voz padrão do Google. É importante ressaltar, que dentro do próprio *framework* existem *hotspots* sendo implementados, diretamente, chamando a API utilizada.

Estes e outros *hotspots*, tais como reconhecimento de voz e personalização de resposta aos eventos provenientes da API Dialogflow serão discutidos com mais detalhes ao longo do desenvolvimento deste trabalho.

2.1.2 Coldspots

Num domínio de aplicações Android que utilizam chatbots, apesar de cada aplicativo ter suas peculiaridades, toda a base de desenvolvimento, como invocação de métodos que utilizam recursos nativos do dispositivo, eventos que lidem com interface gráfica e *layout*, são geralmente as mesmas. Neste contexto, temos o conceito de *coldspot*, que consiste em ter toda essa lógica básica implementada de modo que ela não seja modificada pelos usuários que utilizarão o *framework*. Ao contrário dos *hotspots*, os “pontos frios” são pedaços de código imutáveis que fazem parte do núcleo do *framework*. Seu papel consiste em chamar os *hotspots*, quando necessário, apresentando parte de cada instância do *framework* (MARKIEWICZ; LUCENA, 2001).

2.1.3 Classificação

No desenvolvimento de *frameworks*, existem 3 tipos de classificação que relacionam as características do paradigma de orientação a objetos no que diz respeito à forma de implementação e nível de acesso do sistema. Nesta subseção serão abordados conceitos referentes aos três tipos de classificação, assim como suas diferenças e aspectos.

2.1.3.1 Caixa Branca

Frameworks caixa branca utilizam os conceitos de herança e ligação dinâmica, a fim de que o usuário através de uma subclasse tenha a responsabilidade em definir o comportamento da superclasse presente no *framework*. Esta abordagem permite ao usuário o conhecimento de detalhes internos da implementação de modo a prover mais

liberdade para que possam ser feitas alterações na implementação existente na superclasse. Do ponto de vista do desenvolvedor do *framework*, essa estratégia torna o projeto mais simples de implementar, visto que o usuário deve ter um entendimento a respeito de como o *framework* funciona para usá-lo. Isso faz com que a usabilidade seja mais presente em *frameworks* caixa cinza e preta.

Para exemplificar esse conceito, no trabalho das autoras [Assis e Maciel \(2008\)](#), é ressaltada a dificuldade no desenvolvimento de uma *middleware* CORBA (arquitetura utilizada em sistemas distribuídos) de um *framework* caixa branca. Segundo as autoras, a compreensão e o estudo dos requisitos e funcionalidades avançadas do CORBA foram cruciais na construção do projeto. Apesar disso, um dos pontos positivos encontrados foi a facilidade em instanciar diversas aplicações com diferentes funcionalidades visto que havia um conhecimento a respeito do padrão de desenvolvimento.

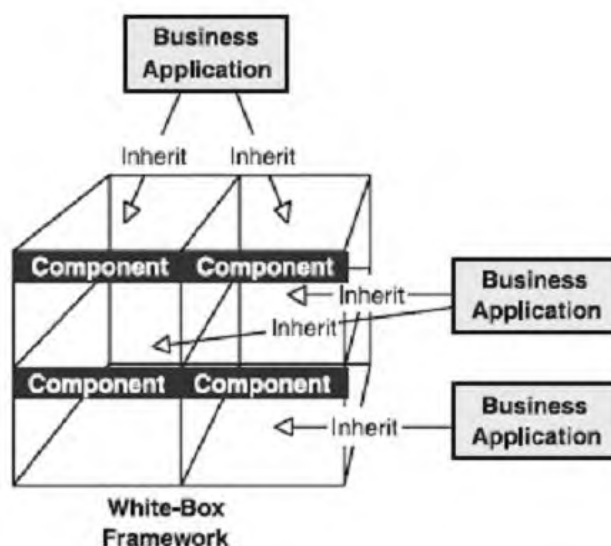


Figura 2 – Arquitetura de um *framework* caixa branca ([PURBIA, 2012](#)).

2.1.3.2 Caixa Preta

Diferentemente dos *frameworks* caixa branca, estes ocultam do usuário detalhes referentes à implementação da superclasse, de modo que o usuário interage com uma interface composta por componentes de software fixada na arquitetura do *framework*.

Um dos exemplos mais comuns de *frameworks* desse tipo, que [Andrade \(2004\)](#) cita, são os formulários em aplicações *web*. Segundo o autor, esse tipo de *framework* necessita possuir flexibilidade em sua composição a fim de alcançar o reúso desejado. Temos ainda *frameworks* que não só implementam formulários, como também fornecem uma interface completa e responsiva a vários outros elementos de uma aplicação *web*. O Bootstrap é um exemplo do tipo caixa preta que implementa essas características.

Criado no Twitter, o Bootstrap teve como origem a busca por resoluções em inconsistências presentes nos códigos trabalhados, visto que não havia uma estrutura padronizada a ser seguida dentro da empresa. Com a criação do Bootstrap, a equipe do Twitter passou a utilizar uma única estrutura padrão para desenvolver os projetos, e com isso houve um ganho de performance ([SILVA, 2014](#)).

Diante do exemplo apresentado, é possível estabelecer a relação entre usabilidade e flexibilidade no reúso de *frameworks*. *Frameworks* caixa-preta facilitam o uso e padronizam estruturas no desenvolvimento de software e são úteis em domínios previsíveis, já o uso de *frameworks* do tipo caixa branca, apesar de reduzir o poder de utilização imediata e aumentar o nível de exigência de conhecimento do código reutilizado, fornece maior maleabilidade em domínios de difícil previsão dos requisitos.

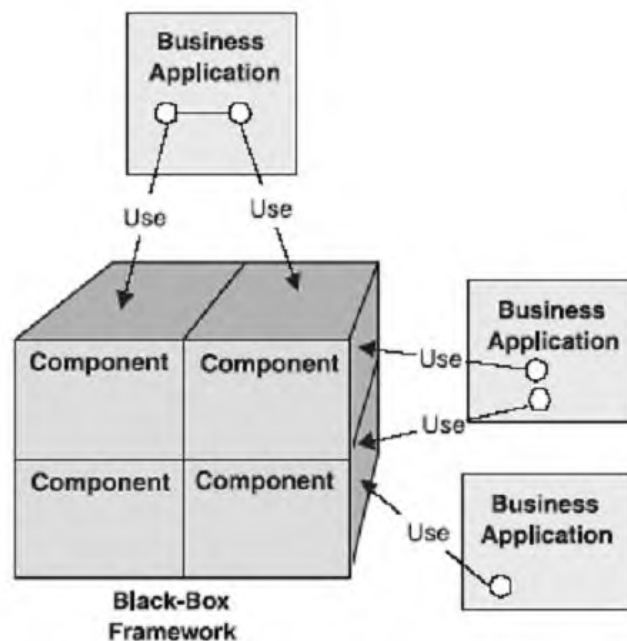


Figura 3 – Arquitetura de um *framework* caixa preta ([PURBIA, 2012](#)).

2.1.3.3 Caixa Cinza

No reúso de software, um dos aspectos mais importantes que um *framework* deve possuir é a usabilidade. Esta característica que determina o sucesso ou fracasso de um projeto designado para esse fim. Entretanto, apesar da facilidade de se projetar um *framework* caixa branca, muitos desenvolvedores buscam aprimorar o sistema de modo que o software torne-se, futuramente, um *framework* caixa preta.

Podemos citar como exemplo o trabalho das autoras [Assis e Maciel \(2008\)](#) discutido numa das subseções anteriores. Nele, é sugerido como um trabalho futuro este tipo de conversão, de modo que os objetos do *framework* passem a invocar métodos implementados nos objetos do usuário no momento da instanciação das aplicações.

Ao longo deste processo, o código pode apresentar características que impossibilitam definir precisamente se o *framework* é caixa preta ou caixa branca. Esses são denominados do tipo caixa cinza, ou seja, além de prover uma interface bem definida com o usuário através de composição de objetos, ele também permite alterações internas relevantes dentro da superclasse utilizada.

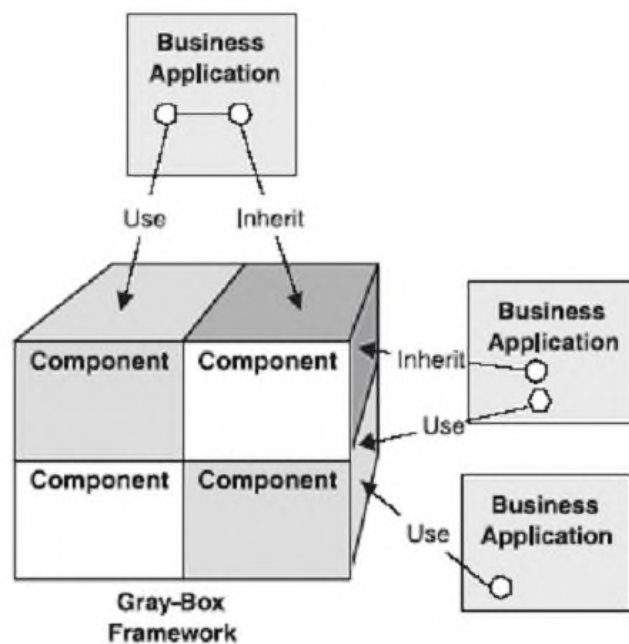


Figura 4 – Arquitetura de um *framework* caixa cinza ([PURBIA, 2012](#)).

2.2 Documentação de *frameworks* OO

A fim de garantir a base de sustentação do *framework* proposto neste trabalho, é necessário, em primeiro lugar, conceituar e detalhar diferentes tipos de abordagens na documentação de *frameworks* e apresentar heurísticas que guiem a produção da documentação utilizada.

Documentar um *framework* consiste em fornecer um guia formal contendo informações relevantes acerca dos resultados e conhecimentos gerados pelo software com objetivo de reutilizá-lo. Detalhes importantes como exemplos de utilização, instruções no uso dos pontos de acesso e definições da arquitetura do sistema devem ser levadas em conta para o bom entendimento do programador.

Os autores [Chen e Xu \(2002\)](#) ressaltam que uma boa documentação deve ser efetiva, compreensível e precisa de modo que o usuário do *framework* possa entender os processos envolvidos e seus resultados. Caso ela não esteja adequada, informações valiosas que dizem respeito ao entendimento dos requisitos da aplicação podem ser perdidas ou prejudicadas. Portanto, para que o programador consiga construir um software concreto e personalizado é primordial que seu aprendizado seja consolidado por meio de uma documentação bem escrita, sucinta e sem ambiguidades.

A documentação, além de servir de guia, economiza tempo de desenvolvimento e auxilia na manutenibilidade do sistema. Para isso, segundo [Chen e Xu \(2002\)](#), é fundamental o entendimento do domínio da aplicação, sua arquitetura e propósito de cada *hotspot* presente no sistema de software. O usuário do *framework* deve conhecer onde encontram-se os componentes do *framework*, bem como os comportamentos que os objetos apresentam, seus relacionamentos e interações. [Chen e Xu \(2002\)](#) lembram que, por ser uma tarefa onerada, a documentação de um *framework* deve ser feita ao mesmo tempo em que é desenvolvido o software reutilizável. Na medida em que melhorias são realizadas, a documentação deve ser atualizada. Isso proporciona uma orientação mais bem fundamentada para que erros de entendimento e implementação não sejam cometidos.

Neste trabalho, por exemplo, a inclusão do tratamento de ações fornecidas pelo Dialogflow no código do *framework* gerou um novo *hotspot*. Consequentemente, foi feita uma atualização no documento de maneira a tratar esse novo ponto de acesso a fim de que o usuário tenha conhecimento e implemente corretamente sua personalização. De certa forma, é importante que o fornecimento de mecanismos simples de personalização de *hotspots* atenda tanto às funcionalidades gerais quanto às específicas.

Para os autores [Chen e Xu \(2002\)](#), uma documentação primorosa e entendível são cruciais no bom aproveitamento do uso de um *framework* altamente escalável. Um documento em que o usuário entenda o propósito do *framework*, como utilizá-lo e como funciona a arquitetura provê uma facilidade na manutenção, no aumento de suas funcio-

nalidades e na inclusão de novas pessoas no desenvolvimento do *framework*.

2.3 Chatbots

Para um *bot* ser capaz de entender o que o usuário comunica existem dois mecanismos :

- Correspondência de padrões de entrada do usuário;
- Classificação de intenções usando processamento de linguagem natural (PLN).

Segundo a [ActiveWizards \(2019\)](#), o primeiro consiste em definir um conjunto de respostas possíveis para um de entradas previstas. Esta abordagem é mais simples de ser implementada, no entanto não abrange os inúmeros tipos de cenários que um *bot* pode encontrar caso necessite de fornecer respostas mais flexíveis. Em situações em que o usuário não descobre o conjunto estruturado de respostas ocorre a dificuldade de agir do usuário ([DIALOGFLOW, 2019](#)).

O segundo, ao contrário do primeiro, necessita de técnicas de aprendizado de máquina aplicadas às entradas fornecidas, a fim de que seja feita uma interpretação adequada para que o bot responda da maneira esperada a um determinado evento. Este mecanismo torna a implementação mais difícil de ser feita, porém existem diversas plataformas disponíveis que implementam a solução bastando apenas o conhecimento de alguns conceitos de PLN como veremos adiante ([ActiveWizards, 2019](#)). Como o primeiro mecanismo possui limitações quanto à escalabilidade, o segundo será utilizado por este trabalho.

CHATBOT CONVERSATION FRAMEWORK



Figura 5 – Ilustração dos tipos de *frameworks* de chatbots ([ActiveWizards, 2019](#)) .

2.3.1 Frameworks e APIs

Para a reutilização de código de plataformas que implementem um *chatbot* que utiliza PLN, o autor [ActiveWizards \(2019\)](#) nos apresenta como opção tanto o uso de *frameworks* como o de serviços específicos de inteligência artificial. Segundo ele, os *frameworks* consistem em abstrair a parte do fluxo necessário de um *chatbot* com o fim de facilitar sua implementação em determinada linguagem de programação. O *framework* proposto, por exemplo, se encaixaria nessa classificação, visto que soluciona a codificação da parte lógica e visual necessária para o uso da aplicação em dispositivos Android.

Quanto aos serviços especializados em inteligência artificial, o autor destaca como principais características as funcionalidades de armazenamento em nuvem de dados de conversação, uma interface gráfica amigável na configuração de modelos conversacionais e suas capacidades de processamento em linguagem natural. A comunicação entre esses serviços e as aplicações que os consomem ocorre via API *REST*.

2.3.2 Dialogflow

Dialogflow é uma API de criação de agentes inteligentes que utilizam processamento e entendimento de linguagem natural para construção de interfaces conversacionais que atuem em diversos cenários. Sua implementação é suportada em diferentes tecnologias, incluindo dispositivos móveis ([JÚNIOR et al., 2018](#)). No site oficial da documentação do [DialogFlow \(2019\)](#) é possível encontrar SDKs presentes em várias plataformas, dentre elas podemos citar Android, C++, Javascript, Python, Unit, iOS, entre outras. Além disso, é de fundamental importância o entendimento dos principais conceitos que envolve

as interfaces conversacionais da plataforma, dentre eles podemos citar os *agents*, *intents*, *entities* e os *contexts*. *Entities* tem como função categorizar ideias e objetos, tais como cores, números e cidades por exemplo (DIALOGFLOW, 2019). Os outros conceitos serão discutidos com mais detalhes nas subseções subsequentes.

2.3.2.1 Agents

Para que o *framework* deste trabalho entenda os padrões de saída e seus significados gerados pela API, é necessária a definição de um agente que capture e entenda a entrada do usuário levando em consideração as mais diversas nuances da linguagem humana. Ou seja, é preciso processar a entrada do usuário em um dado estruturado que possa ser usado para retornar uma resposta adequada. Para isso, o Dialogflow estrutura um agente em um conjunto de padrões de requisições denominado *intent*. Essas *intents* são cruciais para definição do mapeamento de entradas para um conjunto de uma ou mais respostas correspondentes (DIALOGFLOW, 2019).

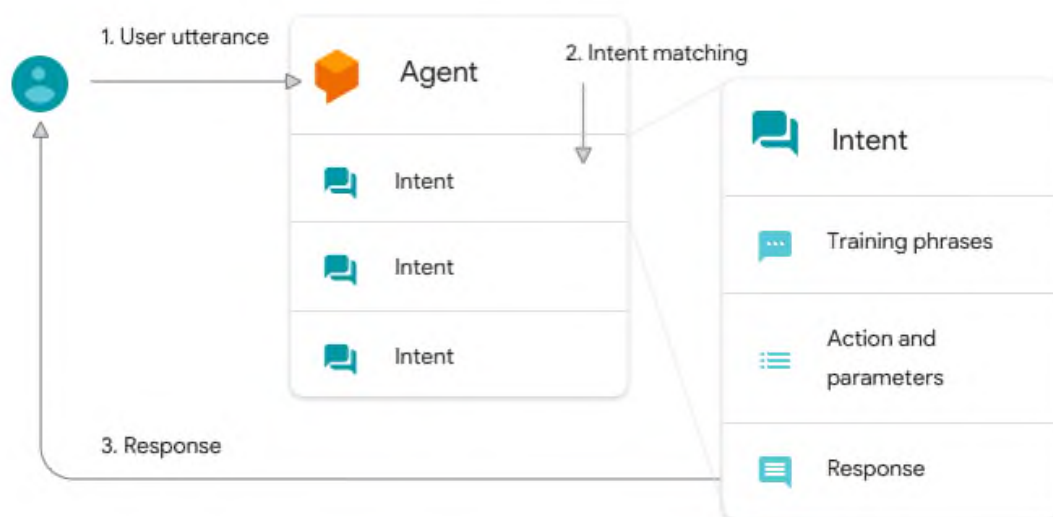


Figura 6 – Ilustração do funcionamento dos *agents* na plataforma Dialogflow (DIALOGFLOW, 2019).

Na documentação oficial do DialogFlow (2019) é dado como exemplo um agente de previsão do tempo que fornece previsões atendendo requisições as quais variam conforme o local onde a pessoa que está interagindo solicita. Essas variadas requisições são traduzidas em uma única *intent* que trata de modo geral o processamento da resposta da previsão do tempo. Cabe portanto, ao programador que integrar a API, em fornecer nesse caso uma consulta de clima através de uma API REST para retornar a resposta esperada pelo usuário (DIALOGFLOW, 2019).



Figura 7 – Ilustração do exemplo de agente de previsão do tempo (DIALOGFLOW, 2019).

2.3.2.2 *Intents e Contexts*

Conforme destaca a [ActiveWizards \(2019\)](#), existem dois conceitos principais da plataforma na construção de diálogos: *intents* e *contexts*. *Intents* são elos construídos para unir modelos de entradas do usuário e possíveis respostas do *bot*. Além disso, é possível definir *actions* e *parameters* que envolvem a interação definida no modelo.

Uma *intent* é composta por três componentes principais que discutiremos com detalhes a seguir:

- Frases de treinamento - Para amplificar a precisão de reconhecimento do *chatbot* são definidas frases que os usuários podem dizer, com isso o agente do DialogFlow reconhece frases similares naturalmente (DIALOGFLOW, 2019).
- Respostas - São as saídas retornadas pelo *bot* na plataforma. Elas podem tanto ser feitas diretamente em um texto fixo, como serem processadas extraindo dados específicos de entrada numa lógica denominada *fulfilment* (DIALOGFLOW, 2019).
- *Actions* e *parameters* - Modelos conversacionais que utilizam processamento de linguagem natural não trabalham apenas com a correspondência de entradas literais fornecidas pelo usuário. Neste contexto surge a necessidade da criação de meios que tratem *intents* e que usem entidades tanto na entrada quanto na resposta do *bot*. Ao definir entidades nas frases de treinamento, a plataforma, automaticamente, transforma-as em *parameters* de uma *action*. A fim de personalizar a resposta que o usuário recebe, esses *parameters* podem ser processados no *fulfilment* (DIALOGFLOW, 2019).

Cada *intent* pode possuir uma única *action* capaz de trabalhar com inúmeros *parameters*. Neste trabalho, será detalhado, no Capítulo de desenvolvimento, o uso e implementação desses conceitos.

Contexts são strings utilizadas na plataforma para melhorar o fluxo de diálogo nas *intents* definidas. Uma *intent* pertencente a um *context* só é acionada se o diálogo estiver no *context*. Suponha que exista uma *intent* que retorne para o usuário informações sobre o clima de determinada região conforme solicitado. É possível criar um *context* “clima” nessa *intent* de forma a possibilitar a definição de novas *intents* dependentes que reutilizem seus parâmetros sem a necessidade do usuário informá-los a cada requisição. Ou seja, definir novos possíveis diálogos a respeito do clima que o *bot* consiga armazenar em memória informações que o usuário passou em uma *intent* relacionada. Quando a *intent* contendo o *context* é acionada, as novas *intents* ligadas a esse *context* podem ser disparadas (ActiveWizards, 2019).

2.3.2.3 Fullfilment

Segundo a documentação do DialogFlow (2019), o agente criado na plataforma escolhe utilizar dois tipos de resposta. Uma delas é feita de forma estática e pode ser útil em casos genéricos, a outra é delegada para o *fullfilment*. *Fullfilment* é uma lógica opcional usada pela plataforma para retornar respostas mais apropriadas e inteligentes de acordo com os dados extraídos na *intent* do usuário. Implementamos o *fullfilment* por meio de um processador de requisições de serviços denominado *webhook*. Um *webhook* consiste em um código executado no servidor do DialogFlow capaz de processar as *intents* recebidas podendo repassá-las para o *backend* os dados extraídos nelas. Os *parameters* da *intent* são enviados em formato JSON para que o *fullfilment* realize tarefas definidas na plataforma, como por exemplo a chamada de uma API REST para manipulação dos *parameters* contidos no JSON transmitido. Consequentemente, o *fullfilment* retornará ao DialogFlow a resposta gerada pelo serviço que ela chamou, podendo ser um texto simples ou uma resposta mais elaborada, contendo imagens e outros dados. Caso o *fullfilment* não consiga retornar uma resposta, o DialogFlow acionará seu mecanismo conhecido como *fallback*, retornando então uma resposta padrão definida para esses casos (DIALOGFLOW, 2019).

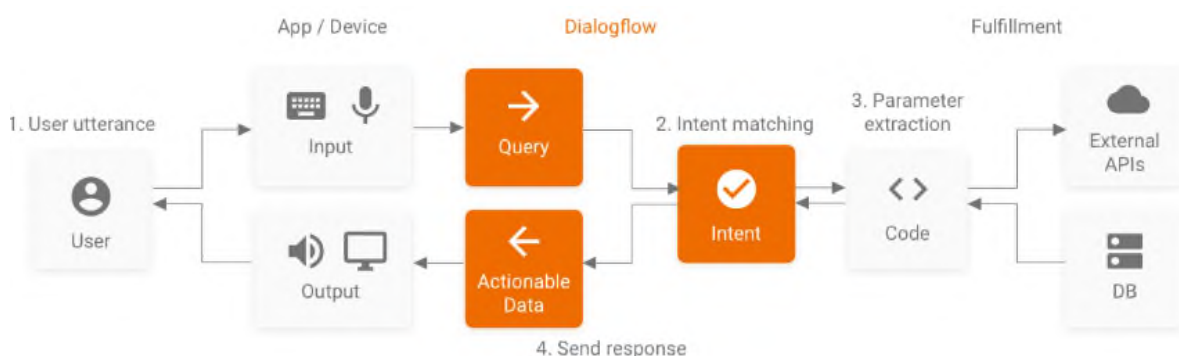


Figura 8 – Ilustração do funcionamento do *fullfilment* na plataforma DialogFlow (DIALOGFLOW, 2019).

3 Desenvolvimento da aplicação

Neste capítulo serão explicados os passos utilizados no desenvolvimento do *framework* proposto. Com base na facilidade de uso, integração com aplicativos Android e manipulação de modelos conversacionais, a API Dialogflow foi escolhida. Com ela foi possível construir um modelo conversacional utilizando os recursos discutidos na subseção [2.3.2](#).

3.1 Escolher o tipo de *framework*

Para este projeto que propõe a implementação de um *framework* de *chatbot*, foi selecionado desenvolver um *framework* do tipo caixa cinza. Do ponto de vista de design gráfico da aplicação teremos um *layout* previamente definido no *chatbot*, porém fica a critério do usuário poder modificar os elementos gráficos existentes no *layout* desenvolvido, de modo a não prejudicar o funcionamento dos já existentes. Este aspecto nos mostra a característica principal de *frameworks* do tipo caixa branca, que no caso seria a responsabilidade do usuário em modificar o *layout* XML para atender suas necessidades. Seu conhecimento de detalhes internos de implementação seria fundamental, já que o modo com que o aplicativo interage com os elementos do *layout* pode ser afetado se a modificação não for feita de maneira correta.

Quanto ao papel do usuário em utilizar o modelo conversacional criado na API, sua usabilidade é vista como simples, pois o usuário não necessita ter conhecimento técnico de como funciona a comunicação do aplicativo com a SDK Android do Dialogflow, basta ele usar os *hotspots* de forma direta. Sendo assim, observamos aspectos distintos numa mesma aplicação, caracterizando o *framework* como do tipo caixa cinza.

3.2 Modelo conversacional

Antes de iniciar o desenvolvimento do *framework*, é necessário definir um modelo conversacional de teste a ser consumido na aplicação de *chatbot*. Definindo o modelo no Dialogflow conseguiremos a chave de acesso necessária para configurar nosso código de teste. O modelo conversacional proposto é o de um *bot* atendente de pizzeria. Seu papel é extrair os dados necessários do pedido do cliente para efetuar o armazenamento em banco de dados de um pedido válido. Para isso é necessário a criação de um fluxo de *intents* que capture as escolhas desejadas do usuário do aplicativo. Veremos com mais detalhes sua implementação no Capítulo [4](#).

3.3 *Layout da aplicação*

Um dos principais aspectos de uma aplicação de *chatbot* é sua interface gráfica com usuário. Interfaces clássicas de uma conversa entre duas pessoas, geralmente, possuem algo que diferencia quais mensagens foram escritas tanto pelo remetente, quanto pelo destinatário. O *layout* do *framework* implementa essas e outras características nos seguintes arquivos definidos a seguir:

- *Activity Splash Screen* - *Layout* de tela inicial da aplicação. Assim que o aplicativo inicia, a tela de abertura do *framework*, ou *SplashScreen*, é instanciada exibindo uma imagem de início enquanto o aplicativo carrega em segundo plano.
- *Activity Main* - *Layout* principal do *framework*. Interface responsável em exibir na tela o campo *EditText* para inserção de mensagem e o botão que dispara o evento de envio. Este botão, quando pressionado, inicialmente dispara um evento de gravação de voz através do uso do microfone do dispositivo enviando o áudio gravado para o Dialogflow. Caso o usuário comece a escrever no campo *EditText* o botão passa a disparar outro evento de clique, passando a requisição para a API em formato de texto. Além disso, são exibidas as mensagens da conversa entre o usuário e o *chatbot* em uma *ListView* a ser manipulada por uma classe do tipo *adapter*. Como veremos adiante na seção subsequente, o *adapter* será responsável em traduzir um objeto do tipo *ChatMessage* para que, através dos *layouts right.xml* e *left.xml* as mensagens possam ser bem posicionadas a fim de diferenciar remetente e destinatário.

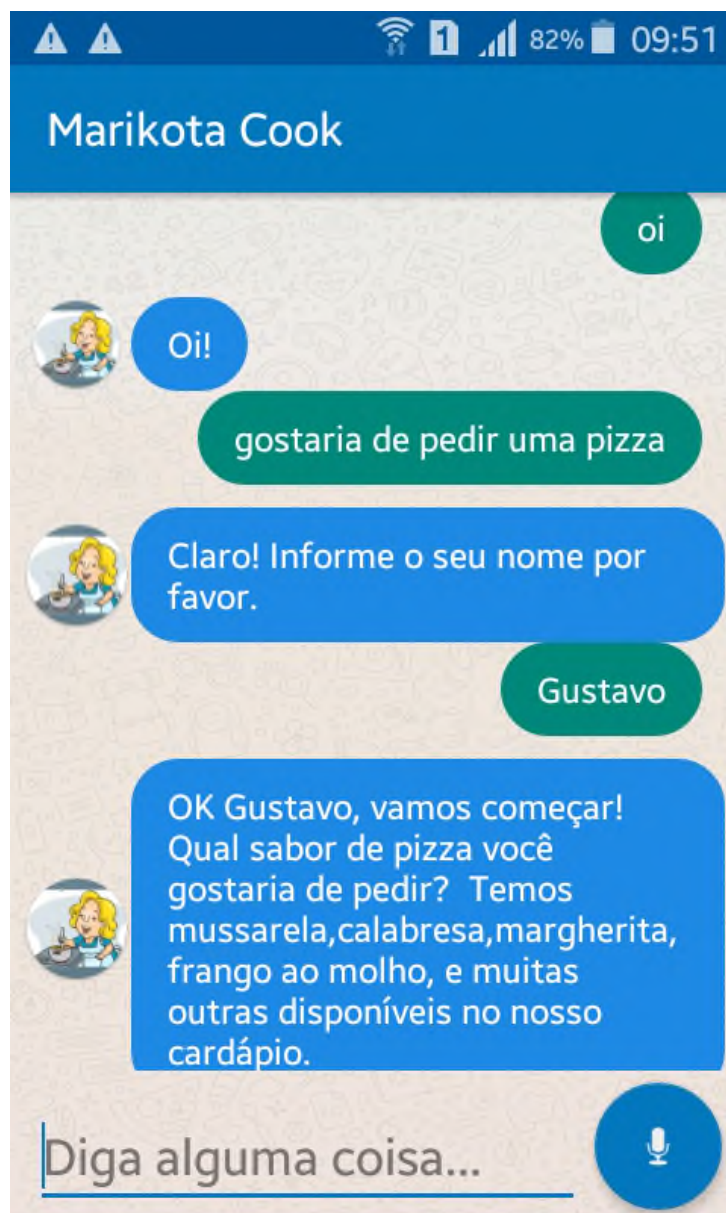


Figura 9 – Mensagens da conversa exibidas no *layout*.

Como observamos nas Figuras 9 e 10, o fluxo de conversa criado no modelo conversacional, anteriormente definido, é exibido na tela de forma a diferenciar as falas colocando as do *chatbot* em um elemento de fundo azul e as do usuário em um elemento de fundo verde. Como o *framework* é do tipo caixa cinza, vale ressaltar que a foto de fundo do *layout*, a foto exibida na *splash screen* e o ícone do *chatbot* pode ser alterado pelo usuário do *framework* diretamente na pasta do módulo, bem como as cores dos fundos.



Figura 10 – Finalização do pedido de pizza sendo exibido no *layout*.

3.4 Diagrama de classes

Com o intuito de formalizar as classes do *framework* é criado um diagrama de classe para que haja entendimento do código. Ele nos ajuda a compreender quais classes são necessárias no projeto, e quais são seus relacionamentos na aplicação desenvolvida. Além disso, o diagrama de classes orienta o programador num nível mais alto evitando que, simplesmente, ele seja forçado a compreender níveis detalhados de implementação de código. Fatores como elementos de interface gráfica no Android (como por exemplo um *RadioButton*) e outras implementações complicadas podem interferir no entendimento de aspectos essenciais do *framework*, caso ele queira personalizar seu comportamento num nível mais baixo de detalhe. O modelo de diagrama foi estruturado em cinco classes presentes no *framework* e três classes que devem ser implementadas pelo usuário. Cada uma delas possui características conforme seguem:

- TTS - Classe responsável em utilizar recursos de áudio nativos do Android para que, dado um contexto e texto de entrada, seja disparada a voz padrão do Google (a mesma que escuta-se no *Google Maps*). Os métodos dessa classe são utilizadas de modo estático, ou seja, não é necessária a sua instanciação no momento em que é utilizada;
- Acao - Classe abstrata utilizada para definir comportamentos personalizados em

actions, definidas no modelo conversacional e criado na plataforma Dialogflow. É no método `onResult()` (método sobrescrito da API Dialogflow) da classe `MainActivity` que a classe é utilizada. Como o usuário não possui acesso direto na implementação desse método, a classe permite o uso de código personalizado através da sobrescrita dos métodos `retornaParametro()`, `executaAcao()` e `RetornaRespostaBot()` em uma subclasse criada pelo usuário a qual definiremos a seguir;

- **AcaoTeste** - Classe concreta que o usuário do *framework* precisa criar para personalizar comportamentos de *actions* definidas no projeto do Dialogflow. Como o diagrama apresenta, seu atributo “nome” visa representar o nome dado à *action* criada na plataforma. O método `retornaParametro` recebe como argumento o nome do parâmetro correspondente à *action* utilizada. Já o método `executaAcao()` tem como objetivo ser sobrescrito para interligar a implementação do usuário com o método `onResult()` da classe `MainActivity`, visto que o utilizador do *framework* não teria acesso direto a ele;

Quanto ao método `RetornaRespostaBot`, sua intenção é recuperar a *string* completa da resposta recebida pela API para a ação disparada. Com essa informação o usuário será capaz de definir um comportamento para uma resposta específica retornada, caso assim deseje.

- **ChatArrayAdapter** - No desenvolvimento de aplicações Android é comum a criação de classes que interligam componentes de interface gráfica com os dados que serão exibidos nestes componentes. Essas classes são denominadas “*adapters*” e servem como elo entre um *layout* personalizado e os dados a serem mostrados neste componente. [Silva \(2015\)](#) por exemplo, utiliza um *adapter* contendo um *ArrayList* de objetos que tem como função carregar os dados desses objetos numa lista definida no *layout* da aplicação.

Semelhante a isso, temos no *layout* do *framework* uma estrutura contendo uma lista de mensagens, e conforme definido na seção de *Layout* é preciso um *adapter* que carregue um *ArrayList* com as informações das mensagens emitidas, tanto pelo remetente quanto pelo destinatário, de forma a diferenciá-los na tela. Para isso, utilizamos no *ArrayList* objetos instanciados da classe `ChatMessage` a qual será definida a seguir.

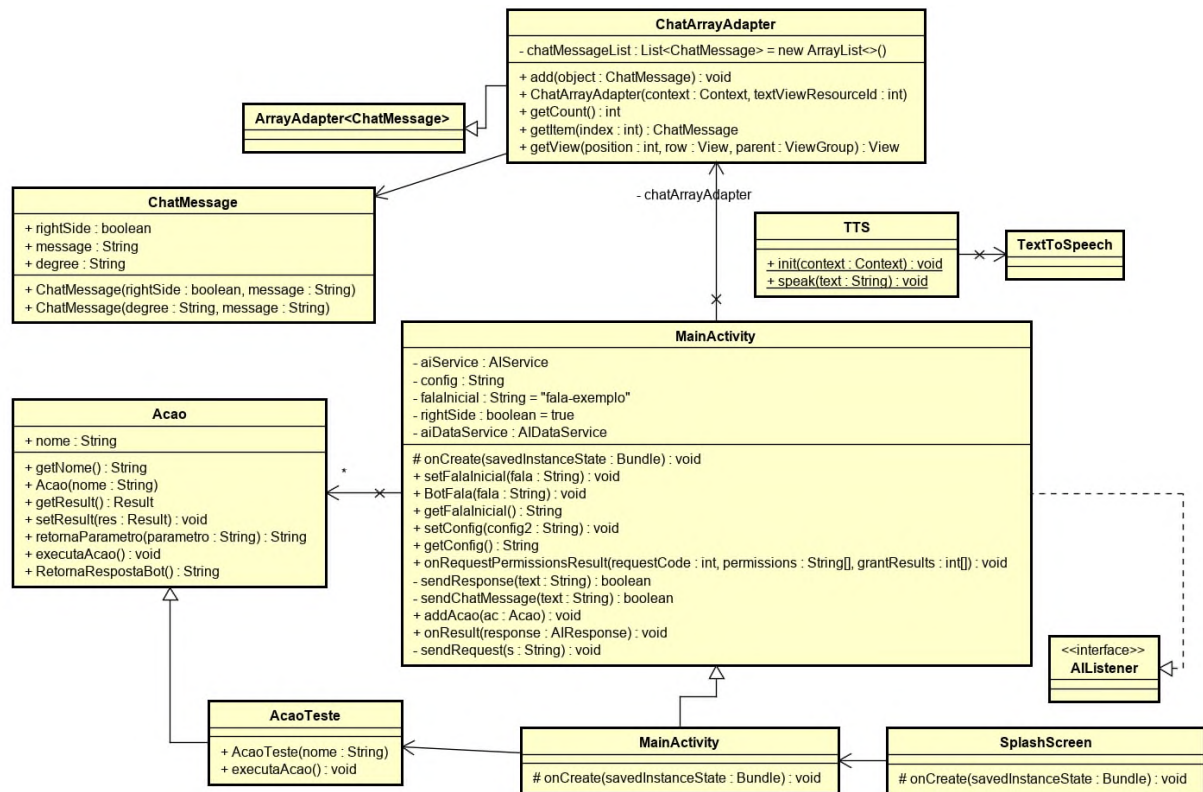
- **ChatMessage** - Classe responsável por passar ao *adapter* a mensagem a ser exibida na tela e o emissor da mesma. Com isso, o *adapter* conseguirá posicionar adequadamente no *layout* as mensagens a serem mostradas para o usuário da aplicação. Isso se faz através do atributo do tipo booleano “rightSide”, que informa se a mensagem deve ser exibida no lado direito ou esquerdo da tela, e do atributo “message”, responsável por receber a mensagem a ser exibida e passá-la para o *adapter* manipular;

- **MainActivity** - Classe principal do *framework* responsável tanto pela integração das classes anteriormente descritas, quanto pela interação do código com os *layouts* XML definidos. O papel dela é sobrescrever os métodos da API Dialogflow e utilizar os métodos das classes da API, tais como as classes “AIService” e “AIDataService”, para efetivar a comunicação entre o *framework*, a interface gráfica da aplicação e a plataforma Dialogflow.

Através dessa classe, temos a base necessária para que a *activity* do usuário do *framework* utilize seus recursos através de herança. O presente estudo passa à essa *activity* de mesmo nome, porém não é necessário usá-lo;

- **MainActivity** - Subclasse da classe principal do *framework* presente no código do usuário. Através dela é possível implementar os *hotspots* definidos, definir a chave de acesso gerada na plataforma e inserir ações criadas em classes do tipo **AcaoTeste**, como definido anteriormente;
- **SplashScreen** - Classe a ser implementada pelo usuário. Seguindo a documentação, o usuário encontrará um modelo de código para chamada da subclasse da classe principal do *framework*. Este consiste em configurar o *layout* atual da aplicação para o *layout* inicial do *framework* através do método `setContentView()`. Consequentemente, é invocada uma *thread* chamando a **MainActivity** do usuário. Sem essa classe, a execução da classe principal do *framework* seria feita antes da execução da classe do usuário, e com isso problemas seriam encontrados no fluxo correto de execução.

O diagrama de classes é ilustrado na Figura 11.

Figura 11 – Diagrama de classes do *framework*.

3.5 Hotspots

Nesta seção definiremos os pontos de acesso do *framework* desenvolvido. Estes pontos são de suma importância para que o usuário possa desenvolver um código personalizado que possibilite a interação com a implementação do *framework*.

3.5.1 setConfig

O ponto de acesso `setConfig()` é o método responsável em definir um atributo do tipo *string* contendo a chave de acesso do agente (definido no Dialogflow) na classe principal do *framework*. Ao colocar o valor da chave de acesso no atributo, este será recuperado através da chamada do método `getConfig()` no corpo do método `onCreate()` da classe, e passado como parâmetro na instanciação da classe “AIConfiguration”. O objeto instanciado desta classe é utilizado como argumento tanto na obtenção da instância da classe “AIService” (feita no método estático `getService()`) quanto na instanciação da classe “AIDataService”, como observamos na Figura 12.

O objeto “aiService” será utilizado para implementar a integração de reconhecimento de voz do Dialogflow usando seu método `setListener()`. Por meio deste método, especificamos qual classe, que necessariamente implementa a interface “AIListener”, receberá a notificação dos eventos do método da interface. Isso é feito passando como

parâmetro na chamada da função a *activity* principal do *framework* através do termo *this*.

```
final AIConfiguration config = new AIConfiguration(this.getConfig(),
    AIConfiguration.SupportedLanguages.PortugueseBrazil,
    AIConfiguration.RecognitionEngine.System);

aiService = AIService.getService(this, config);
aiDataService = new AIDataService(this, config);
aiService.setListener(this);
```

Figura 12 – Trecho da instanciación de objetos das classes “AIService” e “AIDataService”.

3.5.2 setFalaInicial e BotFala

O método `setFalaInicial()` é responsável em passar para a *activity* principal do *framework* uma mensagem de boas vindas definida no código do usuário. A *string* contendo a mensagem será recuperada via método `getFalaInicial()` e passada como parâmetro no método `sendResponse()` presente no corpo do método `onCreate()` para que seja exibida na tela no início da execução do programa. Além de ser exibida em forma de texto, a mensagem será transmitida via voz através do método `speak()` da classe TTS. Com esses dois métodos a comunicação do aplicativo com o Dialogflow não é necessária, podendo ser chamados pelo usuário tanto nesse *hotspot* quanto no outro, o qual definiremos a seguir.

Com a necessidade de criação de um método que possibilite ao usuário personalizar suas próprias respostas sem a dependência da resposta esperada do servidor do Dialogflow, foi desenvolvido o método `BotFala()`. Neste método de natureza estática, o usuário pode invocar respostas que utilizem texto e voz tanto na subclasse principal quanto em qualquer outra classe que seja instanciada na mesma, bastando passar a fala desejada e o contexto da *activity* executada. O trecho de código dos métodos é exibido na Figura 13. Note que a chamada do método `getFalaInicial()` é feita dentro do método `onCreate()`.

```
        sendResponse(this.getFalaInicial()); //DENTRO DO MÉTODO onCreate
        TTS.speak(this.getFalaInicial());
    }
    public void setFalaInicial(String fala)
    {
        this.falaInicial = fala;
    }

    public static void BotFala(String fala, Context context){
        sendResponse(fala);
        TTS.init(context);
        TTS.speak(fala);
    }

    public String getFalaInicial(){
        return this.falaInicial;
    }
}
```

Figura 13 – Implementação dos hotspots `setFalaInicial()` e `BotFala()`.

3.5.3 executaAcao

Em aplicações Android, a maior parte dos códigos que não demandam alto processamento, são executados em cima de uma *thread* principal denominada *UI thread*. Sua principal prioridade é a exibição de elementos na tela, e realizar processamento pesado, como acesso a arquivos grandes, banco de dados e Internet. Podem causar problemas para liberação de recursos no momento em que é necessário atualizar os elementos de interface gráfica do aplicativo. Devido a esse problema, para submeter as requisições POST do usuário para o *agent* definido no Dialogflow via API REST, necessitamos implementar uma tarefa assíncrona que envie e receba os dados necessários de modo que sua execução seja feita em uma *thread* separada da principal, ou seja, que sua execução seja feita em segundo plano. Esta tarefa assíncrona é denominada *AsyncTask*.

Conforme afirma [Queiroz \(2016\)](#), estender uma classe *AsyncTask* envolve implementar o método `doInBackground()` - responsável por executar as tarefas desejadas em segundo plano - e o método `onPostExecute()` - responsável em fazer a sincronização da *thread* executada assincronamente com a *thread* principal. Os dois métodos são executados somente após chamada do método `execute()`. Nesse trabalho, como observado na Figura 14, essa tarefa assíncrona foi codificada dentro de um método denominado `sendRequest()`, que recebe como parâmetro a *string* da *query* que o usuário passou, através de texto ou voz, para que seja utilizada pela tarefa no método `doInBackground()` quando o método `execute()` for executado.

```
private void sendRequest(String s) {  
    final String queryString = s;  
    final AsyncTask<String, Void, AIResponse> task = new AsyncTask<String, Void, AIResponse>() {  
        private AIError aiError;  
        @Override  
        protected AIResponse doInBackground(final String... params) {  
            final AIRequest request = new AIRequest();  
            String query = params[0];  
            if (!TextUtils.isEmpty(query))  
                request.setQuery(query);  
            try {  
                return aiDataService.request(request);  
            } catch (final AIServiceException e) {  
                aiError = new AIError(e);  
                return null;  
            }  
        }  
        @Override  
        protected void onPostExecute(final AIResponse response) {  
            if (response != null) {  
                setResult(response);  
            } else {  
                onError(aiError);  
            }  
        }  
    };  
    task.execute(queryString);  
}
```

Figura 14 – Implementação da *AsyncTask* dentro do método `sendRequest()`.

O primeiro método a ser executado na *AsyncTask* é o `doInBackground()`, ele recebe a *string* contendo a requisição a ser repassada ao servidor do Dialogflow. Isso ocorre através do método `request()` da classe “*AIDataService*” instanciada anteriormente. Com esse envio, após término de execução do método anterior, o próximo a ser executado é o método `onPostExecute()`, que receberá a resposta vinda da API REST do Dialogflow. Com o objeto do tipo “*AIResponse*” obtido, o mesmo é repassado para o método `onResult()` caso ele não seja nulo. Caso contrário, o método `onError()` fica responsável em tratar a exceção gerada por este evento. É bom destacar que estes dois métodos são definidos na interface implementada “*AIListener*”.

Quando o usuário cria uma nova *action* para uma determinada *intent* na plataforma Dialogflow, é interessante, em alguns domínios de aplicação, a inclusão de um processamento inteligente no código das informações de entrada recebidas na API. Por exemplo: suponha um modelo conversacional de agendamento de reservas de hotel. Neste modelo é comum observar a necessidade do robô em receber dados do usuário como entrada no momento de agendamento. A partir desse momento o *bot* precisa processar esses dados e armazená-los ou realizar consultas em um banco de dados para que possa ser feito um controle do hotel em questão. Ao longo desse processo, se feito através de um *chatbot*, é fundamental que o processamento e armazenamento de dados seja feito entre o espaço de tempo em que a solicitação do cliente é realizada e a resposta do *chatbot* é enviada. Para esse e diversos outros cenários é possível utilizar *actions* no tratamento desse tipo de *intent*.

O uso de uma *action* em modelos conversacionais do Dialogflow pode receber diversos *parameters* recebidos na *intent*. O valor dos campos das *actions* é passado no *fullfilment* e pode ser utilizado posteriormente para disparar alguma lógica específica (DIALOGFLOW, 2019). Através do método `executaAcao()` o usuário do *framework* definirá essa lógica para cada *action* que criar, sendo possível obter os parâmetros envolvidos na *action* por meio do método definido na classe abstrata *Acao* denominado `retornaParametro()`. Se o usuário criar um *action* chamada “reservaQuarto” numa aplicação de hotéis, e nessa *action* exigir dados pessoais do cliente, tais como nome, cidade, número do cartão de crédito, ele necessariamente terá que definir esses dados como *parameters* de uma *action*, e no código do aplicativo definir no método `executaAcao()` um processo que use os valores de cada campo recebidos na *action* para armazená-los e efetivá-los de fato. Consequentemente será retornada uma resposta válida ao cliente contendo a confirmação da operação.

O fluxo de execução na *activity* principal do *framework* para tratar as respostas que chegam da API é feito através de sobrescrita do método `onResult()` presente na interface a qual a classe implementa. Através desse método, serão tratadas as respostas que chegam da API para que as *actions* sejam identificadas e disparadas.

Conforme a Figura 15, podemos observar o modo como as *actions* do usuário se encaixam na execução do método. É criado um *ArrayList* contendo os objetos do tipo *Acao* que terão seu método `executaAcao()` executado caso a ação recebida no objeto “Result” da classe “AIResponse” tenha sido chamada na plataforma Dialogflow. É importante salientar que o nome definido na subclasse concreta de *Acao* deve ser o mesmo da *action* definida na plataforma. O *ArrayList* recebe esses objetos através do *hotspot* `addAcao()` a ser invocado pelo o usuário em sua classe principal.

```
public void onResult(final AIResponse response) {  
    result = response.getResult();  
  
    sendMessage(response.getResult().getResolvedQuery());  
  
    for(Acao a : this.acoes) {  
        if(result.getAction().equals(a.getNome())){  
            a.setResult(result);  
            a.executaAcao();  
        }  
    }  
}
```

Figura 15 – Trecho do método sobrescrito `onResult()`.

3.6 Documentação para instanciação do *framework*

A documentação de *frameworks*, atualmente, corresponde a um padrão adotado pelas grandes empresas que disponibilizam seus códigos *open source*. A abordagem mais adotada pelas empresas de software é disponibilizar seu código e documentação no site mais popular de versionamento e compartilhamento de código conhecido como GitHub. O Google, por exemplo, com o Dialogflow, disponibiliza uma documentação básica que trata o modo de implementar os pontos de acesso da API. Além do Google, temos outras empresas que seguem essa abordagem conforme a Figura 16 nos mostra. Neste trabalho seguimos essa abordagem, e ao disponibilizarmos o nosso código em um repositório público e gratuito, permitimos o uso aberto do *framework* desenvolvido.

A documentação produzida, inicialmente, introduz o conceito do *framework*, detalhando sua utilidade e resumindo seu funcionamento. Após isso, são mostrados dois passos fundamentais antes de iniciar o uso efetivo do *framework*. Um deles é a configuração da chave de acesso fornecida pelo Dialogflow, após criação do modelo conversacional na plataforma. O outro passo é incluir nas dependências do aplicativo a referência à pasta contendo o módulo Android para que a mesma possa ser reconhecida pelo Gradle e compilada junto ao projeto do aplicativo. Posteriormente, a documentação passa a mostrar um tutorial de como utilizar o código exemplo desenvolvido para uso do *framework*. A documentação orienta o usuário do *framework* utilizar tanto o código de exemplo do repositório, quanto a importação do módulo em seu próprio aplicativo. A documentação completa está presente no apêndice A.

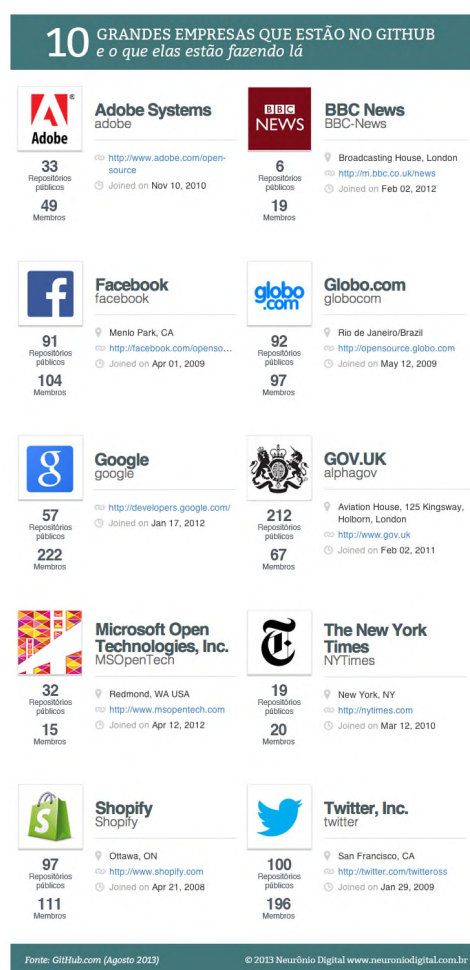


Figura 16 – Grandes empresas que disponibilizam seus códigos no Github (RIBEIRO, 2013).

4 Resultados

Neste capítulo será apresentado o resultado da implementação de um programa de teste usando o framework desenvolvido bem como a documentação produzida. Primeiramente, serão apresentados os dados obtidos em pesquisa realizada com desenvolvedores de *software* acerca da documentação e código disponibilizados [aqui](#). Posteriormente, é apresentado o estudo de caso que utiliza o *framework* a fim de verificar o funcionamento do sistema.

4.1 Validação

Após considerada a finalização do *framework* e documentação, foi feito um questionário *online*, constituindo como objeto de estudo apenas desenvolvedores de *software* que possuem familiaridade com alguma linguagem de programação orientada a objetos.

A pesquisa concentra sua análise em duas dimensões específicas: o funcionamento e usabilidade do sistema e a qualidade da documentação produzida. Para tanto, a elaboração do questionário se deu junto a 19 programadores que responderam perguntas baseadas na escala de Lickert.

Entre os desenvolvedores que responderam, como vemos na Figura 17, 47,4% concordam totalmente que a documentação está clara e compreensível. 42,1% apenas concordam, enquanto os outros 10,5% posicionaram-se com neutralidade a respeito do assunto. Isso revela o quão fundamental é o uso de exemplos de código e procedimentos de preparação de ambiente de desenvolvimento em documentações de *frameworks*.

A documentação do framework está clara e fácil de entender

19 respostas

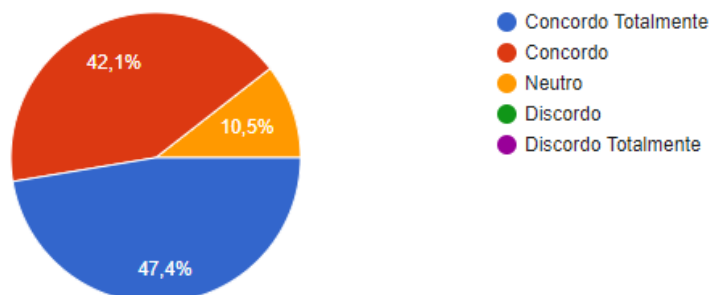


Figura 17 – Gráfico de pizza ilustrando o resultado da pergunta a respeito da documentação.

Outro ponto abordado com respeito à documentação, refere-se a sua capacidade em demonstrar o uso personalizado de *actions* nas respostas retornadas ao usuário. 68.4% afirmam concordar que a documentação esclarece possíveis dúvidas quanto ao uso de *actions* no desenvolvimento. Já 26,3% posicionaram-se de forma neutra ao assunto, enquanto que 5,3% discorda da afirmativa conforme observamos na Figura 18. Esses dados revelam que, por um lado, o uso de *actions* no aplicativo pode ser útil em casos em que é possível definir comportamentos personalizados, visto que é possível manipular parâmetros obtidos na interação. Por outro lado, caso o usuário não tenha claro o conceito de uso dessa funcionalidade torna-se difícil sua implementação.

A documentação é capaz de resolver dúvidas pertinentes ao uso personalizado de Actions criadas no DialogFlow

19 respostas

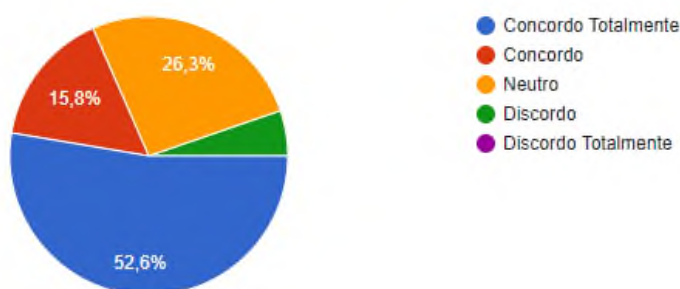


Figura 18 – Gráfico de pizza ilustrando o resultado da pergunta a respeito do uso personalizado de *actions* no código.

Quanto ao código do sistema foram feitas duas perguntas: uma a respeito da capacidade do *framework* em integrar um aplicativo Android à plataforma Dialogflow, e outra sobre o funcionamento e usabilidade no reuso do módulo. Na primeira, 94,7% dos usuários (somando as porcentagens das respostas “Concordo” e “Concordo totalmente”) concorda com a afirmação, enquanto que apenas 5,3 % posicionaram-se neutros a essa questão. Isso comprova a eficácia obtida com a simplificação do uso da SDK Android que o Dialogflow fornece. Com respeito à segunda pergunta, apenas 10,5 % posicionou-se de maneira neutra no questionamento, enquanto os outros 89,5% posicionou-se de forma favorável à afirmação. A partir dessa validação, o uso do *framework* torna-se relevante para implementação de *chatbots* em aplicações para dispositivos móveis usando a API Dialogflow. Os gráficos contendo os dados das duas perguntas se encontram nas Figuras 19 e 20. O estudo referencia o questionário completo no apêndice B.

o framework tem bom funcionamento e usabilidade

19 respostas

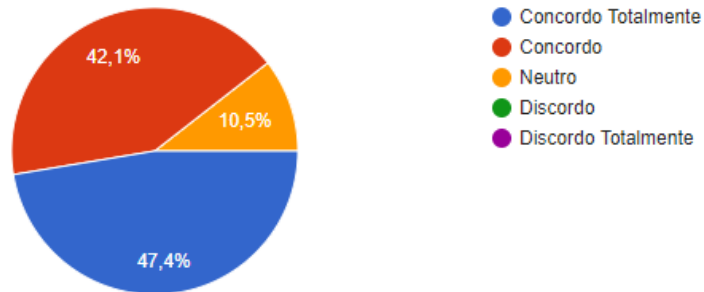


Figura 19 – Gráfico de pizza ilustrando o resultado da pergunta a respeito do funcionamento e usabilidade do *framework*.

O framework é capaz de integrar um projeto criado na plataforma DialogFlow com sucesso

19 respostas

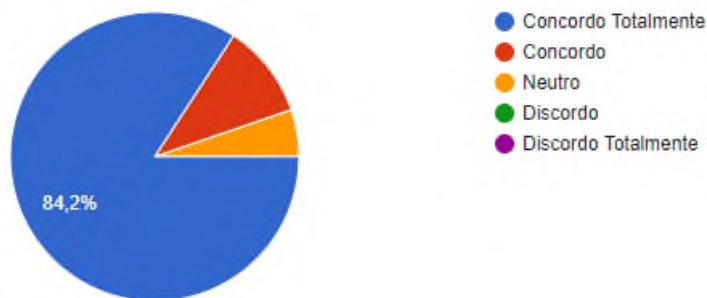


Figura 20 – Gráfico de pizza ilustrando o resultado da pergunta a respeito da integração do Dialogflow com o código.

4.2 Estudo de caso

Nesta seção será abordado o desenvolvimento de um modelo conversacional na plataforma Dialogflow de forma a utilizar os principais recursos fornecidos por ela. Como introduzido na subseção 3.2, é usado como estudo de caso um *chatbot* atendente de pedidos de pizza. Inicialmente, é construído um fluxograma para guiar a construção de *intents*, *contexts*, *entities* e os *parameters* usados ao longo da interação. Cada *intent* e *context* criado será discutido com mais detalhes nesta seção.

Posteriormente, na *intent* final referente à finalização do pedido, com a confirmação do mesmo, o presente estudo apresentará o mecanismo de inserção do pedido em um banco de dados em nuvem que o Google integra com a plataforma denominado **Firestore**.

4.2.1 Fluxo conversacional

Antes da criação de *intents* é necessário um cuidado em relação ao diálogo que o *bot* terá com as pessoas na interação. Para um atendente de pedidos de pizza, um passo fundamental é definir a conclusão de um pedido capturando as informações necessárias de forma gradual, ou seja, para extrair os dados do cliente e do pedido é preciso a simulação de um diálogo de atendimento da forma como é feita tradicionalmente em pedidos por telefone ou aplicativos de entrega. Com a construção de um fluxograma, presente na Figura 21, obtemos os detalhes necessários a serem seguidos na conclusão do pedido, e cada desvio que possa ocorrer nesse fluxo deve ser tratado de forma correta na plataforma Dialogflow.

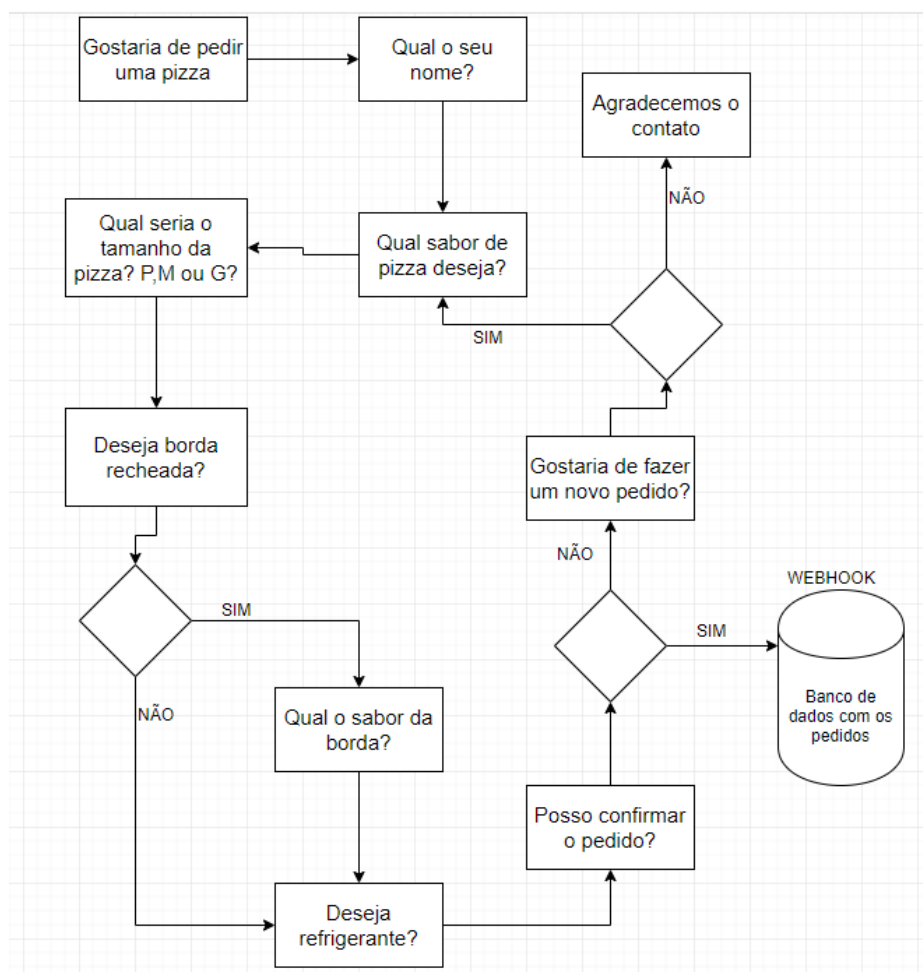


Figura 21 – Diagrama do fluxo conversacional do atendente de pedidos de pizza

Como é observado na figura, o fluxo inicia-se com o cliente informando que gostaria de realizar um pedido. O *bot* neste momento pergunta ao cliente seu nome para que esta informação seja propagada até a conclusão do pedido. Além disso é solicitado o sabor da pizza que será pedida, e posteriormente seu tamanho. No momento seguinte configura-se uma bifurcação na inclusão opcional de borda recheada na pizza. Caso a resposta seja

afirmativa o cliente deverá informar na resposta da próxima *intent* qual sabor deseja para a borda, caso contrário será lhe perguntado se o pedido inclui refrigerante. Observe no fluxo, que após o questionamento da inclusão de borda recheada na pizza, independente de qual resposta seja obtida, o cliente deverá receber a pergunta a respeito do refrigerante. Com a captura dessas informações é feita a solicitação de confirmação do pedido, o cliente pode tanto confirmar (e ocorrer a persistência dos dados do pedido em banco de dados) quanto cancelá-lo recebendo como resposta a opção de iniciar um novo pedido. Por fim é decidido se o fluxo se encerra ou retorna ao ponto de escolher o sabor da pizza de um novo pedido.

4.2.2 Criação de *intents* e *contexts*

A primeira *intent* a ser disparada, de acordo com o fluxograma construído na seção anterior, possuirá como entrada uma das frases de treinamento presentes na Figura 22. Consequentemente, o usuário receberá como resposta uma mensagem solicitando a informação do nome do cliente ao qual o pedido estará associado. Observe que são criados dois *contexts* a serem utilizados nas próximas *intents* do fluxo. O *context* “ctx-pedido” será utilizado ao longo de todo o fluxo, sendo o responsável em guardar os *parameters* referentes aos dados passados pelo cliente até a efetuação ou cancelamento do pedido. Já o *context* “ctx-nome” será utilizado apenas para capturar o nome do usuário através da *intent* “pedido.nome” (mostrada na Figura 23). Note que em cada *context* existe um valor associado a ele que define seu tempo de vida na interação, este tempo é denominado *lifespan*. Dependendo do caso é preciso definir um valor adequado para esse tempo.

- pedido.novo

Contexts ?

5 ctx-pedido ✕

1 ctx-nome ✕

Events ?

Training phrases ?

Responses ?

DEFAULT +

Text response

1

Claro! Informe o seu nome por favor.

2

Enter a text response variant

ADD RESPONSES

Figura 22 – *Intent* “pedido.novo”.

Para o *context* “ctx-pedido”, em cada *intent* seu tempo de vida é de 5 interações, com isso, na *intent* de confirmação de pedido este *context* fornecerá todos os dados necessários para inserção do pedido em banco de dados. Contudo, em *contexts* como o “ctx-nome” teremos como valor ideal de *lifespan* apenas uma interação. Isso se deve ao fato de que, uma vez capturado o nome não é preciso mais disparar a *intent* novamente, visto que sua frase de treinamento consiste em receber uma *entity* de sistema chamada *sys.any* guardada em um campo “nome”. Portanto, caso seu tempo de vida fosse maior seria possível o agente disparar essa *intent* novamente. O mesmo aplica-se às próximas *intents* receptoras de *parameters*. É importante ressaltar que para quaisquer *parameters*

pedido.nome

Contexts ?

ctx-pedido ctx-nome Add input context

5 ctx-pedido 1 ctx-sabor-pizza Add output context

Events ?

Training phrases ?

” Add user expression

” @sys.any:nome

” meu nome é @sys.any:nome

Action and parameters

Enter action name

REQUIRED ?	PARAMETER NAME ?	ENTITY ?
☐	nome	@sys.any
☐	Enter name	Enter entity

[+ New parameter](#)

Responses ?

DEFAULT +

Text response

1

 OK Snome, vamos começar! Qual sabor de pizza você gostaria de pedir? Temos mussarela,calabresa,margherita,frango ao molho, e muitas outras disponíveis no nosso cardápio.

Figura 23 – *Intent* “pedido.nome”.

a serem propagados através do *context* “ctx-pedido”, é preciso definí-lo como *context* de entrada nas *intents* subsequentes, bem como de saída até a *intent* final.

pedido.sabor_pizza

Contexts ?

ctx-pedido

ctx-sabor-pizza

Add input context

5

ctx-pedido

1

ctx-tamanho-pizza

Add output context

Events ?

Training phrases ?

”

Add user expression

”

calabresa

”

Eu gostaria de calabresa

Action and parameters

REQUIRED ?	PARAMETER NAME ?	ENTITY ?
☐	sabor_pizza	@sabor_pizza
☐	Enter name	Enter entity

+ New parameter

Responses ?

DEFAULT

+

Text response

1

Ok #ctx-pedido.nome , Ssabor_pizza é uma boa pedida. Qual tamanho você gostaria?

2

Enter a text response variant

ADD RESPONSES

Figura 24 – Intent “pedido.sabor-pizza”.

pedido.tamanho_pizza

Contexts ?

Events ?

Training phrases ?

Action and parameters

REQUIRED ?	PARAMETER NAME ?	ENTITY ?	VALUE
☐	tamanho_pizza	@tamanho_pizza	\$tamanho_pizza
☐	Enter name	Enter entity	Enter value

[+ New parameter](#)

Responses ?

DEFAULT +

Text response

- Certo #ctx-pedido.nome , pizza \$tamanho_pizza . Deseja borda recheada por mais 5 reais?
- Enter a text response variant

Figura 25 – Intent “pedido.tamanho-pizza”.

pedido.borda-sim

Contexts ?

ctx-pedido

ctx-borda

Add input context

5

ctx-pedido

1

ctx-sabor-borda

Add output c

Events ?

Training phrases ?

”

Add user expression

”

desejo

”

claro

Responses ?

DEFAULT +

Text response

1 Ok, adicionaremos borda. Qual sabor deseja? Temos de chocolate, banana, 4 queijos e cheddar.

2 Enter a text response variant

Figura 26 – *Intent* “pedido.borda-sim”.

Na *intent* “pedido.nome” é definido um *context* de saída com *lifespan* igual a 1 para que a *intent* responsável por capturar o sabor da pizza possa ser disparada neste momento e uma única vez para cada pedido. Observamos este *context* na entrada da *intent* “pedido.sabor-pizza” conforme observado na Figura 24. Nessa *intent* suas frases de treinamento recebem como *parameter* uma *entity* que representa os sabores disponíveis de pizza que o usuário poderá pedir. A resposta gerada nessa interação usa o *parameter* “nome” obtido no início da interação e o sabor de pizza passado na frase de treinamento. Assim é possível armazenar os valores e perguntar ao cliente qual tamanho de pizza ele gostaria. O tratamento dessa questão será feita na *intent* posterior através de um novo *context* denominado “ctx-tamanho-pizza”.

Da mesma forma que a *intent* “pedido.sabor-pizza” recebeu do usuário o sabor desejado, a *intent* “pedido.tamanho-pizza” recebe o tamanho da pizza do pedido a ser feito. Neste momento o tamanho é representado pela *entity* “tamanho-pizza” e retornado na resposta para o cliente juntamente com seu nome novamente. Desta vez é feito o ques-

pedido.sabor-borda

Contexts ?

ctx-pedido

ctx-sabor-borda

Add input context

5

ctx-pedido

1

ctx-refrigerante

Add

Events ?

Training phrases ?

Add user expression

chocolate

REQUIRED	PARAMETER NAME	ENTITY	VALUE
☐	borda	@borda	Sborda
☐	Enter name	Enter entity	Enter value

New parameter

Responses ?

DEFAULT +

Text response

1

ok, sabor Sborda na borda recheada. Deseja guaraná mineiro 2 litros por mais 6 reais?

Figura 27 – Intent “pedido.sabor-borda”.

pedido.borda-nao

Contexts ?

ctx-pedido ✕ ctx-borda ✕ Add input context

5 ctx-pedido ✕ 1 ctx-refrigerante ✕ Add

Events ?

Training phrases ?

” Add user expression

” quero não

” não, obrigado

” não gosto

Responses ?

DEFAULT +

Text response

1	Ok, não adicionaremos borda. Deseja guaraná mineiro por mais 6 reais?
2	Enter a text response variant

ADD RESPONSES

Figura 28 – *Intent* “pedido.borda-nao”.

pedido.refrigerante

Contexts ?

ctx-pedido ✕ ctx-refrigerante ✕ Add input context

5 ctx-pedido ✕ 1 ctx-pedido-confirmacao ✕ Add

Events ?

Training phrases ?

” Add user expression

” não

REQUIRED	PARAMETER NAME	ENTITY	VALUE
☐	refrigerante	@refrigerante	Srefrigerante
☐	Enter name	Enter entity	Enter value

+ New parameter

Responses ?

DEFAULT +

Text response

- 1 Ok #ctx-pedido nome, então uma pizza #ctx-pedido tamanho_pizza de #ctx-pedido sabor_pizza certo? Posso confirmar?
- 2 Ok #ctx-pedido nome, então uma pizza #ctx-pedido tamanho_pizza de #ctx-pedido sabor_pizza com borda recheada de #ctx-pedido borda certo? Posso confirmar?

Figura 29 – *Intent* “pedido.refrigerante”.

tionamento ao usuário se o mesmo deseja borda recheada na pizza conforme observado na figura 25. Note pelas figuras 26 e 28 que o *context* de saída dessa *intent* será utilizado como de entrada em duas *intents* distintas que tratam respostas diferentes que o usuário possa transmitir. Isso também ocorre nas *intents* “pedido.confirmacao-nao” e “pedido.refrigerante” conforme ilustrado nas figuras 30, 31, 32 e 33.

A *intent* “pedido.borda-sim” será disparada se o usuário informar que deseja adicionar borda ao pedido. Dessa forma é criado um *context* de saída denominado “ctx-sabor-borda” que será usado como entrada na *intent* “pedido.sabor-borda”, como ilustrado na Figura 27. Esta *intent* será disparada somente caso o usuário tenha aceitado a inclusão do adicional ao pedido, caso contrário a *intent* que será disparada é a “pedido.borda-nao”, respeitando as regras do fluxograma criado.

The screenshot shows the configuration page for the Slack Bot Builder intent named "pedido.confirmacao_sim". The interface is divided into several sections:

- Contexts:** Includes input contexts "ctx-pedido-confirmacao" and "ctx-pedido", and an output context "5 ctx-pedido".
- Events:** A section for configuring events.
- Training phrases:** A list of phrases including "ok", "certo, pode", "certo", and "pode".
- Action and parameters:** Shows the action "pedido.gravar" and a parameter "Enter name" with a required checkbox.
- Responses:** A list of responses, including "OK #ctx-pedido.nome, pedido confirmado." and "Enter a text response variant".
- Fulfillment:** A toggle switch to "Enable webhook call for this intent".

Figura 30 – *Intent* “pedido.confirmacao-sim”.

Em “pedido.sabor-borda”, e “pedido-borda-nao” observamos um mesmo *context* de saída “ctx-refrigerante”, indicando que a próxima *intent* a ser disparada deve ser a “pedido.refrigerante”. Para que o usuário responda de modo que a *intent* seja invocada, na resposta das duas é preciso ter o questionamento de inclusão de refrigerante ao pedido. Feito isso a *intent* “pedido.refrigerante” guardará a escolha do usuário em um *parameter* “refrigerante” e retornará como resposta os dados do pedido, ou seja, os *parameters* obtidos nas *intents* anteriores (sabor, tamanho e borda). No final da resposta é acrescentada a pergunta ao cliente se o mesmo deseja confirmar a requisição.

Se o cliente desistir do pedido e não confirmá-lo na interação, teremos uma *intent* que perguntará ao usuário se o mesmo deseja iniciar um novo pedido. Repare que com a criação do *context* “ctx-novo-pedido-confirmacao”, o usuário poderá iniciar o pedido novamente sem informar o nome, visto que a *intent* “pedido.novo-confirmacao-sim” recebe este *context* como entrada, portanto o parâmetro “nome” será recuperado pelo agente.

pedido.confirmacao_ao

Contexts ?

ctx-pedido ⊗ ctx-pedido-confirmacao ⊗ Add input context

5 ctx-pedido ⊗ 1 ctx-novo-pedido-confirmacao ⊗ Add output context

Events ?

Training phrases ?

” Add user expression

” queria cancelar

” não

Responses ?

DEFAULT +

Text response

1 Ok pedido cancelado, gostaria de iniciar um novo pedido?

Figura 31 – Intent “pedido.confirmacao-nao”.

pedido.novo-pedido-confirmacao-nao

Contexts ?

ctx-pedido ⊗ ctx-novo-pedido-confirmacao ⊗ Add input context

Add output context

Events ?

Training phrases ?

” Add user expression

” não quero

” não

Responses ?

DEFAULT +

Text response

1 Ok, agradecemos seu contato.

Figura 32 – Intent “pedido.novo-confirmacao-nao”.

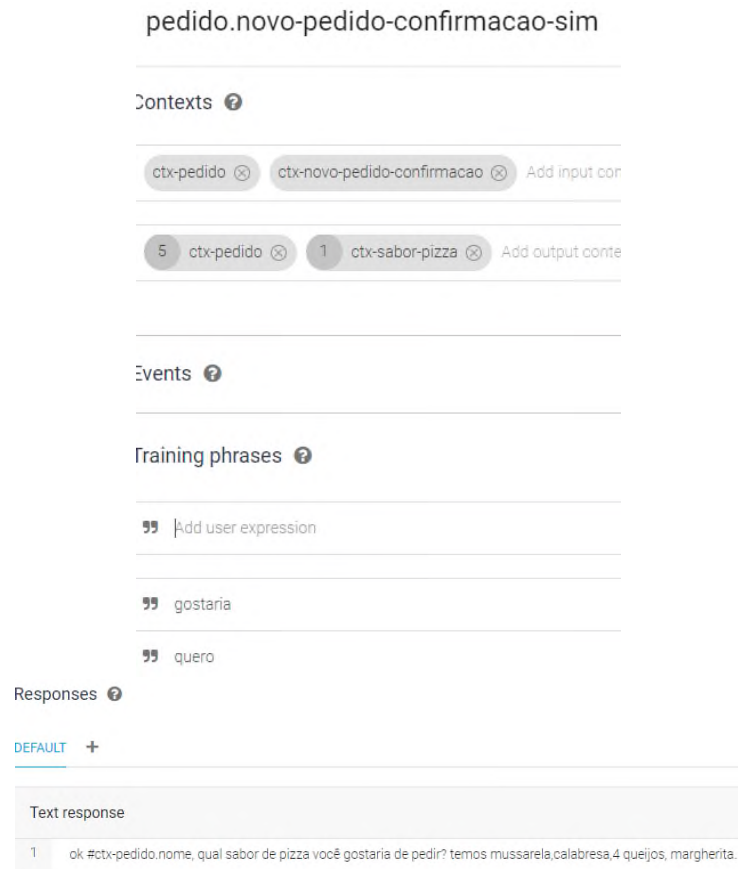


Figura 33 – *Intent* “pedido.novo-confirmacao-sim”.

Caso o cliente queira confirmar o pedido, a *intent* “pedido.confirmacao-sim” será invocada passando os *parameters* do *context* “ctx-pedido” num *fulfilment* que armazenará essas informações num banco de dados **Firestore**. Isso é feito através de um *webhook* escrito em **JavaScript** com base no código fornecido pela plataforma. A fim de que esse código seja executado para essa *intent*, é preciso definir um nome à *action* (“pedido.gravar”) e ativar a opção na seção *fulfilment*, como vemos na Figura 30. Observa-se na Figura 34, a importação do módulo necessário para integração com o **Firestore** e a recepção do JSON contendo os dados da *intent* processada. Deste modo, através da chamada da função `salvaPedido()` (definida como o *handler* da *intent* “pedido.confirmacao-sim”) é identificada a *action* “pedido.gravar” e criado um objeto “pedido” para inserção por meio do método `add()` na *collection* “Pedidos”. Após a execução do método, o **Firestore** retornará o objeto inserido com o identificador do registro do pedido. Assim, o código retorna a mensagem de resposta em formato JSON para a API, que por sua vez irá enviar ao aplicativo para que o mesmo exiba na tela para o usuário. Note pela Figura 35, a visualização da coleção dos registros que foram inseridos ao longo dos experimentos realizados.

```
function salvaPedido(agent){
  var action = acao;
  if(action == "pedido.gravar"){
    var parametros = contextoSaida.parameters;
    var pedido = {
      nome: parametros.nome,
      sabor_pizza:parametros.sabor_pizza,
      tamanho_pizza:parametros.tamanho_pizza,
      refrigerante:parametros.refrigerante,
      criadoEm: new Date().toISOString().slice(0,10)
    };
    if(parametros.borda){
      pedido.borda = parametros.borda;
    }else{
      pedido.borda = "não";
    }
    return db.collection('Pedidos').add(pedido).then(function(docRef){
      return agent.add(`Pedido confirmado com o ID:`+docRef.id);
    }).catch(function(error){
    });
  }else{
    return agent.add(`Não foi possível concluir a solicitacao`);
  }
}

const functions = require('firebase-functions');
const {WebhookClient} = require('dialogflow-fulfillment');
const {Card, Suggestion} = require('dialogflow-fulfillment');
const admin = require('firebase-admin');
admin.initializeApp(functions.config().firebase);
var db = admin.firestore();
process.env.DEBUG = 'dialogflow:debug'; // enables lib debugging statements

exports.dialogflowFirebaseFulfillment = functions.https.onRequest((request, response) => {
  const agent = new WebhookClient({ request, response });
  var acao = request.body.queryResult.action;
  var contextoSaida = request.body.queryResult.outputContexts[0];
```

Figura 34 – Código Javascript implementado no servidor do Google para inserção de pedido.

Pedidos	tVA3BXCMEodaHhEownt
+ Adicionar documento GUJiboFU31DBZNFzHUNW GncusUn6XLFe05dqDiUs r2Sa6P302tnI9LfCzHvy tVA3BXCMEodaHhEownt >	+ Adicionar coleção + Adicionar campo borda: "banana" criadoEm: "2019-06-23" nome: "Gustavo" refrigerante: "não" sabor_pizza: "margherita" tamanho_pizza: "grande"

Figura 35 – Pedidos armazenados no banco de dados Firebase.

5 Conclusão

O uso de plataformas de processamento de linguagem natural para criação de *chatbots*, mostrou aos desenvolvedores de *software* a importância da aplicação e integração delas em seus sistemas que possuem atendentes virtuais. Neste contexto, de modo a prover facilidade na escrita de código em aplicações *web* e *mobile*, surgem iniciativas que visam simplificar o processo de integração dessas plataformas. Uma delas são os *frameworks* que tratam a parte complexa de interagir com esses serviços, fornecendo reuso tanto de *design* quanto de código.

O objetivo principal deste trabalho consistia na construção de um framework para desenvolvimento de *chatbots* na plataforma Android. Esta construção foi feita por meio da criação de um módulo Android que possibilita a reutilização de classes e *layouts* XML em outros aplicativos. Entre as motivações para o uso deste tipo de *framework*, a facilidade em reusar um código que implementa os principais recursos de um *bot* e a possibilidade de personalização do mesmo são as principais. No *framework* desenvolvido neste trabalho, destaca-se o reconhecimento de voz, a sintetização de fala, sobrescrita dos métodos da SDK Android do Dialogflow (para integração entre aplicativo e plataforma), e a possibilidade de construir código que utilizem os *parameters* definidos nas *intents* criadas na plataforma. Sua implementação envolveu tratar eventos dos elementos de interface gráfica, tanto de entrada quanto de saída, bem como a comunicação via API REST executada por uma tarefa assíncrona que manipula requisições e respostas. Além disso, foi feita uma documentação contendo explicações e exemplos de código que permitem o usuário entender como realizar a instanciação do *framework* de forma correta usando seus pontos de acesso. Para avaliar sua eficácia foi realizada uma pesquisa

Uma vez que o framework foi desenvolvido e documentado, foi criado um estudo de caso que engloba como utilizar os principais recursos da plataforma Dialogflow. Nele observamos a construção ideal de um fluxo e suas respectivas *intents*, *contexts* e *parameters*. Além disso, é importante destacar a criação de um *webhook* para fazer uso do *fulfillment* a fim de inserir os dados obtidos do usuário em um banco de dados Firebase.

Uma das dificuldades encontradas para implementação do *framework*, foi encontrar um meio de instanciá-lo antes da execução da classe principal do aplicativo de teste, visto que inicialmente isso não ocorria. Para resolver esse problema foi criada uma *thread* numa classe intermediária à principal que possibilitou a instanciação corretamente. Outro empecilho encontrado foi a requisição de permissão do microfone no dispositivo. Para isso, foi investigada a maneira de requisitar a permissão ao usuário caso o mesmo não tenha autorizado, visto que a permissão de microfone, ao contrário da de internet, é considerada

como perigosa ([DEVELOPERS, 2019](#)).

Uma das lacunas que encontramos na integração completa entre aplicativos Android e Dialogflow, que poderia ser tratada em trabalhos futuros, seria a inclusão da possibilidade do usuário em adicionar *intents* e *entities* através de requisições HTTP para o servidor do Dialogflow. Isso torna a manutenção da configuração do modelo conversacional mais simples, pois não seria preciso a todo momento a edição do modelo através do site da plataforma. Por exemplo, se o usuário deseja adicionar um novo sabor de pizza ao seu modelo de atendente de pizzeria, via código seria possível a inclusão desse sabor.

Referências

- ActiveWizards. *Smartphone market share*. 2019. Disponível em: <<https://activewizards.com/blog/a-comparative-analysis-of-chatbots-apis/>>. Acesso em: 25.03.2019. Citado 4 vezes nas páginas 19, 20, 22 e 23.
- ANDRADE, M. V. d. Um framework para a criação de formulários dinâmicos. uma solução voltada a projetos de sistemas para a gestão municipal. 2004. Citado na página 16.
- ASSIS, S. R. de; MACIEL, R. S. P. Um framework de apoio ao desenvolvimento de aplicações baseadas em documentos. 2008. Citado 2 vezes nas páginas 15 e 17.
- CHEN, Q.; XU, F. *Framework Issue: Framework Documentation*. 2002. Disponível em: <https://www.cse.unr.edu/~chen_q/docu.html>. Acesso em: 2019-04-14. Citado na página 18.
- DEVELOPERS, A. *Permissões do sistema*. 2019. Disponível em: <<https://developer.android.com/guide/topics/security/permissions.html?hl=pt-br>>. Acesso em: 2019-07-03. Citado na página 55.
- DIALOGFLOW. *DialogFlow Docs*. 2019. Disponível em: <<https://dialogflow.com/docs/>>. Acesso em: 2019-05-16. Citado 6 vezes nas páginas 19, 20, 21, 22, 23 e 34.
- FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM*, ACM, v. 40, n. 10, p. 32–38, 1997. Citado na página 13.
- IDC. *Smartphone market share*. 2017. Disponível em: <<https://www.idc.com/promo/smartphone-market-share/os>>. Acesso em: 13.07.2018. Citado na página 9.
- JOHNSON, R. E. Frameworks=(components+ patterns). *Communications of the ACM*, ACM, v. 40, n. 10, p. 39–42, 1997. Citado na página 13.
- JÚNIOR, C. F. de C. et al. Chatbot: uma visão geral sobre aplicações inteligentes. *Revista Sítio Novo*, v. 2, n. 2, p. 68–84, 2018. Citado na página 20.
- KOMMUNICATE. *Features designed to help you serve your customers better*. 2019. Disponível em: <<https://www.kommunicate.io/features>>. Acesso em: 2019-06-08. Citado na página 12.
- MAMGAIN, D. D. *Build Android Chatbot with Dialogflow (API.ai)*. 2018. Disponível em: <<https://hackernoon.com/build-android-chatbot-with-dialogflow-api-ai-97d15eefe6>>. Acesso em: 2019-06-08. Citado na página 12.
- MARCUS, L. *Integrando o DialogFlow a um aplicativo Android*. 2018. Disponível em: <<https://luizmarcus.com/android/integrando-o-dialogflow-a-um-aplicativo-android/>>. Acesso em: 2019-05-16. Citado na página 11.
- MARKIEWICZ, M. E.; LUCENA, C. J. de. Object oriented framework development. *Crossroads*, ACM, v. 7, n. 4, p. 3–9, 2001. Citado 2 vezes nas páginas 13 e 14.

- MARTIN, V. *O chatbot veio para interagir mais os consumidores e as empresas*. 2017. Disponível em: <<http://www.abeoc.org.br/2017/03/o-chatbot-veio-para-interagir-mais-os-consumidores-e-as-empresas-vanessa-martin/>>. Acesso em: 2018-07-16. Citado na página 10.
- OLIVEIRA, S. F. G. *Interfaces conversacionais-chatbot para a casa da música*. 2017. Citado na página 10.
- PAIVA, F. *Brasil tem cerca de 8 mil bots que trafegam 500 milhões de mensagens por mês*. 2017. Disponível em: <<http://www.mobiletime.com.br/22/11/2017/brasil-tem-cerca-de-8-mil-bots-que-trafegam-500-milhoes-de-mensagens-por-mes/481702/news.aspx>>. Acesso em: 2018-05-16. Citado na página 10.
- PURBIA, D. *Difference between white box framework and black box framework*. 2012. Disponível em: <<https://www.sampleexamples.com/2012/04/difference-between-white-box-and-black.html>>. Acesso em: 2019-05-17. Citado 3 vezes nas páginas 15, 16 e 17.
- QUEIROZ, F. D. O. d. *Analisando o tratamento de exceções em aplicações android*. Dissertação (Mestrado) — Brasil, 2016. Citado na página 32.
- RIBEIRO, A. *10 grandes empresas que estão no GitHub e o que elas estão fazendo lá*. 2013. Disponível em: <<https://www.neuroniodigital.com.br/empresas-no-github/>>. Acesso em: 2019-06-08. Citado na página 36.
- SILVA, A. D. A. P. D. *Design responsivo: técnicas, frameworks e ferramentas*. 2014. Citado na página 16.
- SILVA, J. C. M. d. *Aplicativo de chat para aprendizado de idiomas vinculado ao facebook para dispositivos android*. Dissertação (B.S. thesis) — Universidade Tecnológica Federal do Paraná, 2015. Citado na página 28.
- TYAGI, A. *Android chatbot with Dialogflow*. 2018. Disponível em: <<https://medium.com/@abhi007tyagi/android-chatbot-with-dialogflow-8c0dcc8d8018>>. Acesso em: 2019-06-08. Citado na página 11.

Apêndices

APÊNDICE A – Documentação para instanciação do *framework*

O *framework chatbot* inteligente facilita a integração com a API de processamento de linguagem natural DialogFlow em dispositivos Android. Com ela podemos criar diferentes cenários de diálogos que interagem com o usuário diretamente através de *intents* e possíveis ações definidas pelo programador. Através do *framework* é possível definir os comportamentos resultantes de ações programadas bem como o uso de seus parâmetros.

Para utilizar o *framework* inclua em seu projeto a pasta `modulochatbot`, conforme ilustrado na Figura 37, presente no código fonte. Dentro do arquivo `build.gradle` presente na pasta `app` do projeto, inclua a referência da pasta do *framework* nas suas dependências.

```
compile project(path: ':modulochatbot')
```

Figura 36 – Trecho de código ilustrando a importação do *framework*.

Para integrar o aplicativo com o seu projeto do DialogFlow é preciso configurar o app passando seu *client access token* que é gerado pela plataforma e está presente na tela de configuração na parte inferior (Como ilustrado na Figura 37). Selecione na tela de configurações a versão 1 da api (V1 API).



Figura 37 – Tela de configurações do agente no Dialogflow.

A.1 Executando o código de exemplo do *framework chatbot* inteligente

O *framework chatbot* inteligente traz para o usuário um exemplo de código que exemplifica o uso de ações e parâmetros para que o usuário possa manipular e personalizar o comportamento do *bot* criado na plataforma DialogFlow. Use os seguintes passos para executar o código de exemplo:

- Crie um agente inteligente na plataforma DialogFlow;
- Abra o Android Studio;
- Vá até Ferramentas > Android > SDK Manager e certifique-se de que a *build tools* 25.0.0 está instalada. Caso não esteja, instale e a referencie no arquivo `build.gradle` presente na pasta `app` do projeto;
- Dentro da classe `MainActivity` presente na pasta `app/src/main/java/com/example/marikota` acrescente no método `setConfig()` seu *client access token* obtido anteriormente;
- Vincule um dispositivo Android ativando seu modo de *debug*. Caso não tenha um dispositivo, crie um emulador de máquina virtual Android no Android Studio para executar o app.
- Clique no menu `Run>Debug` e escolha o seu dispositivo;
- Escreva textos na parte inferior da tela no app e clique no botão inferior direito para enviar as mensagens ao *bot*.

A.2 Começando com o seu próprio app

Essa seção descreve o que é preciso fazer para integrar seu app com o *framework chatbot* inteligente. Primeiramente, é necessária a criação de duas classes *Activity* no projeto. Uma irá chamar o *splash screen* presente no *framework*. A outra é invocada por essa classe após a *splash screen* do *framework* ser executada.

A.2.1 Primeira classe

É preciso importar o módulo presente na pasta `modulochatbot` que foi incluído, anteriormente, em seu projeto, conforme ilustra a Figura 38.

Posteriormente, como observado na Figura 39, é preciso definir como *layout* da classe o *layout* definido no *framework* para o *splash screen*. Logo depois, criamos uma *thread* que invocará a segunda classe a ser implementada a seguir.

```
import chatbotbiblio.modulochatbot.*;
```

Figura 38 – Trecho de código ilustrando a importação do módulo na classe do usuário.

```
setContentView(chatbotbiblio.modulochatbot.R.layout.activity_splash_screen);
```

Figura 39 – Trecho de código ilustrando a definição do *layout* a ser instanciado na *activity*.

```
Thread myThread = new Thread(){
    @Override
    public void run(){
        try{
            sleep(2000);
            Intent intent = new Intent(getApplicationContext(), MainActivity.class);
            startActivity(intent);
            finish();
        }catch(InterruptedException e){
            e.printStackTrace();
        }
    }
};
myThread.start();
```

Figura 40 – Trecho de código ilustrando a invocação da *thread* a ser executada na *activity*.

A.2.2 Segunda classe

A segunda classe deve estender, necessariamente, a classe principal do *framework*, como mostrado na Figura 41. Dentro da sobrescrita do método principal `onCreate()`,

```
public class MainActivity extends chatbotbiblio.modulochatbot.MainActivity {
```

Figura 41 – Trecho de código ilustrando a extensão da classe do *framework*.

recomenda-se o uso do seguinte modelo de uso ilustrado na Figura 42.

Com o método `setConfig()` devemos passar como parâmetro nosso *client access token* gerado pela API DialogFlow. Já no método `setFalaInicial()` configuramos a fala inicial do *bot* criado. Nas linhas abaixo, com a criação de uma classe de exemplo `AcaoTeste` que estende a classe `Acao` do *framework*, devemos instanciar essas classes de acordo com as ações criadas dentro das *intents* da plataforma. Caso não haja nenhuma, a criação da classe não é necessária. Através da criação de instâncias da classe `AcaoTeste`, é possível trabalhar com ações definidas dentro das *intents* da plataforma DialogFlow.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.setConfig("SEU ACCESS TOKEN AQUI");
    super.setFalaInicial("Olá");
    Acao ac = new AcaoTeste("teste");
    super.addAcao(ac);
    super.onCreate(savedInstanceState);
}
```

Figura 42 – Trecho de código ilustrando a instânciação do *framework*.

Para isso é preciso passar como construtor da classe **AcaoTeste** o mesmo nome das ações criadas na plataforma DialogFlow.

A.2.3 Classe AcaoTeste

Para definir o comportamento das ações entre na classe **AcaoTeste** modifique o método sobrescrito **executaAcao()** de acordo com os nomes das ações criadas na classe principal da aplicação. Para trabalhar com parâmetros use o método **retornaParametro()** como ilustrado na Figura 43. Como podemos observar, seguindo essa estrutura na sobrescrita do método **executaAcao()** podemos definir os comportamentos das ações criadas na classe principal da aplicação. Através da estrutura de “ifs”, como exemplo, definimos um código personalizado para tratar diversas situações que o *bot* pode encontrar. A partir do método **RetornaRespostaBot** nós recuperamos as falas geradas pelo *bot* para responder a interação com o usuário. Com isso, dentro do “if” ilustrado no código da Figura 43, pode-se definir o tratamento de diversas situações a partir de uma resposta fixada na API DialogFlow.

A.2.4 Executando o app

- Vincule um dispositivo Android no seu computador, ou crie uma máquina virtual Android pelo Android Studio;
- Certifique-se de que seu dispositivo Android está com o modo *debug* ativado e ligado ao computador. Ou verifique se sua máquina virtual está pronta;
- Selecione sua aplicação para executar no Android Studio e clique no botão *debug*. O app deve executar em instantes. Escreva textos na parte inferior da tela no app e clique no botão inferior direito para enviar as mensagens ao *bot*.

```
import chatbotbiblio.modulochatbot.Acao;

public class AcaoTeste extends Acao {

    public AcaoTeste(String nome) {
        super(nome);
    }

    @Override
    public void executaAcao(){
        if(this.RetornaRespostaBot() == "resposta-retornada-pelo-bot"){
        }
        if(this.nome == "nome-da-acao"){
            //define o comportamento da ação
            String parametroValor = this.retornaParametro("nome-do-parametro");
        }
        else if(this.nome == "nome-da-acao-2"){
            //define o comportamento da ação
            String parametroValor = this.retornaParametro("nome-do-parametro2");
            //faz alguma coisa com o parametro
        }
    }
}
```

Figura 43 – Trecho de código exemplificando o uso personalizado de respostas usando ações.

APÊNDICE B – Questionário usado para validação do *framework*

O *framework chatbot* inteligente facilita a integração com a API de processamento de linguagem natural DialogFlow em dispositivos Android. Inicialmente, usando sua conta Google, crie um projeto no DialogFlow. Para utilizar o *framework*, siga a documentação e baixe o código fonte disponível em : <https://github.com/gumiranda/frameworkChatbotInteligente>. E seguindo a documentação responda as seguintes questões.

B.1 Questões abertas

A seguir, as perguntas abertas feitas para colher dados básicos dos usuários.

- Nome
- Ocupação

B.2 Questões de múltipla escolha

Seguindo a escala de Lickert, foram feitas perguntas de múltipla escolha com as opções “Concordo totalmente”, “Concordo”, “Neutro”, “Discordo” e “Discordo totalmente”. As perguntas foram:

- A documentação do *framework* está clara e fácil de entender ?
- O *framework* tem bom funcionamento e usabilidade ?
- O *framework* é capaz de integrar um projeto criado na plataforma Dialogflow ?
- A documentação é capaz de resolver dúvidas pertinentes ao uso de *actions* criadas no Dialogflow ?