



**UNIVERSIDADE ESTADUAL DE SANTA CRUZ – UESC**

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** Sistema distribuído para recuperação de informações no âmbito da UESC

**ILHÉUS - BAHIA  
2018**

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** Sistema distribuído para recuperação de informações no âmbito da UESC

Trabalho de Conclusão de Curso apresentado à Universidade Estadual de Santa Cruz - UESC, como parte das exigências para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Me. Leard de Oliveira Fernandes

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** Sistema distribuído para recuperação de informações no âmbito da UESC

Trabalho de Conclusão de Curso apresentado à Universidade Estadual de Santa Cruz - UESC, como parte das exigências para obtenção do título de Bacharel em Ciência da Computação.

Ilhéus, 5 de dezembro de 2018.

---

Prof. Me. Leard de Oliveira Fernandes  
UESC/DCET  
(Orientador)

---

Prof. Dr. Francisco Bruno Souza Oliveira  
UESC/DCET

---

Prof. Dr. Marcelo Ossamu Honda  
UESC/DCET

À minha família e à minha companheira Nilana que, com muito amor e paciência, sempre me deram todo o suporte quando necessário durante toda a minha etapa de graduação.

## AGRADECIMENTOS

À Universidade Estadual de Santa Cruz, pela oportunidade de fazer o curso.

Ao NBCGIB, pelo ambiente criativo e amigável que me proporcionou desde o início do curso.

Ao meu orientador Prof. Me. Leard de Oliveira Fernandes, pelo suporte no pouco tempo que lhe coube, pelas suas correções e incentivos.

Ao Prof. Dr. Marcelo Ossamu Honda, pelo apoio e oportunidades dadas durante todo meu percurso na universidade e pelas orientações de iniciações científicas.

Aos meus pais, Júnior e Mara, e irmãos, Vinícius e Neto, pelo amor, incentivo e apoio incondicional.

À minha companheira Nilana, pelo suporte emocional e incentivador nas horas difíceis, de desânimo e cansaço, durante toda a minha trajetória nesta universidade e pela ajuda dada na escrita e revisão deste projeto.

À minha cunhada Laylla, pela ajuda e apoio fornecidos para a escrita e revisão deste projeto.

Aos integrantes da “*semi.pro*”: Adshow, Pedreca, Medina, Bida e Pague, também aos meus amigos, em especial Alberto, Levy e Tulio, por todos esses anos de apoio e companheirismo e pela ajuda no amadurecimento e testes deste projeto.

Aos professores e funcionários pelos ensinamentos e pela convivência durante este período.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

|

“Stay hungry, stay foolish.”

(Steve Jobs)

**LANA, ASSISTENTE PESSOAL:** Sistema distribuído para recuperação de informações no âmbito da UESC

**RESUMO**

Com a disseminação de computadores pessoais e dispositivos móveis conectados à internet, junto ao avanço da tecnologia cognitiva na área de linguagem natural, softwares de assistência pessoal surgiram para uma variedade de propósitos. A interação entre usuário e assistente pessoal, normalmente, se dá através da troca de mensagens, onde o usuário faz uma pergunta ou uma requisição de serviço, e o assistente a responde, ou executa o serviço requisitado. O objetivo deste trabalho é implementar um *software* de assistência pessoal distribuído, nomeado como Lana, capaz de receber mensagens textuais, interpretá-las, executar serviços quando necessário, e retornar ao usuário final uma resposta trivial ou um resultado da execução de um serviço requisitado. Serviços de recuperação de informações sobre a UESC disponíveis no *site* da própria universidade e informações do portal sagres, do aluno e do professor, foram implementados nos extratores de dados. Para a implementação do *software* de assistência pessoal foi utilizado um serviço de entendimento de linguagem natural, uma solução *backend* como serviço para o armazenamento de dados dos usuários, *web scraping* para recuperar informações, serviços de hospedagem e as linguagens de programação JavaScript e Python. Por fim, foi desenvolvido um *software* distribuído que se mostrou capaz de receber mensagens textuais a partir de duas aplicações de troca de mensagens, extrair as intenções e entidades das mensagens dos usuários e recuperar informações do site da UESC e do portal acadêmico sagres quando necessário.

**Palavras-chave:** Assistente Pessoal. IBM Watson. Sistemas Distribuídos. Web Scraping.

**LANA, PERSONAL ASSISTANT:** Distributed system for information retrieval  
regarding UESC

**ABSTRACT**

With the spread of personal computers and mobile devices connected to the internet, along with the advancement of cognitive technology in the area of natural language, personal assistance softwares has emerged for a variety of purposes. The interaction between the user and a personal assistant usually occurs through the exchange of text messages, where the user asks a question or request a service, and the assistant responds or performs the requested service. The objective of this work is to implement distributed personal assistance software, named Lana, capable of receive textual messages, interpret them, perform actions when necessary, and return to the user a trivial response or the result of a requested service. Information retrieval services about the UESC available on the university's own website and information from the portal sagres, for students and teachers, were implemented in data extractors. For the implementation of the personal assistance software, were used a natural language understanding service, a backend as a service solution for storing users' data, web scraping to retrieve information, hosting services and the JavaScript and Python programming languages. Finally, one distributed software was developed that was able to receive textual messages from two messaging applications, extract the intents and entities from the user's messages and retrieve information from the UESC website and the sagres academic portal when necessary.

**Keywords:** Personal Assistant. IBM Watson. Distributed Systems. Web Scraping.



## LISTA DE FIGURAS

Figura 1 - Painel do serviço Watson Assistant para a criação de Intents. ....	21
Figura 2 - Painel do serviço Watson Assistant para a criação de Entities. ....	23
Figura 3 - Painel do serviço Watson Assistant para a criação de Dialogs. ....	24
Figura 4 - Criação de um novo bot a partir do Telegram em uma conversa com o BotFather.....	25
Figura 5 - Painel de controle de uma aplicação no Back4App.....	27
Figura 6 - Painel de controle de um Droplet na plataforma do DigitalOcean. ....	29
Figura 7 - Painel de controle de um Dyno na plataforma do Heroku. ....	30
Figura 9 - Painel do serviço Watson Assistant com algumas Intents criadas para o workspace Lana. ....	40
Figura 10 - Painel do serviço Watson Assistant com todas as Entities criadas para o workspace Lana. ....	41
Figura 11 - Painel do serviço Watson Assistant com parte do fluxo de primeira conversa criada para o workspace Lana.....	42
Figura 11 - Topologia de comunicação entre módulos. ....	49
Figura 12 - Conversação mostrando a Lana em funcionamento no aplicativo Telegram. ....	50
Figura 13 - Lana informando suas principais funcionalidades. ....	51
Figura 14 - Lana realizando ações para professor no Telegram. ....	52
Figura 15 - Lana realizando ações para professor no Telegram. ....	53
Figura 16 - Conversação mostrando a Lana em funcionamento no aplicativo Messenger.....	54

## LISTA DE SIGLAS

AP	Assistente Pessoal
API	Application Programming Interface
BaaS	Backend as a Service
CPU	Central Processing Unit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IaaS	Infrastructure as a Service
IBM	Internacional Business Machines
IDL	Interface Definition Language
IP	Internet Protocol
PaaS	Platform as a Service
REST	Representational State Transfer
SAP	Software de Assistência Pessoal
SSH	Secure Shell
UESC	Universidade Estadual de Santa Cruz
UML	Unified Modeling Language
URL	Uniform Resource Locator
VPS	Virtual Private Server

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO .....</b>	<b>11</b>
<b>1.1</b>	<b>Objetivos .....</b>	<b>13</b>
1.1.1	Gerais .....	13
1.1.2	Específicos .....	14
<b>2</b>	<b>EMBASAMENTO TEÓRICO.....</b>	<b>15</b>
<b>2.1</b>	<b>Entendimento de Linguagem Natural.....</b>	<b>15</b>
<b>2.2</b>	<b>Backend as a Service.....</b>	<b>16</b>
<b>2.3</b>	<b>VPS Hosting.....</b>	<b>16</b>
<b>2.4</b>	<b>Web Scraping.....</b>	<b>17</b>
<b>2.5</b>	<b>Sistemas Distribuídos.....</b>	<b>18</b>
<b>2.6</b>	<b>Metodologias Ágeis.....</b>	<b>19</b>
<b>3</b>	<b>MATERIAIS E MÉTODOS .....</b>	<b>20</b>
<b>3.1</b>	<b>Materiais .....</b>	<b>20</b>
3.1.1	Watson Assistant.....	20
3.1.2	Aplicativos de Troca de Mensagens .....	24
3.1.2.1	Telegram .....	24
3.1.2.2	Messenger.....	26
3.1.3	Back4App .....	26
3.1.4	DigitalOcean Droplet .....	28
3.1.5	Heroku Dyno .....	29
3.1.6	Selenium WebDriver.....	31
3.1.7	Linguagens de Programação .....	32
3.1.7.1	JavaScript.....	32
3.1.7.2	Python .....	32
3.1.8	Kanban .....	33
<b>3.2</b>	<b>Métodos.....</b>	<b>33</b>
3.2.1	Interface .....	34
3.2.1.1	Telegram .....	35
3.2.1.2	Messenger.....	36
3.2.2	Banco de Dados .....	37
3.2.3	Entendimento de Linguagem Natural .....	39
3.2.4	Assistente Pessoal .....	43
3.2.5	Indexador de Bots .....	44
3.2.6	Bots Scrapers .....	45
3.2.6.1	BotSagres.....	46
3.2.6.2	BotUESC .....	47
<b>4</b>	<b>RESULTADOS E DISCUSSÕES.....</b>	<b>49</b>
<b>5</b>	<b>CONCLUSÃO .....</b>	<b>58</b>
<b>5.1</b>	<b>Trabalhos Futuros .....</b>	<b>58</b>
	<b>REFERÊNCIAS.....</b>	<b>59</b>
	<b>APÊNDICE A – CÓDIGO FONTE DO BOT DO TELEGRAM.....</b>	<b>63</b>
	<b>APÊNDICE B – CÓDIGO FONTE DO BOT DO MESSENGER .....</b>	<b>67</b>
	<b>APÊNDICE C – CÓDIGO FONTE DAS CLOUD FUNCTIONS DO BACK4APP .....</b>	<b>70</b>

## 1 INTRODUÇÃO

Com a disseminação de computadores pessoais e dispositivos móveis conectados a internet, junto ao avanço da tecnologia cognitiva na área de linguagem natural, *softwares* de assistência pessoal (SAP) surgiram para uma variedade de propósitos, tais como: gerenciamento de trabalho, organização e recuperação de informações e agendamento de atividades (MITCHELL *et al.*, 1994).

O objetivo destes *softwares* é auxiliar ou substituir seus usuários em determinadas tarefas, deixando-os livres para realizar atividades mais importantes. Converging aos conceitos de SAP, expostos por ZAMBIASI e RABELO (2012), Enembreck e Barthes (2002) explicam que o papel principal de um assistente pessoal (AP) é isentar o usuário de realizar tarefas repetitivas ou entediadas. Além disso, os autores deixam claro que as aplicações de um AP podem variar de pesquisas na internet até mesmo tarefas colaborativas.

A interação entre usuário e AP, normalmente, se dá através da troca de mensagens, onde o usuário faz uma pergunta ou uma requisição de serviço, e o assistente a responde, ou executa o serviço requisitado. Desta maneira, a construção de um SAP cria a necessidade de implementação de um meio de comunicação entre usuário e assistente, além da criação de serviços que possam ser requisitados pelo usuário. Em uma comunicação realizada por meio da troca de mensagens textuais, fazendo somente o uso da língua natural sem caracteres especiais para identificar comandos específicos, é necessário o entendimento de linguagem natural por parte do assistente, objetivando a compreensão das requisições realizadas pelo usuário, como é mostrado por USACHEV *et al.* (2018) e REATEGUI, RIBEIRO e BOFF (2008), onde os autores desenvolveram em seu módulo de comunicação um sistema para o entendimento de linguagem natural.

Tendo em vista os objetivos de um AP e as suas principais necessidades, em conjunto ao modelo de sistema exposto por REATEGUI, RIBEIRO e BOFF (2008), pode-se realçar que para a implementação de um SAP é indispensável o desenvolvimento de um meio de comunicação, armazenamento de dados, entendimento de linguagem natural e por fim a execução dos serviços necessários. Desta maneira, um SAP pode ser visto como uma grande aplicação monolítica, e a construção de um *software* desta maneira gera acoplamento e dificuldade de manutenção (IBM, [S.d.]). Entretanto, visando evadir-se destes problemas, as

necessidades do SAP podem ser divididas em agentes capazes de executar tarefas específicas, criando assim um sistema multiagente.

Wooldridge (2002, p. XIII) apresenta sistemas multiagente como sistemas compostos por múltiplos elementos computacionais capazes de interagir, conhecidos como agentes. O autor também conceitua um agente como:

Um sistema computacional com dois importantes recursos. Primeiro, eles são pelo menos até certo ponto capazes de ações autônomas – de decidir por si mesmos o que eles precisam fazer para satisfazer seus objetivos. Segundo, eles são capazes de interagir com outros agentes – não simplesmente trocando dados, mas com engajamento em atividades sociais que nós realizamos diariamente em nossa vida: cooperação, coordenação, negociação e coisas do gênero.

Nesse sentido, Reateguil, Ribeiro e Boff (2008) propõem um sistema multiagente para o controle de um AP, que é explicado pelos autores: “cada agente controla uma funcionalidade específica, e um agente mediador define qual deles deve entrar em ação a cada momento” (REATEGUI; RIBEIRO; BOFF, 2008).

Desta maneira, pode-se obter soluções adquiridas a partir de ferramentas de terceiros, por exemplo, o recurso de entendimento de linguagem natural pode ser obtido com o IBM Watson, como é apresentado por YAN *et al.* (2016), PITON (2017) e MOSTAÇO *et al.* (2018). Não restrito ao entendimento de linguagem, outros recursos como a interface de comunicação, i.e. o meio de troca de mensagens pode ser modularizada e obtido a partir de outras ferramentas, como é mostrado por MOSTAÇO *et al.* (2018) onde é utilizado a *Application Programming Interface* (API) do aplicativo de mensagens Telegram para a comunicação com o usuário.

A comunicação entre os agentes, ou módulos, de um SAP geralmente é realizada por meio da internet, pois quase sempre os módulos estão sendo executados em computadores e locais distintos. Sendo assim, um SAP implementado de maneira modularizada pode ser considerado um sistema distribuído. Segundo TANENBAUM e VAN STEEN (2013) um sistema distribuído é como uma coleção de computadores independentes que cooperam para resolver uma tarefa, entretanto o usuário final os enxerga como um só. Ainda COULOURIS, DOLLIMORE e KINDBERG (2012) acrescentam ao conceito de sistema distribuído como sendo um sistema no qual *softwares* ou *hardwares* conectados à rede comunicam-se e coordenam as suas ações por meio de troca de mensagens.

Dada a necessidade de disponibilizar serviços pela internet, se faz necessária a criação de serviços de rede (*web services*), ou seja, servidores conectados à internet, construídos com o propósito de suprir as necessidades de um *site* ou uma aplicação. Programas clientes utilizam APIs para se comunicar com estes *web services*. De modo geral, uma API disponibiliza dados e funções para facilitar a interação entre os *softwares* e permite que eles troquem informações. Uma API *web* é a interface de um serviço *web*, que recebe e responde requisições de clientes (MARK, 2013).

Como é exposto por USACHEV *et al.* (2018) um AP pode ser utilizado para obter informações para um usuário. Nesse sentido, um SAP pode utilizar uma API disponibilizada por outro sistema para a obtenção de dados ou para a realização de alguma ação.

Entretanto, nem todos os *sites* existentes fornecem uma API, e para extrair informações destes sistemas *web* uma das principais técnicas é o uso do *web scraping* para obter estes dados de forma automatizada (MALIK; RIZVI, 2011). VARGIU e URRU (2012) apresentam *web scraping* como “[...] uma técnica de *software* destinada a extrair informações de *sites*. *Web scrapers* simulam a exploração humana na internet” (VARGIU; URRU, 2012, tradução nossa). Os autores apresentam o termo *web scrapers*, este é designado a *softwares* programados para buscar, em uma página da *web*, informações e/ou executar ações utilizando a técnica de *web scraping*. Este tipo de *software* automatizado é popularmente conhecido como *um bot scraper*. Tais *bots* percorrem o *site* através do código fonte, normalmente em formato HTML, e também podem executar instruções em JavaScript na página.

## 1.1 Objetivos

### 1.1.1 Gerais

O objetivo deste trabalho é implementar um *software* de assistência pessoal distribuído, nomeado como Lana, capaz de receber mensagens textuais, interpretá-las, executar serviços quando necessário, e retornar ao usuário final uma resposta trivial ou um resultado da execução de um serviço requisitado.

### 1.1.2 Específicos

Os objetivos específicos deste projeto são:

- a) estabelecer um meio de comunicação entre usuário e assistente pessoal a partir de aplicativos de troca de mensagens;
- b) armazenar informações referentes aos usuários em banco de dados;
- c) obter o entendimento de linguagem natural a respeito das ações que possam ser realizadas;
- d) extrair informações do portal acadêmico Sagres, como, por exemplo, obter horários do semestre e listar disciplinas;
- e) extrair informações do site da UESC, como, por exemplo, listar cursos de graduação ofertados e os últimos editais publicados.

Este trabalho de conclusão de curso foi dividido em 4 tópicos gerais, apresentando-se no primeiro um aprofundamento a respeito da teoria utilizada neste projeto. No segundo tópico é abordado a implementação de cada módulo do projeto, como e quais ferramentas foram utilizadas para o desenvolvimento de cada parte do projeto. O terceiro tópico caracteriza a exposição dos resultados obtidos e uma discussão sobre eles, além de exibir possíveis melhorias e os impedimentos sofridos durante o projeto. Por fim, no ultimo tópico é apresentado as conclusões acerca dos resultados obtidos e proposto implementações futuras.

## 2 EMBASAMENTO TEÓRICO

Neste tópico será apresentada a fundamentação teórica necessária para o desenvolvimento deste projeto.

### 2.1 Entendimento de Linguagem Natural

O entendimento de linguagem natural, também conhecido como processamento de linguagem natural, é uma área que estuda o desenvolvimento de *softwares* que analisam e reconhecem linguagens humanas, ou linguagens naturais (PERNA; DELGADO; FINATTO, 2010).

Esta área trata computacionalmente os aspectos da comunicação humana considerando estruturas e contextos das sentenças. De uma maneira geral, o objetivo do entendimento de linguagem natural é estabelecer uma comunicação com o computador por meio da linguagem humana (GONZALEZ; LIMA, 2003).

Devido à rica ambiguidade da linguagem natural, o entendimento desta se torna uma tarefa não trivial. Essa ambiguidade faz com que o entendimento de linguagem natural seja diferente do entendimento de linguagens de programação, já que estas foram criadas com definições que evitem justamente a ambiguidade (PERNA; DELGADO; FINATTO, 2010).

Ainda PERNA, DELGADO e FINATTO (2010) acrescentam que:

Vários métodos ou técnicas computacionais podem ser usados para analisar e interpretar a linguagem natural. Existe uma clara distinção, por exemplo, entre as técnicas empregadas na linguagem falada e na linguagem escrita, mas também além destas divisões é possível ver diferenças claras entre as técnicas necessárias para diferentes tipos de aplicação, e que podem envolver diferentes tipos de textos, como por exemplo, textos científicos, jornalísticos, literários, etc. Neste sentido, a grande maioria das aplicações é voltada para um tipo específico de linguagem.

Devido a este envolvimento de textos com diferentes aspectos, as empresas que fornecem o serviço de entendimento de linguagem natural, normalmente, requisitam aos clientes que sejam cadastradas quais são as intenções e entidades que devem ser extraídas do texto, como é o caso do Google DialogFlow e IBM Watson Assistant (GOOGLE, [S.d.]) (MILLER, 2017).



As intenções são os objetivos que o usuário terá ao interagir com o serviço, ou seja, as ações que o usuário pretende realizar com o serviço. Para cada *Intent* é necessário incluir elocuções de amostra que refletem a entrada que os clientes possam usar para obter as informações que eles precisam. Já as entidades representam um termo ou objeto que fornece um contexto para uma intenção, i.e., elas são as entidades que podem aparecer durante a conversação, como locais, datas, horários ou códigos específicos de algum objeto.

## 2.2 Backend as a Service

Um *Backend as a Service* (BaaS) pode ser visto como um serviço que auxilia a conexão entre o *backend* e o *frontend* de uma aplicação. O BaaS ajuda os desenvolvedores a acelerar o desenvolvimento de software e simplificar a criação de APIs. Com ele não é necessário desenvolver todo o *backend* de um aplicativo, apenas utilizar o BaaS para criar as APIs e conectar à aplicação (BATSCHINSKI, 2016).

O BaaS fornece aos desenvolvedores de aplicações *web* e móveis uma maneira de conectar seus aplicativos ao armazenamento e processamento em nuvem e ao mesmo tempo fornecem recursos comuns como gerenciamento de usuários, notificações, integração de redes sociais e outros recursos bastante utilizados por aplicações *web* e móveis hoje em dia. Essa nova geração de soluções BaaS são fornecidas por meio de bibliotecas personalizadas e APIs. Em geral, o BaaS é um serviço relativamente novo na área de computação em nuvem, a maioria das empresas que fornecem essa solução surgiram a partir do ano de 2011 (LANE, 2013).

## 2.3 VPS Hosting

*Virtual Private Server (VPS) Hosting* é a hospedagem de máquinas virtuais, vendido por empresas como um serviço. Cada VPS possui seu sistema operacional dedicado e os clientes destas empresas possuem acesso de usuário com direitos de administrador destas máquinas. Um VPS pode usar uma *Central Processing Unit* (CPU) compartilhada com outros VPSs, isso significa que cada um receberá uma porção de *threads* compartilhadas de uma CPU virtualizada. Entretanto, os VPSs podem utilizar *threads* dedicadas de uma CPU virtualizada, aumentando assim o seu

desempenho e consistência durante processamento intensivo da CPU (DIGITALOCEAN, 2018b).

Um VPS fornece maior custo-benefício em comparação com servidores dedicados, além disso, os VPSs fornecem recursos computacionais previamente garantidos. Até pouco tempo atrás, o VPS *Hosting* não possuía um esquema de arquitetura flexível, a alocação de recursos era realizada no momento da criação do contrato com a empresa e caso o cliente quisesse atualizar algum recurso somente durante um período de pico de uso, ele tinha que pagar até mesmo pelos momentos em que o VPS estava em desuso (TELENYK *et al.*, 2013).

Atualmente, grande parte das empresas que vendem o serviço de VPS *Hosting* fornecem ao cliente a possibilidade de alterar recursos de maneira mais dinâmica por meio de um painel de controle, criando assim a capacidade de alterar os recursos de um VPS somente nos momentos necessários e reduzindo os custos de utilização do serviço (TELENYK *et al.*, 2013).

## 2.4 Web Scraping

O *web scraping* é uma técnica de *software* que objetiva a extração de informações de páginas da *web*. Normalmente, o uso da técnica consiste na imitação de um ser humano utilizando a internet e interagindo com componentes de sistemas *web* ou até mesmo realizando requisições e enviando informações.

*Web scraping* está relacionado com uma outra técnica chamada de indexação da *web*, esta é uma técnica adotada por diversos mecanismos de pesquisa para indexar informações da internet por meio de um *bot*. Entretanto, o *web scraping* é mais voltado para transformação de dados não estruturados na internet, geralmente dispostos em formato HTML, em dados estruturados que possam ser armazenados e analisados (VARGIU; URRU, 2012).

Tal técnica também pode ser utilizada como maneira de automatizar a extração ou inserção de dados em sistemas. Normalmente, é criado um *software* capaz de percorrer estruturas HTML e executar funções JavaScript, estes são conhecidos como *bots scrapers*. Por fim, estes *bots* são utilizados para realizar ações específicas em alguma página para diversas finalidades.

Atualmente o *web scraping* é comumente utilizado para comparação de preços online, monitoramento de dados meteorológicos, detecção de mudanças em sistemas

*web*, buscas na internet, compilação de diferentes fontes de informação em um só lugar e integração de dados da internet (VARGIU; URRU, 2012). Para o uso da técnica de *web scraping* existem diversos *softwares* que fornecem maior facilidade na criação de *bots scrapers*.

## 2.5 Sistemas Distribuídos

Um sistema distribuído é um sistema que é composto por componentes que estão localizados em computadores diferentes, interligados por uma rede, que coordenam as suas ações a partir do repasse de mensagens e para o usuário final parecem ser apenas um único sistema. Desta maneira, um sistema distribuído possui concorrência de componentes, falta de um relógio global e falhas de componentes independentes (COULOURIS; DOLLIMORE; KINDBERG, 2012) (TANENBAUM; VAN STEEN, 2013).

Basicamente, um sistema distribuído é composto por diversos componentes, i.e., computadores, autônomos transparentes ao usuário final, de maneira que eles enxerguem todos os componentes como um único. Desta maneira, os componentes precisam cooperar. A maneira na qual esta colaboração é realizada é um dos princípios de sistemas distribuídos (TANENBAUM; VAN STEEN, 2013).

Os computadores envolvidos em um sistema distribuído podem ser dos mais variados tipos e possuir as suas mais variadas diferenças, de *mainframes* até mesmo pequenos sensores conectados a uma rede. Entretanto, para que haja uma comunicação entre eles é necessário que exista uma *Interface Definition Language* (IDL), um padrão uniforme de comunicação que descreve as interfaces assim como suas assinaturas (TANENBAUM; VAN STEEN, 2013) (MESSERSCHMITT, 1999).

Os sistemas distribuídos podem ser implementados para os mais variados propósitos, como por exemplo, a própria internet é um sistema distribuído, assim como jogos online, e-mails, redes sociais entre outros (COULOURIS; DOLLIMORE; KINDBERG, 2012).

Os desafios na criação de um sistema distribuído aparecem devido a necessidade de permitir que novos componentes sejam adicionados ou substituídos, prover segurança em todo o sistema, possuir uma boa escalabilidade, i.e., funcionar bem quando a carga ou o número de usuários aumenta, tratar concorrência entre os

componentes e ofertar um serviço de qualidade (COULOURIS; DOLLIMORE; KINDBERG, 2012).

## 2.6 Metodologias Ágeis

As metodologias ágeis são apontadas como uma alternativa as formas tradicionais de desenvolvimento de *softwares*. Estas são voltadas para projetos em que possam haver muitas mudanças, os requisitos são passíveis de alterações, modificar partes de códigos não são tarefas de alto custo, as equipes são pequenas, datas de entregas são curtas e é essencial que a implementação seja realizada de maneira rápida sem requisitos estáticos (SOARES, 2004).

Em 2001, dezessete especialistas em processos de desenvolvimento de *softwares* apresentaram estabeleceram princípios comuns a serem compartilhados por todos os métodos ágeis. Este documento contendo os princípios é conhecido como manifesto ágil. Os principais conceitos apresentados por ele são os de indivíduos e interações ao invés de processos e ferramentas, *software* executável ao invés de documentação, colaboração do cliente ao invés de negociação de contratos e respostas rápidas e mudanças ao invés de seguir planos (SOARES, 2004) (BECK *et al.*, 2001).

O manifesto ágil coloca algumas das principais preocupações das metodologias clássicas como tarefas de importância secundária. De acordo com as metodologias ágeis, não é obrigatório a criação de diagramação UML, somente é criado estes diagramas quando é necessário e geralmente de forma automatizada (BECK *et al.*, 2001).

As metodologias ágeis são comumente usadas por pequenas e grandes empresas no cenário atual devido ao seu aumento na velocidade de produção e entrega. Existem diversas metodologias ágeis atualmente, as principais são Scrum, Test Driven Development, Extreme Programming e Kanban.

### 3 MATERIAIS E MÉTODOS

Para a realização deste projeto foi necessária a utilização de algumas ferramentas de trabalho, as principais estão listadas no subtópico materiais e em seguida, no subtópico métodos, é apresentado o uso delas no trabalho.

#### 3.1 Materiais

##### 3.1.1 Watson Assistant

Anteriormente conhecido como Watson Conversation, em 2016, foi criado o serviço Watson Assistant. Disponibilizado pelo IBM Watson por meio da plataforma IBM Cloud, com este serviço é possível construir soluções que entendam linguagem natural e responda de maneira similar a uma conversa entre humanos (MILLER, 2017).

Serviços similares são disponibilizados por outras empresas, como, por exemplo, o Google DialogFlow, entretanto, para este projeto optou-se pelo IBM Watson Assistant, pois este apresentou uma interface mais amigável para a configuração do serviço, em conjunto a possibilidade de criar uma conta gratuita para utilizar o sistema.

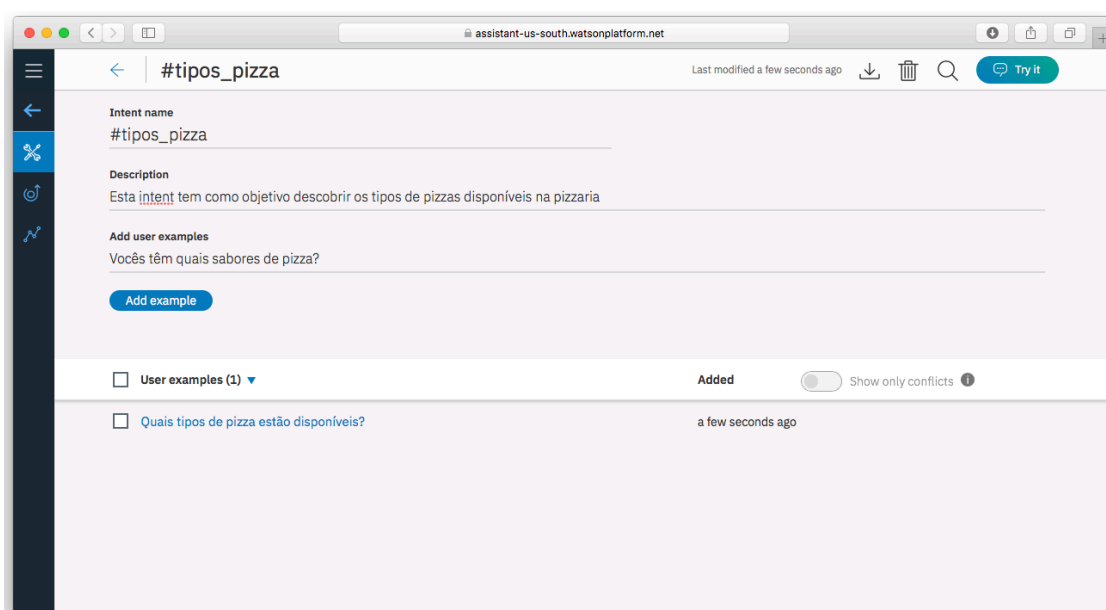
O funcionamento proposto pelo serviço é que a interação com o usuário seja realizada através da interface de algum meio de comunicação implementada pelo desenvolvedor, como por exemplo uma janela de bate-papo simples. O Aplicativo envia a mensagem de texto do usuário para um *workspace* do serviço, o Watson Assistant interpreta a entrada do usuário, direciona o fluxo da conversa, reúne as informações necessárias e retorna estes dados à aplicação. Por fim, o aplicativo pode interagir com seus outros sistemas com base no entendimento da intenção do usuário e utilizar as informações adicionais obtidas para, por exemplo, abrir chamados, atualizar informações de conta ou realizar pedidos.

Para a utilização do serviço é necessário a criação de um *workspace*, assim como suas *Intents* e *Entities*. A configuração de *workspace* é realizada através de um painel disponível na plataforma IBM Cloud, neste é possível criar, editar ou remover *workspaces*.

Após a criação de um novo *workspace* é necessário configurar os dados de treinamentos, *Intents* e *Entities*, e o fluxo de diálogo das conversações, *Dialogs*.

As *Intents* são os objetivos que o usuário terá ao interagir com o serviço, i.e., elas são as intenções do usuário, ou seja, as ações que o usuário pretende realizar com o serviço. Para cada *Intent* é necessário incluir elocuições de amostra que refletem a entrada que os clientes possam usar para obter as informações que eles precisam, este painel pode ser visualizado na Figura 1. Por exemplo, uma *Intent* “tipos\_pizza” pode ser criada para representar a intenção do usuário de saber os tipos de pizzas disponíveis em uma pizzaria, para que o Watson identifique esta *Intent* é preciso incluir vários exemplos de perguntas, como: “quais tipos de pizza estão disponíveis?” ou “você têm quais sabores de pizza?”.

Figura 1 - Painel do serviço Watson Assistant para a criação de Intents.



Fonte: Plataforma do IBM Watson<sup>1</sup>.

Já as *Entities* representam um termo ou objeto que fornece um contexto para uma *Intent*, i.e., elas são as entidades que podem aparecer durante a conversação, como locais, datas, horários ou códigos específicos de algum objeto. O serviço já disponibiliza algumas *Entities* de sistema, para identificar números, datas,

<sup>1</sup> Disponível em: <<https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/intents>>. Acesso em nov. 2018.

percentagens, dinheiro e tempo. As *Entities* possuem valores associados à sinônimos ou padrões, sendo assim, ao cadastrar uma nova *Entitie* é preciso informar os sinônimos, i.e., as maneiras como um valor pode aparecer durante o dialogo, ou uma expressão regular que padroniza este valor. Por exemplo, uma *Entitie* “aeroportos\_brasil” pode ser criada para identificar siglas dos aeroportos brasileiros em uma conversação, nesta *Entitie* seria cadastrado o valor “aeroporto de Ilhéus” associado ao sinônimo “IOS”, desta maneira ao receber a mensagem “Chegarei no dia 20 de dezembro em IOS” o Watson Assistant reconhecerá na mensagem a data 20 de dezembro do ano corrente, a partir da *Entitie* de sistema, e o aeroporto de Ilhéus a partir da *Entitie* cadastrada. Um outro exemplo é a criação de uma *Entitie* para a identificação de informações pessoais, neste caso seria incluído o valor como “telefone” e ao invés do uso de sinônimos é utilizado o reconhecimento de padrões, sendo assim é criado uma expressão regular que reconheça um número de telefone e ao enviar uma mensagem contendo um número de telefone qualquer para o serviço este valor é reconhecido podendo ser armazenado no contexto.

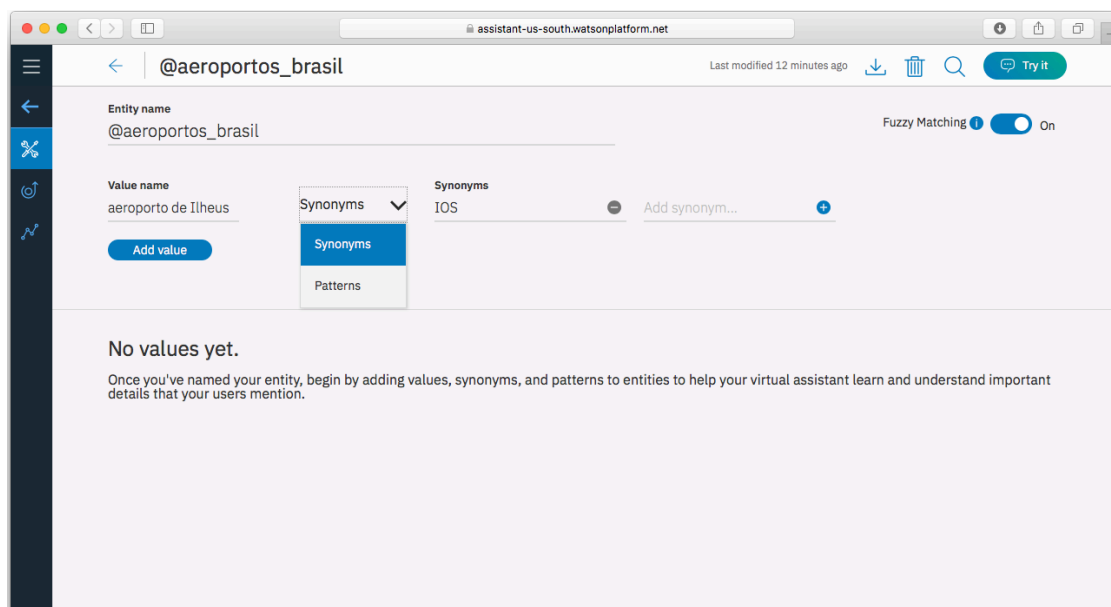
A Figura 2 apresenta a criação de uma *Entitie* “aeroportos\_brasil”, utilizada como exemplo, na imagem pode ser observado que é criado o valor “aeroporto de Ilhéus” associado ao sinônimo “IOS”, também é possível observar que na caixa de seleção de tipo foi selecionado a opção “Synonyms”, que significa “sinônimos” em português, para adicionar novos sinônimos que vão possuir este valor, entretanto caso seja selecionada a opção “Patterns”, que significa “padrões” em português, no lugar de novos sinônimos seriam adicionados expressões regulares para a identificação destes padrões.

Conforme são incluídos novos dados de treinamento, um classificador de língua natural é automaticamente incluído no *workspace* e é treinado para entender as solicitações indicadas.

Para finalizar a configuração do *workspace* é necessário montar fluxos de diálogos, ou *Dialogs*, que incorporam as *Intents* e *Entities*. A criação destes fluxos é realizada através de um painel de criação de *dialogs*, que pode ser visto na Figura 3. O fluxo de diálogo é representado graficamente na ferramenta como uma árvore, sendo possível incluir novas ramificações para processar cada uma das *Intents* incluídas. Também é possível incluir nós de ramificações que tratam diversas permutações possíveis de um pedido com base em outros fatores, como por exemplo,

entidades localizadas na mensagem de entrada ou informações extras passadas ao requisitar o serviço.

Figura 2 - Painel do serviço Watson Assistant para a criação de Entities.



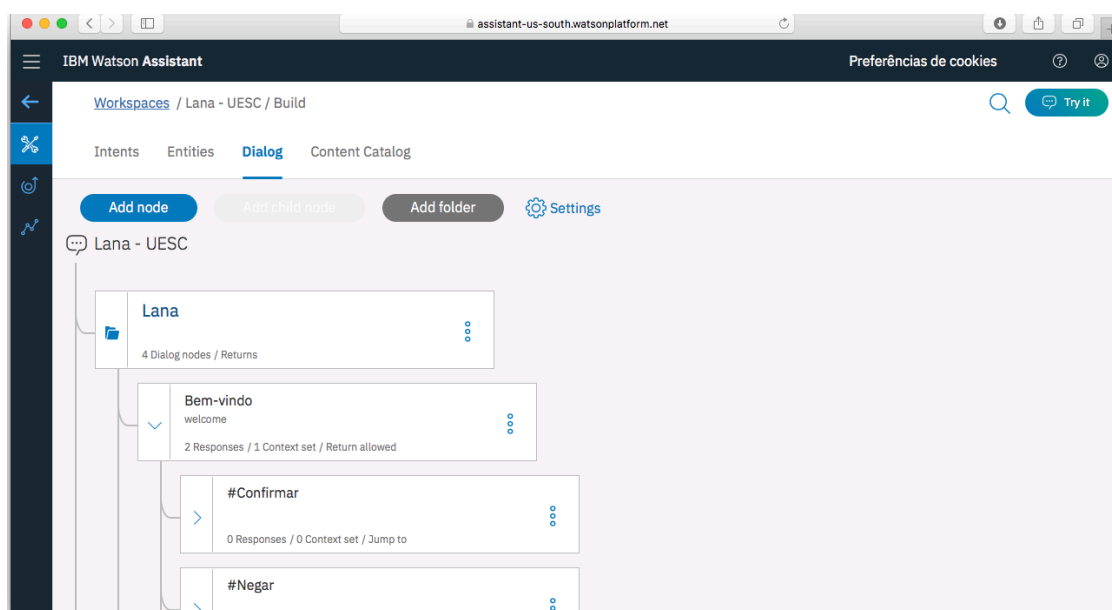
Fonte: Plataforma do IBM Watson<sup>2</sup>.

Finalmente, após finalizar toda a configuração do *workspace*, é possível utilizar o serviço Watson Assistant. Com o serviço já configurado, ao requisita-lo enviando uma mensagem de texto, o Watson vai avaliar a entrada buscando contextos utilizando os *Dialogs* e retornar quais são as *Intents* e *Entities* identificadas na mensagem e a sua taxa de confiabilidade para cada.

<sup>2</sup> Disponível em: <<https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/entities/user>>. Acesso em nov. 2018.



Figura 3 - Painel do serviço Watson Assistant para a criação de Dialogs.



Fonte: Plataforma do IBM Watson<sup>3</sup>.

### 3.1.2 Aplicativos de Troca de Mensagens

Para o desenvolvimento deste projeto foi proposto o uso de aplicativos de troca de mensagens como meio de comunicação entre o usuário final e o SAP, visando eliminar a necessidade de criação de uma nova aplicação que serviria apenas como interface, aliado à possibilidade de contatar a AP facilmente a partir de aplicativos com API pública, sem forçar o usuário a adquirir um novo aplicativo para este propósito.

#### 3.1.2.1 Telegram

O Telegram é um aplicativo popular de troca de mensagens baseado em plataforma de código livre (SUTIKNO *et al.*, 2016). Ele foi escolhido para este projeto pois é uma aplicação gratuita e com uma interface simples, disponível para *smartphones* e computadores pessoais com aplicação *desktop* ou aplicação *web*.

Além disso, o Telegram também disponibiliza uma API para criação de *bots* na plataforma, desta maneira usuários podem interagir com os *bots*, enviando

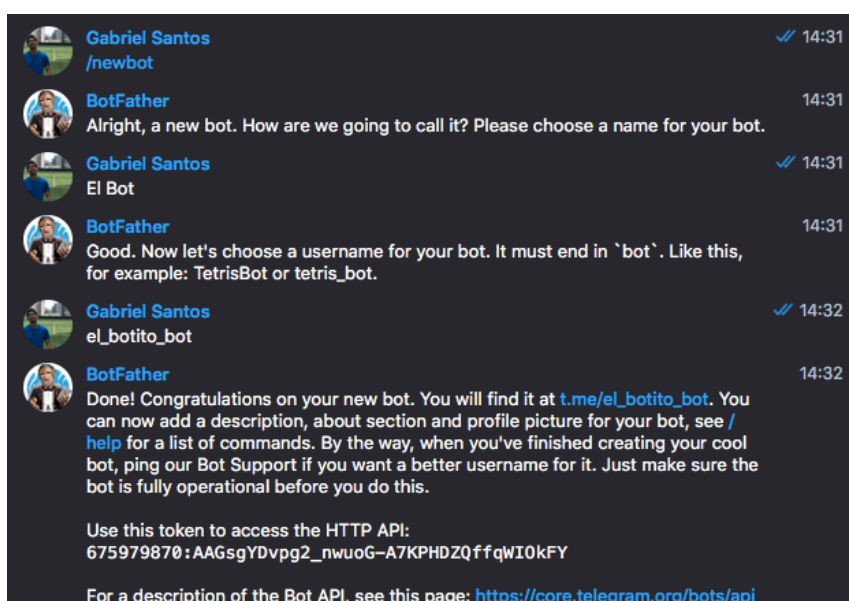
<sup>3</sup> Disponível em: <<https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/dialog>>. Acesso em nov. 2018.

mensagens e comandos. O controle dos *bots* é feito através de requisições HTTPS para a API pública do Telegram.

O Telegram não possui um painel de controle para a configuração de novos *bots*, entretanto, a criação de um novo *bot* no Telegram é feita através de um outro *bot* disponibilizado pelo aplicativo, o BotFather. Ao enviar o comando de criar um novo *bot* para o BotFather, ele pede informações básicas como nome e nome de usuário do *bot* a ser criado e em seguida é informado a chave de acesso, *token*, para controle deste *bot*.

A Figura 4 mostra a criação de um novo *bot* através do envio do comando “/newbot”, passagem do nome “El bot” e o nome de usuário “el\_botito\_bot”. Em seguida o BotFather confirma a criação do novo *bot* e envia o *token* “675979870:AAGsgYDvpg2\_nwuoG-A7KPHDZQffqWIOkFY” que será utilizado para acessar e controlar as funcionalidades deste *bot*.

Figura 4 - Criação de um novo bot a partir do Telegram em uma conversa com o BotFather.



Fonte: Conversa com o BotFather a partir do Telegram<sup>4</sup>.

A implementação de novas funcionalidades do *bot* pode ser realizada em qualquer linguagem de programação onde seja possível fazer requisições a API pública do Telegram. Além disso, também pode ser utilizado *frameworks* para a

<sup>4</sup> Disponível em: <<https://t.me/botfather>>. Acesso em nov. 2018.

facilitar o uso da API, como por exemplo o TGFancy, *framework* disponibilizado para a linguagem de programação JavaScript.

### 3.1.2.2 Messenger

Messenger é o aplicativo da rede social Facebook para a troca de mensagens entre os seus usuários. Ele está disponível para acesso via aplicação *web* ou aplicativo móvel. O Messenger foi escolhido para este projeto devido a sua popularidade gerada pelo Facebook, além disso, ele também é totalmente gratuito e possui API pública.

Para a criação de um novo *bot* na plataforma do Messenger, primeiro é necessário criar uma página no Facebook, em seguida, essa página é associada a uma nova aplicação através do painel de controle disponibilizado pelo Facebook. Após a criação da aplicação pelo painel é necessário a configuração de chaves de autenticação para que o *bot* tenha acesso as funcionalidades do Messenger.

Com o Messenger devidamente configurado através do painel de controle do Facebook, finalmente pode-se dar início ao desenvolvimento das funcionalidades do *bot*. Para isto é necessário configurar um *webhook*, um ponto de entrada na *web* que recebe requisições HTTP/HTTPS de outros sistemas ou clientes. Este *webhook*, que obrigatoriamente deve estabelecer comunicações utilizando o protocolo HTTPS, será utilizado pelo Messenger para repassar as mensagens recebidas ao *bot*, assim como enviar mensagens ao cliente. Assim como no Telegram, a criação do *webhook* e suas funcionalidades podem ser realizadas em qualquer linguagem que possua funcionalidade de realizar e receber requisições pela *web* e também é possível utilizar *frameworks* para facilitar o uso da API do Messenger, como por exemplo o BootBot que é disponibilizado para a linguagem de programação JavaScript.

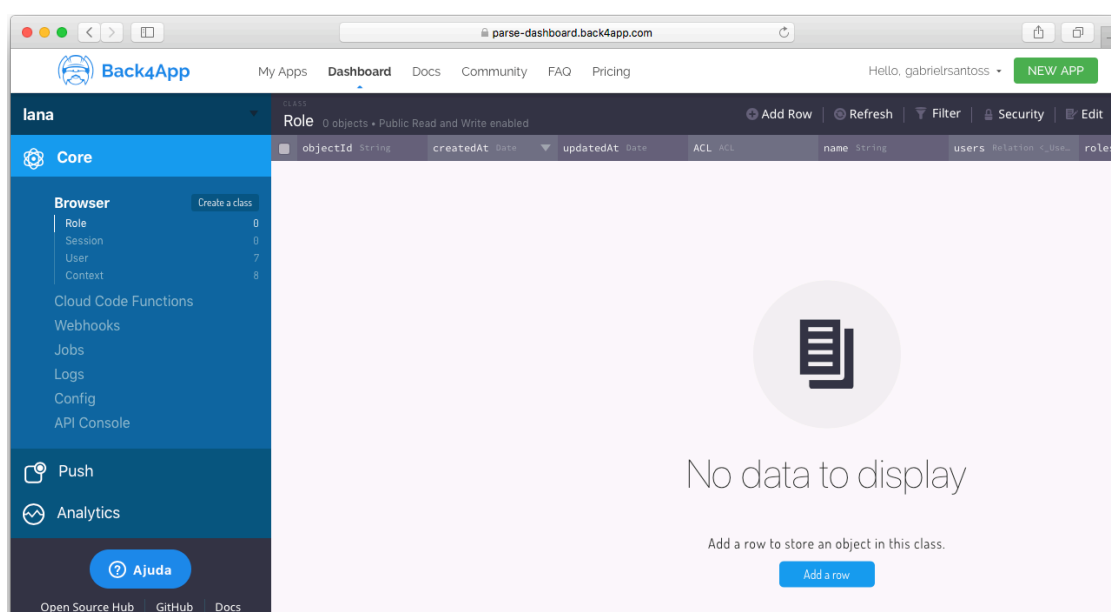
### 3.1.3 Back4App

Back4App é uma plataforma BaaS, baseada no Parse, um *framework* popular de *backend*, onde é possível criar e hospedar APIs para aplicações *web* e móveis de maneira mais rápida. O Back4App cria toda a estruturação de banco de dados MongoDB e sua API, além de outros recursos mais avançados para facilitar o gerenciamento de sistemas e acelerar o desenvolvimento. O sistema é bem

documentado e fornece uma enorme facilidade, comodidade e também é gratuito, portanto, estes foram os principais motivos pela escolha de uso do Back4App neste projeto.

A plataforma possibilita a criação de *Cloud Functions*, funções criadas pelo desenvolvedor que são armazenadas no Back4App e ficam disponíveis para serem executadas dentro da própria plataforma, sem a necessidade de criar uma API para realizar tais tarefas ou executa-las localmente. Além disso o Back4App conta com um painel de controle, apresentado na Figura 5, para gerenciamento de banco de dados, gerenciamento de *Cloud Functions*, controle de *logs*, configurações do servidor, envio de notificações entre outros diversos serviços ofertados.

Figura 5 - Painel de controle de uma aplicação no Back4App.



Fonte: Plataforma Back4App<sup>5</sup>.

Utilizando o painel de controle é possível criar novas classes para o banco de dados, novos campos para as classes e adicionar regras de segurança. O Back4App cria automaticamente uma classe de usuário já pré-configurada, e esta possui um campo criptografado para senha, um campo de email e um campo de nome de usuário. Além disso, todas as classes criadas possuem um campo definido automaticamente que armazena data de criação do objeto e um campo que armazena

<sup>5</sup> Disponível em: <<https://parse-dashboard.back4app.com/apps/{app-id}>>. Acesso em nov. 2018.

a data da última modificação, os valores atribuídos a estes campos são definidos automaticamente.

A integração com a plataforma pode ser realizada a partir das bibliotecas do Parse disponibilizadas para diversas linguagens de programação como JavaScript, PHP, C# e C, ou por meio de requisições HTTP/HTTPS através da REST API oferecida pelo serviço.

O uso de uma biblioteca cria maior facilidade ao lidar com o uso da plataforma, estas já possuem funções prontas para criação de novos objetos de uma classe, assim como alterar os campos deste objeto, criar novos campos e por fim enviar tudo ao banco de dados para ser armazenado. Além disso, a biblioteca conta com funções específicas para tratar com a classe de usuário, como funções de registro de novos usuários, autenticação e recuperação de senha. Por fim, a biblioteca oferece maior facilidade para a criação de rotinas de banco de dados e adição de novas funções para a API do banco. Estas funcionalidades não são exclusivas às bibliotecas, elas também podem ser utilizadas a partir da REST API, entretanto, será necessário a criação de novas funções para realizar estas requisições.

#### 3.1.4 DigitalOcean Droplet

A empresa DigitalOcean fornece um serviço de VPS *Hosting* chamado DigitalOcean Droplet. Um Droplet é um VPS com recursos adicionais de armazenamento, segurança e monitoramento para executar facilmente os aplicativos em produção (DIGITALOCEAN, 2018a).

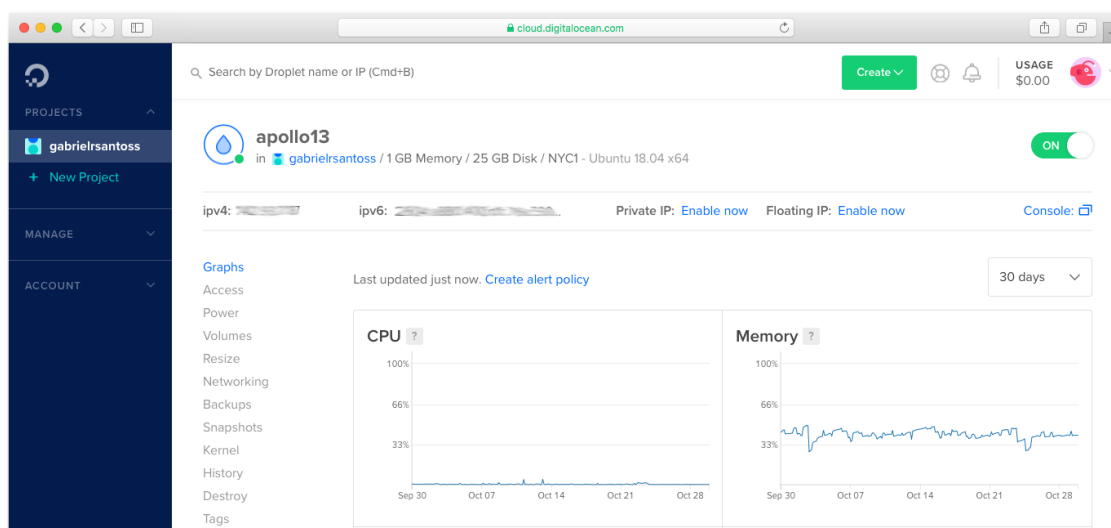
A DigitalOcean fornece o VPS com o sistema operacional já instalado, entretanto, para utilizar o Droplet é necessário configurá-lo de acordo com a sua finalidade, i.e., para utilizar o VPS como um servidor *web*, por exemplo, é responsabilidade do cliente instalar as ferramentas necessárias, assim como configurar o Droplet para receber requisições e atualizações de *softwares*.

Ao criar um Droplet, o DigitalOcean atribui a ele um novo endereço IP e um usuário e senha de administrador, estes podem ser utilizados para estabelecer uma conexão SSH com o VPS, tendo assim acesso a máquina e podendo configurá-la como desejar. Além disso, o DigitalOcean fornece um painel de controle, Figura 6, onde é possível monitorar o uso de atributos do Droplet, assim como realizar algumas

configurações como alteração do tamanho da memória, de CPUs, desligar o VPS ou até mesmo mudar o sistema operacional utilizado.

Apesar das tarefas de configuração e manutenção que um Droplet precisa, este continua sendo uma boa proposta pois oferece maior liberdade ao usuário quando se necessita instalar ferramentas que possuem restrições de sistema operacional ou dependências. Esta liberdade, aliada ao fato de ter conseguido um Droplet gratuito foram os motivos da escolha do DigitalOcean neste projeto.

Figura 6 - Painel de controle de um Droplet na plataforma do DigitalOcean.



Fonte: Plataforma do DigitalOcean<sup>6</sup>.

### 3.1.5 Heroku Dyno

“Heroku é uma plataforma de nuvem baseada em um sistema de contêiner gerenciado, com serviços e dados integrados e um poderoso ecossistema, para implementar e executar aplicativos modernos.”(HEROKU, 2018). O Heroku oferece um tipo de VPS, o Dyno. Ele é mais limitado que um DigitalOcean Droplet, entretanto com o Dyno não é necessário desenvolver toda a estruturação de um servidor *web*, somente a integração da aplicação nele.

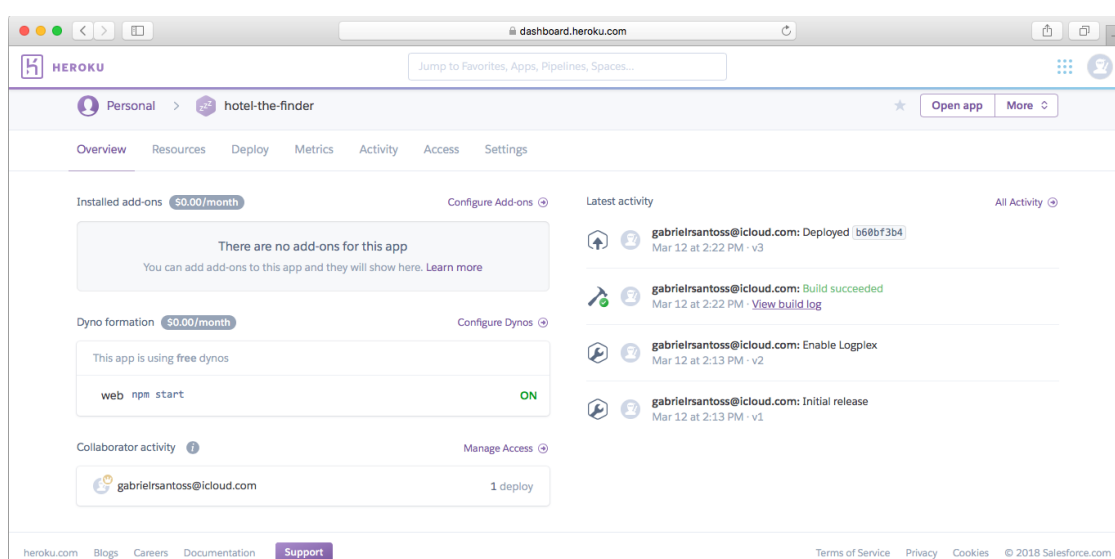
Um Dyno oferece facilidade ao desenvolvedor para lançar e manter serviços *web*, ele é um VPS já configurado como um servidor *web*, no qual somente é

<sup>6</sup> Disponível em: <<https://cloud.digitalocean.com/droplets/{droplet-id}>>. Acesso em nov. 2018.

necessário executar nele uma aplicação *web* que receba requisições em uma porta específica, esta porta é alterada sempre que ocorre alguma modificação na aplicação que esta sendo executada, ou sempre que o Dyno é reiniciado, o número da porta fica disponível no próprio servidor armazenado em uma variável de ambiente de nome “PORT”.

Todas as configurações do VPS podem ser acessadas através do painel de controle oferecido pelo Heroku, Figura 7. A partir deste painel é possível adicionar configurações de linguagens de programação que serão utilizadas no Dyno, adicionar novas variáveis de ambiente, monitorar *logs* e atividade do servidor, entre outras opções.

Figura 7 - Painel de controle de um Dyno na plataforma do Heroku.



Fonte: Plataforma do Heroku<sup>7</sup>.

A criação da aplicação é realizada localmente, e ao finalizar a implementação é enviado então o código fonte ao Dyno, utilizando o Git, e a aplicação enviada será então executada no VPS. O Heroku deixa o Dyno em repouso enquanto não existem novas requisições destinadas a ele, quando uma nova requisição chega então o Dyno é iniciado novamente para atender ao pedido. Ao enviar uma aplicação para o VPS, é informada uma URL ao usuário, por fim, as requisições HTTPS para o serviço podem ser realizadas para o Dyno por meio desta URL.

<sup>7</sup> Disponível em: <<https://dashboard.heroku.com/apps/{app-name}>>. Acesso em nov. 2018.

A facilidade para subir novos serviços para a *web* sem a necessidade de configuração de um servidor para aplicações mais simples junto a gratuidade do Heroku Dyno foram os motivos para a escolha desta ferramenta.

### 3.1.6 Selenium WebDriver

Selenium é uma ferramenta criada para testes automatizados de sistemas *web*, porém, o seu uso não se limita a isto, ela também pode ser utilizada para criação de *bots* extratores de informações, visto que é uma ferramenta que utiliza a técnica de *web scraping*. A documentação desta ferramenta está disponível online, o Selenium WebDriver possui bibliotecas para as linguagens de programação: Java, C#, Python, Ruby, PHP, Perl e JavaScript.

O WebDriver fornece ao desenvolvedor, funcionalidades para facilitar o *web scraping*. A biblioteca conta com funções para abrir URLs, clicar em elementos HTML, escrever em caixas de texto, arrastar elementos para algum local da página e também fornece funções que verificam se um elemento existe ou não, se está visível, ou até mesmo para monitorar o comportamento de algum elemento. Além disso, com o Selenium também é possível executar comandos JavaScript na página *web* (HOLMES; KELLOGG, 2006).

Para utilizar o Selenium WebDriver é necessário, além da instalação do próprio Selenium WebDriver, a instalação de um driver do navegador *web* a ser utilizado, como por exemplo, para utilizar o Google Chrome é necessário o *Chrome Driver* e para o Mozilla Firefox é preciso o *Firefox Driver*. Ao executar o Selenium é aberto o navegador e realizado automaticamente todas as ações que foram implementadas, estas ações podem ser visualizadas normalmente, pois por padrão a janela do navegador fica aberta, entretanto pode-se configurar o navegador, através do Selenium, para iniciar em modo *headless*, desta maneira somente o navegador é executado em segundo plano sem interface gráfica.

O Selenium foi escolhido para este projeto pois possui suporte a diversas linguagens de programação, além de uma sintaxe simples e uma variedade de funcionalidades, tornando-o assim uma ferramenta fácil de usar e ao mesmo tempo completa.



### 3.1.7 Linguagens de Programação

Para a criação de alguns componentes deste projeto, foram utilizadas as linguagens de programação JavaScript e Python.

#### 3.1.7.1 JavaScript

JavaScript é uma linguagem de programação de alto nível, dinâmica, não tipificada, interpretada, que fornece todos os recursos necessários para ser orientada a objeto e funcional (FLANAGAN, 2011). Originalmente ela foi criada como parte dos navegadores *web* para execução de instruções *client-side*, utilizando o interpretador de JavaScript do Google conhecido como V8.

Entretanto com o objetivo de tornar JavaScript uma linguagem que possa ser executada *server-side* foi criado o Node.JS, um interpretador *server-side* de JavaScript, baseado na implementação do V8, com ênfase em fornecer um bom desempenho e baixo consumo de memória (TILKOV; VINOSKI, 2010).

Com o Node.JS é possível criar aplicações *web* ou desktop inteiramente em JavaScript, sem a necessidade de uma outra linguagem *server-side*.

JavaScript foi escolhido para este projeto devido a possibilidade de uso dos *frameworks* TGFancy e BootBot disponíveis para a linguagem, o seu alto desempenho e a abstração de instruções mais complexas, visto que esta é uma linguagem de alto nível.

#### 3.1.7.2 Python

Python é uma poderosa linguagem de programação de fácil entendimento. Ela possui estruturas de dados de alto nível e uma abordagem simples, mas efetiva, de programação orientada a objeto. Python tem uma natureza interpretada, ideal para desenvolvimento de aplicações de maneira rápida e é comumente utilizada na maioria das plataformas (SWAROOP, 2003).

O Python foi escolhido para este projeto devido a sua compatibilidade com o Selenium WebDriver, além de possuir uma sintaxe fácil e de fácil entendimento.

### 3.1.8 Kanban

O desenvolvimento deste projeto foi realizado com base na metodologia ágil Kanban. Esta metodologia foi escolhida para este projeto pois ela é voltada para equipes pequenas ou somente um desenvolvedor, como é o caso deste projeto, além disto ela tem como objetivo aumentar consideravelmente a produtividade dividindo grandes trabalhos em pequenas tarefas, fazendo com que melhore as estimativas de tempo para a finalização de uma tarefa e diminuir os problemas trazidos por um grande trabalho.

Inicialmente o método Kanban foi aplicado em empresas japonesas de fabricação em série. A Toyota, empresa de automóveis, foi a responsável pela introdução do Kanban. Este método permite um maior controle de produção com informações precisas da quantidade de tarefas que foram realizadas, quantas estão em progresso e quantas ainda irão ser iniciadas (7GRAUS, 2015).

Para o funcionamento do Kanban é necessário utilizar um quadro para fixar cartões. Este quadro é dividido em 3 colunas, tarefas a fazer, tarefas em andamento e tarefas terminadas. Cada cartão representa uma tarefa e a cada etapa de produção de uma tarefa esta é movida para a coluna correspondente.

## 3.2 Métodos

Como se trata de um sistema distribuído, a implementação deste SAP foi modularizada, de forma que cada módulo possua uma funcionalidade específica. Portanto, foram implementados módulos de interface, banco de dados, entendimento de linguagem natural, assistente pessoal, indexador de *bots* e os *bots scrapers*.

Para estabelecer comunicação entre os módulos, foi criado uma IDL. Desta maneira, para um módulo se comunicar com outro basta ambos seguirem o padrão de comunicação estabelecido.

Para melhor interatividade com o Kanban foi utilizado a ferramenta de quadro virtual disponível no GitHub, um repositório Git gratuito que oferece diversas ferramentas para auxiliar o desenvolvedor a se organizar. Para o desenvolvimento foi dividida a implementação de cada módulo em pequenas tarefas e foram criados cartões para cada uma. Assim, a cada nova implementação o quadro era atualizado posicionando os cartões nas colunas correspondentes ao estado de cada tarefa. Além

disso, todos os códigos fontes dos módulos implementados foram versionados com o Git e estão armazenados no GitHub em repositório privado.

Este tópico tem como objetivo descrever o funcionamento e a implementação dos módulos criados, assim como quais e de que maneira as ferramentas foram utilizadas em cada módulo.

### 3.2.1 Interface

O módulo de interface tem como objetivo lidar com a troca de mensagens de texto entre o AP e o usuário final. As mensagens recebidas não são interpretadas neste módulo, apenas são encaminhadas em forma de requisições HTTPS para o módulo de assistente pessoal e em seguida a resposta obtida desta requisição é repassada ao usuário. Além da implementação do encaminhamento de mensagem, também foi necessário a criação de um ponto de acesso, este é utilizado pela Lana para enviar mensagens ao usuário assim que algum serviço requisitado finalizar sua execução.

A fim de realizar troca de informações entre um agente de interface e o AP, o agente deve seguir o padrão de comunicação estabelecido pela IDL do AP. A requisição para o AP sempre deve conter as seguintes informações:

- a) nome do agente de interface (ex.: Telegram ou Messenger);
- b) nome do usuário;
- c) número de identificação do usuário;
- d) URL do ponto de acesso;
- e) mensagem a ser encaminhada.

E a resposta recebida desta requisição sempre contém as seguintes informações:

- a) tipo de mensagem (ex.: texto ou imagem);
- b) variável booleana informando se a mensagem possui marcação de estilo;
- c) mensagem.

Já o ponto de acesso direto deve receber requisições que contenham as seguintes informações:

- a) número de identificação de usuário destinatário;
- b) tipo de mensagem (ex.: texto ou imagem);
- c) variável booleana informando se a mensagem possui marcação de estilo;
- d) mensagem.

### 3.2.1.1 Telegram

Visando o funcionamento do módulo de interface, primeiramente, foi implementado um *bot* para o aplicativo de mensagens Telegram, este foi desenvolvido utilizando a linguagem de programação JavaScript e o *framework* TGFancy.

A escolha de trabalhar a partir de um *framework* e não diretamente com a API do Telegram foi feita pois o TGFancy já implementa alguns tratamentos que são necessários para o envio de mensagens longas (com mais de 4096 caracteres), imagens ou mensagens com marcação de estilo, diferentemente das requisições diretas a API onde seria preciso implementar esses tratamentos.

Após escolher o modo de uso da API do Telegram e a linguagem de programação a ser utilizada, o passo seguinte para a implementação foi a criação de um novo *bot* a partir do próprio Telegram. Foi requisitado ao BotFather um novo *bot* com o nome “Lana” e nome de usuário “lana\_pa\_bot” e juntamente a confirmação de criação do novo *bot* foi recebido o *token* de acesso dele.

Com o *token* de acesso já informado, foi dado início à implementação das funcionalidades do *bot*, i.e., as funcionalidades necessárias para este *bot* se tornar um agente de interface da Lana. O TGFancy fornece eventos que são ativados ao receber novas mensagens e ao ativar um destes eventos ele pode realizar a chamada de uma outra função passando para ela a mensagem recebida e as informações de usuário.

Foi desenvolvida uma função, chamada “fowardMessageToLana”, que recebe uma mensagem de texto juntamente ao usuário remetente, faz uma requisição a uma API passando a mensagem recebida e por fim envia ao usuário remetente a resposta obtida da API requisitada. A implementação da requisição e recebimento de resposta da API seguiram os padrões estabelecidos pela IDL do AP. Por fim, o evento de

recebimento de mensagens, oferecido pelo TGFancy, foi configurado para executar “forwardMessageToLana” ao receber qualquer mensagem.

Após implementar a funcionalidade de encaminhamento de mensagens, foi criado um ponto de acesso chamado “sendMessageEndpoint”, este recebe uma requisição, que segue o padrão da IDL do AP, encaminha a mensagem recebida na requisição para o usuário de destino e por fim responde se houve algum erro ao encaminhar a mensagem.

Ao finalizar o desenvolvimento das funcionalidades do agente de interface, este foi hospedado em um Heroku Dyno para que possa estar disponível sempre que houver uma nova requisição ao ponto de acesso ou um novo evento de mensagem. O código fonte deste agente está disponível no Apêndice A.

### 3.2.1.2 Messenger

A implementação do agente de interface do Telegram já é o suficiente para o funcionamento do SAP, entretanto, o agente do Messenger foi desenvolvido como prova de conceito de que a implantação de um novo agente de interface não acarretará em nenhuma mudança nos outros módulos já desenvolvidos e para demonstrar a facilidade de migrar ou adicionar novos meios de comunicação ao SAP.

A criação do agente de interface do Messenger foi realizada utilizando a linguagem de programação JavaScript e o *framework* BootBot. O uso do *framework* deu-se devido ao leque de funções auxiliares implementadas nele, o BootBot faz o tratamento do envio de imagens, textos, cria indicadores de digitação proporcionando ao usuário final a sensação de que existe alguém realmente conversando com ele, além das opções de marcar mensagens como lidas ou recebidas.

Para criar o *bot* do Messenger, o primeiro passo realizado foi a criação de uma página no Facebook, o nome dado a página foi Lana. A partir do painel de controle do Facebook foi configurado um *webhook* para receber mensagens dos usuários para a página a partir do Messenger.

Ao finalizar a configuração do *webhook* no painel do Messenger, foi iniciada a implementação das funcionalidades requisitadas para que o *bot* se torne um agente de interface. Similar ao TGFancy, o BootBot também fornece eventos de recebimentos de mensagem com algumas diferenças. Ao receber uma mensagem pelo evento, esta não possui informações sobre o usuário remetente, entretanto, o evento recebe um

outro objeto além da mensagem em si, o “chat”. O objeto “chat” possui diversos métodos relacionados a atual conversação, entre eles o método “getUserProfile” que retorna o perfil do usuário que enviou aquela mensagem, além disso o “chat” fornece métodos de enviar novas mensagens ao usuário naquela conversa.

Utilizando o evento de recebimento de mensagens, foi implementado à funcionalidade o encaminhamento necessário ao agente de interface. Ao receber uma nova mensagem, este evento é ativado e então o *bot* usa o método “getUserProfile” do objeto “chat” para buscar mais informações sobre o remetente da mensagem. Em seguida é chamada uma nova função, de nome “forwardMessageToLana”, que tem o mesmo objetivo da função “forwardMessageToLana” implementada no agente do Telegram. Entretanto, a função implementada no agente do Messenger recebe não somente a mensagem e o usuário, como também recebe o objeto “chat” para continuar a conversação sem a necessidade da criação de um novo objeto.

Similar à implementação do agente do Telegram, neste agente de interface também foi criada uma função “sendMessageEndpoint”, que tem o mesmo propósito: criar um ponto de acesso para o envio de mensagens da AP ao finalizar algum serviço para o usuário.

A implementação deste agente de interface também seguiu os padrões estabelecidos pela IDL da AP, a fim de manter um meio de comunicação funcional entre os módulos. Ao finalizar a implementação das funcionalidades o *bot* foi implantado em um Heroku Dyno com a finalidade de estabelecer uma comunicação HTTPS com o Facebook e estar disponível sempre que necessário. O código fonte deste agente está disponível no Apêndice B.

### 3.2.2 Banco de Dados

O objetivo do módulo de banco de dados é armazenar informações referentes aos usuários do SAP e armazenar os contextos das conversas, estruturas que auxiliam o módulo de entendimento de linguagem natural a acompanhar o fluxo das conversações.

Para a implementação deste módulo foi utilizado a plataforma BaaS Back4App, pois esta proporciona facilidade e conforto para criar e gerenciar bancos de dados em nuvem. Também foi utilizado a linguagem de programação JavaScript para a criação de *Cloud Functions* a partir da biblioteca fornecida pelo Back4App.

O banco de dados provido pela pelo Back4app é o MongoDB, um banco de dados não relacional. O MongoDB apresenta em sua documentação dois tipos de modelagem de dados, o modelo de dados incorporado onde as classes criadas possuem todos o conteúdo sem necessitar acessar outras classes, e o modelo de dados normalizado, onde existe a separação de conteúdo entre as classes e é criado um esquema de relacionamento, parecido com a relação chave primária e estrangeira, apresentado pelos bancos de dados relacionais. Para este projeto foi utilizado o modelo de dados incorporado, onde as classes devem possuir todos os atributos necessários para uma consulta sem a necessidade de acessar outras classes através de referências.

Para dar início à implementação deste módulo, primeiramente, a partir do painel de controle do Back4App foi criado um novo projeto “Iana”. Para ajustar o projeto as necessidades do AP, a classe de usuário, criada por padrão, foi modificada adicionando um novo campo “nome” com o objetivo de armazenar os nomes dos usuários do SAP. Por fim, ainda no painel de controle, foi criada uma nova classe, chamada “Context”; nesta classe somente foi necessário criar um campo de nome “context” que armazena um valor do tipo objeto, com o objetivo de atribuir a este campo as estruturas de contexto das conversações.

Após ajustar as configurações do Back4App pelo painel de controle, deu-se início à implementação das *Cloud Functions* necessárias para o controle das operações no banco de dados. Foram criadas funções com o objetivo de obter um usuário a partir de um valor de um campo, adicionar, modificar ou remover o valor de um campo de um usuário, registrar novos usuários, fazer o *login* de um usuário, criar ou alterar um “context”, obter um “context” a partir de um número de identificação única de interface de um usuário e por fim foi criada uma rotina que é executada a cada 24 horas que tem como objetivo apagar do banco de dados informações que contenham senhas do usuário de sistemas de terceiros, como por exemplo uma senha de usuário do portal Sagres.

Apesar de não ter previamente configurado nenhuma conexão entre a classe “Context” e a classe de usuário, uma relação entre elas é criada a partir do momento que é registrado um novo usuário no banco de dados. Ao registrar um novo usuário, utilizando a *Cloud Function* responsável, é criado um novo campo na tabela de usuário. Este campo é construído para ter o nome do agente de interface que o usuário está utilizando e o valor atribuído a este campo é o número identificador único

informado pelo agente de interface. Por exemplo, se um usuário entrar em contato a partir do Telegram é criado um novo campo na classe usuário chamado “telegram” e o valor deste campo é atribuído para esse usuário como o número de identificação informado pelo agente de interface do Telegram, mesmo com a comunicação dos dois módulos não sendo realizadas diretamente, a Lana repassa essas informações a este módulo.

Similarmente a criação de um novo usuário, ao criar uma nova entrada para a classe “Context” a *Cloud Function* implementada também adiciona um novo campo a esta classe com o nome do agente de interface responsável por aquela conversação e o valor atribuído é o número de identificação única do usuário que possui o objeto “context” a ser armazenado.

Desta maneira, ao procurar por um contexto de conversa ou procurar informações de um usuário, é possível buscar a partir do número de identificação único do agente de interface e obter as informações referentes ao mesmo usuário em classes diferentes.

Por fim, todas as *Cloud Functions* criadas foram implantadas no Back4App e podem ser executadas via requisições HTTPS utilizando a REST API fornecida pelo próprio Back4App, a partir do painel de controle da plataforma ou utilizando o método “run” do objeto “Cloud” da biblioteca de desenvolvimento do Back4App. O código fonte das *Cloud Functions* pode ser observado no Apêndice C.

### 3.2.3 Entendimento de Linguagem Natural

O módulo de entendimento de linguagem natural tem como objetivo interpretar mensagens textuais, recebidas pelos usuários do SAP, identificando as intenções do usuário com o serviço de assistência pessoal e manter um fluxo contextual de conversação.

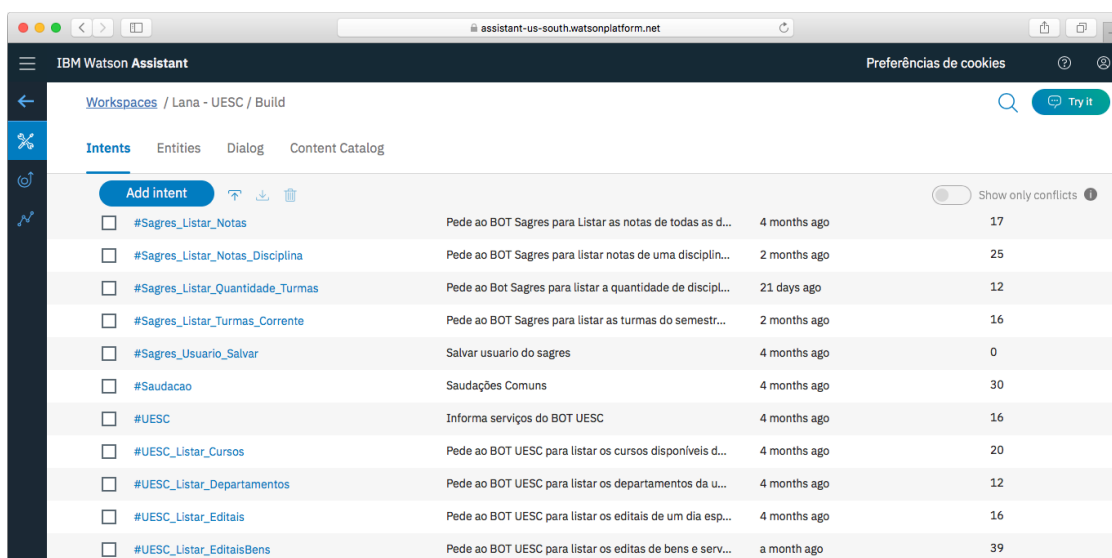
A implementação deste módulo foi realizada a partir do serviço do IBM Watson, o Watson Assistant. Através da plataforma IBM Cloud, foi criado um novo *workspace* para tratar das conversações da AP a respeito dos seus serviços disponíveis e algumas conversas triviais.

Com o *workspace* criado, deu-se início à configuração dele, foram criadas as *Intents*, *Entities* e *Dialogs* necessários para o funcionamento do serviço. Inicialmente foram cadastradas as *Intents* que demonstravam ações triviais em conversas, como



agradecimentos, negações, afirmações e saudações. Em seguida foram cadastradas as *Intents* que representavam os serviços disponíveis pela Lana, estas têm como objetivo identificar nas mensagens a intenção do usuário requisitar ao SAP a execução de algum serviço. Além das *Intents* de serviço, também foram criadas *Intents* para o registro do usuário ao SAP. Algumas podem ser vistas na Figura 8.

Figura 8 - Painel do serviço Watson Assistant com algumas Intents criadas para o workspace Lana.



Intents	Entities	Dialog	Content Catalog
<a href="#">Add intent</a>			
<input type="checkbox"/> #Sagres_Listar_Notas	Pede ao BOT Sagres para Listar as notas de todas as d...	4 months ago	17
<input type="checkbox"/> #Sagres_Listar_Notas_Disciplina	Pede ao BOT Sagres para listar notas de uma disciplin...	2 months ago	25
<input type="checkbox"/> #Sagres_Listar_Quantidade_Turmas	Pede ao Bot Sagres para listar a quantidade de discipl...	21 days ago	12
<input type="checkbox"/> #Sagres_Listar_Turmas_Corrente	Pede ao BOT Sagres para listar as turmas do semestr...	2 months ago	16
<input type="checkbox"/> #Sagres_Usuario_Salvar	Salvar usuario do sagres	4 months ago	0
<input type="checkbox"/> #Saudacao	Saudações Comuns	4 months ago	30
<input type="checkbox"/> #UESC	Informa serviços do BOT UESC	4 months ago	16
<input type="checkbox"/> #UESC_Listar_Cursos	Pede ao BOT UESC para listar os cursos disponíveis d...	4 months ago	20
<input type="checkbox"/> #UESC_Listar_Departamentos	Pede ao BOT UESC para listar os departamentos da u...	4 months ago	12
<input type="checkbox"/> #UESC_Listar_Editais	Pede ao BOT UESC para listar os editais de um dia esp...	4 months ago	16
<input type="checkbox"/> #UESC_Listar_EditaisBens	Pede ao BOT UESC para listar os editas de bens e serv...	a month ago	39

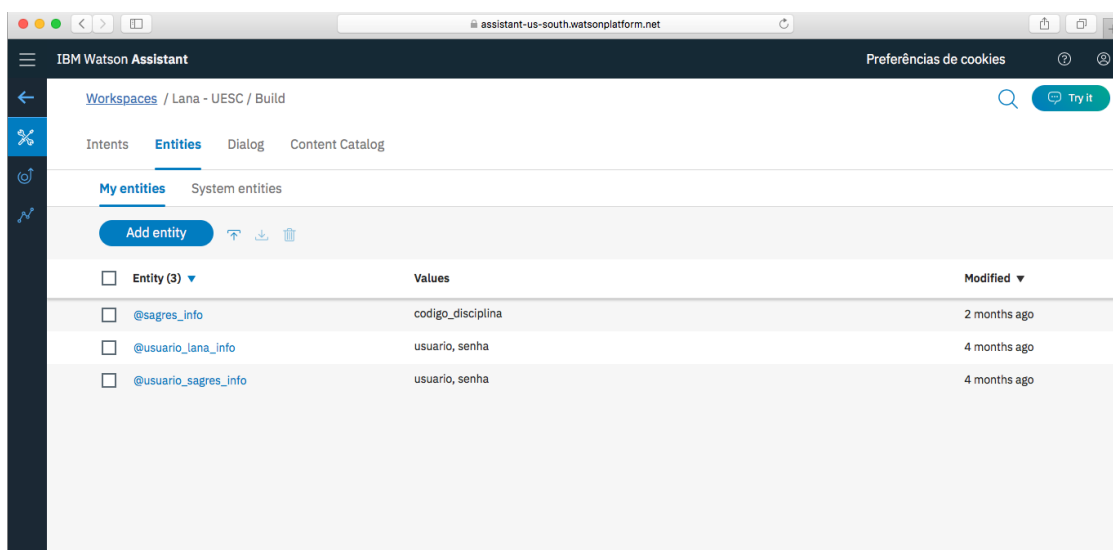
Fonte: elaborado pelo autor.

Em seguida, foram cadastradas novas *Entities* para identificar informações necessárias à execução de um serviço e para o cadastro de usuário do SAP. Foram criadas as *Entities* para reconhecer códigos de disciplinas da UESC “sagres\_info”, nome de usuário e senha do portal Sagres “usuario\_sagres\_info” e do próprio SAP “usuario\_lana\_info”, estas podem ser observadas na Figura 9.

Após a criação de *Intents* e *Entities*, foram implementados os *Dialogs*, objetivando a criação do fluxo de conversação de acordo com cada *Intent* identificada nas mensagens textuais.

Primeiramente foram implementados os fluxos de conversa para o cadastro ou *login* de um usuário, Figura 10, o nó responsável pelo inicio deste fluxo de conversa é o “Bem-vindo”, este é o nó criado por padrão que sempre é primeiro a ser executado em novas conversações.

Figura 9 - Painel do serviço Watson Assistant com todas as Entities criadas para o workspace Lana.



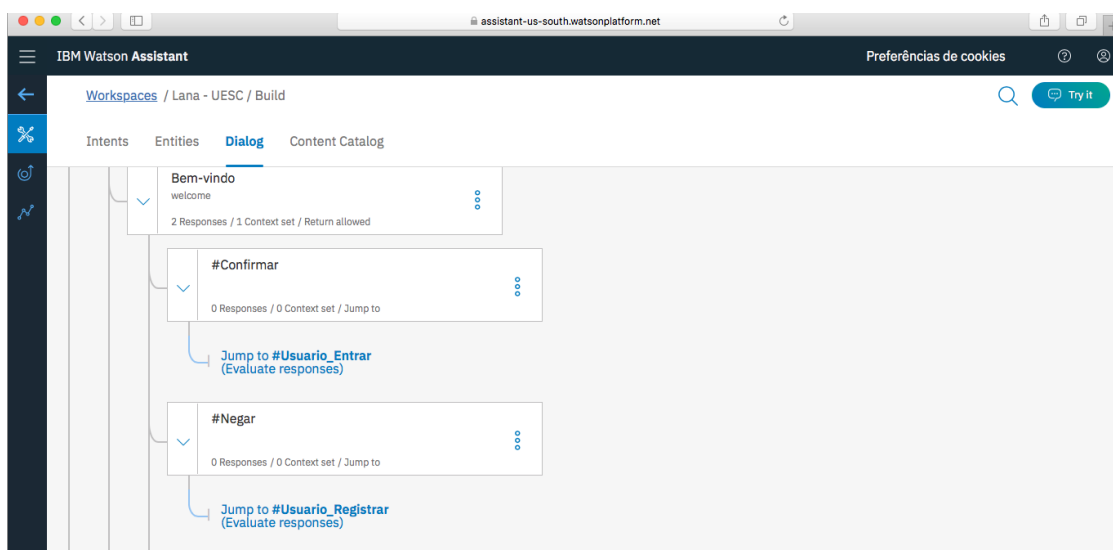
Fonte: elaborado pelo autor.

Ao entrar no nó “Bem-vindo” é perguntado ao usuário se ele já possui uma conta com o SAP, caso seja confirmado a existência de uma conta então o fluxo muda para o nó responsável pela conversação com usuários que desejam fazer *login*, nó “#Usuario\_Entrar”, caso seja negada a existência de uma conta então o a conversa será movida para o nó responsável por criações de novas contas, nó “#Usuario\_Registrar”.

As criações dos nós responsáveis pela identificação de intenções triviais foram realizadas em seguida, todos estes nós possuem ações parecidas, onde somente é necessário identificar a intenção do usuário e responder algo que tenha sentido à sua intenção de pergunta, por exemplo, a criação do nó de saudação somente necessita responder uma outra saudação ao usuário, assim como um nó que identifica a intenção de despedida precisa somente responder mensagem que contenha uma despedida.

Ao finalizar a criação dos fluxos básicos de conversa, foram implementados os nós responsáveis por gerar respostas quando identificada a necessidade de execução de um serviço, e para cada serviço disponível foi criado um novo nó.

Figura 10 - Painel do serviço Watson Assistant com parte do fluxo de primeira conversa criada para o workspace Lana.



Fonte: elaborado pelo autor.

As implementações de respostas dos serviços foram realizadas de maneira diferente das respostas dadas a *Intents* triviais, todos os serviços implementados respondem textualmente ao usuário. Entretanto, junto com a mensagem são enviadas algumas informações necessárias para a execução do serviço, sendo estas:

- a) nome do serviço;
- b) *entities* necessárias para a realização do serviço;
- c) nome do Indexador de Bot responsável pela execução do serviço.

Finalmente, com o *workspace* totalmente configurado, foi criada uma classe de fachada com o objetivo de facilitar o uso da API do serviço Watson Assistant. Esta classe foi implementada com a linguagem de programação JavaScript e tem por finalidade ser utilizada pelo módulo de assistente pessoal, nela foram implementados métodos que utilizam a biblioteca do Watson Assistant para estabelecer comunicação com o serviço e desta maneira usufruir do *workspace* criado anteriormente. Desta maneira, o módulo de assistente pessoal não depende do IBM Watson diretamente, mas sim de uma classe de fachada que deve realizar o serviço de entendimento de linguagem natural.

### 3.2.4 Assistente Pessoal

O módulo de assistente pessoal é a implementação da Lana em si, e ele é responsável por estabelecer comunicação com os outros módulos e realizar toda a lógica de operação para o funcionamento do AP. Este módulo realiza suas ações baseadas nas respostas obtidas pelo módulo de entendimento de linguagem natural. Desta maneira, ele é responsável pela decisão de criar novos usuários, inserir novas informações no banco de dados e iniciar serviços. A implementação deste módulo foi realizada com a linguagem de programação JavaScript.

Primeiramente, foi necessário desenvolver métodos para criar um meio de comunicação entre todos os módulos do sistema. Foram criadas classes que serviram como fachada para acessar funções específicas dos módulos de banco de dados e de entendimento de linguagem natural. O funcionamento do assistente pessoal segue um fluxo lógico de comunicação, ao receber uma requisição de um módulo de interface o assistente envia a mensagem recebida ao módulo de entendimento de linguagem natural e baseando-se na sua resposta decide se é necessário armazenar uma nova informação no banco de dados, dar início a um serviço, criar uma nova conta ou não realizar nenhuma ação e somente responder de volta ao usuário.

Caso a mensagem recebida tenha a intenção de iniciar um novo serviço, a Lana responde ao usuário informando que o serviço requisitado será realizado e faz uma requisição ao módulo indexador de *bots*, o Bothub, pedindo a execução de um serviço, passando a ele as informações necessárias para o funcionamento deste.

Por fim, ao receber a resposta do Bothub, a Lana encaminha a resposta do serviço realizado, através do ponto de acesso direto do agente de interface, ao responsável pela comunicação com o usuário requisitante.

As requisições realizadas a um Bothub devem seguir ao padrão estabelecido pela IDL, este pedido deve conter sempre o nome do serviço a ser executado e caso haja alguma informação adicional esta deve ser passada juntamente a requisição.

Finalmente, ao finalizar a implementação do módulo de assistente pessoal, foi criada uma API para que os agentes do módulo de interface possam realizar requisições a Lana. Em seguida, este módulo foi implantado em um Heroku Dyno para que possa estar disponível sempre que for necessário realizar qualquer ação com o SAP.

### 3.2.5 Indexador de Bots

Objetivando a redução de acoplamento e dependência entre o módulo de assistente pessoal e os *bots* extratores de dados, foi criado um módulo indexador de *bots*, denominado Bothub. Um Bothub é a implementação de um índice de *bots* que realizam ações que possuem o mesmo domínio, para este projeto foi implementado somente um indexador, o Bothub UESC, utilizando a linguagem de programação Python. O Bothub UESC é responsável por indexar os *bots* que realizam serviços no contexto da universidade.

Os Bothubs são responsáveis por manter controle das requisições realizadas aos *bots* que estão indexados neles, assim como ter conhecimento de quais são os serviços disponíveis em cada *bot*. Desta maneira, ao receber uma requisição informando qual serviço precisa ser realizado e quais informações devem ser passadas, o Bothub deve ser capaz de encaminhar este pedido ao *bot* que irá atender as necessidades desta requisição.

Ao realizar uma requisição a um Bothub do módulo indexador de *bots*, espera-se sempre que a resposta obtida siga o padrão da IDL estabelecida pela AP, caso o serviço tenha sido executado sem nenhum problema, a resposta obtida deve sempre conter as seguintes informações:

- a) tipo de mensagem (ex.: texto ou imagem);
- b) variável booleana informando se a mensagem possui marcação de estilo;
- c) mensagem.

Caso ao executar um serviço ocorra algum tipo de erro, o Bothub deve responder a requisição com estas informações:

- a) tipo de mensagem (ex.: texto ou imagem);
- b) variável booleana informando se a mensagem possui marcação de estilo;
- c) mensagem;
- d) variável booleana informando que houve um erro ao executar o serviço atribuída com o valor “verdade”;

e) informações enviadas a requisição que ocasionaram este erro.

Desta maneira, a Lana não precisa saber qual *bot* é responsável por um serviço específico, mas somente em qual contexto esse serviço se encaixa e requisitar o serviço ao BotHub deste âmbito, esta comunicação indireta do AP com os *bots* extratores de dados pode ser observada na Figura 11 apresentada no início deste tópico. Por fim, após a implementação do Bothub UESC, este foi hospedado em um Dyno na plataforma Heroku.

### 3.2.6 Bots Scrapers

Visando a execução dos serviços disponibilizados pelo SAP, foram implementados dois *bots* que utilizam a técnica de *web scraping* para extrair informações do site da UESC, BotUESC, e do portal acadêmico Sagres, BotSagres. A implementação destes *bots* foi realizada utilizando a linguagem de programação Python e o *framework* Selenium WebDriver para facilitar o uso do *web scraping*.

Para o funcionamento do Selenium, como já explicado no tópico 3.1.6, é necessário instalar um *driver* do navegador *web* a ser utilizado. Este processo realiza algumas ações que necessitam de maior controle do ambiente que esta sendo configurado, por este motivo, ao finalizar a implementação dos extratores de dados, ambos foram implantados em um DigitalOcean Droplet.

Tendo em vista a automatização de uma implantação em um Droplet, onde diferentemente de um Heroku Dyno não é oferecido esta funcionalidade por padrão, foi criado um repositório Git para cada *bot scraper* no Droplet, e em seguida foram desenvolvidos *scripts* para a implantação automatizada de cada *bot* ao ser enviado ao seu repositório.

A implementação dos *bots* exigiu a criação de suas respectivas APIs para que o Bothub UESC pudesse realizar requisições diretamente aos *bots* quando necessário. Desta maneira, os *bots* precisaram seguir a IDL previamente estabelecida para que a comunicação entre o indexador e o extrator seja realizada sem problemas, sendo assim, a configuração adotada exige que o ponto de acesso a requisição seja o nome do serviço a ser realizado e as informações passadas pela requisição são os parâmetros necessários para cada ação. Por exemplo, uma requisição ao ponto de acesso “uesc\_listar\_ultimos\_editais” deve realizar a ação de listar os últimos editais

do site da UESC. Já a resposta enviada ao resolver uma requisição deve seguir o mesmo padrão de resposta do Bothub, apresentado anteriormente no tópico 3.2.5.

#### 3.2.6.1 BotSagres

O BotSagres foi implementado visando a realização de ações no portal acadêmico Sagres. O desenvolvimento deste *scraper* foi realizado observando o funcionamento do portal e identificando quais ações eram necessárias para extrair os dados do sistema.

Primeiro foi decidido quais seriam as funcionalidades implementadas no BotSagres, elas foram divididas em serviços para alunos e para professores. As ações disponíveis para os alunos decididas foram:

- a) calcular o coeficiente de rendimento acadêmico;
- b) enviar uma imagem com os horários de aula;
- c) listar todas as disciplinas cursadas;
- d) listar as disciplinas que estão sendo cursadas;
- e) listar a quantidade de faltas em todas as disciplinas cursadas;
- f) listar a quantidade de faltas em uma disciplina específica;
- g) listar as médias de todas as disciplinas cursadas;
- h) listar os créditos de uma disciplina específica.

Já para os professores, foi decidido que as seguintes funcionalidades seriam implementadas:

- a) enviar uma imagem com os horários de aula;
- b) listar as turmas do semestre atual;
- c) informar a quantidade de alunos matriculados em uma disciplina específica;
- d) informar a quantidade de disciplinas que já foram ministradas;
- e) calcular a carga horária semanal ministrada.

Em seguida, foi estudado o que seria preciso para implementar estas funcionalidades. Observou-se que para fazer qualquer serviço no portal é necessário

realizar *login* no sistema utilizando o nome de usuário e senha de um aluno ou professor, logo, estas credenciais devem sempre ser informadas ao requisitar a execução de uma das funcionalidades disponíveis. Dentre os serviços oferecidos, somente os que precisam de uma disciplina específica necessitam de outras informações, nestes casos é necessário informar o código desta disciplina.

Por fim, todas as funcionalidades foram implementadas e testadas, e finalmente o BotSagres foi implantado no Droplet para que poder ser acessado a qualquer momento.

### 3.2.6.2 BotUESC

A implementação do BotUESC visa extrair informações diretamente do site da UESC. Similarmente ao BotSagres, o desenvolvimento deste *bot* foi realizado a partir de observações do funcionamento do sistema, na tentativa de descobrir as informações necessárias para realizar cada ação.

Após analisar possíveis utilidade e viabilidade da obtenção de algumas informações no site da UESC, foi decidido a implementação das seguintes funcionalidades:

- a) listar notícias recentes;
- b) listar notícias de um dia específico;
- c) listar resultados recentes;
- d) listar editais recentes;
- e) listar editais de um dia específico;
- f) listar editais de aquisição de bens e serviços recentes;
- g) listar editais de aquisição de bens e serviços de um mês específico;
- h) listar os cursos ofertados pela UESC junto ao *site* do colegiado de cada;
- i) listar os departamentos da UESC e os *sites* de cada.

Observou-se que para realizar algumas ações não seria necessária nenhuma informação adicional, entretanto, serviços que realizam pesquisas em um dia ou mês específico exigiriam que fosse informado a data para realizar tal pesquisa.



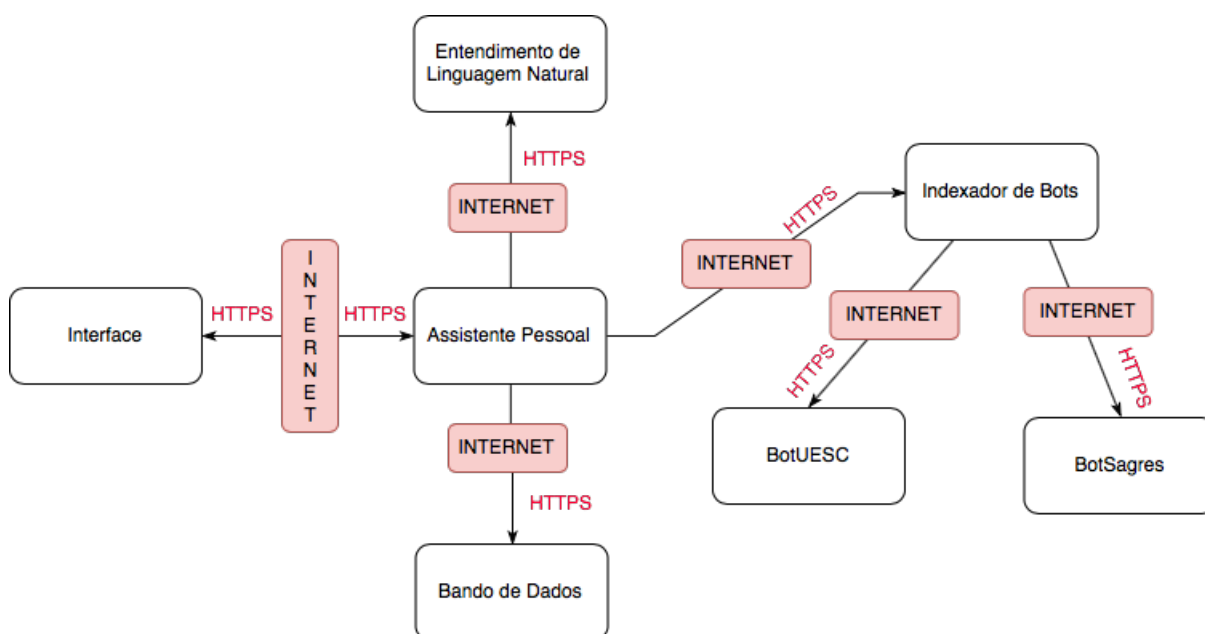
Finalmente, todos os serviços propostos foram devidamente desenvolvidos e testados, por fim, o *bot* foi implantado com sucesso no DigitalOcean Droplet e pode ser requisitado para realizar qualquer um dos serviços disponíveis.

## 4 RESULTADOS E DISCUSSÕES

Ao finalizar a implementação de todos os módulos, e estabelecer a comunicação entre eles, a Lana já estava em funcionamento com todos os seus serviços disponíveis para acesso ao público a partir do *bot* do aplicativo de mensagens Telegram. A topologia de comunicação entre os módulos é representada na Figura 11, nela podemos observar que todas as comunicações entre os módulos são realizadas através da internet utilizando o protocolo HTTPS.

A Figura 11 deixa claro que o módulo de interface realiza requisições ao módulo de assistente pessoal, assim como o módulo de assistente pessoal realiza requisições ao de interface, entretanto, somente o assistente realiza requisições aos outros módulos, exceto aos *bots scrapers* onde as requisições a eles são realizadas pelo indexador, e estes outros módulos não realizam requisições ao assistente, somente enviam respostas.

Figura 11 - Topologia de comunicação entre módulos.

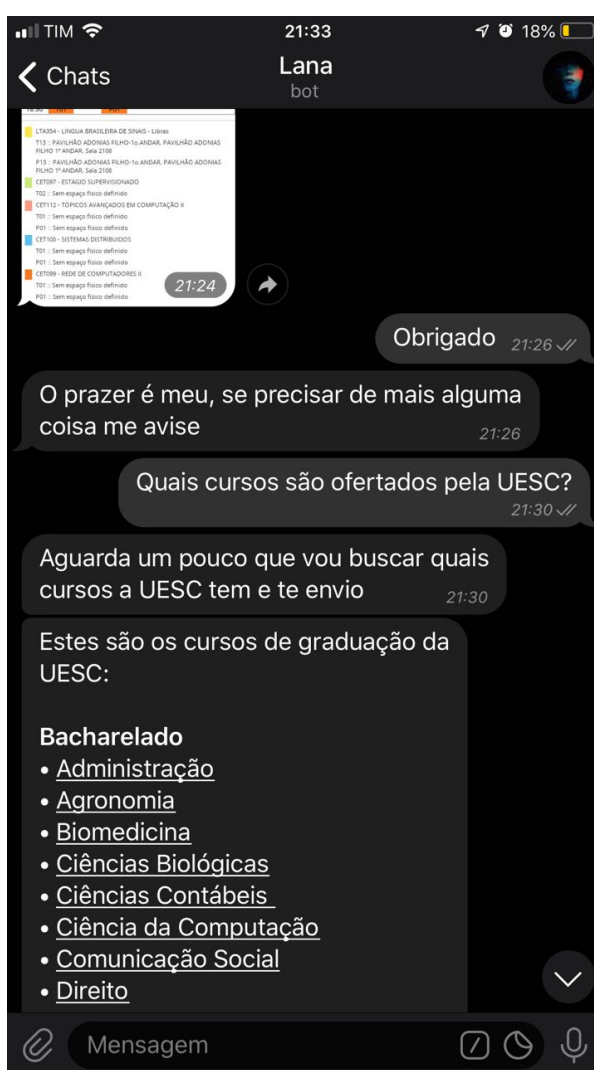


Fonte: elaborado pelo autor.

Para este projeto não foram gerados outros diagramas além da topologia apresentada na Figura 11, visto que este trabalho foi implementado utilizando a metodologia ágil Kanban, onde não é obrigatório a criação de diagramas.

A Lana pode ser vista em funcionamento no Telegram na Figura 12, nesta é mostrado a assistente pessoal respondendo a um agradecimento e em seguida é solicitado que ela liste os cursos disponíveis na universidade, a assistente avisa ao usuário que ela executará esta ação e retornará com a resposta e então, após realizar a busca das informações, ela envia uma mensagem de texto contendo a lista de cursos da universidade com os links para o *site* de cada colegiado.

Figura 12 - Conversação mostrando a Lana em funcionamento no aplicativo Telegram.

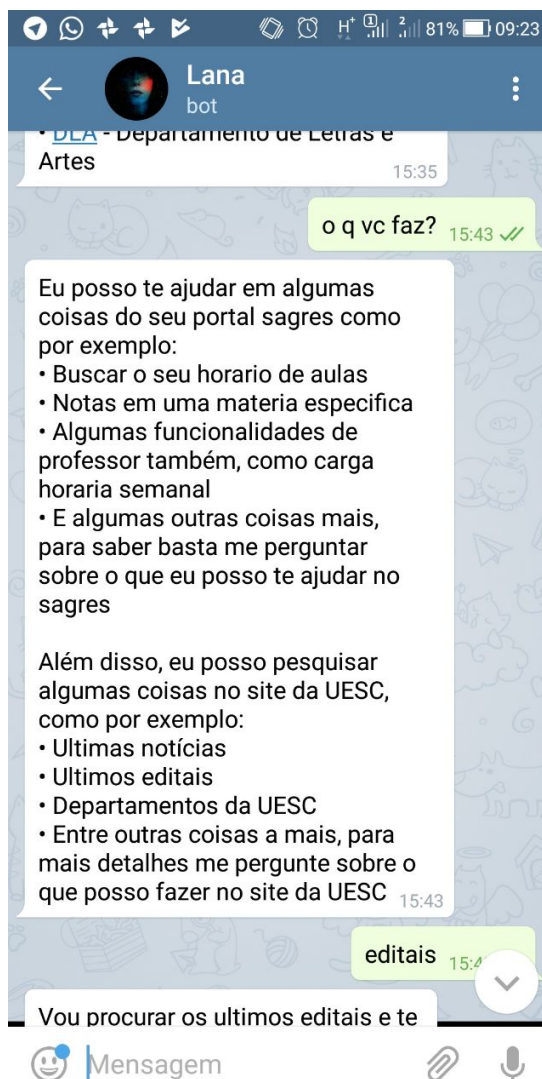


Fonte: elaborado pelo autor.

A Figura 13 mostra um cliente perguntando a Lana, a partir do Telegram, quais são os serviços realizados por ela, então é respondido de maneira geral algumas das principais ações implementadas. Nota-se que apesar da mensagem conter

abreviações como “q” e “vc” a Lana ainda consegue extrair a intenção do usuário de saber quais serviços ela pode realizar.

Figura 13 - Lana informando suas principais funcionalidades.



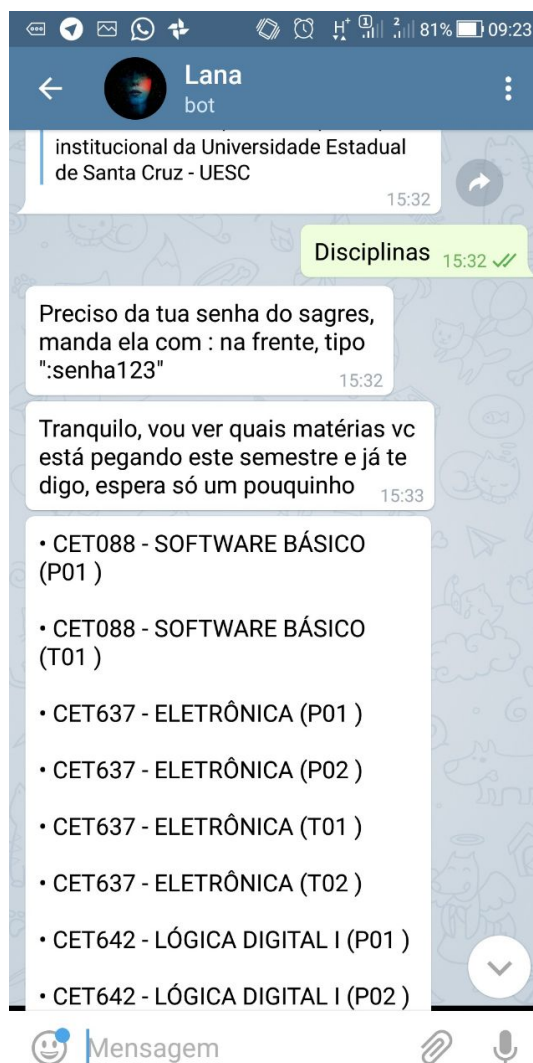
Fonte: elaborado pelo autor.

A

Figura 14 apresenta a Lana realizando ações para um professor por meio da aplicação Telegram, nela é possível observar a requisição da listagem de disciplinas ministradas com uma mensagem mais direta, somente falando “Disciplinas” e ainda assim foi possível extrair a intenção, em seguida a AP requisita ao cliente a senha do

portal Sagres para realizar tal operação, visto que este usuário já havia utilizado a Lana posteriormente não foi necessário informar o nome de usuário, após receber a senha foi realizado o serviço e enviado a resposta ao cliente.

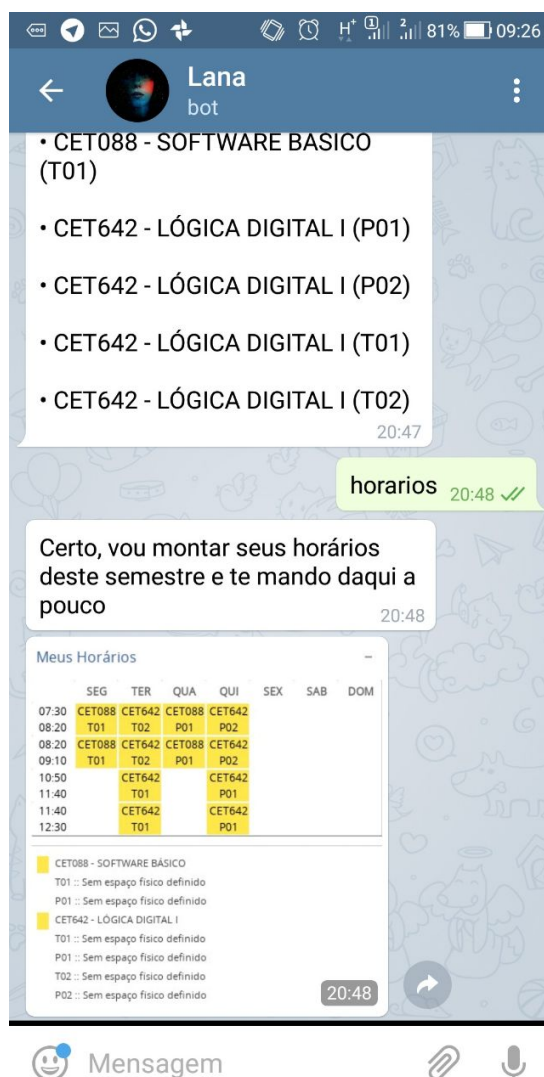
Figura 14 - Lana realizando ações para professor no Telegram.



Fonte: elaborado pelo autor.

Em seguida a Figura 15 apresenta a realização de uma outra requisição para o envio dos horários de aulas a serem ministrados no semestre, por fim a Lana responde a essa requisição enviando uma imagem contendo estes horários. Nesta Figura também se observa que foi possível extrair a intenção de buscar os horários mesmo em uma mensagem mais direta.

Figura 15 - Lana realizando ações para professor no Telegram.

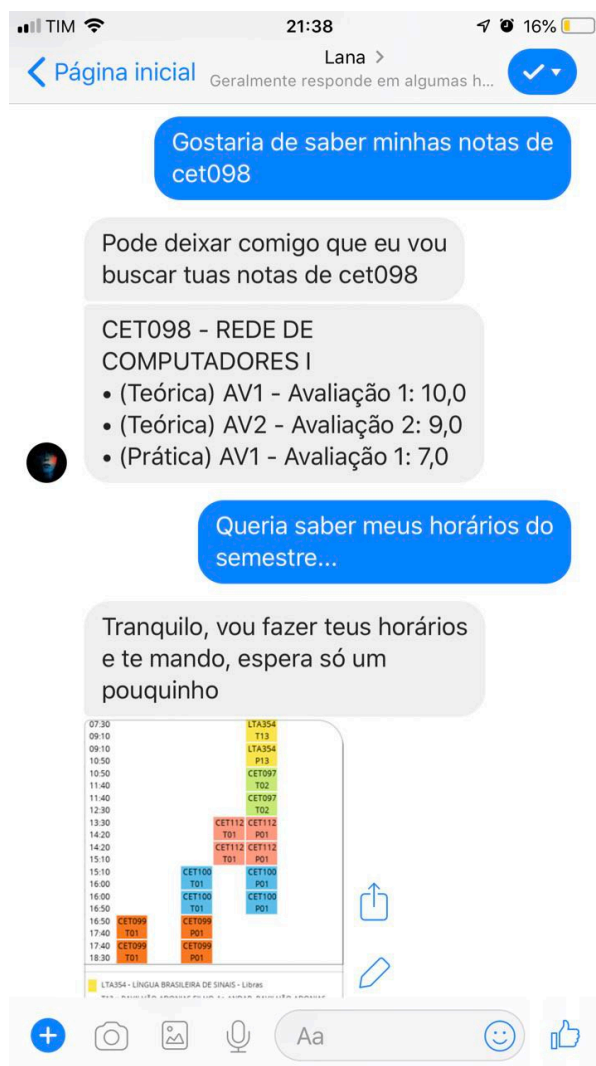


Fonte: elaborado pelo autor.

Na Figura 16, podemos observar o funcionamento da Lana no aplicativo de troca de mensagens Messenger, nesta é apresentado duas requisições de serviço à Lana, uma para a busca de notas da disciplina de código “CET098”, onde é retornado ao usuário a lista de notas referentes as avaliações desta matéria. Em seguida, outra requisição para que a assistente busque os horários de aula no portal Sagres, nesta a Lana envia como resposta uma imagem que contem o horário de aulas semanais do requisitante. Pode-se notar que a código da disciplina foi extraído da mensagem em conjunto a intenção, desta maneira o AP já conseguiu identificar que deveria executar um serviço para o código em questão, além disso, a Figura também mostra

que é possível requisitar o mesmo serviço de maneiras diferentes, assim como na Figura 15, é requisitado a Lana os horários do semestre, entretanto na Figura 16 o pedido foi realizado diferente, de maneira mais contextualizada.

Figura 16 - Conversação mostrando a Lana em funcionamento no aplicativo Messenger.



Fonte: elaborado pelo autor.

Apesar do agente de interface do Messenger ter sido implementado, este só pode ser utilizado pelo criador da página do Facebook, para liberar o acesso ao público é necessário enviar uma requisição de análise ao Facebook, e somente após esta análise e liberação é concedido o uso do *bot* do Messenger ao público geral. O

desenvolvimento dos agentes de interface foi realizado assim como o esperado, sem ocorrer nenhum problema.

A utilização da plataforma BaaS Back4app facilitou e acelerou a implementação do módulo de banco de dados, visto que não foi necessária criar a infraestrutura de um banco de dados, assim como não foi preciso criar uma API para as operações básicas do banco. Mesmo sendo necessário o estudo de novas ferramentas para o uso da plataforma, como a biblioteca do Parse, somente foi preciso estudar a documentação disponibilizada pelo Back4app para a criação das funcionalidades deste módulo.

O Watson Assistant realizou todo o trabalho de entendimento de linguagem natural. Graças a este serviço foi possível focar grande parte do desenvolvimento deste trabalho na estruturação do projeto em si e nas suas funcionalidades. A interface oferecida pela IBM para o uso do serviço é intuitiva e possui documentação explicando todos os conceitos necessários para o uso do Watson Assistant, o que ajudou na configuração do serviço para o uso no módulo de entendimento de linguagem natural.

Entretanto, alguns problemas foram enfrentados ao decorrer da configuração do Watson Assistant. Ao identificar a intenção de um usuário realizar algum serviço do BotSagres, era sempre necessário buscar as informações de autenticação do portal Sagres do cliente e só depois realizar o serviço, apesar do Watson Assistant dar suporte a mudança de contextos e pulos entre nós, só foi possível resolver este problema manualmente no módulo de assistente pessoal.

Também foi enfrentado um problema para identificar disciplinas da universidade, pelo nome, inserida em uma mensagem de texto sem nenhum tipo de marcação, visto que a UESC possui diversas disciplinas e cada uma tem um nome diferente, assim como os usuários poderiam escrever o nome de uma mesma disciplina de forma diferente ou abreviada, desta maneira. Uma solução possível seria a criação de uma *Entitie* para cada disciplina e nessa seria cadastrada as formas diferentes de escrever o nome da matéria, entretanto esta solução fica inviável devido ao grande número de disciplinas por curso da UESC e pelo número limite de *Entities* por *workspace* no plano gratuito do Watson Assistant, desta maneira, foi necessário utilizar o código da disciplina para a identificação de uma menção a uma matéria em uma mensagem de texto, já que os códigos de disciplinas da universidade seguem um padrão de no mínimo 3 e no máximo 4 letras seguidas de 3 números. Além disso, as intenções algumas vezes não são identificadas corretamente quando as



mensagens de texto possuem erro de ortografia. Apesar dos problemas enfrentados, um maior estudo do funcionamento do Watson Assistant poderia ajudar a resolver estes mesmos problemas com uma abordagem diferente sem a necessidade de repassar a competência de resolução a outro módulo.

Assim como planejado, o módulo de assistente pessoal foi totalmente construído para realizar grande parte das suas ações de forma genérica, sendo as únicas ações específicas as de realizar operações com as informações dos usuários, como remover, salvar ou alterar, entretanto essas funcionalidades não são específicas a variáveis e informações pré-determinadas, mas sim de forma dinâmica, bastando seguir aos padrões de comunicação da IDL.

Porém, foi necessária a criação de uma resolução para o problema de gerenciamento de contexto ao tratar de serviços do BotSagres, que foi apresentado neste tópico. Entretanto, até mesmo este problema específico foi tratado de maneira genérica, podendo ser aplicada a mesma solução, sem a mudança de código no módulo de assistente pessoal, para futuros serviços que possuam a mesma característica de necessitar informações de autenticação para ser executado.

Durante o desenvolvimento do BotUESC foram enfrentados alguns problemas e empecilhos gerados pela má estruturação HTML e falta de padronização apresentada pelo *site* da universidade. Algumas funcionalidades pensadas foram descartadas durante a análise de viabilidade devido à falta de organização das informações. Por exemplo, uma ação que viria a ser implementada seria a de encontrar o nome, localização no campus e currículo dos discentes de cada curso. Entretanto, esta não é viável pois o *site* não possui nenhuma padronização para dispor estas informações, alguns cursos disponibilizam estes dados em forma de tabela, outros em arquivos do tipo PDF e até mesmo nomes diferentes no menu de opções que possuem a mesma funcionalidade. Estas incoerências na disposição dos dados no site dificultam a automatização da recuperação de informações, pois seria necessário implementar diferentes maneiras de buscar dados semelhantes para cada curso e departamento da universidade, o que deixaria o código fonte mais complexo e dificultaria a manutenção do *bot* caso houvesse novas mudanças na estruturação das páginas. A dificuldade e extensão de implementações específicas foram os motivos para a inviabilização desta e de outras mais ações. Por fim, todas as funcionalidades propostas foram implementadas sem problemas e obtiveram

resultados corretos, como já esperado, visto que parte das funcionalidades desejadas já haviam sido consideradas inviáveis.

Já o desenvolvimento do BotSagres foi realizado sem empecilhos, todas as funcionalidades desenvolvidas foram testadas e obtiveram resultados corretos. Somente um problema foi identificado durante a implementação deste extrator de dados. A comunicação entre todos dos módulos do SAP é realizada de maneira criptografada utilizando o protocolo de comunicação HTTPS e todas as informações de usuários armazenadas são criptografadas, entretanto o *site* do portal Sagres não implementa nenhum tipo de segurança com criptografia das informações de autenticação para acesso ao sistema, toda a sua comunicação é realizada por meio do protocolo HTTP, o que gera uma falha de segurança expondo os nomes de usuário e senhas de alunos e professores para a rede ao acessar o sistema.

## 5 CONCLUSÃO

Neste trabalho foi proposto a implementação de um *software* de assistência pessoal distribuído capaz de receber mensagens textuais, interpretá-las, executar serviços quando necessário, e retornar ao usuário final uma resposta trivial ou um resultado da execução de um serviço requisitado, assim como os seus componentes responsáveis por estabelecer um meio de comunicação entre o usuário e o assistente, armazenar dados dos usuários, processar linguagem natural a fim de extrair as intenções a respeito dos serviços propostos e recuperar informações do portal acadêmico Sagres e do site da UESC.

O módulo de interface implementado se mostrou capaz de estabelecer uma comunicação entre o usuário e o módulo de assistente pessoal. Já o armazenamento de dados dos usuários foi tratado no módulo de banco de dados, o módulo de entendimento de linguagem se mostrou capaz de identificar as intenções relacionadas aos serviços dos extratores de dados e algumas intenções triviais. O BotSagres se mostrou capaz de recuperar informações no portal acadêmico sagres, o BotUESC também se mostrou funcional para buscar dados no site da UESC. Por fim, o módulo de assistente pessoal foi responsável por orquestrar a comunicação entre os outros módulos, criando assim um sistema distribuído de assistência pessoal capaz de receber mensagens de texto, interpreta-las e realizar as ações necessárias para responder ao usuário final, alcançando assim todos os objetivos propostos para este trabalho.

### 5.1 Trabalhos Futuros

Por fim, pode-se identificar possíveis melhorias a serem implementadas futuramente, como melhoramento do entendimento de linguagem natural, implementação de novas conversações, implementar conversação a partir de mensagens de áudio visando melhorar a acessibilidade do SAP, criação de novos serviços para o SAP, assim como a criação de novos indexadores de *bots* com temas diferentes, calcular eficiência em tempo de resposta do assistente e a implementação de testes automatizados que sejam realizados periodicamente para a identificação de erros nos serviços visando maior rapidez na correção de novos problemas.

## REFERÊNCIAS

7GRAUS. **Significado de Kanban**: o que é, conceito e definição. Disponível em: <<https://www.significados.com.br/kanban/>>. Acesso em: 8 nov. 2018.

BATSCHINSKI, George. **What is a Backend as a Service?** Disponível em: <<https://blog.back4app.com/2016/01/11/what-is-a-backend-as-a-service/#more-705>>. Acesso em: 8 nov. 2018.

BECK, K et al. **Agile Manifesto**. The Agile Manifesto, 2001.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems: Concepts and Design**, 5 ed. [S.l: s.n.], 2012.

DIGITALOCEAN. **DigitalOcean Droplets**. Disponível em: <<https://www.digitalocean.com/products/droplets/>>. Acesso em: 8 nov. 2018.

DIGITALOCEAN. **Droplet Overview**: DigitalOcean Product Documentation. Disponível em: <<https://www.digitalocean.com/docs/droplets/overview/>>. Acesso em: 20 nov. 2018.

ENEMBRECK, F.; BARTHES, J.-P. **Personal assistant to improve CSCW**. The 7th International Conference on Computer Supported Cooperative Work in Design, p. 329–335, 2002.

FERRUCCI, D. A. **Introduction to “This is Watson”**. IBM Journal of Research and Development, 2012.

FLANAGAN, David. **JavaScript: The Definitive Guide**, 6 ed. [S.l: s.n.], 2011.

GONZALEZ, Marco; LIMA, Vera L. S. De. **Recuperação de Informação e Processamento da Linguagem Natural**. XXIII Congresso da Sociedade Brasileira de Computação. Anais da III Jornada de Mini-Cursos de Inteligência Artificial. Campinas:[sn], v. 3, p. 347–395, 2003. Disponível em: <[http://www.erfelipe.com.br/artigos/RI\\_Processamento\\_de\\_linguagem\\_natural.pdf](http://www.erfelipe.com.br/artigos/RI_Processamento_de_linguagem_natural.pdf)>. Acesso em: 5 de dez. 2018.

GOOGLE. **DialogFlow - Docs**. Disponível em: <<https://dialogflow.com/docs>>. Acesso em: 5 dez. 2018.

GRIFFIN, Keith; FLANAGAN, Colin. **Defining a call control interface for browser-based integrations using representational state transfer**. Computer Communications, 2011.

HAHN, Rodrigo Machado; BARBOSA, Jorge Luis Victória. **Uma Arquitetura de Assistente Pessoal Orientada a Ambientes de Aprendizagem Ubíqua**. Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE), 2008.

HEROKU. **The Heroku Platform**. Disponível em: <<https://www.heroku.com/platform>>. Acesso em: 8 nov. 2018.

HOLMES, Antawan; KELLOGG, Marc. **Automating functional tests using selenium**. [S.l: s.n.], 2006.

IBM. **IBM Cloud**. Disponível em: <<https://console.bluemix.net/docs/overview/ibm-cloud.html#overview>>. Acesso em: 8 nov. 2018.

IBM. **IBM Knowledge Center - What is distributed computing**. Disponível em: <[https://www.ibm.com/support/knowledgecenter/en/SSAL2T\\_8.2.0/com.ibm.cics.tx.doc/concepts/c\\_wht\\_is\\_distd\\_comptg.html](https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html)>. Acesso em: 19 nov. 2018.

LANE, Kin. **Overview Of The Backend as a Service (BaaS) Space**. API Evangelist, v. 2013, n. May, 2013. Disponível em: <<https://s3.amazonaws.com/kinlane-productions/whitepapers/API+Evangelist+-+Overview+of+the+Backend+as+a+Service+Space.pdf>>. Acesso em: 5 de dez. 2018.

MARK, Massé. **REST API Design Rulebook**. [S.l: s.n.], 2013.

MALIK, Sanjay Kumar; RIZVI, Sam. **Information extraction using web usage mining, web scrapping and semantic annotation**. 2011, [S.l: s.n.], 2011.

MESSERSCHMITT, David G. **Interface Definition Language**. University of California, Oakland CA, 1999.

MILLER, M. **IBM Cloud Docs Conversation**. Disponível em: <<https://console.bluemix.net/docs/services/conversation/index.html#about>>. Acesso em: 8 nov. 2018.

MITCHELL, Tom M. et al. **Experience with a learning personal assistant**. Communications of the ACM, 1994.

MOSTAÇO, Gustavo Marques et al. **AgronomoBot: a smart answering Chatbot applied to agricultural sensor networks**. 2018, [S.l: s.n.], 2018.

PERNA, CL; DELGADO, HK; FINATTO, MJ. **Linguagens especializadas em corpora: modos de dizer e interfaces de pesquisa**. [S.l: s.n.], 2010. Disponível em: <<http://books.google.com/books?hl=en&lr=&id=2gV5PGfSk0QC&oi=fnd&pg=PA71&dq=LINGUAGENS+ESPECIALIZADAS+EM+CORPORA+MODOS+DE+DIZER+E+INTERFACES+DE+PESQUISA&ots=iC8iVie-J2&sig=jVEwkEEndakr6Z4vuPdTLzJN7qQ>>. Acesso em: 5 de dez. 2018.

PITON, Otávio Henrique Gotardo. **AUTOMAÇÃO RESIDENCIAL UTILIZANDO A PLATAFORMA EM NUVEM IBM BLUEMIX**. 2017.

REATEGUI, Eliseo; RIBEIRO, Alexandre; BOFF, Elisa. **Um Sistema Multiagente Para Controle De Um Assistente Pessoal Aplicado a Um Ambiente Virtual De Aprendizagem**. Renote, 2008.

SOARES, MICHEL DOS SANTOS. **Comparação entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. Idesia (Arica), v. 3, p. 6, 2004. Disponível em: <<http://www.dcc.ufra.br/infocomp/index.php/INFOCOMP/article/view/68>>. Acesso em: 5 de dez. 2018.

SOEDIONO, Budi. **Web Scraping with Python**. [S.l: s.n.], 1989.

SUTIKNO, Tole et al. **WhatsApp, Viber and Telegram which is Best for Instant Messaging?** International Journal of Electrical and Computer Engineering (IJECE), v. 6, n. 3, p. 909, 1 jun. 2016.

TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed Systems: Principles and Paradigms**. [S.l: s.n.], 2013.

TELENYK, Sergii et al. **MODELS AND METHODS OF RESOURCE MANAGEMENT FOR VPS HOSTING**. 2013.

USACHEV, Denis et al. **Open source platform Digital Personal Assistant**. 2018. Disponível em: <<http://arxiv.org/abs/1801.03650>>. Acesso em: 19 nov. 18.

VARGIU, Eloisa; URRU, Mirko. **Exploiting web scraping in a collaborative filtering- based approach to web advertising**. Artificial Intelligence Research, v. 2, n. 1, 20 nov. 2012.

WOOLDRIDGE, Michael. **Introduction to Multiagent Systems**. Information Retrieval, 2002.

YAN, Mengting et al. **Building a Chatbot with Serverless Computing**. 2016, [S.l: s.n.], 2016.

ZAMBIASI, Saulo Popov; RABELO, Ricardo J. **Uma Arquitetura Aberta e Orientada a Serviços para Softwares Assistentes Pessoais**. Revista de Informática Teórica e Aplicada, 2012.



## APÊNDICE A – CÓDIGO FONTE DO BOT DO TELEGRAM

O *bot* está em funcionamento e pode ser acessado a partir da URL: <https://t.me/lana\_pa\_bot>.

```

1 // Arquivo index.js
2 const lana = require('./lana');
3 require('./web')(lana);
4 // Fim do arquivo index.js

1 // Arquivo web.js
2 const express = require('express');
3 const packageInfo = require('./package.json');
4 const bodyParser = require('body-parser');
5
6 const app = express();
7 app.use(bodyParser.json());
8
9 app.get('/', (req, res) => {
10   res.json({ version: packageInfo.version });
11 });
12
13 const server = app.listen(process.env.PORT, () => {
14   const host = server.address().address;
15   const port = server.address().port;
16
17   console.log("Web server started at http://%s:%s", host, port);
18 });
19
20 module.exports = (lana) => {
21   app.post('/' + lana.token, (req, res) => {
22     lana.processUpdate(req.body);
23     res.sendStatus(200);
24   }),
25   app.post('/' + lana.token + '/sendMessage', (req, res) => {
26     lana.sendMessageEndpoint(req.body)
27       .then(resp => {
28         res.sendStatus(200);
29       })
30       .catch(err => {
31         res.sendStatus(400);
32       });
33   })
34 };
35 // Fim do arquivo web.js

1 // Arquivo config/keys.js
2 module.exports = {
3   tokenTelegramBot : process.env.tokenTelegramBot,
4   urlTelegramBot : process.env.urlTelegramBot,

```



```

5     urlMessageEndpoint : process.env.urlTelegramMessageEndpoint,
6     lana_api_key : process.env.lana_API_MASTER_KEY
7   }
8   // Fim do arquivo config/keys.js

1   // Arquivo lana.js
2   const keys = require("./config/keys");
3   const token = keys.tokenTelegramBot;
4   const url = keys.urlTelegramBot;
5   const urlMessageEndpoint = keys.urlMessageEndpoint;
6   const lana_api_key = keys.lana_api_key;
7   const request = require("request");
8
9   const Bot = require("tgfancy");
10  let lana;
11
12  if (process.env.NODE_ENV === "production") {
13    lana = new Bot(token);
14    lana.setWebHook(url + "/" + lana.token);
15  } else {
16    lana = new Bot(token, { polling: true });
17  }
18
19  console.log("lana telegram bot server started...");
20
21  lana.on("polling_error", error => {
22    console.error(error);
23  });
24
25  //Any message
26  lana.on("message", msg => {
27    // lana.sendMessage(msg.chat.id, 'Recebi a mensagem "' + msg.text + '" de: ' +
msg.from.first_name + ' ' + msg.from.last_name);
28    let msgJson = {
29      interface: "telegram",
30      type: "incoming",
31      message: msg.text,
32      from: {
33        name: msg.from.first_name + " " + msg.from.last_name,
34        username: msg.from.username,
35        id: msg.from.id,
36        chatId: msg.chat.id
37      },
38      date: new Date(msg.date * 1000)
39    };
40    console.log(JSON.stringify(msgJson));
41
42    fowardMessageToLana(msg);
43  });
44
45  const fowardMessageToLana = msg => {
46    let options = {
47      method: "POST",
48      uri: "https://lana-api.herokuapp.com/api/message",
49      headers: {
50        "x-api-key": lana_api_key
51      },
52      body: {
53        interface: "telegram",

```

```

54     message: msg.text,
55     user: {
56       name: `${msg.from.first_name} ${msg.from.last_name}`,
57       username: msg.from.username,
58       id: msg.from.id,
59       chatId: msg.chat.id
60     },
61     messageEndpoint: urlMessageEndpoint
62   },
63   json: true // Automatically stringifies the body to JSON
64 };
65
66 request(options, (err, resp, body) => {
67   if (!resp || resp.statusCode !== 200 || err) {
68     let msgJson = {
69       interface: "telegram",
70       type: "outgoing",
71       message: msg.text,
72       errMessage: err,
73       to: {
74         chatId: msg.chat.id
75       },
76       date: new Date()
77     };
78     console.log(JSON.stringify(msgJson));
79     lana.sendMessage(msg.chat.id, "Ops, algo deu errado, tente novamente");
80   } else {
81     let msgJson = {
82       interface: "telegram",
83       type: "outgoing",
84       message: body.message,
85       to: {
86         chatId: body.chatId
87       },
88       date: new Date()
89     };
90     console.log(JSON.stringify(msgJson));
91     let markdown = {};
92     if (body.markdown) {
93       markdown = { parse_mode: "Markdown" };
94     }
95     if (body.message !== undefined) {
96       lana.sendMessage(body.chatId, body.message, markdown);
97     } else {
98       lana.sendMessage(body.chatId, "Ops, algo deu errado, tente novamente");
99     }
100   }
101 });
102 };
103
104 lana.sendMessageEndpoint = messageBody => {
105   return new Promise((resolve, reject) => {
106     if (messageBody.chatId !== undefined && messageBody.message !== undefined) {
107       let msgJson = {
108         interface: "telegram",
109         type: "outgoing",
110         message: messageBody.message,
111         to: {
112           chatId: messageBody.chatId
113         },

```

```
114     date: new Date()
115   };
116   console.log(JSON.stringify(msgJson));
117   let markdown = {};
118   if (messageBody.markdown) {
119     markdown = { parse_mode: "Markdown" };
120   }
121   if (messageBody.type == "image") {
122     lana
123       .sendPhoto(messageBody.chatId, messageBody.message, markdown)
124       .then(res => {
125         resolve();
126       })
127       .catch(err => {
128         reject(err);
129       });
130   } else {
131     lana
132       .sendMessage(messageBody.chatId, messageBody.message, markdown)
133       .then(res => {
134         resolve();
135       })
136       .catch(err => {
137         reject(err);
138       });
139   }
140   } else {
141     console.log("Unable to Send Message Body in Wrong Format");
142     reject();
143   }
144   });
145 };
146
147 module.exports = lana;
148 // Fim do arquivo lana.js
```

## APÊNDICE B – CÓDIGO FONTE DO BOT DO MESSENGER

```

1 // Arquivo config/keys.js
2 module.exports = {
3   PAGE_ACESS_TOKEN: process.env.TOKEN_LANA_MESSENGER_BOT,
4   VERIFY_TOKEN: process.env.TOKEN_LANA_MESSENGER_BOT_VERIFY,
5   APP_SECRET: process.env.TOKEN_LANA_MESSENGER_BOT_APP_SECRET,
6   LANA_API_KEY : process.env.lana_API_MASTER_KEY,
7   URL_MESSAGE_ENDPOINT: process.env.URL_MESSENGER_ENDPOINT
8 };
9 // Fim do arquivo config/keys.js

```

```

1 // Arquivo index.js
2 const BootBot = require('bootbot');
3 const request = require('request');
4 const keys = require('./config/keys');
5 const bodyParser = require('body-parser');
6
7 const bot = new BootBot({
8   accessToken: keys.PAGE_ACESS_TOKEN,
9   verifyToken: keys.VERIFY_TOKEN,
10  appSecret: keys.APP_SECRET
11 });
12 bot.app.use(bodyParser.json());
13
14 bot.on('message', (payload, chat) => {
15   const text = payload.message.text;
16   chat.getUserProfile()
17     .then(user => {
18     const msgJson = {
19       interface : 'messenger',
20       type : 'incoming',
21       message : text,
22       from : {
23         name : user.first_name + ' ' + user.last_name,
24         id : user.id,
25         chatId : user.id
26       },
27       date : new Date()
28     };
29     console.log(JSON.stringify(msgJson));
30     chat.sendAction('mark_seen');
31     fowardMessageToLana(text, chat, user);
32   });
33 });
34
35 const fowardMessageToLana = (text, chat, user) => {
36   const options = {
37     method : 'POST',
38     uri : 'https://lana-api.herokuapp.com/api/message',
39     headers : {
40       'x-api-key': keys.LANA_API_KEY
41     },
42     body : {
43       interface : 'messenger',

```

```

44     message : text,
45     user : {
46       name : `${user.first_name} ${user.last_name}`,
47       id : user.id,
48       chatId : user.id
49     },
50     messageEndpoint : keys.URL_MESSAGE_ENDPOINT,
51   },
52   json : true // Automatically stringifies the body to JSON
53 };
54
55 request(options, (err, resp, body) => {
56   if (!resp || resp.statusCode !== 200 || err) {
57     const msgJson = {
58       interface : 'messenger',
59       type : 'outgoing',
60       message : text,
61       errorMessage : err,
62       to : {
63         chatId : user.id
64       },
65       date : new Date()
66     };
67     console.log(JSON.stringify(msgJson));
68     chat.say('Ops, algo deu errado, tente novamente', { typing : true });
69   }
70   else {
71     let msgJson = {
72       interface : 'messenger',
73       type : 'outgoing',
74       message : body.message,
75       to : {
76         chatId : body.chatId
77       },
78       date : new Date()
79     };
80     console.log(JSON.stringify(msgJson));
81     let markdown = {};
82     if (body.markdown) {
83       markdown = { parse_mode: 'Markdown' };
84     }
85     chat.say(body.message, { typing : true });
86   }
87 });
88 };
89
90 // LANA'S ENDPOINT
91 bot.app.post('/' + keys.APP_SECRET + '/sendMessage', (req, res) => {
92   sendMessageEndpoint(req.body)
93     .then(resp => {
94       res.sendStatus(200);
95     })
96     .catch(err => {
97       res.sendStatus(400);
98     });
99 });
100
101 const sendMessageEndpoint = (messageBody) => {
102   return new Promise((resolve, reject) => {
103     if (messageBody.chatId && messageBody.message) {

```

```

104     let msgJson = {
105         interface : 'messenger',
106         type : 'outgoing',
107         message : messageBody.message,
108         to : {
109             chatId : messageBody.chatId
110         },
111         date : new Date()
112     };
113     console.log(JSON.stringify(msgJson));
114     let markdown = {};
115     if(messageBody.markdown) {
116         markdown = {parse_mode: 'Markdown'};
117     }
118     if(messageBody.type == 'image') {
119         bot.say(messageBody.chatId, { attachment: 'image', url: messageBody.message })
120             .then(res => {
121                 resolve();
122             })
123             .catch(err => {
124                 reject(err);
125             });
126     }
127     else {
128         bot.say(messageBody.chatId, messageBody.message)
129             .then(res => {
130                 resolve();
131             })
132             .catch(err => {
133                 reject(err);
134             });
135     }
136 }
137 else {
138     console.log("Unable to Send Message Body in Wrong Format");
139     reject();
140 }
141 });
142 };
143
144 const port = process.env.PORT || 3000;
145 bot.app.get('/', (req, res) => res.send("Hello World!"));
146 bot.start(port);
147 // Fim do arquivo index.js

```

## APÊNDICE C – CÓDIGO FONTE DAS CLOUD FUNCTIONS DO BACK4APP

```

1 // Arquivo main.js
2 const cloudFunctions = require('./cloud-functions');
3
4 Parse.Cloud.define('echo', cloudFunctions.echo(Parse));
5
6 Parse.Cloud.define('getUserByField', cloudFunctions.getUserByField(Parse));
7
8 Parse.Cloud.define('setUserField', cloudFunctions.setUserField(Parse));
9
10 Parse.Cloud.define('unsetUserField', cloudFunctions.unsetUserField(Parse));
11
12 Parse.Cloud.define('registerUser', cloudFunctions.registerUser(Parse));
13
14 Parse.Cloud.define('loginUser', cloudFunctions.loginUser(Parse));
15
16 Parse.Cloud.define('setContext', cloudFunctions.setContext(Parse));
17
18 Parse.Cloud.define('getContextByInterfaceld', cloudFunctions.getContextByInterfaceld(Parse));
19
20 Parse.Cloud.job('removeStoredPasswords', cloudFunctions.removeStoredPasswords(Parse));
21 // Fim do arquivo main.js

```

```

1 // Arquivo cloud-functions.js
2 const destroyAllSessions = (Parse) => {
3   let query = new Parse.Query(Parse.Session);
4
5   query.find({
6     success: (results) => {
7       for (let session of results) {
8         session.destroy({useMasterKey: true});
9       }
10      return true;
11    },
12    error: (error) => {
13      return false;
14    },
15    useMasterKey: true
16  });
17 };
18
19 module.exports.echo = (Parse) => {
20   return (request, response) => {
21     response.success({ code: 200, message: request.params.string });
22   };
23 };
24
25 module.exports.getUserByField = (Parse) => {
26   return (request, response) => {
27     let userQuery = new Parse.Query(Parse.User);
28     userQuery.equalTo(request.params.field, request.params.value);
29     userQuery.first({
30       success: (user) => {
31         if(user != undefined) {

```

```

32     response.success({ code: 200, message: user });
33   }
34   else
35     response.error(404, "Usuario não encontrado");
36   },
37   error: (err) => {
38     response.error(err);
39   }
40   });
41 };
42 };
43
44 module.exports.setUserField = (Parse) => {
45   return (request, response) => {
46     let userQuery = new Parse.Query(Parse.User);
47     userQuery.equalTo("objectId", request.params.userId);
48     userQuery.first()
49       .then(userToUpdate => {
50         if(userToUpdate !== undefined) {
51           userToUpdate.set(request.params.field, request.params.value);
52           userToUpdate.save(null, {useMasterKey:true})
53             .then(userToUpdate => {
54               response.success({ code: 200, message: userToUpdate });
55             })
56             .catch(err => {
57               response.error(err);
58             });
59         }
60         else
61           response.error(404, "Usuario não encontrado");
62       })
63       .catch(err => {
64         response.error(err);
65       });
66   };
67 };
68
69 module.exports.unsetUserField = (Parse) => {
70   return (request, response) => {
71     let userQuery = new Parse.Query(Parse.User);
72     userQuery.equalTo("objectId", request.params.userId);
73     userQuery.first()
74       .then(userToUpdate => {
75         if(userToUpdate !== undefined) {
76           userToUpdate.unset(request.params.field);
77           userToUpdate.save(null, {useMasterKey:true})
78             .then(userToUpdate => {
79               response.success({ code: 200, message: userToUpdate });
80             })
81             .catch(err => {
82               response.error(err);
83             });
84         }
85         else
86           response.error(404, "Usuario não encontrado");
87       })
88       .catch(err => {
89         response.error(err);
90       });
91   };

```



```

92   };
93
94   module.exports.registerUser = (Parse) => {
95     return (request, response) => {
96       if(request.params.username == undefined ||
97         request.params.password == undefined ||
98         request.params.interface == undefined ||
99         request.params.interfaceId == undefined ||
100        request.params.name == undefined) {
101          response.error(400, "Usuario precisa de username, password, interface, interfaceId e
name");
102        }
103        let user = new Parse.User();
104        user.set("username", request.params.username);
105        user.set("password", request.params.password);
106        user.set(request.params.interface, request.params.interfaceId);
107        user.set("name", request.params.name);
108
109        user.signUp(null)
110          .then((newUser) => {
111            destroyAllSessions(Parse);
112            response.success({ code: 200, message: newUser });
113          })
114          .catch((err) => response.error(err));
115      };
116    };
117
118    module.exports.loginUser = (Parse) => {
119      return (request, response) => {
120        Parse.User.logIn(request.params.username, request.params.password)
121          .then((user) => {
122            destroyAllSessions(Parse);
123            response.success({ code: 200, message: user.toJSON() });
124          })
125          .catch((err) => response.error(err));
126      };
127    };
128
129    module.exports.setContext = (Parse) => {
130      return (request, response) => {
131        if(request.params.interface == undefined ||
132          request.params.interfaceId == undefined ||
133          request.params.context == undefined) {
134          response.error(400, "Precisa de interface, interfaceId e Context");
135        }
136        let Context = Parse.Object.extend('Context');
137        let contextQuery = new Parse.Query(Context);
138        let myContext = new Context();
139        contextQuery.equalTo(request.params.interface, request.params.interfaceId);
140        contextQuery.first()
141          .then(contextToUpdate => {
142            if(contextToUpdate != undefined)
143              myContext = contextToUpdate;
144            else
145              myContext.set(request.params.interface, request.params.interfaceId);
146
147            myContext.set("context", request.params.context);
148
149            myContext.save(null)
150              .then(newContext => {

```

```

151         response.success({code: 200, message: newContext});
152     })
153     .catch(err => {
154         response.error(err);
155     });
156 })
157 .catch(err => {
158     response.error(err);
159 });
160 };
161 };
162
163 module.exports.getContextByInterfaceId = (Parse) => {
164     return (request, response) => {
165         let Context = Parse.Object.extend('Context');
166         let contextQuery = new Parse.Query(Context);
167         contextQuery.equalTo(request.params.interface, request.params.interfaceId);
168         contextQuery.first()
169             .then(contextToFind => {
170                 if(contextToFind != undefined)
171                     response.success({code: 200, message: contextToFind.toJSON()});
172                 else
173                     response.error(404, "Context não encontrado");
174             })
175             .catch(err => {
176                 response.error(err);
177             });
178     };
179 };
180
181 module.exports.removeStoredPasswords = (Parse) => {
182     return (request, status) => {
183         const date = new Date();
184         const timeNow = date.getTime();
185         const intervalOfTime = 24*60*60*1000; // the time set is 24 hours in milliseconds
186         const timeThen = timeNow - intervalOfTime;
187
188         // Limit date
189         const queryDate = new Date();
190         queryDate.setTime(timeThen);
191
192         // Will query all users then check for '_password' on fields and unset all
193         const query = new Parse.Query(Parse.User);
194         query.find()
195             .then(result => {
196                 result.forEach(user => {
197                     Object.keys(user.attributes).forEach(attr => {
198                         if (attr.includes('_password'))
199                             user.unset(attr);
200                     });
201                     user.save(null, {useMasterKey:true});
202                 });
203                 status.success({ code: 200, message: 'Senhas apagadas com sucesso' });
204             })
205             .catch(err => status.error({ code : 500, message: 'Erro ao buscar usuarios' }));
206     };
207 };
208 // Fim do arquivo cloud-functions.js

```