

**UNIVERSIDADE ESTADUAL DE SANTA CRUZ – UESC**

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** IBM Watson e web scraping para extração de informações no âmbito da UESC

ILHÉUS - BAHIA

**2018**

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** IBM Watson e web scraping para extração 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. Leard de Oliveira Fernandes

ILHÉUS - BAHIA

**2018**

**GABRIEL RODRIGUES DOS SANTOS**

**LANA, ASSISTENTE PESSOAL:** IBM Watson e web scraping para extração 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. Leard de Oliveira Fernandes

UESC/DCET

(Orientador)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Dr. Francisco Bruno Souza Oliveira

UESC/DCET

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Prof. Dr. Marcelo Ossamu Honda

UESC/DCET

<Dedicatória>

**AGRADECIMENTOS**

<Epigrafe>

**LANA, ASSISTENTE PESSOAL:** IBM Watson e web scraping para extração 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. Este projeto implementa um *software* de assistência pessoal, nomeada como Lana, assim como o sistema de extração de dados*,* para realização dos serviços disponibilizados. Serviços de busca de informações sobre a instituição disponíveis no *site* da própria universidade e informações retiradas do portal Sagres, do aluno e do professor, foram implementados nos extratores de dados. Para a implementação do *software* de assistência pessoal foi utilizado o serviço Watson Assistant da plataforma IBM Cloud, a solução ***backend***como serviçoBack4App, o *framework* Selenium para a técnica de *web scraping*, os serviços de hospedagem do Heroku e DigitalOcean e as linguagens de programação JavaScript e Python.

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

**LANA, PERSONAL ASSISTANT:** IBM Watson and web scraping to extract information 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. This project implements one personal assistance software, named as Lana, as well as the data extraction system, to perform the provided services. Information search services about UESC, available on the university’s website, and informations taken from Sagres, for students and teachers, were implemented in the data extraction system. For the implementation of the personal assistance software, were used the Watson Assistant service from IBM Cloud’s platform, the backend as a service solution Back4App, the Selenium framework for web scraping technique, Heroku and DigitalOcean hosting services and the programming languages JavaScript and Python.

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

**LISTA DE FIGURAS**

Figura 1 - Painel do serviço Watson Assistant para a criação de Intents. 18

Figura 2 - Painel do serviço Watson Assistant para a criação de Entities. 19

Figura 3 - Painel do serviço Watson Assistant para a criação de Dialogs. 20

Figura 4 - Criação de um novo bot a partir do Telegram em uma conversa com o BotFather. 22

Figura 5 - Painel de controle de uma aplicação no Back4App. 24

Figura 6 - Painel de controle de um Droplet na plataforma do DigitalOcean. 26

Figura 7 - Painel de controle de um Dyno na plataforma do Heroku. 27

Figura 8 - Topologia de comunicação entre módulos. 30

Figura 9 - Painel do serviço Watson Assistant com algumas Intents criadas para o workspace Lana. 37

Figura 10 - Painel do serviço Watson Assistant com todas as Entities criadas para o workspace Lana. 38

Figura 11 - Painel do serviço Watson Assistant com parte do fluxo de primeira conversa criada para o workspace Lana. 39

Figura 12 - Conversação mostrando a Lana em funcionamento no aplicativo Telegram. 49

Figura 13 - Lana informando suas principais funcionalidades. 50

Figura 14 - Lana realizando ações para professor no Telegram. 51

Figura 15 - Lana realizando ações para professor no Telegram. 52

Figura 16 - Conversação mostrando a Lana em funcionamento no aplicativo Messenger. 53

**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 Respresentational 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**

1 INTRODUÇÃO 11

2 EMBASAMENTO TEÓRICO 15

2.1 IBM Cloud 15

2.1.1 IBM Watson 15

2.1.1.1 Watson Assistant 16

2.2 Aplicativos de Troca de Mensagens 20

2.2.1 Telegram 21

2.2.2 Messenger 22

2.3 Back4App 23

2.4 VPS Hosting 25

2.4.1 DigitalOcean Droplet 25

2.4.2 Heroku Dyno 26

2.5 Selenium WebDriver 28

2.6 Kanban 29

3 MATERIAIS E MÉTODOS 30

3.1 Interface 31

3.1.1 Telegram 32

3.1.2 Messenger 33

3.2 Banco de Dados 35

3.3 Entendimento de Linguagem Natural 37

3.4 Assistente Pessoal 40

3.5 Indexador de Bots 41

3.6 Bots Scrapers 42

3.6.1 BotSagres 43

3.6.2 BotUESC 44

4 RESULTADOS E DISCUSSÕES 46

5 CONCLUSÃO 54

REFERÊNCIAS 55

APÊNDICE A – Código Fonte do bot do Telegram 58

APÊNDICE B – Código Fonte do bot do Messenger 62

APÊNDICE C – Código Fonte das Cloud Functions do Back4App 65

# 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. Convergindo 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 entediantes. 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 especificas, 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, ReateguiI, Ribeiro e Boff (2008) propõem um sistema multiagente para o controle de um AP, que é explicado pelos autores: “cada agente controla uma funcionalidade especifica, 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 em 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) conceituam um sistema distribuído 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 épopularmente conhecido como *um* *bot*. Tais *bots* percorrem o *site* através do código fonte, normalmente em formato HTML, e também podem executar instruções em JavaScript na página.

Este trabalho apresenta a implementação de um SAP, nomeada como Lana, assim como o sistema de extração de dados*,* para realizar os serviços disponibilizados, e dois *bots,* que utilizam API de *softwares* de troca de mensagens, a fim de estabelecer uma comunicação entre o usuário final e a Lana. Para realizar a troca de informações entre os *bots* e o AP foram implementadas APIs para ambos.

A Lana recebe as mensagens do usuário final, as interpreta e responde com o que for necessário, de acordo com o seu entendimento. A implementação foi realizada visando a criação de um produto genérico, isto é, somente os extratores de dados possuem funções para serviços específicos, a assistente trata todas as mensagens recebidas de maneira única, desta forma, a implementação de novos serviços, dos mais variados contextos, não acarretará na necessidade de modificar o SAP, somente à criação de novos extratores de dados e na integração de contexto, facilitando assim o escalonamento da aplicação na adição de novas funcionalidades. A fim de fechar um escopo de serviços realizados pela assistente, visto que seria possível implementar diversas ações para diferentes finalidades, foram escolhidos serviços dentro do contexto da UESC, serviços de busca de informações sobre a instituição disponíveis no *site* da própria universidade e informações retiradas do portal Sagres, do aluno e do professor, foram implementados nos extratores de dados.

Para a implementação do SAP foi utilizado um serviço de entendimento de linguagem natural, uma solução *backend* como serviço (BaaS), um *framework* para a técnica de *web scraping* e serviços de hospedagem.

Este trabalho de conclusão de curso foi dividido em 4 tópicos gerais, apresentando-se no primeiro um aprofundamento teórico a respeito das plataformas e serviços utilizados. 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.

# EMBASAMENTO TEÓRICO

Neste tópico será apresentada a fundamentação teórica necessária para o entendimento e manuseio das principais ferramentas e serviços utilizados para desenvolver este projeto.

## IBM Cloud

O IBM Cloud, anteriormente conhecido como IBM Bluemix, é uma plataforma de computação em nuvem oferecida pela IBM, ela combina o conceito de plataforma comoserviço (PaaS) e infraestrutura como serviço(IaaS). Além disso, disponibiliza serviços em nuvem que podem ser integrados ao PaaS e IaaS para a construção de aplicações (IBM, 2018).

A plataforma fornece um painel com acesso aos serviços do IBM Cloud disponibilizados pela IBM e terceiros. Estes incluem serviços do IBM Watson, internet das coisas, análise de dados, móveis e outros mais.

Outras empresas fornecem serviços similares ao IBM Cloud, como por exemplo o Amazon AWS e Microsoft Azure. Entretanto, elas não possuem o serviço de entendimento de linguagem natural, que é o principal serviço buscado nestas plataformas para este projeto. Sendo assim, o IBM Cloud foi escolhido, tanto por disponibilizar gratuitamente o uso da ferramenta quanto por fornecer serviços cognitivos de entendimento de linguagem natural através do IBM Watson.

### IBM Watson

“O IBM Watson é um supercomputador que combina inteligência artificial e software analítico para oferecer serviços diversos” (PITON, 2017). O Watson foi criado pela IBM com o objetivo de auxiliar desenvolvedores e empresas a construírem sistemas cognitivos que possa melhorar processos, interações e ações de maneira mais simples, poupando ao desenvolvedor o processamento de dados para soluções cognitivas.

O Watson foi apresentado mundialmente em 2011 durante o programa de televisão americano de perguntas e respostas, Jeopardy! Em sua participação, o Watson precisava agir como um dos participantes do programa, respondendo às perguntas a partir de pistas totalmente sozinho, sem utilizar busca por internet e sem interação com ninguém para ajuda-lo a entender ou responder as questões. O Watson deveria analisar uma pista, entender o sentido do que está sendo perguntado, determinar a melhor resposta e computar a sua taxa de confiança naquele resultado em uma média de 3 segundos (FERRUCCI, 2012).

Inicialmente o Watson contava com um serviço de perguntas e respostas que foi removido em 2015. Esta função foi distribuída para os serviços de classificação de linguagem natural, conversação, recuperação e classificação e conversão de documentos (PITON, 2017). O Watson está sempre recebendo novas modificações, seja a adição de novos serviços ou a atualização de serviços já conhecidos, como é o caso dos serviços de conversação, recuperação e classificação e conversão de documentos, estes foram substituídos por outros novos serviços que englobam seus objetivos.

Atualmente o Watson conta com serviços de entendimento de linguagem natural, analisador de emoções, tradução de linguagens, reconhecimento visual, aprendizado de máquina, transformação de textos para voz, entre outras várias soluções cognitivas. Todas as documentações dos serviços do Watson estão disponíveis abertamente na plataforma IBM Cloud. A configuração de serviços do Watson precisa ser realizada pela plataforma IBM Cloud, somente após as configurações é possível utilizar as APIs dos serviços.

#### Watson Assistant

Anteriormente conhecido como Watson Conversation, em 2016, foi criado o serviço Watson Assistant. Com este serviço é possível construir soluções que entendam linguagem natural e responda de maneira similar a uma conversa entre humanos (MILLER, 2017).

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 elocuções 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 “vocês têm quais sabores de pizza?”.

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



Fonte: Plataforma do IBM Watson[[1]](#footnote-1).

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, 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.

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



Fonte: Plataforma do IBM Watson[[2]](#footnote-2).

Conforme são inclusos 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 - Painel do serviço Watson Assistant para a criação de Dialogs.



Fonte: Plataforma do IBM Watson[[3]](#footnote-3).

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* identificadasna mensagem e a sua taxa de confiabilidade para cada.

## 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.

### 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 totalmente grátis 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 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 - Criação de um novo bot a partir do Telegram em uma conversa com o BotFather.



Fonte: Conversa com o BotFather a partir do Telegram[[4]](#footnote-4).

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

### 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 inicio 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.

## Back4App

Um 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).

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 - Painel de controle de uma aplicação no Back4App.



Fonte: Plataforma Back4App[[5]](#footnote-5).

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 a data da ultima 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.

## 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).

### 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 configura-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 maquina e podendo configura-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.

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



Fonte: Plataforma do DigitalOcean[[6]](#footnote-6).

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.

### 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 necessário executar nele uma aplicação *web* que receba requisições em uma porta especifica, 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 - Painel de controle de um Dyno na plataforma do Heroku.



Fonte: Plataforma do Heroku[[7]](#footnote-7).

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.

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.

## 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.

## 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 divido 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.

Como o Kanban se trata de uma metodologia ágil, não é preciso criar diagramas UML, esses tipos de diagramas geralmente são criados somente quando necessário e de maneira automática.

# MATERIAIS E 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 linguagem de definição de interface (IDL), um padrão uniforme de comunicação que descreve as interfaces assim como suas assinaturas (MESSERSCHMITT, 1999). Desta maneira, para um módulo se comunicar com outro basta ambos seguirem o padrão de comunicação estabelecido. A topologia de comunicação entre os módulos é representada na Figura 8, nela podemos observar que todas as comunicações entre os módulos são realizadas através da internet utilizando o protocolo HTTPS. A figura 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 - Topologia de comunicação entre módulos.



Fonte: elaborado pelo autor.

Para este projeto não foram gerados outros diagramas além da topologia apresentada na Figura 8, visto que este trabalho foi implementado utilizando a metodologia ágil Kanban, onde não é obrigatório a criação de diagramas.

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.

## 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:

1. nome do agente de interface (ex.: Telegram ou Messenger);
2. nome do usuário;
3. número de identificação do usuário;
4. URL do ponto de acesso;
5. mensagem a ser encaminhada.

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

1. tipo de mensagem (ex.: texto ou imagem);
2. variável booleana informando se a mensagem possui marcação de estilo;
3. mensagem.

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

1. número de identificação de usuário destinatário;
2. tipo de mensagem (ex.: texto ou imagem);
3. variável booleana informando se a mensagem possui marcação de estilo;
4. mensagem.

### 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 “fowardMessageToLana” 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.

### 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 “fowardMessageToLana”, que tem o mesmo objetivo da função “fowardMessageToLana” 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.

## 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 inicio à implementação deste módulo, primeiramente, a partir do painel de controle do Back4App foi criado um novo projeto “lana”. 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.

## 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 9.

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



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 10.

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



Fonte: elaborado pelo autor.

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 11, 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. 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”.

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



Fonte: elaborado pelo autor.

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. 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:

1. nome do serviço;
2. *entities* necessárias para a realização do serviço;
3. 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.

## 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 especificas 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 inicio 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.

## 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:

1. tipo de mensagem (ex.: texto ou imagem);
2. variável booleana informando se a mensagem possui marcação de estilo;
3. mensagem.

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

1. tipo de mensagem (ex.: texto ou imagem);
2. variável booleana informando se a mensagem possui marcação de estilo;
3. mensagem;
4. variável booleana informando que houve um erro ao executar o serviço atribuída com o valor “verdade”;
5. informações enviadas a requisição que ocasionaram este erro.

Desta maneira, a Lana não precisa saber qual *bot* é responsável por um serviço especifico, 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 inicio deste tópico. Por fim, após a implementação do Bothub UESC, este foi hospedado em um Dyno na plataforma Heroku.

## 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 2.5, 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 3.5.

### 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:

1. calcular o coeficiente de rendimento acadêmico;
2. enviar uma imagem com os horários de aula;
3. listar todas as disciplinas cursadas;
4. listar as disciplinas que estão sendo cursadas;
5. listar a quantidade de faltas em todas as disciplinas cursadas;
6. listar a quantidade de faltas em uma disciplina específica;
7. listar as médias de todas as disciplinas cursadas;
8. listar os créditos de uma disciplina específica.

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

1. enviar uma imagem com os horários de aula;
2. listar as turmas do semestre atual;
3. informar a quantidade de alunos matriculados em uma disciplina específica;
4. informar a quantidade de disciplinas que já foram ministradas;
5. 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 especifica 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.

### 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:

1. listar noticias recentes;
2. listar noticias de um dia específico;
3. listar resultados recentes;
4. listar editais recentes;
5. listar editais de um dia específico;
6. listar editais de aquisição de bens e serviços recentes;
7. listar editais de aquisição de bens e serviços de um mês específico;
8. listar os cursos ofertados pela UESC junto ao *site* do colegiado de cada;
9. 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 especifico 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.

# RESULTADOS E DISCUSSÕES

Ao finalizar a implementação de todos os módulos, e estabelecer a comunicação entre eles, a Lana já estava em pleno funcionamento com todos os seus serviços disponíveis para acesso ao público a partir do *bot* do aplicativo de mensagens Telegram. Apesar do agente de interface do Messenger ter sido implementado também, 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 analise 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. Apesar dos problemas enfrentados, um maior estudo do funcionamento do Watson Assistant poderia ajudar a resolver estes mesmo 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 especificas 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.

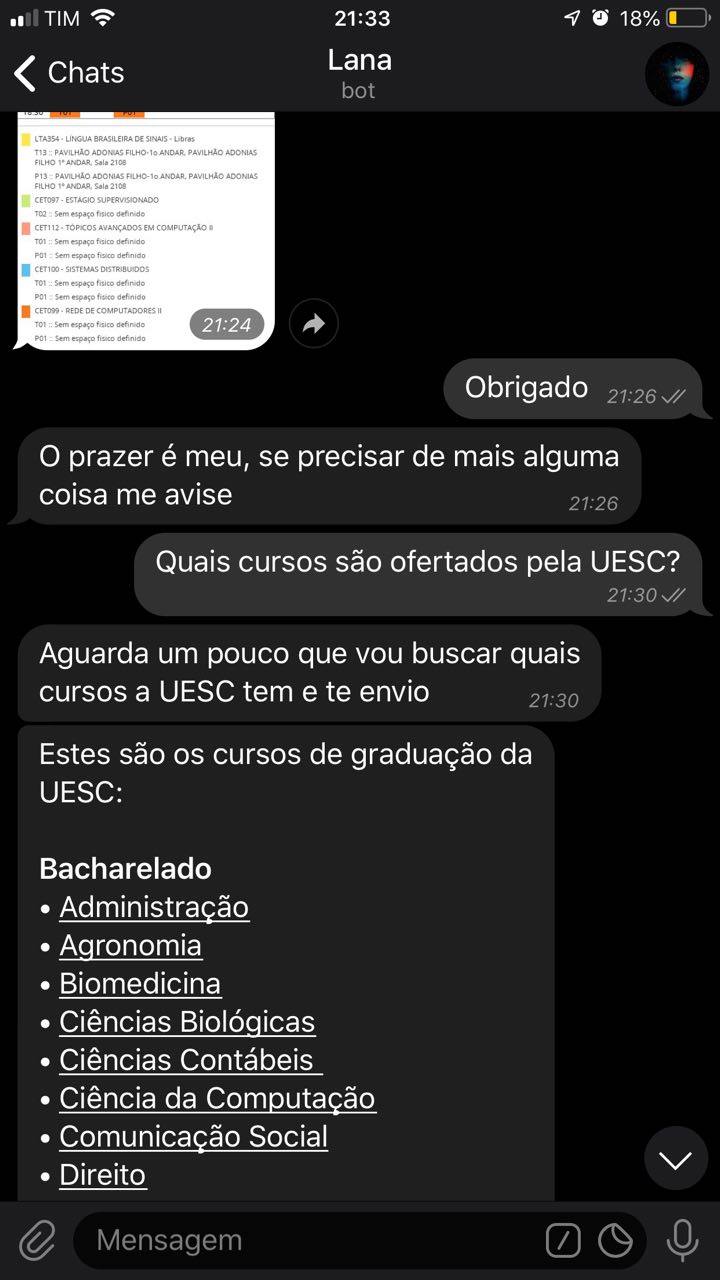
Porém, foi necessária a criação de uma resolução para o problema de gerenciamento de contexto ao tratar de serviços do BotSagres, que foi apresentado neste tópico. Entretanto, até mesmo este problema especifico foi tratado de maneira genérica, podendo ser aplicada a mesma solução, sem a mudança de código 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 especificas 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.

A Lana pode ser vista em funcionamento no Telegram na Figura 12, a figura mostra 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.

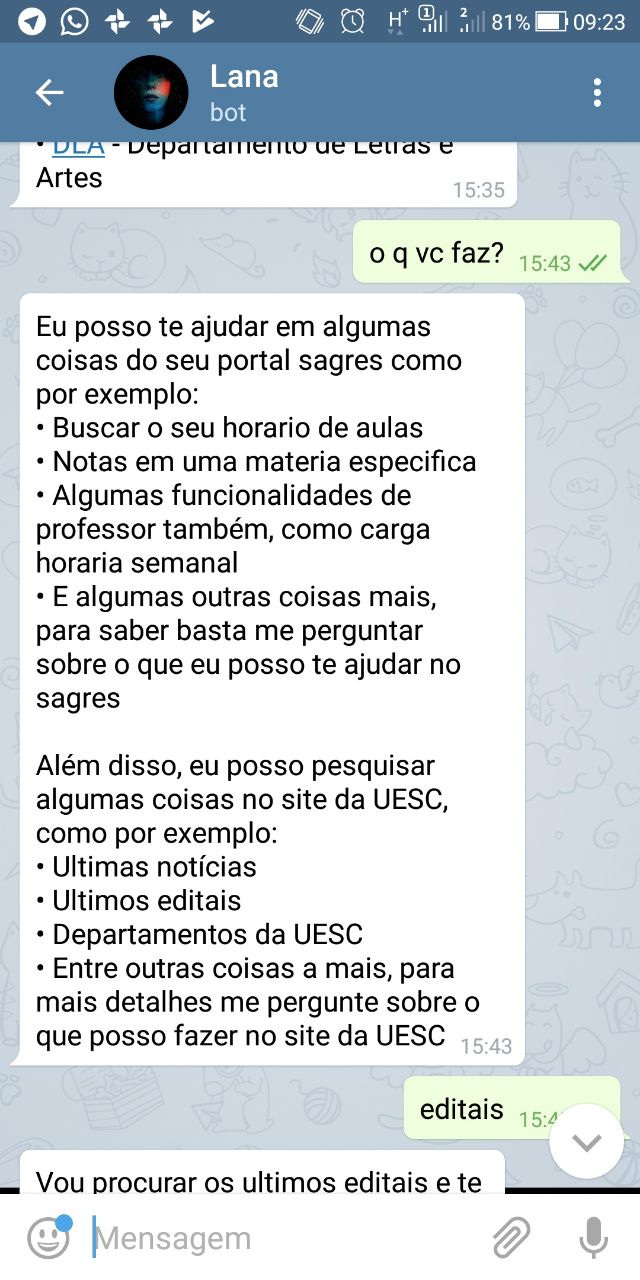
Figura - Conversação mostrando a Lana em funcionamento no aplicativo Telegram.



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.

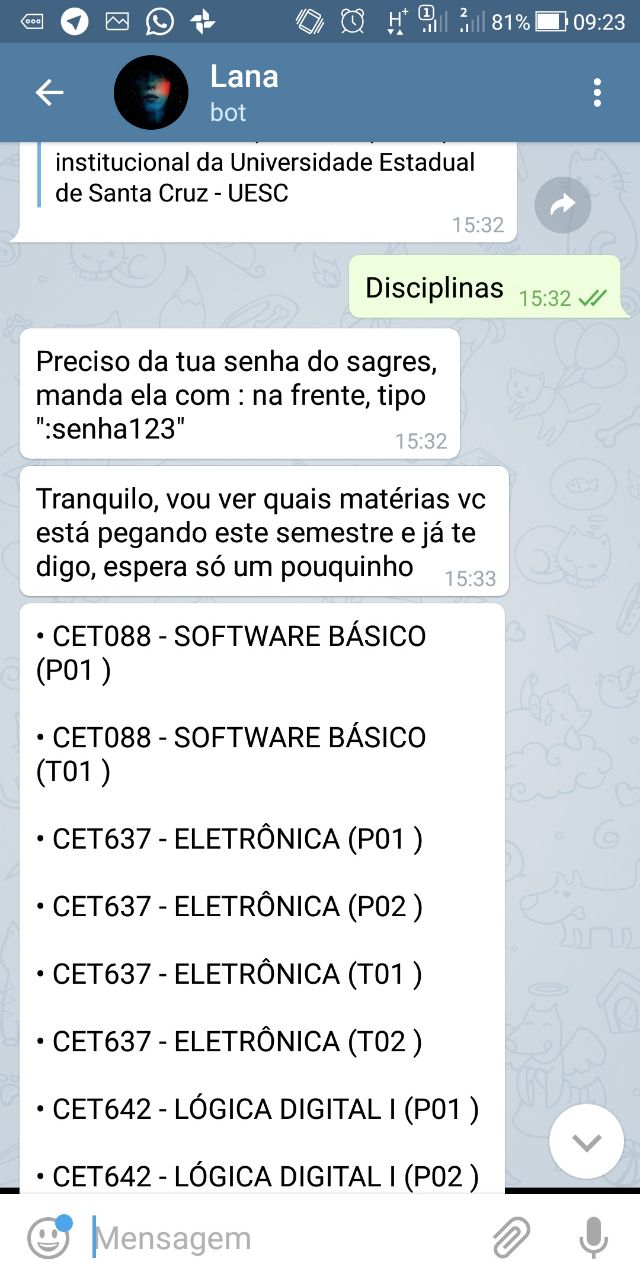
Figura - Lana informando suas principais funcionalidades.



Fonte: elaborado pelo autor.

A Figura 14 apresenta a Lana realizando ações para um professor por meio da aplicação Telegram, na figura é possível observar a requisição da listagem de disciplinas ministradas, para isso 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.

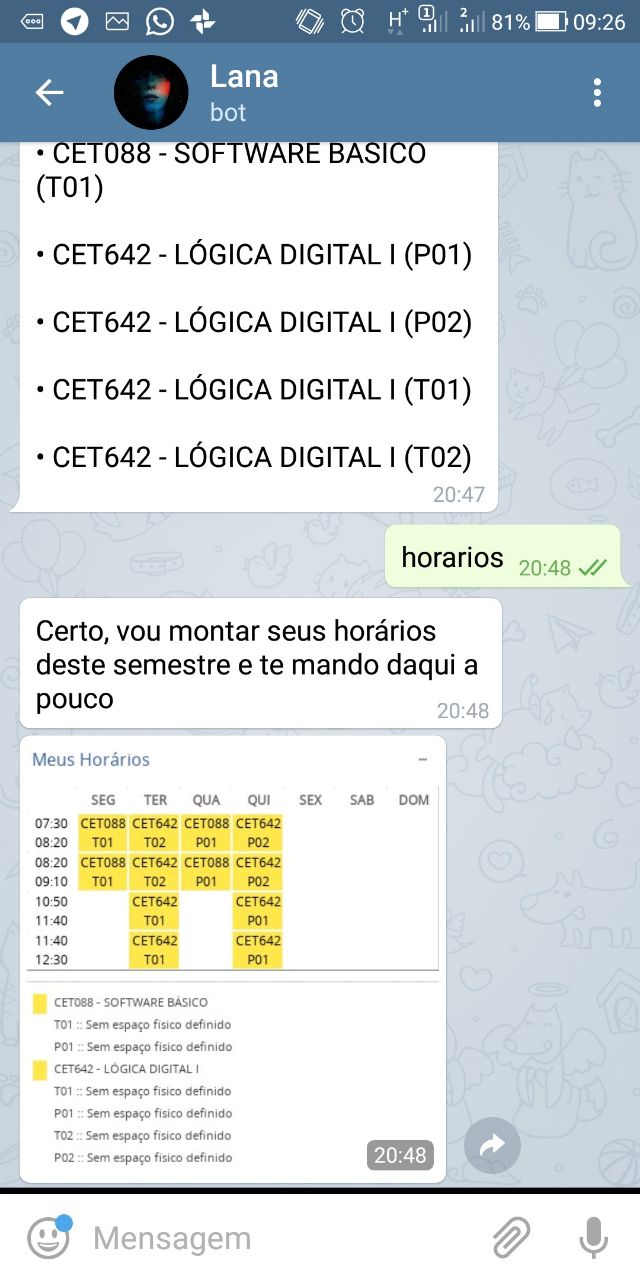
Figura - Lana realizando ações para professor no Telegram.



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.

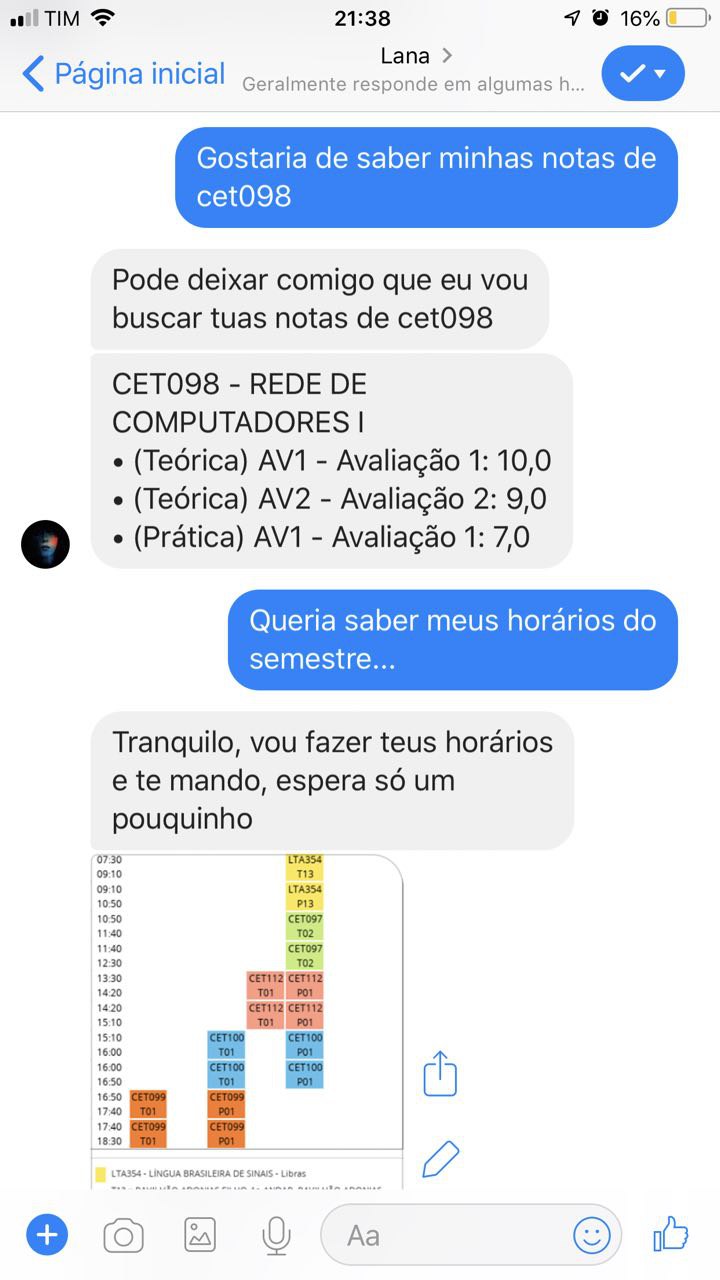
Figura - Lana realizando ações para professor no Telegram.



Fonte: elaborado pelo autor.

Na Figura 16, podemos observar o funcionamento da Lana no aplicativo de troca de mensagens Messenger, nesta figura 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.

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



Fonte: elaborado pelo autor.

Ao fim, o SAP é capaz de receber mensagens textuais através dos aplicativos de mensagens Telegram e Messenger, buscar as intenções e os contextos das mensagens mantendo uma conversação com o usuário, mesmo que somente para temas simples como agradecimentos e elogios ou temas específicos que abordam a realização de seus serviços, armazenar informações de usuários e realizar serviços de busca de informações através de seus *bots* extratores de dados.

# CONCLUSÃO

De uma maneira geral, os serviços, plataformas e bibliotecas utilizadas durante a execução do projeto se mostraram eficientes e essenciais para o funcionamento do SAP. O Back4app ofereceu toda uma infraestrutura de banco de dados de maneira simples e fácil agilizando o processo de desenvolvimento. O Watson Assistant ofereceu suporte ao entendimento de intenções nas mensagens textuais, fazendo com que a Lana pudesse manter uma conversa que tenha sentido para os seus usuários finais. O Selenium foi essencial para a criação dos serviços propostos, visto que todos os serviços foram realizados a partir dos *bots* extratores de dados utilizando a técnica de *web scraping*. Os aplicativos Telegram e Messenger foram importantes para se estabelecer uma comunicação entre os usuários e o SAP sem ser preciso criar uma aplicação móvel para isto. Por fim, os hospedeiros DigitalOcean e Heroku foram essenciais para a disponibilidade e meio de acesso a internet dos módulos implementados, visto que todos ficaram hospedados em ao menos uma das plataformas.

Finalmente, conclui-se que é viável a criação de um SAP de maneira distribuída modularizando os seus principais mecanismos como interface, armazenamento de dados, entendimento de linguagem natural e execução de ações. Ao finalizar a implementação deste projeto foi possível obter resultados satisfatórios de todos os módulos implementados, alcançando assim os objetivos propostos. Por fim, pode-se identificar possíveis melhorias a serem implementadas futuramente, como melhoramento do entendimento de linguagem a partir de um estudo mais aprofundado do serviço Watson Assistant e da implementação de novas conversações, criação de novos serviços para o SAP, assim como a criação de novos indexadores de *bots* com temas diferentes, acrescentando assim, novos contextos para o AP 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.

BAKER, Mark; APON, Amy. **Middleware**. International Journal of High Performance Computing Applications, 2001.

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.

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.

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>. Acesso em: 19 nov. 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.

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.

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.

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
5. // Arquivo web.js
6. const express = require('express');
7. const packageInfo = require('./package.json');
8. const bodyParser = require('body-parser');
10. const app = express();
11. app.use(bodyParser.json());
13. app.get('/', (req, res) => {
14. res.json({ version: packageInfo.version });
15. });
17. const server = app.listen(process.env.PORT, () => {
18. const host = server.address().address;
19. const port = server.address().port;
21. console.log('Web server started at http://%s:%s', host, port);
22. });
24. module.exports = (lana) => {
25. app.post('//' + lana.token, (req, res) => {
26. lana.processUpdate(req.body);
27. res.sendStatus(200);
28. }),
29. app.post('/' + lana.token + '/sendMessage', (req, res) => {
30. lana.sendMessageEndpoint(req.body)
31. .then(resp => {
32. res.sendStatus(200);
33. })
34. .catch(err => {
35. res.sendStatus(400);
36. });
37. })
38. };
39. // Fim do arquivo web.js
40. // Arquivo config/keys.js
41. module.exports = {
42. tokenTelegramBot : process.env.tokenTelegramBot,
43. urlTelegramBot : process.env.urlTelegramBot,
44. urlMessageEndpoint : process.env.urlTelegramMessageEndpoint,
45. lana\_api\_key : process.env.lana\_API\_MASTER\_KEY
46. }
47. // Fim do arquivo config/keys.ks
48. // Arquivo lana.js
49. const keys = require("./config/keys");
50. const token = keys.tokenTelegramBot;
51. const url = keys.urlTelegramBot;
52. const urlMessageEndpoint = keys.urlMessageEndpoint;
53. const lana\_api\_key = keys.lana\_api\_key;
54. const request = require("request");
56. const Bot = require("tgfancy");
57. let lana;
59. if (process.env.NODE\_ENV === "production") {
60. lana = new Bot(token);
61. lana.setWebHook(url + "/" + lana.token);
62. } else {
63. lana = new Bot(token, { polling: true });
64. }
66. console.log("lana telegram bot server started...");
68. lana.on("polling\_error", error => {
69. console.error(error);
70. });
72. //Any message
73. lana.on("message", msg => {
74. // lana.sendMessage(msg.chat.id, 'Recebi a mensagem "' + msg.text + '" de: ' + msg.from.first\_name + ' ' + msg.from.last\_name);
75. let msgJson = {
76. interface: "telegram",
77. type: "incoming",
78. message: msg.text,
79. from: {
80. name: msg.from.first\_name + " " + msg.from.last\_name,
81. username: msg.from.username,
82. id: msg.from.id,
83. chatId: msg.chat.id
84. },
85. date: new Date(msg.date \* 1000)
86. };
87. console.log(JSON.stringify(msgJson));
89. fowardMessageToLana(msg);
90. });
92. const fowardMessageToLana = msg => {
93. let options = {
94. method: "POST",
95. uri: "https://lana-api.herokuapp.com/api/message",
96. headers: {
97. "x-api-key": lana\_api\_key
98. },
99. body: {
100. interface: "telegram",
101. message: msg.text,
102. user: {
103. name: `${msg.from.first\_name} ${msg.from.last\_name}`,
104. username: msg.from.username,
105. id: msg.from.id,
106. chatId: msg.chat.id
107. },
108. messageEndpoint: urlMessageEndpoint
109. },
110. json: true // Automatically stringifies the body to JSON
111. };
113. request(options, (err, resp, body) => {
114. if (!resp || resp.statusCode != 200 || err) {
115. let msgJson = {
116. interface: "telegram",
117. type: "outgoing",
118. message: msg.text,
119. errMessage: err,
120. to: {
121. chatId: msg.chat.id
122. },
123. date: new Date()
124. };
125. console.log(JSON.stringify(msgJson));
126. lana.sendMessage(msg.chat.id, "Ops, algo deu errado, tente novamente");
127. } else {
128. let msgJson = {
129. interface: "telegram",
130. type: "outgoing",
131. message: body.message,
132. to: {
133. chatId: body.chatId
134. },
135. date: new Date()
136. };
137. console.log(JSON.stringify(msgJson));
138. let markdown = {};
139. if (body.markdown) {
140. markdown = { parse\_mode: "Markdown" };
141. }
142. if (body.message != undefined) {
143. lana.sendMessage(body.chatId, body.message, markdown);
144. } else {
145. lana.sendMessage(body.chatId, "Ops, algo deu errado, tente novamente");
146. }
147. }
148. });
149. };
151. lana.sendMessageEndpoint = messageBody => {
152. return new Promise((resolve, reject) => {
153. if (messageBody.chatId != undefined && messageBody.message != undefined) {
154. let msgJson = {
155. interface: "telegram",
156. type: "outgoing",
157. message: messageBody.message,
158. to: {
159. chatId: messageBody.chatId
160. },
161. date: new Date()
162. };
163. console.log(JSON.stringify(msgJson));
164. let markdown = {};
165. if (messageBody.markdown) {
166. markdown = { parse\_mode: "Markdown" };
167. }
168. if (messageBody.type == "image") {
169. lana
170. .sendPhoto(messageBody.chatId, messageBody.message, markdown)
171. .then(res => {
172. resolve();
173. })
174. .catch(err => {
175. reject(err);
176. });
177. } else {
178. lana
179. .sendMessage(messageBody.chatId, messageBody.message, markdown)
180. .then(res => {
181. resolve();
182. })
183. .catch(err => {
184. reject(err);
185. });
186. }
187. } else {
188. console.log("Unable to Send Message Body in Wrong Format");
189. reject();
190. }
191. });
192. };
194. module.exports = lana;
195. // 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
10. // Arquivo index.js
11. const BootBot = require('bootbot');
12. const request = require('request');
13. const keys = require('./config/keys');
14. const bodyParser = require('body-parser');
16. const bot = new BootBot({
17. accessToken: keys.PAGE\_ACESS\_TOKEN,
18. verifyToken: keys.VERIFY\_TOKEN,
19. appSecret: keys.APP\_SECRET
20. });
21. bot.app.use(bodyParser.json());
23. bot.on('message', (payload, chat) => {
24. const text = payload.message.text;
25. chat.getUserProfile()
26. .then(user => {
27. const msgJson = {
28. interface : 'messenger',
29. type : 'incoming',
30. message : text,
31. from : {
32. name : user.first\_name + ' ' + user.last\_name,
33. id : user.id,
34. chatId : user.id
35. },
36. date : new Date()
37. };
38. console.log(JSON.stringify(msgJson));
39. chat.sendAction('mark\_seen');
40. fowardMessageToLana(text, chat, user);
41. });
42. });
44. const fowardMessageToLana = (text, chat, user) => {
45. const options = {
46. method : 'POST',
47. uri : 'https://lana-api.herokuapp.com/api/message',
48. headers : {
49. 'x-api-key': keys.LANA\_API\_KEY
50. },
51. body : {
52. interface : 'messenger',
53. message : text,
54. user : {
55. name : `${user.first\_name} ${user.last\_name}`,
56. id : user.id,
57. chatId : user.id
58. },
59. messageEndpoint : keys.URL\_MESSAGE\_ENDPOINT,
60. },
61. json : true // Automatically stringifies the body to JSON
62. };
64. request(options, (err, resp, body) => {
65. if (!resp || resp.statusCode != 200 || err) {
66. const msgJson = {
67. interface : 'messenger',
68. type : 'outgoing',
69. message : text,
70. errMessage : err,
71. to : {
72. chatId : user.id
73. },
74. date : new Date()
75. };
76. console.log(JSON.stringify(msgJson));
77. chat.say('Ops, algo deu errado, tente novamente', { typing : true });
78. }
79. else {
80. let msgJson = {
81. interface : 'messenger',
82. type : 'outgoing',
83. message : body.message,
84. to : {
85. chatId : body.chatId
86. },
87. date : new Date()
88. };
89. console.log(JSON.stringify(msgJson));
90. let markdown = {};
91. if(body.markdown) {
92. markdown = {parse\_mode: 'Markdown'};
93. }
94. chat.say(body.message, { typing : true });
95. }
96. });
97. };
99. // LANA'S ENDPOINT
100. bot.app.post('/' + keys.APP\_SECRET + '/sendMessage', (req, res) => {
101. sendMessageEndpoint(req.body)
102. .then(resp => {
103. res.sendStatus(200);
104. })
105. .catch(err => {
106. res.sendStatus(400);
107. });
108. });
110. const sendMessageEndpoint = (messageBody) => {
111. return new Promise((resolve, reject) => {
112. if(messageBody.chatId && messageBody.message) {
113. let msgJson = {
114. interface : 'messenger',
115. type : 'outgoing',
116. message : messageBody.message,
117. to : {
118. chatId : messageBody.chatId
119. },
120. date : new Date()
121. };
122. console.log(JSON.stringify(msgJson));
123. let markdown = {};
124. if(messageBody.markdown) {
125. markdown = {parse\_mode: 'Markdown'};
126. }
127. if(messageBody.type == 'image') {
128. bot.say(messageBody.chatId, { attachment: 'image', url: messageBody.message })
129. .then(res => {
130. resolve();
131. })
132. .catch(err => {
133. reject(err);
134. });
135. }
136. else {
137. bot.say(messageBody.chatId, messageBody.message)
138. .then(res => {
139. resolve();
140. })
141. .catch(err => {
142. reject(err);
143. });
144. }
145. }
146. else {
147. console.log("Unable to Send Message Body in Wrong Format");
148. reject();
149. }
150. });
151. };
153. const port = process.env.PORT || 3000;
154. bot.app.get('/', (req, res) => res.send('Hello World!'));
155. bot.start(port);
156. // Fim do arquivo index.js

APÊNDICE C – Código Fonte das Cloud Functions do Back4App

1. // Arquivo main.js
2. const cloudFunctions = require('./cloud-functions');
4. Parse.Cloud.define('echo', cloudFunctions.echo(Parse));
6. Parse.Cloud.define('getUserByField', cloudFunctions.getUserByField(Parse));
8. Parse.Cloud.define('setUserField', cloudFunctions.setUserField(Parse));
10. Parse.Cloud.define('unsetUserField', cloudFunctions.unsetUserField(Parse));
12. Parse.Cloud.define('registerUser', cloudFunctions.registerUser(Parse));
14. Parse.Cloud.define('loginUser', cloudFunctions.loginUser(Parse));
16. Parse.Cloud.define('setContext', cloudFunctions.setContext(Parse));
18. Parse.Cloud.define('getContextByInterfaceId', cloudFunctions.getContextByInterfaceId(Parse));
20. Parse.Cloud.job('removeStoredPasswords', cloudFunctions.removeStoredPasswords(Parse));
21. // Fim do arquivo main.js
22. // Arquivo cloud-functions.js
23. const destroyAllSessions = (Parse) => {
24. let query = new Parse.Query(Parse.Session);
26. query.find({
27. success: (results) => {
28. for (let session of results) {
29. session.destroy({useMasterKey: true});
30. }
31. return true;
32. },
33. error: (error) => {
34. return false;
35. },
36. useMasterKey: true
37. });
38. };
40. module.exports.echo = (Parse) => {
41. return (request, response) => {
42. response.success({ code: 200, message: request.params.string });
43. };
44. };
46. module.exports.getUserByField = (Parse) => {
47. return (request, response) => {
48. let userQuery = new Parse.Query(Parse.User);
49. userQuery.equalTo(request.params.field, request.params.value);
50. userQuery.first({
51. success: (user) => {
52. if(user != undefined) {
53. response.success({ code: 200, message: user });
54. }
55. else
56. response.error(404, "Usuario não encontrado");
57. },
58. error: (err) => {
59. response.error(err);
60. }
61. });
62. };
63. };
65. module.exports.setUserField = (Parse) => {
66. return (request, response) => {
67. let userQuery = new Parse.Query(Parse.User);
68. userQuery.equalTo("objectId", request.params.userId);
69. userQuery.first()
70. .then(userToUpdate => {
71. if(userToUpdate !== undefined) {
72. userToUpdate.set(request.params.field, request.params.value);
73. userToUpdate.save(null, {useMasterKey:true})
74. .then(userToUpdate => {
75. response.success({ code: 200, message: userToUpdate });
76. })
77. .catch(err => {
78. response.error(err);
79. });
80. }
81. else
82. response.error(404, "Usuario não encontrado");
83. })
84. .catch(err => {
85. response.error(err);
86. });
87. };
88. };
90. module.exports.unsetUserField = (Parse) => {
91. return (request, response) => {
92. let userQuery = new Parse.Query(Parse.User);
93. userQuery.equalTo("objectId", request.params.userId);
94. userQuery.first()
95. .then(userToUpdate => {
96. if(userToUpdate !== undefined) {
97. userToUpdate.unset(request.params.field);
98. userToUpdate.save(null, {useMasterKey:true})
99. .then(userToUpdate => {
100. response.success({ code: 200, message: userToUpdate });
101. })
102. .catch(err => {
103. response.error(err);
104. });
105. }
106. else
107. response.error(404, "Usuario não encontrado");
108. })
109. .catch(err => {
110. response.error(err);
111. });
112. };
113. };
115. module.exports.registerUser = (Parse) => {
116. return (request, response) => {
117. if(request.params.username == undefined ||
118. request.params.password == undefined ||
119. request.params.interface == undefined ||
120. request.params.interfaceId == undefined ||
121. request.params.name == undefined) {
122. response.error(400, "Usuario precisa de username, password, interface, interfaceId e name");
123. }
124. let user = new Parse.User();
125. user.set("username", request.params.username);
126. user.set("password", request.params.password);
127. user.set(request.params.interface, request.params.interfaceId);
128. user.set("name", request.params.name);
130. user.signUp(null)
131. .then((newUser) => {
132. destroyAllSessions(Parse);
133. response.success({ code: 200, message: newUser });
134. })
135. .catch((err) => response.error(err));
136. };
137. };
139. module.exports.loginUser = (Parse) => {
140. return (request, response) => {
141. Parse.User.logIn(request.params.username, request.params.password)
142. .then((user) => {
143. destroyAllSessions(Parse);
144. response.success({ code: 200, message: user.toJSON() });
145. })
146. .catch((err) => response.error(err));
147. };
148. };
150. module.exports.setContext = (Parse) => {
151. return (request, response) => {
152. if(request.params.interface == undefined ||
153. request.params.interfaceId == undefined ||
154. request.params.context == undefined) {
155. response.error(400, "Precisa de interface, interfaceId e Context");
156. }
157. let Context = Parse.Object.extend('Context');
158. let contextQuery = new Parse.Query(Context);
159. let myContext = new Context();
160. contextQuery.equalTo(request.params.interface, request.params.interfaceId);
161. contextQuery.first()
162. .then(contextToUpdate => {
163. if(contextToUpdate != undefined)
164. myContext = contextToUpdate;
165. else
166. myContext.set(request.params.interface, request.params.interfaceId);
168. myContext.set("context", request.params.context);
170. myContext.save(null)
171. .then(newContext => {
172. response.success({code: 200, message: newContext});
173. })
174. .catch(err => {
175. response.error(err);
176. });
177. })
178. .catch(err => {
179. response.error(err);
180. });
181. };
182. };
184. module.exports.getContextByInterfaceId = (Parse) => {
185. return (request, response) => {
186. let Context = Parse.Object.extend('Context');
187. let contextQuery = new Parse.Query(Context);
188. contextQuery.equalTo(request.params.interface, request.params.interfaceId);
189. contextQuery.first()
190. .then(contextToFind => {
191. if(contextToFind != undefined)
192. response.success({code: 200, message: contextToFind.toJSON()});
193. else
194. response.error(404, "Context não encontrado");
195. })
196. .catch(err => {
197. response.error(err);
198. });
199. };
200. };
202. module.exports.removeStoredPasswords = (Parse) => {
203. return (request, status) => {
204. const date = new Date();
205. const timeNow = date.getTime();
206. const intervalOfTime = 24\*60\*60\*1000;  // the time set is 24 hours in milliseconds
207. const timeThen = timeNow - intervalOfTime;
209. // Limit date
210. const queryDate = new Date();
211. queryDate.setTime(timeThen);
213. // Will query all users then check for '\_password' on fields and unset all
214. const query = new Parse.Query(Parse.User);
215. query.find()
216. .then(result => {
217. result.forEach(user => {
218. Object.keys(user.attributes).forEach(attr => {
219. if (attr.includes('\_password'))
220. user.unset(attr);
221. });
222. user.save(null, {useMasterKey:true});
223. });
224. status.success({ code: 200, message: 'Senhas apagadas com sucesso' });
225. })
226. .catch(err => status.error({ code : 500, message: 'Erro ao buscar usuarios' }));
227. };
228. };
229. // Fim do arquivo cloud-functions.js

1. Disponível em: <[https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/intents](https://assistant-us-south.watsonplatform.net/us-south/%7Buser-id%7D/workspaces/%7Bworkspace-id%7D/build/intents)>. Acesso em nov. 2018. [↑](#footnote-ref-1)
2. Disponível em: <[https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/entities/user](https://assistant-us-south.watsonplatform.net/us-south/%7Buser-id%7D/workspaces/%7Bworkspace-id%7D/build/entities/user)>. Acesso em nov. 2018. [↑](#footnote-ref-2)
3. Disponível em: <[https://assistant-us-south.watsonplatform.net/us-south/{user-id}/workspaces/{workspace-id}/build/dialog](https://assistant-us-south.watsonplatform.net/us-south/%7Buser-id%7D/workspaces/%7Bworkspace-id%7D/build/dialog)>. Acesso em nov. 2018. [↑](#footnote-ref-3)
4. Disponível em: <<https://t.me/botfather>>. Acesso em nov. 2018. [↑](#footnote-ref-4)
5. Disponível em: <[https://parse-dashboard.back4app.com/apps/{app-id}](https://parse-dashboard.back4app.com/apps/%7Bapp-id%7D)>. Acesso em nov. 2018. [↑](#footnote-ref-5)
6. Disponível em: <[https://cloud.digitalocean.com/droplets/{droplet-id}](https://cloud.digitalocean.com/droplets/%7Bdroplet-id%7D)>. Acesso em nov. 2018. [↑](#footnote-ref-6)
7. Disponível em: <[https://dashboard.heroku.com/apps/{app-name}](https://dashboard.heroku.com/apps/%7Bapp-name%7D)>. Acesso em nov. 2018. [↑](#footnote-ref-7)