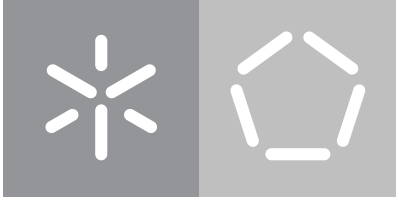
Universidade do Minho
Escola de Engenharia

Rui Miguel da Costa Meira

ChatbotWizard - O Orquestrador de Chatbots

Dezembro, 2022



Universidade do Minho
Escola de Engenharia

Rui Miguel da Costa Meira

ChatbotWizard - O Orquestrador de Chatbots

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação do(a)
José João Antunes Guimarães Dias de Almeida
Alberto Manuel Brandão Simões

Dezembro, 2022

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Braga, 14 de Dezembro de 2022

(Localização) (Data)

Rui Miguel da Costa Meira

(Rui Miguel da Costa Meira)

Agradecimentos

Estes agradecimentos são para as pessoas que me ajudaram e apoiaram ao longo do meu percurso académico, que termina com a conclusão desta dissertação e do respetivo Mestrado em Engenharia Informática.

Primeiramente agradeço aos meus pais pelo esforço que fizeram para me possibilitar estudar e também pelos alicerces proporcionados para a minha vida adulta. Agradeço também ao meu irmão, resto da família, amigos, professores e colegas pelo apoio que me proporcionaram.

Para a conclusão desta tese agradeço ao professor José João Almeida, professor Alberto Simões e ao Pedro Verruma.

Os agradecimentos ao professor José João devem-se por ter-me aceite como seu orientando, todo o conhecimento que me transmitiu e a ajudar para a conclusão desta dissertação.

Além do professor Alberto Simões ter-me ajudado bastante nesta dissertação, sendo co-orientador, o maior agradecimento para ele é por ter sido como um tutor para mim desde a licenciatura. Foi a pessoa que mais me ensinou sobre Software, a pessoa que me permitiu ver a beleza no desenvolvimento de Software.

Por último, agradeço ao Pedro Verruma. O Pedro Verruma teve a disponibilidade e gosto de acompanhar o progresso do ChatbotWizard durante mais de um ano. Colaborou bastante para melhorar o ChatbotWizard e foi incansável em ter reuniões comigo semanalmente por mais de um ano para ajudar.

Resumo

Atualmente os *chatbots* são usados por diversas organizações para automatizar tarefas. Os *chatbots* são desenvolvidos para diversos casos de uso, desde ajudar os utilizadores a navegar nas aplicações até resolver problemas que os utilizadores encontram. No entanto, a criação de um *chatbot* exige recursos monetários e de conhecimento. Assim, a motivação deste projeto passa é permitir a democratização de criação de *chatbots* com o desenvolvimento da ferramenta ChatbotWizard, permitindo que um utilizador possa criar um *chatbot* sem grande conhecimento tecnológico, seja o *chatbot* de elevado grau de complexidade ou não.

O ChatbotWizard usa o Rasa como sistema de diálogo, permitindo integrar vários módulos para a criação de um *chatbot*. Os módulos disponíveis no ChatbotWizard são: módulo para a extração de entidades, módulo realizar pedidos a API, módulo de template para construir texto a partir de JSON e módulo de *Question Answering* (QA) baseado em Transformers (BERT). Estes módulos podem ser conectados para criar o fluxo do *chatbot* desejado. Do ChatbotWizard fazem parte dois componentes: o *backend* e o ChatbotWizard web. O ChatbotWizard web permite a utilizadores criarem os seus *chatbots* integrando e configurando os diversos módulos. O *backend* tem a responsabilidade de receber o fluxo do *chatbot* e criar um *chatbot* baseado no Rasa.

Com o desenvolvimento do ChatbotWizard conseguiu-se uma aplicação que permite o utilizador criar *chatbots* e integrar os mesmos no Telegram. E por fim, foi criado um caso de estudo baseado numa API pública.

Palavras-chave: Chatbots, Rasa, Transformers, BERT

Abstract

In this age, chatbots are used by several organizations to automate tasks. The chatbots are developed for a variety of use cases, from helping users navigate applications to solve issues that the users find. Nonetheless, the development of a chatbot require monetary and knowledge resources. So, the motivation of this project is to allow the democratization of creation of chatbots with the development of our tool named ChatbotWizard, allowing a user to create a chatbot without great technological knowledge, whether the chatbot is of a high degree of complexity or not.

The ChatbotWizard uses Rasa as dialog system, allowing to integrate several modules for a criation of a chatbot. The modules available in ChatbotWizard are: a module to extract entities, a module to do API requests, a template module to create text from JSON and a module for question and answering based in Transformers (BERT). This modules can be connected to create a chatbot flow.. ChatbotWizard have two components: the backend and the ChatbotWizard web. The ChatbotWizard web allows the users to create chatbots, integrating the various modules. The backend is responsible to receive a chatbot flow and create the chatbot using Rasa as template.

With the development of the ChatbotWizard, the application allows a user to create chatbots and integrate them into Telegram. Finally, a case study based on a public API was created to show how to use the ChatbotWizard

Keywords: Chatbots, Rasa, Transformers, BERT

Índice

Lista de Figuras

Capítulo 1

Introdução

Numa grande parte das organizações, a primeira primeira linha de resposta a questões de um utilizador é através de um chatbot. Os chatbots permitem ajudar os utilizadores a navegar numa aplicação, aceder a páginas de documentação, resolver problemas frequentes, responder a questões frequentes (*Frequently Asked Questions*), entre outros. Os chatbots também permitem que uma organização não precise de um funcionário para responder a pedidos frequentes dos utilizadores. A existência de um chatbot numa organização permite responder aos utilizadores de forma mais eficiente do que com um funcionário da organização a realizar as mesmas tarefas. Além de ser mais rápido que um humano a responder a questões dos utilizadores, também consegue ser mais produtivo e eficaz a automatizar certas tarefas. Outra vantagem que uma organização usufrui ao ter um chatbot é o baixo custo de manutenção e a possibilidade de poder adicionar mais funcionalidades ao chatbot. No entanto, nem todas as organizações podem ter um chatbot pois exige recursos financeiros e organizar uma equipa para o desenvolvimento do mesmo, pois não existe uma ferramenta no mercado que permita a criação de um chatbot sem conhecimentos tecnológicos. Por isso essa equipa tem que ter uma especialização tecnológica para a criação do chatbot.

A motivação desta dissertação é a democratização da criação de chatbots através do desenvolvimento da ferramenta ChatbotWizard. O objetivo do ChatbotWizard é permitir orquestrar chatbots sem grande conhecimento tecnológico. Orquestrar no sentido de permitir aos utilizadores criar e gerir chatbots através duma aplicação web, acessível apenas com uma conexão à Internet. A democratização passa por permitir a organizações com poucos recursos a criação de um chatbot sem necessidade de organizar uma equipa e grandes conhecimentos tecnológicos. No entanto, não descurando grandes organizações, o ChatbotWizard deve permitir criar chatbots complexos qualquer que seja o caso de uso que a organização ou utilizadores tenham em mente. O ChatbotWizard deve ser uma ferramenta completa, permitindo através de uma aplicação web criar e gerir chatbots, assim como de os integrar com redes sociais.

1.1 Objetivos

O objetivo central desta dissertação de mestrado é desenvolver um orquestrador de chatbots, que permita criar chatbots para diversos casos de uso. Para este desafio é necessários cumprir os seguintes objetivos:

- Pesquisa de conceitos para o densenvolvimento de um chatbot
- Investigação sobre ferramentas que auxiliem o desenvolvimento de um chatbot
- Desenvolvimento de módulos que permitam o utilizador criar um chatbot
- Desenvolvimento do ChatbotWizard, que permite o utilizador criar chatbots
- Possibilitar o utilizador criar um chatbot no ChatbotWizard e integrar o chatbot numa rede social/aplicação de chat

No fim deste projeto espera-se que o ChatbotWizard permita a criação de chatbots, independentemente do caso de uso do chatbot em questão.

1.2 Estrutura do Documento

No capítulo 2 é apresentado o estado de arte. Começando por uma introdução ao chatbots, os tipos de chatbots e a história dos chatbots. Neste capítulo tambem se descreve sistemas de dialogo, que permitem simplificar o desenvolvimento de um chatbot. De seguida, são apresentados os Transformers e BERT, que possibilitam um chatbot interpretar questões do utilizador. Ainda no tópico de BERT, apresenta-se o projeto HuggingFace, que permite usar modelos BERT já treinados.

No capítulo 3, de objetivos e funcionalidades, descreve-se os objetivos e funcionalidades que o ChatbotWizard deve suportar. Desde as funcionalidade do backend até ao ChatbotWizard web. Neste capítulo também são enumerados os módulos que o ChatbotWizard deve suportar para permitir o utilizador criar um chatbot.

No capítulo 4 são descritos os detalhes da implementação do ChatbotWizard. De entre os detalhes também fazem parte as ferramentas escolhidas para os objetivos em questão.

No capítulo 5 é apresentado um exemplo da utilização do ChatbotWizard para a criação de um chatbot que deve responder a questões de meteorologia. Neste capítulo também são apresentados detalhes de como o ChatbotWizard configura certos ficheiros para a criação do chatbot.

Por fim, no capítulo 6, são apresentadas as conclusões da dissertação e o trabalho futuro a ser realizado.

Capítulo 2

Estado de Arte

Um *chatbot* é um programa de computador desenvolvido para simular a conversa entre utilizadores humanos (Adamopoulou e Moussiades, 2020a). Cada vez se encontram mais aplicações conversacionais em situações do dia à dia, desde comprar bilhetes para um espetáculo a comprar roupa numa aplicação *e-commerce* (Adamopoulou e Moussiades, 2020a). A inteligência artificial (IA) é uma grande impulsionadora de como interagimos nas nossas atividades diárias desenvolvendo aplicações avançadas e dispositivos inteligentes, denominados agentes inteligentes, com a capacidade de desempenhar diversas funcionalidades. Numa definição mais técnica pode-se descrever um *chatbot* como um programa de inteligência artificial que integra um modelo de interação humano-máquina, do inglês *Human–computer interaction* (HCI) (Adamopoulou e Moussiades, 2020a). Um *chatbot* usa processamento de linguagem natural (PLN) e, em alguns casos, análise de sentimento para comunicar em linguagem humana percetível por texto ou voz, seja com humanos ou outros agentes inteligentes.

À parte de imitar a interação humana, os *chatbots* são úteis em diversos campos: educação, negócios, *e-commerce*, saúde e entretimento. A produtividade é o vetor motivacional para o desenvolvimento de *chatbots* (Shawar e Atwell, 2007). Em negócios os *chatbots* tornaram-se bastante comuns porque reduzem o custo dos serviços e podem lidar com diversos clientes simultaneamente (Shawar e Atwell, 2007).

2.1 Tipos de chatbots

Um *chatbot* pode ser classificado usando diferentes parâmetros: o domínio do conhecimento, o tipo de serviço, o objetivo, os métodos de processamento do *input* e respetiva resposta e o método de desenvolvimento do mesmo (Adamopoulou e Moussiades, 2020b).

A classificação de um *chatbot* baseada no domínio do conhecimento considera os dados a que o mesmo tem acesso. O *chatbot* pode ter um domínio aberto, ou seja, pode responder a questões de variados tópicos, enquanto *chatbots* de domínio fechado sãp desenvolvidos para um domínio especifico

(Nimavat e Champaneria, 2017).

A classificação baseada no tipo de serviço considera a relação do *chatbot* com o utilizador, sendo a mesma dependente da tarefa que o *chatbot* realiza. Os *chatbots* interpessoais realizam tarefas bem definidas, sem grande interação com o utilizador, como *chatbots* de reservas e *chatbots* de *Frequently Asked Questions* (FAQ). Os *chatbots* interpessoais não são companhia para os utilizador, o seu único objetivo é obter a informação e responder às perguntas dos utilizadores. Os *chatbots* intrapessoais existem no domínio pessoal do utilizador e funcionam como um parceiro de conversação para o utilizador. Os *chatbots* intrapessoais servem de companhia para o utilizador e percebem o utilizador como um humano (Nimavat e Champaneria, 2017).

A classificação baseada em objetivos considera o objetivo primário que o *chatbot* deve alcançar. Os *chatbots* informativos são desenvolvidos para apresentar informação ao utilizador que foi guardada antecipadamente ou de fontes de dados fixas, tal como em *chatbots* de FAQ. Os *chatbots* conversacionais mantêm uma conversa com o utilizador, replicando o comportamento humano, sendo que o seu objetivo é responder corretamente aos pedidos do utilizador. Os *chatbots* de tarefas executam uma tarefa especifica como reservar um voo. Todos estes *chatbots* são inteligentes no contexto de se questionar o mesmo sobre informação e perceber o *input* do utilizador (Kucherbaev et al., 2018).

Classificar um *chatbot* de acordo com o processamento da frase do utilizador e método de resposta tem em conta o método de processamento dos *inputs* e geração de respostas.

Há três métodos usados para produzir as respostas adequadas: baseado em regras, *retrieval-based* e *generative* (Hien et al., 2018). Os *chatbots* baseados em regras são o tipo de arquitetura em que a maioria dos primeiros *chatbots* foram desenvolvidos. Estes *chatbots* têm o seu sistema de resposta baseado num conjunto fixo de regras predefinidas, fazem reconhecimento do *input* do utilizador. O conhecimento num *chatbot* baseado em regras é codificado à mão e está organizado em padrões conversacionais (Ramesh et al., 2017). É possível ter uma base de dados com regras para as respostas, permitindo agregar mais informação. A limitação deste tipo de modelo é não permitir ser robusto a erros gramaticais do utilizador.

O modelo *retrieval-based* oferece mais flexibilidade pois obtém e analisa fontes de dados disponíveis usando *Application Programming Interface* (API). Este modelo obtém algumas repostas candidatas de um conjunto de dados e algoritmos de classificação para selecionar a resposta ideal.

O modelo *generative* gera respostas de uma forma mais eloquente, baseado nas mensagens anteriores do utilizador. Os *chatbots* que implementam esta arquitetura agem mais como um humano, usando técnicas de *Machine Learning* (ML) e *Deep Learning* (DL). No entanto, desenvolver um *chatbot* com esta arquitetura é mais difícil.

2.2 História dos chatbots

Em 1950, Alan Turing investigou como um programa de computador poderia comunicar com um grupo de pessoas sem perceber que o seu interlocutor era artificial. Esta questão, denominada teste de Turing, é considerado por muitos como a ideia genérica de *chatbots* (TURING, 1950).

A questão de Alan Turing teve a primeira resposta em 1966, com o primeiro *chatbot* denominado ELIZA com o objetivo de simular a atuação de um psicoterapeuta (Weizenbaum, 1966). A habilidade para comunicar era limitada mas foi uma fonte de inspiração para o desenvolvimento de *chatbots* que surgiram posteriormente. O ELIZA usa *pattern matching*[1] e a resposta é selecionada em esquemas com base em *templates* (Brandtzaeg e Følstad, 2017). Um defeito deste *chatbot* prende-se com a limitação do conhecimento do mesmo, sendo que só poderia comunicar sobre tópicos particulares, e também não conseguia manter conversas longas ou aprender e descobrir o contexto da conversa.

Em 1972 foi apresentado o *chatbot* PARRY, também no campo da saúde. O objetivo do mesmo era agir como um paciente com esquizofrenia (Colby et al., 1971). O PARRY é considerado mais avançado que o ELIZA pois é suposto ter personalidade e uma melhor estrutura de controlo. Este *chatbot* define as suas respostas com base num sistema de suposições ativado pela alteração da relevância das frases do utilizador. O PARRY foi usado numa experiência em 1979 com 4 psiquiatras que tentassem julgar se eventualmente se tratava de um programa de computador ou um paciente com esquizofrenia. No entanto, a amostra de 4 psiquiatras é pequena, e o significado dos resultados não são claros pois as pessoas com esquizofrenia têm um grau elevado de incoerência no seu discurso. Em geral, PARRY é considerado um *chatbot* com poucas capacidades relativamente ao entendimento da linguagem e habilidade de expressar emoções.

A IA tem o primeiro impacto no desenvolvimento de *chatbots* em 1988 com o *chatbot* Jabberwacky. O Jabberwacky foi desenvolvido numa linguagem baseada em folhas de cálculo que facilita o desenvolvimento de *chatbots*, e é usado *pattern matching* para responder com base em conversas anteriores. Apesar dos avanços que este *chatbot* trouxe, ele não é capaz de responder muito rapidamente e funcionar nas suas plenas capacidades com um grande número de utilizadores.

Apesar da já existência dos *chatbots* só em 1991 é que o termo *chatterbot* foi mencionado (Mauldin, 1994). O termo *chatterbot* foi o termo original que deu origem ao termo *chatbot*.

Outro passo relevante na história dos *chatbots* foi a criação em 1995 da *Artificial Linguistic Internet Computer Entity* (ALICE), o primeiro *chatbot* inspirado na ELIZA. A ALICE é baseado em *pattern-matching* sem qualquer perceção da conversação no seu todo mas com a capacidade de incluir qualquer tópico. A ALICE foi desenvolvida com uma nova linguagem criada exclusivamente para este propósito, *Artificial Intelligence Markup Language* (AIML), que é a sua principal diferença com o ELIZA. O conhecimento de ALICE baseia-se em cerca de 14000 *templates* e padrões relacionados, um grande número comparando com ELIZA que tem apenas 200 palavras-chave e regras. No entanto, ALICE não tem características inteligentes e não pode gerar respostas idênticas às humanas, expressando emoções e atitudes (Heller et al., 2005).

Em 2001 houve uma evolução em tecnológica de *chatbots* com o desenvolvimento de SmarterChild, que estava disponível em aplicações de conversação como America Online (AOL) e Microsoft Network (MSN). Foi pela primeira vez que um *chatbot* podia ajudar pessoas em tarefas práticas diárias assim como poderia devolver informação de bases de dados acerca de filmes, resultados de jogos desportivos,

[1] Técnica de ciência da computação para encontrar um padrão num conjunto de dados

preços de ações, novidades e meteorologia. Esta funcionalidade marcou um desenvolvimento significativo em inteligência computacional e interação entre homem e máquina, em que sistemas de informação podem ser acedidos através da discussão com o *chatbot* (Molnár e Szüts, 2018).

O desenvolvimento de *chatbots* baseados em inteligência artificial deu mais um largo passo com a criação de assistentes de voz inteligentes, inseridos nos *smartphones* ou *home speakers*, os quais compreendem comandos de voz, e tratam tarefas como monitorizar dispositivos da habitação, calendários, email e outros Hoy, 2018.

A forma como os *chatbots* hoje em dia conversam é totalmente diferente da sua antecessora ELIZA. Os *chatbots* dos dias de hoje podem partilhar os seus pensamentos pessoais, dramas familiares, eventos relevantes, entre outros tal como os humanos.

2.3 Sistemas de diálogo

Os sistemas de diálogo são ferramentas que permitem definir as ações que o *chatbot* deve tomar na conversação com um utilizador. Estes sistemas também podem possibilitar a integração de outros serviços, como serviços externos, integrar numa rede social e/ou aplicação de chat, entre outros. Estas ferramentas são centrais para rapidamente se desenvolver um *chatbot*. De entre as ferramentas existentes no mercado destacam-se Dialogflow, Lex e Rasa.

2.3.1 Dialogflow

Dialogflow foi desenvolvida pela Google e faz parte do Google Cloud Platform (GCP). As características mais relevantes do Dialogflow consiste em permitir criar *chatbots* baseados em voz ou texto, ter um componente de *Natural Language Undestanding* (NLU) avançado, conter um editor do fluxo da conversação avançado e permite os programadores analisar dados analíticos sobre as conversações em *dashboards* (Singh et al., 2019). A maior desvantagem do Dialogflow é o facto de não ser *open source* e ser uma *blackbox*.

2.3.2 Lex

Amazon Lex é um serviço para construir interfaces conversacionais usando texto ou voz. Amazon Lex disponibiliza funcionalidades para *Automatic Speech Recognition* (ASR) que permite converter áudio em texto, e *Natural Language Understanding* (NLU) para reconhecer as entidades do texto (Sreeharsha e Kesapragada, 2022). A maior desvantagem do Amazon Lex é não ser *open source* e ser uma *blackbox*.

2.3.3 Rasa

O Rasa é uma *framework open-source* de *Machine Learning* usada para criar *chatbots* e automatizar a criação de assistentes de texto e voz.

2.3.3.1 Arquitetura do Rasa

A arquitetura do Rasa é modular por design, permitindo uma fácil integração com outros sistemas. Pode-se usar o Rasa Core do ecossistema Rasa como gestor de dialogo mas também se pode fazer uso de outros sistema para o mesmo propósito. Assim pode-se fazer apenas uso do Rasa NLU em conjunção com um serviço de gestão de dialogo fora do ecossistema Rasa.

Quando um utilizador envia uma mensagem ao Rasa o processo que o sistema segue é descrito no diagrama da figura 2.1, sendo que apenas a etapa 1 é tratada pelo Rasa NLU e as restantes pelo Rasa Core. Na etapa 1 a mensagem é recebida e passada ao interpretador (*interpreter*), por exemplo o uso do Rasa NLU para extrair o *intent*[2], as entidades e qualquer outra informação estruturada. Na etapa 2 o *tracker* mantém o estado da conversa, recebe uma notificação que uma nova mensagem foi recebida. A *policy*[3] recebe o estado atual do *tracker*, descrita na etapa 3. Na etapa 4, a *policy* decide qual a ação a ser tomada de seguida. Na etapa 5 a ação escolhida é guardada pelo *tracker*. Por fim na etapa 6 a ação é executada, incluindo o envio da resposta ao utilizador e no caso da ação não ser identificada volta ao passo 3. O estado do dialogo é guardado num objeto (*tracker* da figura 2.1), havendo apenas um objeto por conversa que é o único componente do sistema com estado. O *tracker* guarda *slots* assim como os *logs* dos eventos que levaram a dar-se esse estado na conversa. O estado da conversa pode ser reconstruido voltando a executar todos os eventos.

A equipa de desenvolvimento do Rasa definiu o gestor de dialogo como um problema de classificação, em que a cada iteração o Rasa Core prevê qual a ação a tomar de uma lista predefinida. Uma ação pode ser uma simples frase ou uma ação arbitraria a executar. Quando uma ação é executada, é passada a uma instância do *tracker* de forma a fazer uso relevante de informação coletada na história do dialogo: *slots*, frases anteriores e resultados de ações anteriores. As ações não podem alterar diretamente o *tracker*, mas quando executadas enviam uma lista de eventos ao *tracker* que os consome e atualiza o seu estado.

O Rasa NLU contém um conjunto de módulos de NLU e bibliotecas de ML numa API consistente. O objetivo da equipa no desenvolvimento do Rasa NLU é permitir um balanceamento entre a possibilidade de configuração e a facilidade de utilização, sendo que existem fluxos pré-definidos para este fim.

O objetivo da *policy* é selecionar a próxima ação a ser executada dado o objeto *tracker*. Uma *policy* é instanciada com um *featurizer*, que cria um vetor que representa o estado atual do dialogo. O *featurizer* concatena as seguintes características:

- Qual foi a última ação
- O *intent* e as entidades na mensagem mais recente
- Quais os *slots* definidos

[2] No contexto de PLN, um *intent* refere-se a um objetivo ou intenção do *input* do utilizador.

[3] No contexto do Rasa, uma *policy* descreve um conjunto de regras que o *usa* para tomar decisões sobre como responder ao *input* do utilizador

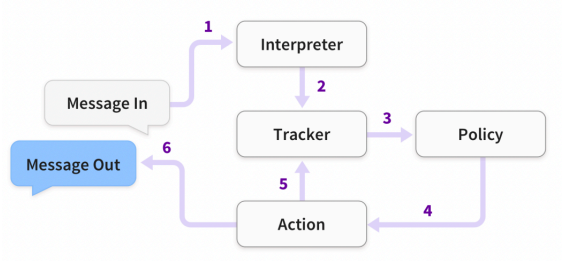


Figura 2.1: Arquitetura do Rasa

2.3.3.2 Elementos do Rasa

Rasa funciona com 3 elementos distintos:

- Natural Language Understanding (NLU)
- Natural Language Generation (NLG)
- Gestão de diálogo

O elemento de NLU converte o texto em vetores para identificar a intenção da frase e converter o texto que recebe de *input* em *tokenizers* com a extração de entidades. Isto é possível com *part-of-speech tagger* que é usado para colocar uma *tag* com uma parte do discurso como substantivos, verbos, etc. De seguida, o *chunker* agrupa os substantivos com palavras relacionadas com eles.

O elemento de NLG é um subconjunto de IA que é capaz de receber formatos não linguísticos como *input* e converter para formato capaz de ser percetível pelos humanos.

O sistema de diálogo como o nome indica é usado para gerir quando os dados do *intent* e entidade são recebidos os consegue encaminhar corretamente entre perguntas e respostas.

2.4 Transformers

O modelo de *Neural Networks* (NN) denominado por *Transformer* tem capacidades especiais para problemas de PLN.

O artigo de Vaswani et al., 2017 descreve os *transformers* como uma arquitetura *sequence-to-sequence* (Seq2Seq). A arquitetura Seq2Seq é uma NN que transforma uma sequência de elementos, tal como uma sequência de palavras numa frase em outra possível frase. Uma escolha popular para estes tipos de modelos são baseados em *Long Short Term Memory* (LSTM). Frases são dependentes da sequência, tal que a ordem das palavras é crucial para entender a frase, assim as LSTM são a escolha natural para este tipo de dados. Os modelos Seq2Seq consistem num *encoder* e um *decoder*. O *encoder* recebe a sequencia e mapeia para um vetor com n dimensões. O vetor é enviado para o *decoder* que o torna numa sequência de *output*, sendo que este *output* pode estar noutra linguagem, com símbolos, uma cópia do *input* de entre outras inúmeras possibilidades.

2.4.1 Attention-Mechanism

O *attention-mechanism* verifica quais as partes relevantes de uma sequência. Por exemplo quando se lê um texto, a atenção está na palavra que estamos no instante a ler mas ao mesmo tempo a mente precisa de manter as palavras-chave relevantes em memória de forma a ter contexto. O mecanismo de atenção funciona de forma similar para uma dada sequência. Ou seja, para cada *input* de uma LSTM (*Encoder*), o *attention-mechanism* tem em conta os outros *inputs* relevantes ao mesmo tempo que decide quais são importantes para atribuir diferentes "pesos" para os *inputs*. O *decoder* vai ter em conta o *input*, a sequência *encoded* e os "pesos" obtidos pelo *attention-mechanism* (Luong et al., 2015).

2.4.2 Funcionamento de Transformers

O *Transformer* usa o mecanismo de atenção anteriormente descrito. Tal como as LSTM, o *Transformer* é uma arquitetura para transformar uma sequência noutra com a ajuda de duas partes, o *encoder* e o *decoder*, mas difere dos outros modelos Seq2Seq existentes porque não implica nenhuma *Recurrent Neural Network* (RNN), tais como *Gated Recurrent Units* (GRU), LSTM, etc..

As RNN foram, até ao momento, uma das melhores abordagens para obter dados sequenciais. No entanto, os investigadores que publicaram o artigo Vaswani et al., 2017 provaram que é possível uma arquitetura com apenas mecanismo de atenção, sem RNN, ter um melhor desempenho para tarefas de PLN, tal como tradução. Uma melhoria em tarefas de PLN foi introduzida pela equipa que apresentou *Bidirectional Encoder Representations from Transformers* (BERT) (Devlin et al., 2019).

A figura 2.2 representa a arquitetura do Transformer. O *encoder* e o *decoder* estão respetivamente representados à esquerda e direita. Ambos são compostos de módulos que podem ser empilhados no topo de outros múltiplas vezes, descritos por Nx na figura 2.2. Estes módulos consistem maioritariamente em *Multi-Head Attention* e camadas *Feed Forward*. Os *inputs* e *outputs* são primeiramente embebidos num espaço com n dimensões dado que não podemos usar como texto diretamente.

As operações realizadas pelo *Multi-Head Attention* são representadas pelas figuras 2.3 e 2.4.

A operação *Scaled Dot-Product Attention* é dada pela equação 2.1, em que Q (*query*) é a matriz que contem a representação vetorial de uma palavra numa sequência, K (*keys*) são as representações vetoriais de todas as palavras na sequência e V são os valores que são representações de todas as palavras numa sequência. Para o *encoder* e o *decoder*, nos módulos de *Multi-Head Attention*, V consiste na mesma palavra da sequência que Q. No entanto para o módulo *Attention* que tem em conta o *encoder* e o *decoder*, V é diferente das sequências representadas por Q.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

Pode-se simplificar um pouco, afirmando que os valores em V são multiplicados e somados com *attention-weights*, a, onde os pesos são definidos pela equação 2.2. Isto significa que os pesos a são definidos como cada palavra da sequência, Q, é influenciada por todas as outras palavras na sequência,

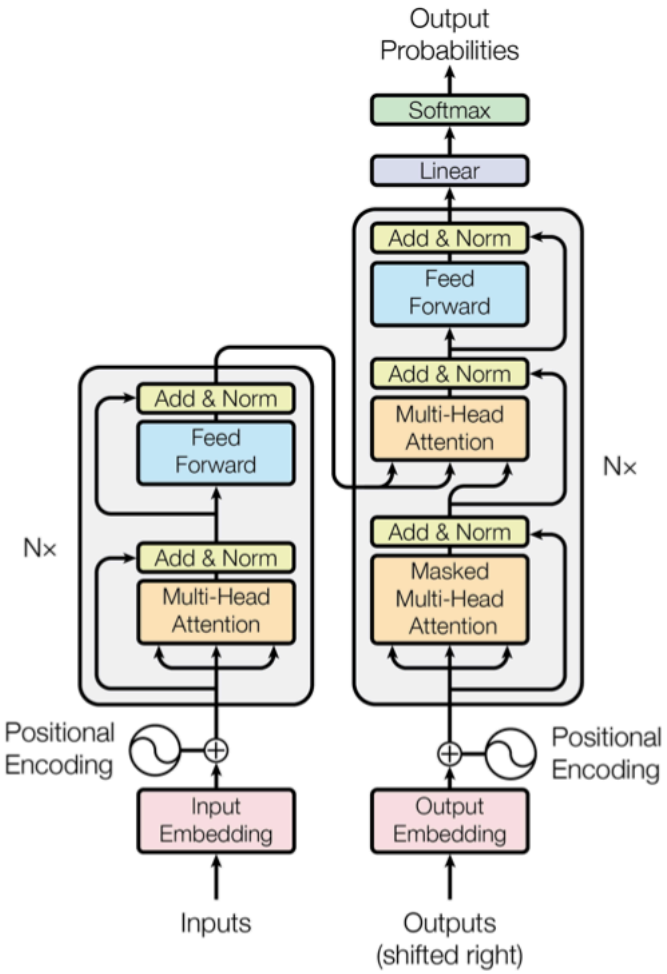


Figura 2.2: Arquitetura do Transformer (Vaswani et al., 2017)

K. Adicionalmente a função softmax é aplicada aos pesos a, que têm uma distribuição $\in [0, 1]$. Estes pesos são aplicados a todas as palavras na sequência que foi introduzida em V (os mesmos vetores que Q para o *encoder* e *decoder* mas diferentes para o módulo que tem os *inputs* do *encoder* e *decoder*).

$$a = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.2}$$

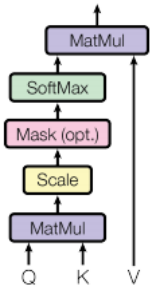


Figura 2.3: Scaled Dot-Product Attention (Vaswani et al., 2017)

A figura 2.4 descreve como o mecanismo de atenção pode ser paralelizado em múltiplos mecanismos que podem ser usados lado a lado. Mecanismo este que é repetido múltiplas vezes com projeções lineares

de Q, K e V, permitindo que o sistema possa aprender de diferentes representações de Q, K e V. As representação lineares são obtidas com a multiplicação de Q, K e V por matrizes de pesos, W, que são aprendidas durante o treino.

As matrizes Q, K e V são diferentes para cada posição dos módulos de atenção na estrutura dependendo se estão no *encoder*, *decoder* ou entre o *encoder* e *decoder*. O motivo passa por se querer ter atenção a toda a sequência de entrada do *encoder* ou parte da sequência de entrada do *decoder*. O módulo *Multi-Head Attention* que conecta o *encoder* e o *decoder* vai garantir que a sequência de entrada do *encoder* é tida em conta tal como a sequência de entrada do *decoder*, até pelo menos uma determinada posição.

Após **Multi-Attention heads** em ambos o *encoder* e *decoder* tem-se uma camada *feed-forward*. Esta rede *feed-forward* tem parâmetros idênticos para cada posição, que podem ser descritos como separados, aplicando-se uma transformação linear idêntica para cada elemento da sequência dada.

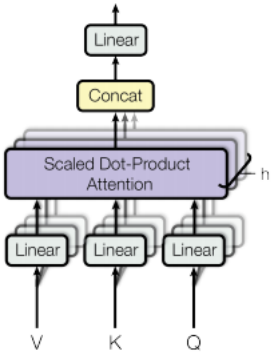


Figura 2.4: Multi-Head Attention (Vaswani et al., 2017)

2.5 BERT

As *Bidirectional Encoder Representations from Transformers* (BERT) foram introduzidas por investigadores da Google AI Language através do artigo Devlin et al., 2019. Isto causou grande excitação na comunidade de IA ao apresentar resultados de última geração numa ampla variedade de tarefas de PLN, incluindo *Question Answering* (QA) com o modelo SQuAD v1.1, inferência de linguagem natural (MNLI) e outras.

A inovação chave das BERT passa por aplicar o treino bidirecional do Transformer. Os resultados do artigo demonstram que o modelo que é treinado bidirecional tem um maior sentido do contexto da linguagem e fluxo que modelo de única direção. No artigo é demonstrada uma técnica novel denominada *Masked LM* (MLM) que permite treino bidirecional em modelos que previamente era impossível.

BERT faz uso de Transformer. Como descrito na secção anterior, o Transformer tem 2 mecanismos separados: o *encoder* que lê o texto recebido como *input* e o *decoder* que produz o resultado para a tarefa. Dado que o objetivo das **BERT** é gerar um modelo de linguagem, apenas o *encoder* é necessário. Em oposição aos modelos direcionais, que lêm o *input* sequencialmente (esquerda para a direita ou direita para a esquerda), o *encoder* do Transformer lê toda a sequência de palavras numa única vez. Esta

caraterística permite ao modelo aprender o contexto de uma palavra baseada em todas as palavras que a rodeiam.

O *input* é uma sequência de *tokens*, os quais são primeiramente embebidos em vetores e depois processados numa *neural network*.

Quando se treina modelos de linguagens há o desafio de definir qual o objetivo a prever. Diversos modelos prevêm a próxima palavra na sequência (por exemplo "The child came home from ..."), uma abordagem direcional que limita a aprendizagem do contexto. Para ultrapassar este desafio, as **BERT** usam duas estratégias de treino: *Masked LM* (MLM) e *Next Sentence Prediction* (NSP).

2.5.1 Masked LM

Antes de inserir as sequências de palavras nas **BERT**, 15% das palavras em cada sequência são alteradas por um *token* [MASK]. O modelo tenta então prever o valor original das palavras com mascaras, baseado no contexto dado pela outra, sem máscara, palavras na sequência (figura 2.5).

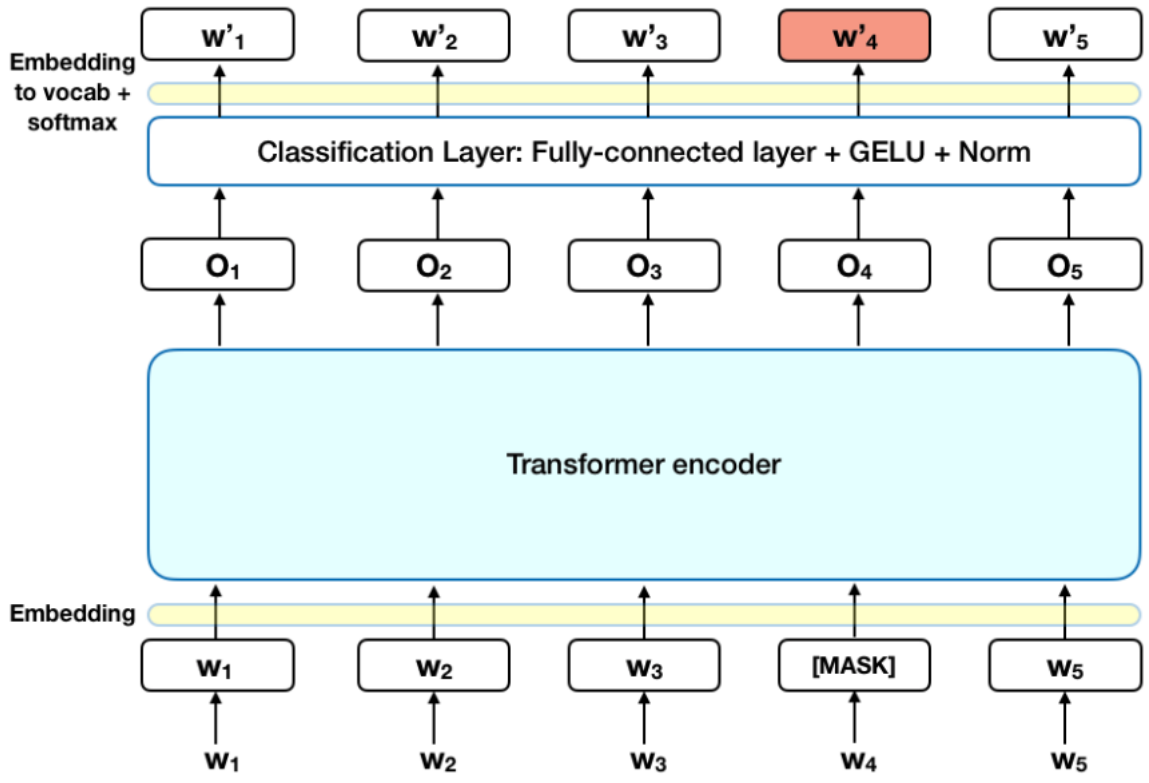


Figura 2.5: Arquitetura da Masked LM

Em termos técnicos, a previsão das palavras requer:

1. Adicionar uma camada de classificação no topo do *output* do *encoder*
2. Multiplicar os vetores de *output* por uma matriz embebida, transformando na dimensão do vocabulário
3. Calcular a probabilidade de cada palavra no vocabulário com a função de ativação softmax

A função de custo BERT tem em consideração apenas a previsão dos valores mascarados e ignora a previsão das palavras sem máscara. Como consequência, o modelo converge mais lentamente que modelos direcionais, uma característica que é compensada pela sua maior perceção do contexto.

2.5.2 Next Sentence Prediction

No processo de treino das BERT, o modelo recebe pares de sequências como *input* e aprende a prever se a segunda sequência no par é uma parte da sequência no documento original. Durante o treino, 50% dos *inputs* são um par no qual a segunda sequência é uma parte da sequência no documento original, enquanto nos outros 50%, uma sequência aleatória do corpo de texto é escolhida como a segunda sequência. A hipótese é que uma sequência aleatória vai ser desconectada da primeira sequência.

Para ajudar o modelo a distinguir entre duas frases no treino, o *input* é processado na seguinte forma antes de entrar no modelo (figura 2.6):

1. Um *token* [CLS] é inserido no inicio da primeira frase e um *token* [SEP] é inserido no fim de cada frase

2. Uma frase embebida indicando que a frase A ou a frase B é adicionada a cada *token*. Frases embebidas são semelhantes no conceito aos *tokens* embebidos com um vocabulário de 2

3. Um *embedding* é adicionado a cada *token* para indicar a sua posição na frase.

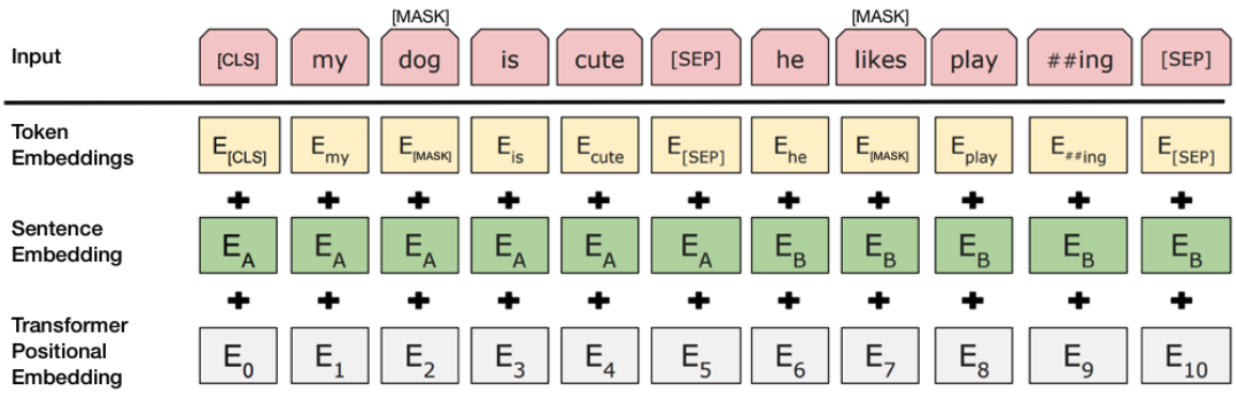


Figura 2.6: Arquitetura de NSP

Para prever se a segunda frase está efetivamente conectada à primeira, os seguintes passos são necessários verificar:

1. Toda a frase de *input* percorre o modelo Transformer

2. O *output* do *token* [CLS] é transformado num vetor de $2x1$, usando uma simples camada de classificação (aprendendo matrizes de pesos e de tendência (*bias*)

3. Calcular a probabilidade de *IsNextSequence* com a função softmax

Quando se treina o modelo BERT, Masked LM e NSP são treinadas em conjunto com o objetivo de minimizar a função de custo combinada com as duas estratégias.

2.5.3 Treinar BERT

Utilizar BERT para uma especifica tarefa é relativamente acessível. BERT podem ser usadas num grande variedade de tarefas em linguagens, apenas adicionando uma camada ao *core* do modelo:

1. Tarefas de classificação como análise de sentimento são realizadas de forma similar a NSP, adicionando uma camada de classificação no topo do *output* do Transformer para o *token* [CLS].

2. Em tarefas de QA (e.g. SQuAD v1.1), o *software* recebe a questão de acordo com a sequência de texto e é requerido para marcar a resposta na sequência. Usando BERT, um modelo para uma tarefa de *question answering* (QA) pode ser treinado aprendendo dois vetores extra que marcam o início e fim da resposta.

3. Em *Named Entity Recognition* (NER), o *software* recebe a sequência de texto e é requerido marcar diversos tipos de entidades (pessoas, organizações, datas, etc) que aparecem no texto. Usando BERT, um modelo NER pode ser treinado preenchendo o vetor de saída com cada *token* numa camada de classificação que prevê a *label* NER.

No treino de *fine-tunning*, a maioria dos hiper-parâmetros continuam os mesmo que no treino da BERT, sendo que na secção 3.5 do artigo Devlin et al., 2019 descreve um guia para os hiper-parâmetros que requerem *tunning*

O gráfico da figura 2.7 demonstra como se deve ter em contra a aprendizagem para um modelo BERT.

Através do mesmo é possível perceber que o tamanho do modelo importa, mesmo para larga escala. `BERT_large`, com 345 milhões de parâmetros, é o maior modelo do seu género. É superior em pequenas tarefas de pequena escala em relação em `BERT_base`, a qual usa a mesma arquitetura com "apenas"110 milhões de parâmetros.

Com suficientes dados de treino, mais passos de treino está correlacionado com maior precisão do modelo. Para uma tarefa MNLI, a precisão de `BERT_base` melhora em 1% quando treinada com um milhões de passos (128 mil palavras por *batch*) comparando com 500 mil passos para o mesmo tamanho de *batch*.

A abordagem bidirecional de BERT (MLM) converge mais lentamente do que abordagens de esquerda para direita (porque 15% das palavras são previstas em cada *batch*) mas o treino bidirecional continua a ter um melhor desempenho que treino da esquerda para a direita após um pequeno número de passos pré treinados.

2.5.4 Word Masking

Treinar o modelo da linguagem em BERT é feito prevendo 15% dos *tokens* no *input*, que são aleatoriamente escolhidos. Estes *tokens* são processados como: 80% substituidos com o *token* [MASK], 10% com uma palavra aleatória, e 10% com a palavra original. A intuição que leva os autores a escolher esta abordagem é:

- Se se usar 100% [MASK] o modelo não vai necessariamente produzir boas representações dos *tokens* para palavras sem mascara. Os *tokens* sem mascara ainda são usados para contexto, mas o modelo foi otimizado prevendo palavras com mascara.
- Se se usar 90% [MASK] e 10% palavras aleatórias vai ensinar o modelo que a palavra observada nunca está correta.
- Se se usar 90% [MASK] e 10% a mesma palavra, o modelo pode apenas copiar o *embedding* sem contexto.

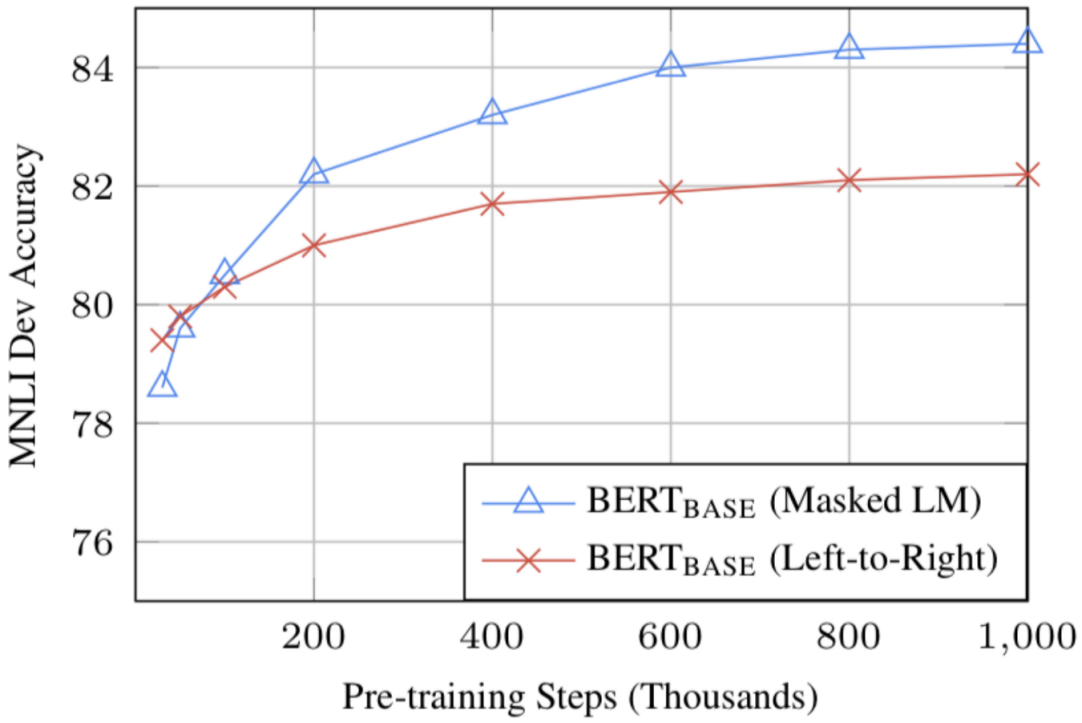


Figura 2.7: Aprendizagem de BERT (Devlin et al., 2019)

2.5.5 Hugging Face

Apesar de ser possível treinar o nosso próprio BERT, existem bibliotecas que contém modelos deste tipo treinados para uso imediato. Hugging Face é uma biblioteca *open-source* para desenvolver, treinar e submeter modelos de ML, especialmente modelos de PLN. Pode-se instalar esta biblioteca em python com o comando `pip install transformers[tf-cpu]`. O BERT é ideal para QA, sendo benéfico incluir tal

funcionalidade num *chatbot* de forma a que consiga responder a questões do utilizador dado as fontes de informação do *chatbot*.

O código para se alcançar a funcionalidade de QA com o uso da biblioteca Hugging Face está apresentado na listagem 2.1. Este exemplo faz uso de um modelo e um *tokenizer*. Neste caso o modelo é o `bert-large-uncased-whole-word-masking-finetuned-squad`. O papel do *tokenizer* é atribuir *tokens* a um texto que é dividido nas palavras ou sub-palavras, que depois são convertidas em `ids` através de uma tabela *look-up*.

Listagem 2.1: Uso de BERT com o auxilio da biblioteca Hugging Face

```
from transformers import AutoTokenizer, TFAutoModelForQuestionAnswering
import tensorflow as tf

tokenizer = AutoTokenizer.from_pretrained(
   "bert-large-uncased-whole-word-masking-finetuned-squad"
)
model = TFAutoModelForQuestionAnswering.from_pretrained(
   "bert-large-uncased-whole-word-masking-finetuned-squad"
)

text = r"""
Transformers (formerly known as pytorch-transformers and pytorch-pretrained-bert)
provides general-purpose architectures (BERT, GPT-2, RoBERTa, XLM, DistilBERT, …XLNet)
for Natural Language Understanding (NLU) and Natural Language Generation (NLG)
with over 32+ pretrained models in 100+ languages
and deep interoperability between TensorFlow 2.0 and PyTorch.
"""

questions = [
   "How many pretrained models are available in Transformers?",
   "What does Transformers provide?",
   "Transformers provides interoperability between which frameworks?",
]

for question in questions:
   inputs = tokenizer.encode_plus(
      question,
      text,
      add_special_tokens=True,
      return_tensors="tf"
   )
   input_ids = inputs["input_ids"].numpy()[0]

   text_tokens = tokenizer.convert_ids_to_tokens(input_ids)
   answer_start_scores, answer_end_scores = model(inputs)
```

```
36
37    answer_start = tf.argmax(
38       answer_start_scores, axis=1
39    ).numpy()[0] # Get the most likely beginning of answer with the argmax of the score
40    answer_end = (
41       tf.argmax(answer_end_scores, axis=1) + 1
42    ).numpy()[0] # Get the most likely end of answer with the argmax of the score
43    answer = tokenizer.convert_tokens_to_string(
44       tokenizer.convert_ids_to_tokens(
45          input_ids[answer_start:answer_end]
46       )
47    )
48
49    print(f"Question: {question}")
50    print(f"Answer: {answer}\n")
```

Capítulo 3

Objetivos e Funcionalidades

O objetivo deste projeto é permitir a utilizadores sem *background* tecnológico criar *chatbots* e permitir a sua gestão. Apesar de já existirem *frameworks* bastante robustas para criar um *chatbot* e disponibilizar o mesmo *online*, para a maioria das soluções continua a ser necessário realizar trabalho de baixo nível, ou seja, é necessário conhecimento tecnológico para desenvolver o *chatbot* com o uso dessas *framework*. Além disso a maioria das *frameworks* apenas permite a criação de *chatbots* para contextos bastante específicos, sendo que o ponto de destaque do ChatbotWizard é permitir criar *chatbot* para qualquer contexto que o utilizador possa imaginar. Assim, este projeto tem como objetivo permitir criar um *chatbot* apenas com acesso à Internet, incorporando diversos módulos com uma solução de *drag and drop*.

Metaforicamente, cada módulo é uma peça do puzzle que constrói um chatbot, ou seja, um módulo pode ser uma função que obtém as cidades de uma frase ou executar uma API remota. O objetivo dos módulos do ChatbotWizard é permitir aos utilizadores usarem módulos para construir o fluxo de conversação do chatbot, e além disso permitir que utilizadores com conhecimentos mais técnicos possam desenvolver novos módulos para o ChatbotWizard. Dessa forma permite o desenvolvimento de módulos para a comunidade poder integrar no seu chatbot.

Após a descrição do ChatbotWizard é necessário destacar as funcionalidades que o mesmo deve incorporar. Para se alcançar este objetivo o ChatbotWizard deve incorporar várias funcionalidades:

- Criar o *chatbot* na aplicação web
- Gerir um *chatbot* existente
- Adicionar novos módulos para outros utilizadores integrarem no seu chatbot
- Integrar o *chatbot* com redes sociais, como por exemplo Telegram, Facebook Messenger, WhatsApp etc

É essencial para a arquitetura do ChatbotWizard que tenha dois componentes distintos: a ChatbotWizard *web* em que o utilizador pode criar o seu chatbot, editar o mesmo e adicionar novos módulos, e o backend do ChatbotWizard que vai lidar com toda a lógica de gestão das diversas funcionalidades do ChatbotWizard. A figura 3.1 representa a arquitetura do ChatbotWizard

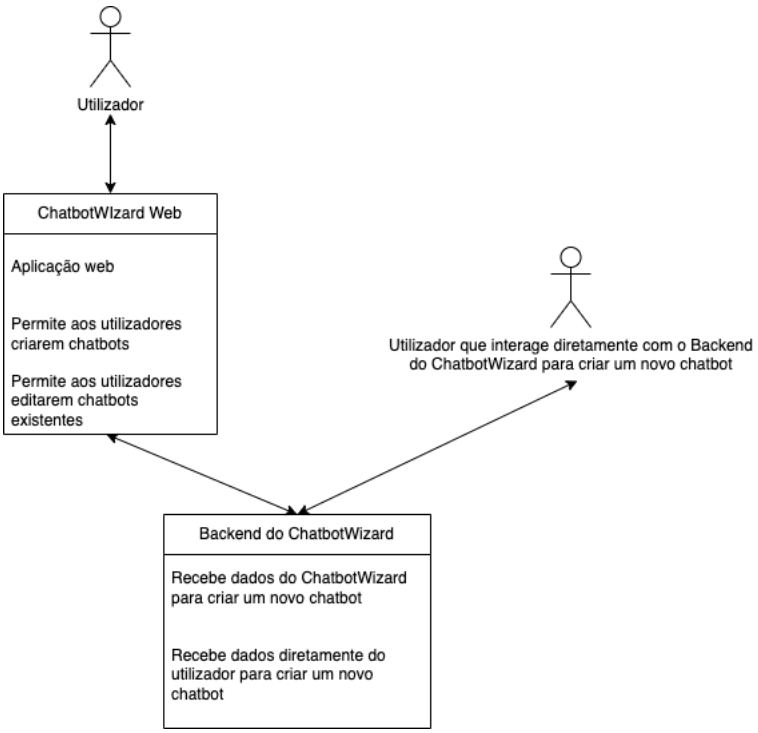


Figura 3.1: Arquitetura do sistema

3.1 Backend do ChatbotWizard

Pretende-se que o *backend* do ChatbotWizard seja um *web service*. Um *web service* é uma solução utilizada na integração de sistemas e na comunicação entre diferentes aplicações. Ao desenvolver o *backend* do ChatbotWizard como um *web service* permite expor as funcionalidades de um sistema através de um protocolo padronizado. Os web services são baseados num conjunto de padrões da internet definidos pelo *World Wide Web Consortium* (W3C) em que o protocolo *Hypertext Transfer Protocol* (HTTP) atua como transportador entre o cliente e o *web service* (Richardson e Ruby, 2007). A comunicação entre sistemas por um *web service* permite seguir vários protocolos: *Simple Object Access Protocol* (SOAP), REST e *Extensible Markup Language Remote Procedure Call* (XML-RPC) (Richardson e Ruby, 2007). No caso do ChatbotWizard vai seguir um *web service* com o protocolo REST. Um *web service* definido com os princípios REST pode ser denominado RESTful *web service*. REST usa os verbos HTTP (GET, POST, PUT, DELETE) para aceder, alterar e eliminar recursos respetivamente.

3.1.1 Endpoints do ChatbotWizard

Para as funcionalidades descritas nos objetivos do ChatbotWizard é necessário que o mesmo contenha diversos *endpoints*.

Começando pela gestão de *chatbots* é preciso que seja possível criar um chatbot, editá-lo e listar os diversos chatbots já criados. Assim, para a gestão de *chatbots* são precisos 3 *endpoints*. O *endpoint* da criação de um novo *chatbot* deve ter uma especificação idêntica a `POST https://www.chatbot.wizard/api/chatbots` com a estrutura do body como da listagem 3.1. No schema do body, o parâmetro `intents` define exemplos de perguntas a que o *chatbot* deve identificar o *intent*, `action_name` é o nome que o utilizador decidiu dar para o chatbot, `actions` são os diversos módulos que estão incluídos no *chatbot* assim como os respetivos parâmetros, sendo que cada item da lista de `actions` tem um **ID** que a identifica. É relevante salientar que cada componente tem um **ID** pois os parâmetros *input* e *output* são os **ID** do componente inicial e final do fluxo do *chatbot* respetivamente. O *endpoint* de editar um *chatbot* tem o propósito de fazer alterações num *chatbot* já existente com a especificação `PUT https://www.chatbot.wizard/api/chatbots`, sendo que o *body* enviado para editar o *chatbot* é o mesmo da criação do *chatbot* presente na listagem 3.1.

Listagem 3.1: Body para criar um novo chatbot

```
{
   "input": "string",
   "output": "string",
   "action_name": "string",
   "intents": [
      "string"
   ],
   "actions": [
      {
         ...
      },
   ]
}
```

Além de permitir criar e editar um *chatbot* é essencial listar os *chatbots* já existentes de forma a poderem ser vizualizados ou mesmo editados. A especificação para listar os *chatbots* existentes é `GET https://www.chatbot.wizard/api/chatbots`, sendo que o schema da resposta está presente na listagem 3.2

Listagem 3.2: Schema da resposta do pedido `GET https://www.chatbot.wizard/api/chatbots`

```
[
   {
      "input": "string",
      "intents": [
         "string"
```

```
      ],
      "actions": [
         {
            ...
         },
      ],
   },
   {
      ...
   }
]
```

Como salientado anteriormente para se criar o fluxo do *chatbot* são necessários módulos. Dado que se quer criar novos módulos, editar módulos e listar módulos existentes para integrar no *chatbot* também são necessários 3 *endpoints* para estas funcionalidades. O *endpoint* definido para listar os módulos é especificado como `GET https://www.chatbot.wizard/api/modules`, obtendo-se um schema idêntico ao da listagem 3.3. Na resposta do pedido GET (listagem 3.3) da listagem dos módulos presentes é possível entender-se que cada módulo é identificado pelo nome do módulo (`name`), a descrição (`description`), o código que permite esse módulo funcionar (`code`) e os parâmetros de entrada (`input_params`). Os parâmetros de entrada são essenciais para quem desejar usar esse módulo no fluxo do seu *chatbot* definir o valor a preencher para a função executar com sucesso. Assim, tanto para criar um novo módulo com a especificação `GET https://www.chatbot.wizard/api/modules` ou alterar um módulo com o pedido `GET https://www.chatbot.wizard/api/modules` deve ser enviado um *body* com um *schema* idêntico ao da listagem 3.4

Listagem 3.3: Schema da resposta do pedido `GET https://www.chatbot.wizard/api/modules`

```
[
   {
      "name": "factorial",
      "description": "Find the factorial of an integer",
      "input_params": ["x"]
      "code": "
            def factorial(x):
                if x == 1:
                    return 1
                return x * factorial(x-1)
        "
   },
   {
      ...
   }
]
```

Listagem 3.4: Schema de body do pedido `POST https://www.chatbot.wizard/api/modules`

```
{
   "name": "subtract_bigger_to_smaller",
   "description": "Function that substract the bigger number to the smaller one",
   "input_params": ["a", "b"]
   "code": "
        def foo(a, b):
            if a >= b:
                return a - b
            return b -a
    "
}
```

3.1.2 Módulos do ChatbotWizard

Os módulos do ChatbotWizard permitem tornar esta *framework* mais escalável do que as *frameworks* anteriormente mencionadas ao longo da dissertação. Estes módulos dão hipótese infindáveis aos utilizadores de criar *chatbots* para qualquer caso de uso. No caso de ser um utilizador com conhecimentos técnicos pode ajudar todos os outros utilizadores criando novos módulos para uso.

No entanto, para permitir o uso do ChatbotWizard, um dos objetivos da aplicação é permitir logo de inicio uso de módulos feitos no desenvolvimento da aplicação. Estes módulos devem permitir criar infindáveis hipóteses de *chatbots* para o caso de uso que o utilizador desejar. Os seguintes módulos foram considerados cruciais:

- Extrator de entidades - permite extrair entidades mencionadas no texto
- Acesso a fontes de dados externas através de uma API - permite aceder e manipular dados de serviços externos expostos por API
- *template* para construir texto a partir de JSON - permite converter JSON para texto dado um template definido pelo utilizador
- Módulo de questão e resposta - permite responder a questões dado uma pergunta e um texto

3.1.2.1 Extrator de entidades do ChatbotWizard

Uma grande parte de informação digital é expressa como texto em linguagem natural o que não é facilmente processado por computadores. Grafos de conhecimento, *Knowledge Graphs* (KG), oferecem um formato amplamente utilizado para representar a informação que o computador entenda (Al-Moslmi et al., 2020).

Processamento de linguagem natural, *Natural Language Processing* (NLP), é preciso para extrair os KG dos textos de linguagem natural com o objetivo central de extrair as entidades do texto. NLP tem o

objetivo de permitir os computadores processar linguagem humana de maneiras significativas (Abu-Salih, 2021). NLP pode ser descrita como uma área de pesquisa e aplicação que explora como os computadores podem ser usados para perceber e manipular texto ou áudio para realizar algo significativo (Braşoveanu, 2021). NLP é geralmente usado para derivar semânticas do texto ou áudio e que o *encoda* num formato estruturado que é adequado para pesquisa semântica e outros tipos de processamento de computador (Sreeram et al., 2021).

KG foram introduzidos pela Google em 2012 (Singhal et al., 2012, Yan et al., 2018) para interligar dados que poderiam ser usados para auxiliar nas pesquisas. Nos grafos de conhecimento os nós representam objetos concretos, conceitos, recursos de informação ou dados sobre os mesmos e as arestas representam as relações semânticas entre os nós. Para permitir que o texto esteja disponível para os computadores num formato entendido pelos mesmos requer NLP e outros tipos de técnicas de extração de informação (figura 3.2), sendo o desafio central identificar as entidades mencionadas no texto. Estas entidades podem ser entidades nomeadas que se referem a conceitos individuais ou abstratos. Nos grafos de conhecimento as entidades podem ser representadas como os nós e as relações como as áreas, grafos de conhecimento são uma forma natural de representar texto natural de forma que o computador consiga processar.

Figura 3.2: Fluxo de linguagem natural até grafos de conhecimento (Al-Moslmi et al., 2020

As entidades nomeadas podem ser pessoas, organizações, localizações ou eventos. Uma menção é uma peça de texto que se refere a uma entidade. Como mencionado anteriormente, extrair as entidades de texto e as representar como nós em KG envolve três tarefas:

- Named Entity Recognition (NER)
- Named Entity Disambiguation (NED)
- Named Entity Linking (NEL)

Named Entity Recognition (NER) tem como objetivo extrair cada segmento no texto que mencione a entidade. *Named Entity Disambiguation* (NED) deve determinar qual a entidade nomeada é referida na menção, ou seja, Trump pode-se referir a uma pessoa, empresa ou edifício. *Named Entity Linking* (NEL) tem como objetivo providenciar um IRI para cada entidade sem ambiguidade, ou seja, NEL tem que aceder a fontes de dados para dado o segmento de texto identificar se a entidade Trump se refere à pessoa, edifício ou empresa. NED e NEL estão interligados, porque uma entidade ambígua deve ser desambiguada antes de poder ser ligada e porque um IRI é uma ótima forma de representar o resultado sem ambiguidade. A figura 3.3 representa a sequência de tarefas desde texto natural até grafos de conhecimento. A mesma figura ainda mostra as mais recentes abordagens *end to end* (também mencionadas *zero-shot*) que junta tudo nas três tarefas, usando normalmente *deep neural networks*.

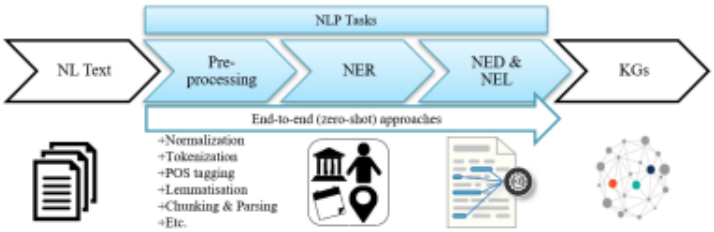


Figura 3.3: Tarefas de NLP (Al-Moslmi et al., 2020)

NER foi introduzido em Grishman e Sundheim, 1996. De acordo com Yadav e Bethard, 2019, o propósito de NER é identificar entidades em texto como pessoas, localizações, organizações, tempo, procedimentos clínicos, dinheiro, proteínas biológicas, entre outros. NER é um campo de pesquisa em rápido desenvolvimento Goyal et al., 2018. A maioria das abordagens propostas têm sido especificas a um domínio, limitando a por exemplo noticias, avaliações. Pode-se dividir NER em três categorias (Goyal et al., 2018), como demonstrado na figura 3.4:

- A primeira e mais antiga categoria é de baseada em regras (Sharnagat, 2014 e Nadeau e Sekine, 2007). A maioria dos estudos nesta categoria são de regras escritas à mão (Goyal et al., 2018 e Nadeau e Sekine, 2007). A vantagem de tais métodos é que eles não requerem dados de treino anotados porque estes métodos não dependem de recursos léxicos. Outra vantagem é a precisão de métodos escritos manualmente devido ao conhecimento do domínio especifico. A óbvia desvantagem destes métodos é serem de domínio especifico (Yadav e Bethard, 2019) e a outra desvantagem deve-se a construir e manter tais recursos para diversas linguagens é bastante custoso (Zheng et al., 2016).

- A segunda categoria dos sistemas NER são baseados em aprendizagem (Ando et al., 2005 e Passos et al., 2014). Estes modelos são usados para substituir regras estáticas precisas pela primeira categoria. Os métodos nesta categoria podem ser divididos em três tipos: supervisionadas, semi-supervisionadas e não supervisionadas. Em métodos supervisionados e semi-supervisionados, um modelo de machine learning é treinado em exemplos de *input* em conjunto com os respetivos *outputs* pretendidos. Support Vector Machines (SVM), Hidden Markov Models (HMM), Conditional Random Fields (CRF) e árvores de decisão são modelos comuns nesta categoria (Yadav e Bethard, 2019). A precisão de NER é normalmente limitada pelo uso do classificador. Por exemplo quando HMM e SVM são empregadas, as dependências entre palavras não são consideradas. Os métodos não supervisionados são mais autónomos (Sharnagat, 2014, Etzioni et al., 2005 e Zhang e Elhadad, 2013), apesar de necessitarem de dados de treino mínimos, denominados de *seeds*.

- A última e mais recente categoria é baseada em abordagens de *feature inferring neural-network* (Alexei et al., 2019 e Akbik et al., 2018). Nesta categoria tal como na anterior depende de machine learning, mas diferem das abordagens baseadas em regras e aprendizagem inferindo características automaticamente através de deep learning. Artigos recentes demonstram que eles têm um

desempenho superior aos métodos anteriores (Yadav e Bethard, 2019 e Goyal et al., 2018). Ao contrário dos métodos anteriores esta abordagem não requer *seeds*, ontologias, ou léxicos de um domínio especifico e por isso são mais independentes do domínio. As abordagens nesta categoria beneficiam da precisão das características inferidas no entanto são necessários grandes conjuntos de dados para construir modelos robustos.

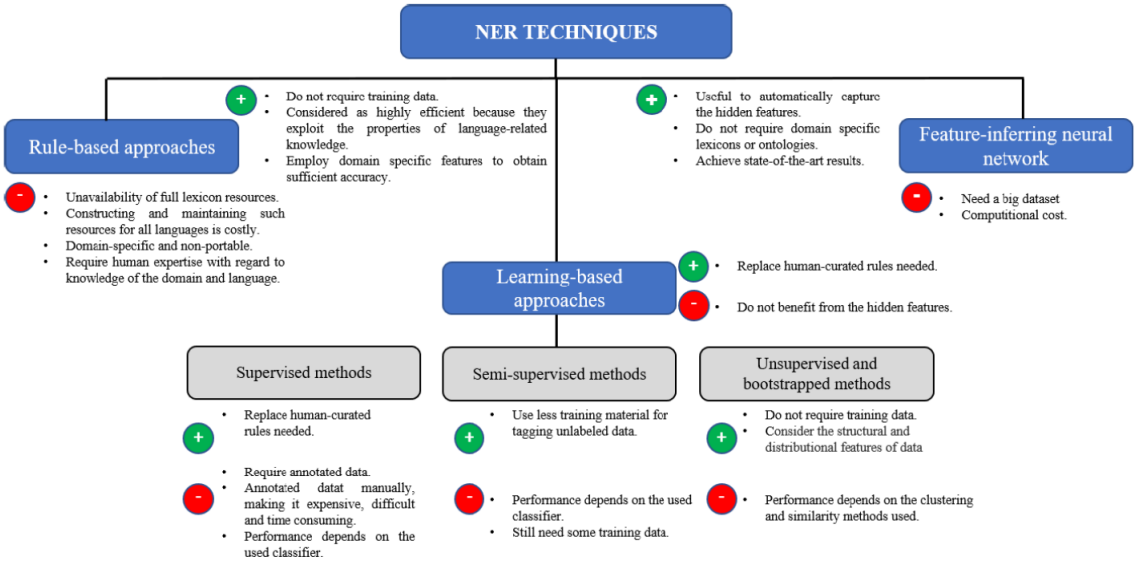


Figura 3.4: Abordagens para NER (Al-Moslmi et al., 2020)

Um extrator de entidades ou NER são chave para qualquer chatbot, pois permitem identificar as entidades das questões do utilizador. Identificar a(s) entidade(s) permite ao *chatbot* saber onde obter os dados para a resposta ou dar a própria resposta. Módulo de extração de entidades por si só deve extrair uma ou mais entidades recebendo como input o tipo de entidade a extrair e o texto para extrair a(s) mesma(s). O extrator de entidades precisa de ser bastante versátil de forma a permitir extrair diversos tipos de entidades, tais como:

- Organizações - A Apple é sediada em Cupertino, Califórnia.
- Pessoas - O 44° presidente dos estados unidos da América foi o Barack Obama.
- Datas - O implantação da republica ocorreu a 5 de outubro de 1910.
- Localizações - O distrito de Braga situa-se no norte de Portugal.

A figura 3.5 representa o comportamento esperado deste módulo.

3.1.2.2 Question and Answering do ChatbotWizard

A procura por conhecimento é profundamente humana, sendo por isso óbvio que assim que apareceram os computadores começamos a fazer-lhes questões. No inicio dos anos 60, os sistemas usavam principalmente dois paradigmas baseado em recuperações de informações e baseados em informações para

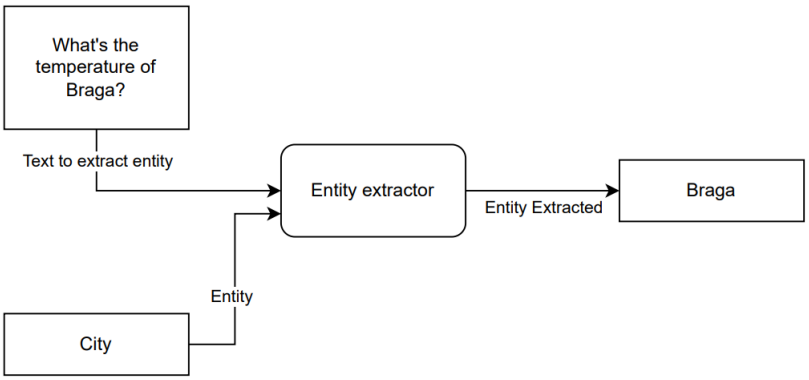


Figura 3.5: Arquitetura do módulo de extrair entidades.

responder a questões de basebol e de factos científicos (Jurafsky e Martin, 2000). Os sistemas de *question answering* são desenvolvidos para preencher as necessidades de informação humana que podem surgir em situações como conversar com um agente virtual, um motor de pesquisa ou realizar pesquisas a uma base de dados. A maioria dos sistemas de QA focam-se num assunto em particular, *factoid questions*, questões que podem ser respondidas com factos expressos em pequenos textos, tal como:

- Onde fica o museu de Louvre?
- Onde fica a universidade do Minho?

Os dois principais paradigmas de *factoid questions* são *Information-retriviel (IR) based QA*, também por vezes denominados *open domain QA* e baseados em informações *knowledge-based QA* (Jurafsky e Martin, 2000).

Information retrievel (IR) é o nome do campo de estudo que engloba a recuperação de todas as informações de todas as formas de media de que o utilizador precisa. O sistema resultante é por vezes denominado um motor de pesquisa. A tarefa de IR considerada é denominada *ad hoc retrieval*, em que o utilizador possui uma *query* a um sistema IR, a qual devolve um conjunto de documentos ordenados de uma coleção. Um documento refere-se a qualquer unidade de texto que o sistema indexa e devolve (páginas web, documentos científicos, noticias ou até pequenas passagens como parágrafos). Uma coleção refere-se a um conjunto de documentos usados para satisfazer os pedidos do utilizado. Um termo refere-se a uma palavra na coleção, mas também pode incluir frases. Finalmente, uma *query* representa a informação expressa pelo utilizador tal como um conjunto de termos. A arquitetura de um sistema IR está representada na figura 3.6. A arquitetura de um sistema IR usa um modelo de espaço vetorial que mapeia as *queries* e documentos em vetores e que usa a semelhança cosseno entre os vetores para criar um *ranking* dos documentos. Isto é um exemplo do modelo *bag-of-words*.

IR-based factoid QA, pode vezes denominado open domain QA, tem como objetivo responder à questão do utilizador encontrando pequenos segmentos de texto da internet ou de outras coleções de dados extensas. O paradigma dominante de open domain QA é recuperar e ler informação como na figura 3.7. Na primeira etapa obtém-se as passagens relevantes de uma coleção de texto, normalmente usando

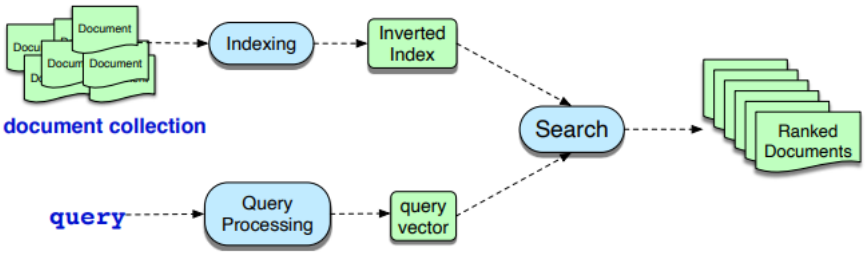


Figura 3.6: Arquitetura ah hoc de um sistem IR (Jurafsky e Martin, 2000)

motores de pesquisa. Na segunda etapa, um algoritmo de compreensão, tal como o Bert, passa sobre cada passagem e encontra partes que são capazes de responder à questão.

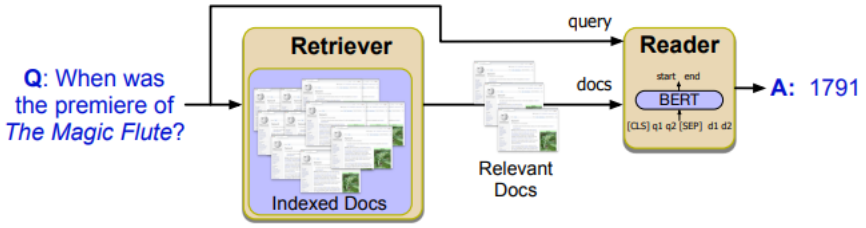


Figura 3.7: Etapas de um sistema IR (Jurafsky e Martin, 2000)

Entity Linking é um paradigma em que a tarefa de associar uma menção no texto com a representação de alguma entidade está na ontologia (Jurafsky e Martin, 2000). A ontologia mais comum para QA de factos é o Wikipédia, em que cada pagina do Wikipédia serve como um único `id` para um entidade particular. A tarefa de decidir qual página do Wikipédia corresponde a uma entidade é denominada por *wikification* (Mihalcea e Csomai, 2007). Desde os sistemas iniciais (Mihalcea e Csomai, 2007, Cucerzan, 2007, Milne e Witten, 2008), *entity linking* é realizada através de duas etapas: deteção da menção e identificação da entidade sem ambiguidade. Para esta tarefa existem dois algoritmos: *anchor dictionaries* com informação do Wikipédia em grafos (Ferragina e Scaiella, 2011) e um modelo de redes neuronais (Li et al., 2020).

Enquanto uma grande parte da informação está presente numa enorme quantidade de texto na internet, a informação também existe em formas mais estruturadas. O termo knowledge-based question answering é usado para a ideia de responder a uma questão de linguagem natural mapeando a mesma numa *query* sobre uma base de dados estruturada. Tal como paradigmas baseados em texto para QA, esta abordagem data dos primórdios do processamento de linguagem natural com sistemas como BASEBALL (Green Jr et al., 1961) que respondia a questões através de uma base de dados estruturada a questões de basebol. Existem dois paradigmas bastante comuns para *Knowledge-based QA*. Um dos paradigmas é baseado em grafos, *graphed-based QA*, com as entidades representadas por nós e as relações como as arestas do grafo. O segundo paradigma é *QA by semantic parsing*, usando métodos de *parsing* semânticos. Sendo que ambos os métodos requerem *entity linking*.

Uma alternativa para QA é realizar *queries* a um modelo de uma linguagem pré-treinada, forçando o modelo a responder a uma questão apenas apresente nos parâmetros do modelo. Apesar de exemplos

tal como o modelo da linguagem T5 (Roberts et al., 2020), figura 3.8, para esta alternativa continua a não ser uma alternativa com uma solução completa para QA, pois sofre de fraca interpretabilidade, não conseguindo fornecer aos utilizadores mais contexto informando de qual passagem a resposta foi obtida.

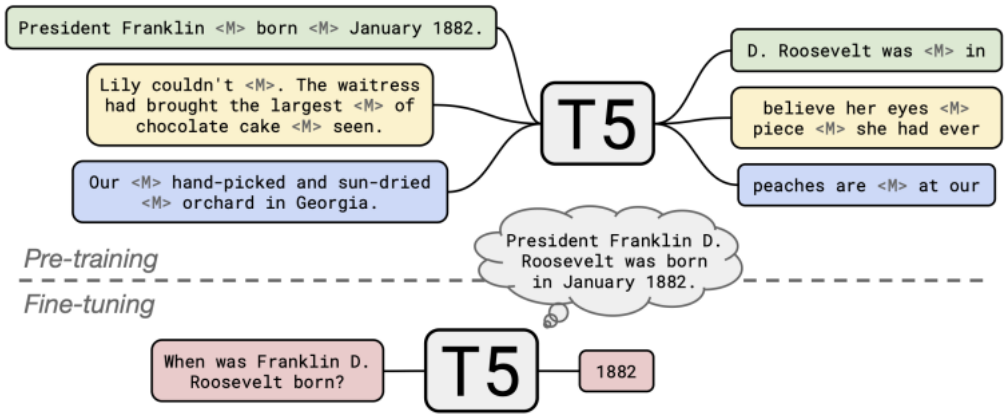


Figura 3.8: O sistema T5 tem uma arquitetura de encoder-decoder (Jurafsky e Martin, 2000)

Apesar de arquiteturas baseadas em modelos de *machine learning* serem o estado de arte para soluções de QA, como o Bert, os modelos clássicos de QA usavam regras e classificadores baseados em características o que para domínios específicos continuam a ser bastante eficientes. O exemplo de um modelo clássico está presente na figura .

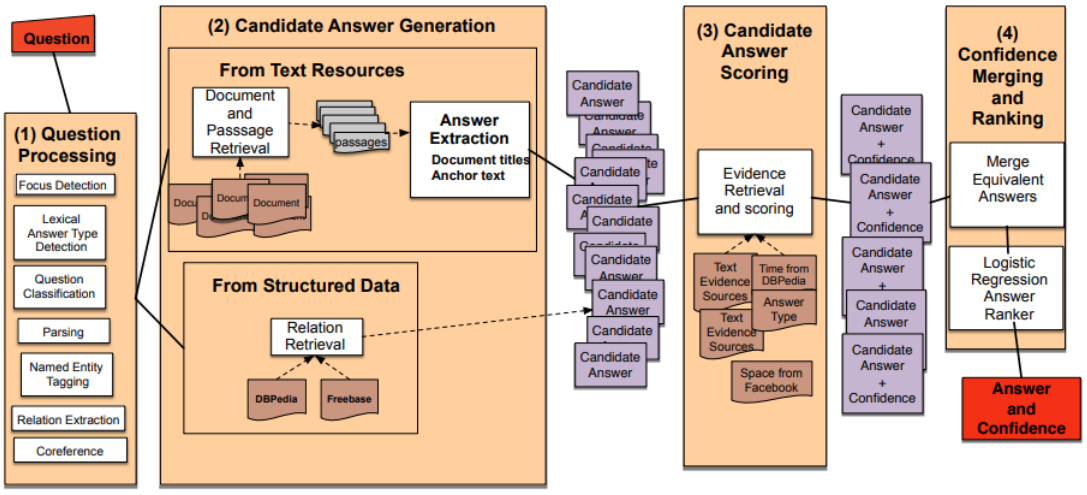


Figura 3.9: The 4 broad stages of Watson QA: (1) Question Processing, (2) Candidate Answer Generation, (3) Candidate Answer Scoring, and (4) Answer Merging and Confidence Scoring. (Jurafsky e Martin, 2000)

Um módulo de QA é essencial para um *chatbot* de forma a responder às perguntas do utilizador através de diversas fontes de dados como descrito anteriormente. Este módulo deve dado uma pergunta e um texto para responder a essa questão responder com elevada precisão, tal como representado na figura 3.10.

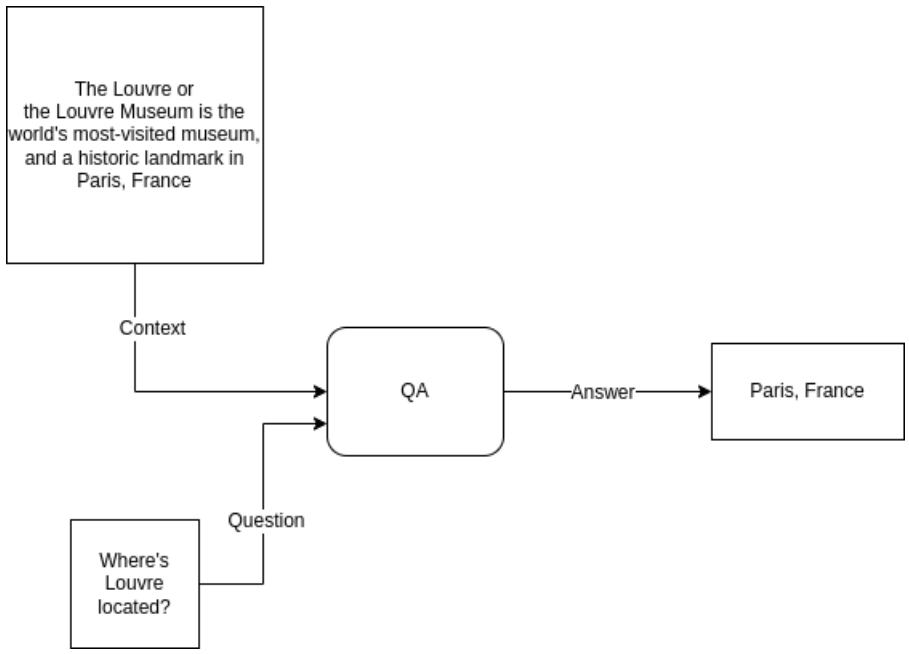


Figura 3.10: Arquitetura do módulo de QA

3.1.2.3 Pedidos a API do ChatbotWizard

O módulo de pedidos a API do ChatbotWizard é um módulo desenvolvido especificamente para o ChatbotWizard. A adição deste módulo como uma funcionalidade pretendida para o ChatbotWizard prende-se com o facto de permitir aceder a outros serviços disponíveis pela internet, serviços esses que contenham fontes de dados para o ChatbotWizard aceder de forma a responder a questões do utilizador. No entanto, este módulo não só permite aceder a outras fontes de dados através do verbo HTTP GET, mas também permite escrever, editar e eliminar recursos de outras API com o respetivo uso dos verbos POST, PUT ou PATCH e DELETE consoante o utilizar desejar integrar o módulo no seu *chatbot* personalizado. Este módulo deve também permitir definir os *query parameters* e o *body* se o *endpoint* da API necessitar. A figura 3.11 representa o comportamento esperado deste módulo.

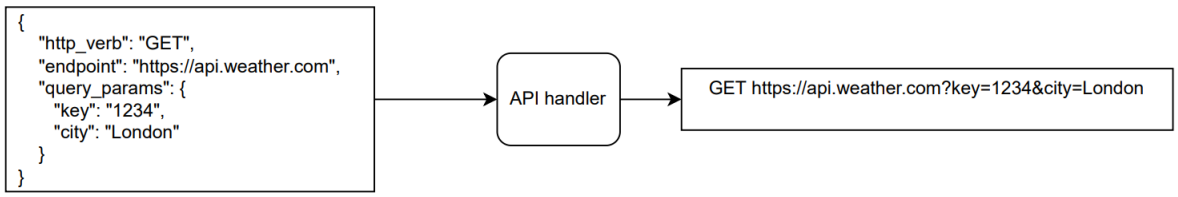


Figura 3.11: Arquitetura do módulo para chamadas a API extrenas

3.1.2.4 Template para construir texto a partir de JSON do ChatbotWizard

Assim como o módulo de API Request, o módulo de *template* para construir texto a partir de JSON foi pensado exclusivamente para o ChatbotWizard. Este módulo tem como objetivo permitir converter JSON em texto, por exemplo quando se usa o módulo API Request para obter dados de outros serviços

e esses dados vêm em formato JSON, o formato mais comum de resposta de API, conhecendo-se a especificação do *endpoint* do serviço é possível converter esses dados JSON para texto com um *template* para a conversão.

A figura 3.12 representa o comportamento esperado deste módulo.

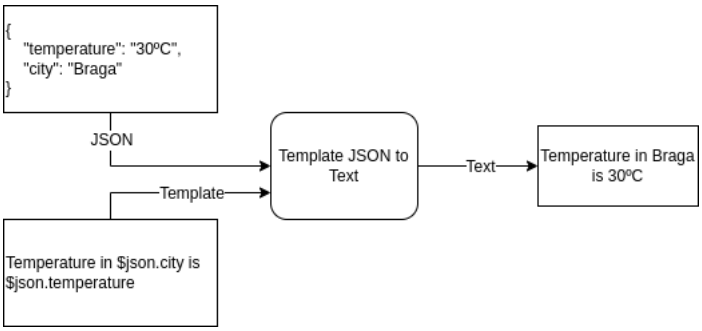


Figura 3.12: Arquitetura do módulo de template para construir texto a partir de JSON

3.2 ChatbotWizard na Web

Um objetivo chave do ChatbotWizard é ser capaz de orquestrar *chatbots* na web. A aplicação web do ChatbotWizard deve permitir a qualquer utilizador criar, editar e visualizar *chatbots* assim como editar, visualizar os respetivos módulos usados para auxiliar a criação de um chatbot. Pretende-se que a aplicação seja de fácil uso. Como o que se pretende é manipular *chatbots* e módulos, a aplicação deve ter uma barra lateral que permita aceder tantos aos módulos como aos chatbots. Pretende-se que quando o utilizador clicar na opção *chatbots* lhe mostre a lista de *chatbots* existentes, sendo que esta página também será a página inicial quando um utilizador acede à aplicação de modo a facilitar o desenvolvimento da aplicação *web*. Na listagem dos *chatbots* deve ser possível visualizar a configuração do *chatbot* ou editar. Além disso a página também deve ter um botão que permita o utilizador criar um novo *chatbot* sendo reecaminhado para outra página. Ao clicar na opção de módulos na página inicial deve aceder a uma página com a listagem dos módulos e com possibilidades similares permitidas na página dos chatbots.

A pagina inicial da aplicação web do ChatbotWizard deve permitir visualizar a lista de *chatbots* existentes com a informação do nome do chatbot, o seu criador e respetivas ações, como por exemplo visualizar e editar. Esta página inicial deve também permitir criar novos chatbots. O mockup da página inicial da aplicação web está presente na figura 3.13, e tal como visível da mesma, do lado esquerdo temos uma barra lateral em que o utilizador pode aceder ao *chatbots* e aos módulos.

Quando o utilizador clica em criar um novo *chatbot* deve-se apresentar uma página idêntica à do mockup da figura 3.14. Tal como é demonstrado na figura o utilizador deve ter a possibilidade de configurar integrações com redes sociais, neste caso para simplicidade do mockup manteve-se apenas o Telegram, possíveis *intents* a que o Chatbot deve responder e o flow do chatbot. O *flow* do *chatbot* é o fluxo entre os diversos componentes que o *chatbot* deve seguir. Quando o utilizador escolhe um módulo para adicionar ao *flow* do *chatbot* deve ser possível configurar o mesmo.

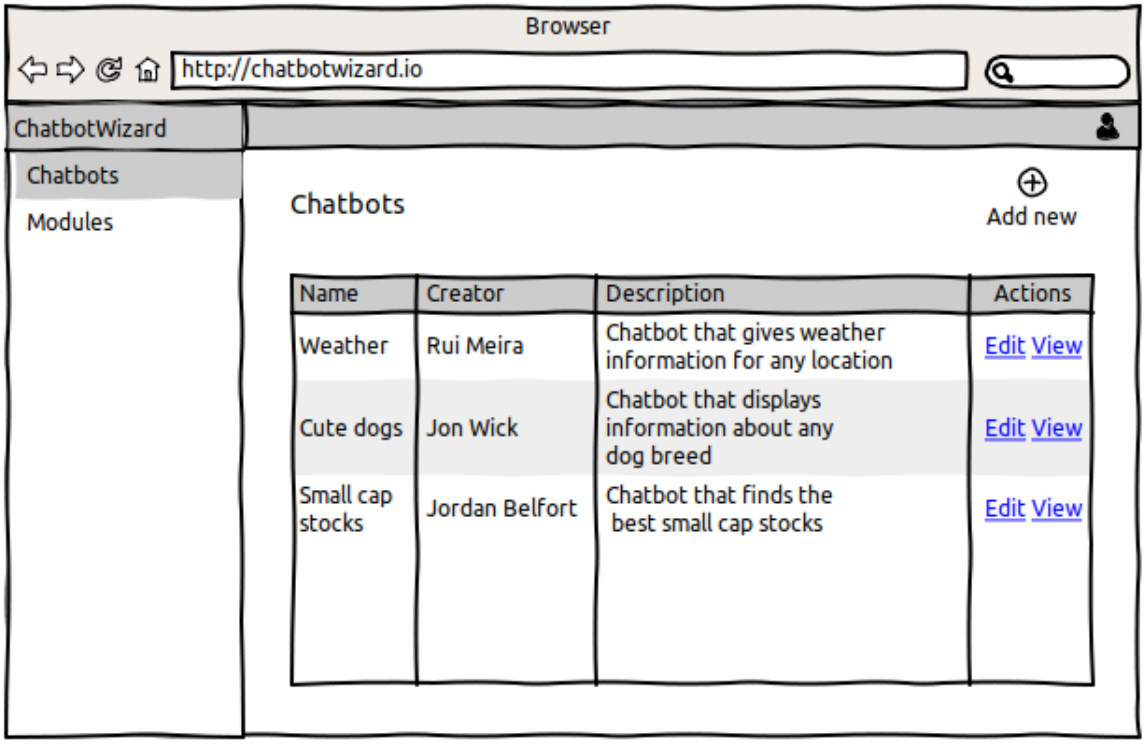


Figura 3.13: Mockup da página com a lista de chatbots

A figura 3.15 representa como o utilizador pode configurar o módulo de extração de entidades. O campo *input* permite o utilizador definir o texto para extrair a entidade. No campo *entity* deve-se definir o nome da entidade a extrair

A figura 3.16 representa o mockup da configuração do módulo API request. O campo *HTTP verb* deixa o utilizador definir o método HTTP: GET, PUT, POST, PATCH e DELETE. O campo *endpoint* permite o utilizador definir o URL da API do serviço extreno. O campo *Query params* permite o utilizador definir *query* parâmetros necessários. No campo *body* pode-se definir o body se necessário.

A figura 3.17 representa o mockup da configuração do módulo de *template* para construir texto a partir de JSON. Tal como se pode ver pela figura é possível definir o JSON e o *template* para converter o JSON em texto.

Por fim a figura 3.18 representa o mockup da configuração do módulo de QA. Neste módulo o utilizador pode definir a questão e o texto para extrair a resposta nas propriedades *Question* e *Text*.

A página com a lista de módulos presente na figura 3.19 é idêntica à pagina com a lista de chatbots. Esta página apresenta o nome, criador e descrição do módulo, assim como as ações que se podem realizar sobre o mesmo. De salientar que módulos com o criador ChatbotWizard advém da própria implementação do *backend* e não são possíveis de editar.

Para se criar um novo módulo é necessário escrever o código do mesmo e as respetivas dependências das bibliotecas. Também é necessário descrever os parâmetros de entrada além dos óbvios parâmetros do nome e descrição do módulo. O mockup para a criação de um novo módulo está presente na figura 3.20.

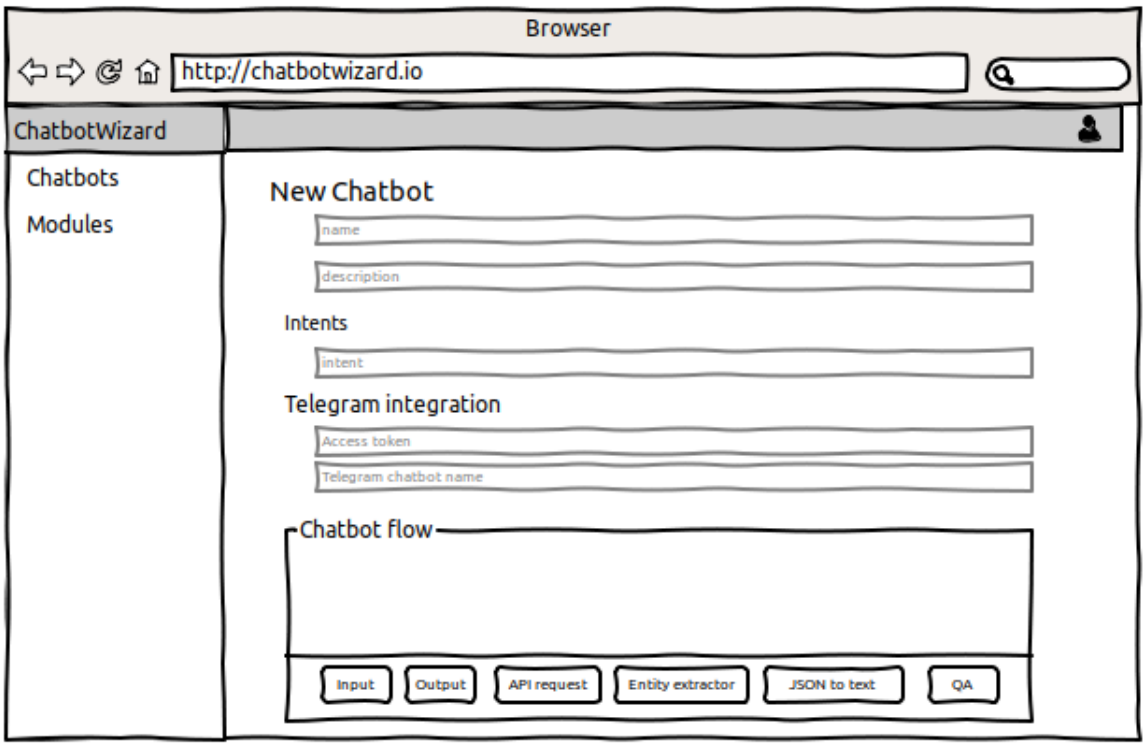


Figura 3.14: Mockup da página que permite a criação de um novo chatbot

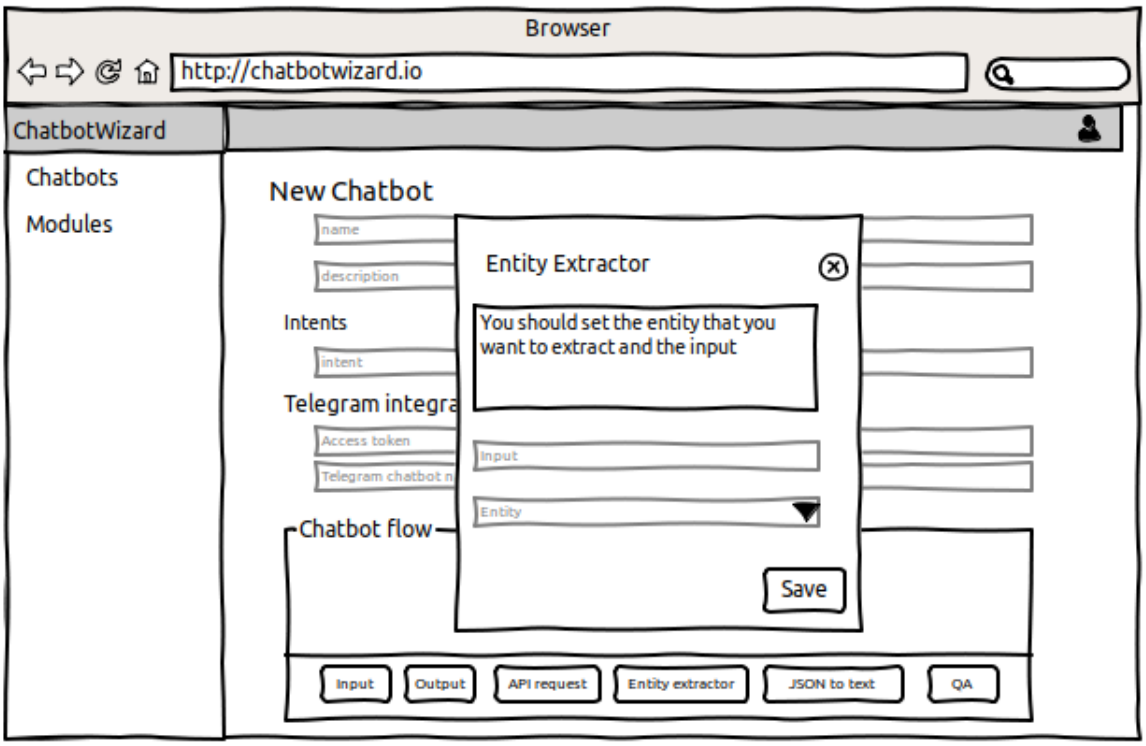


Figura 3.15: Mockup para configurar o módulo de extração de entidades

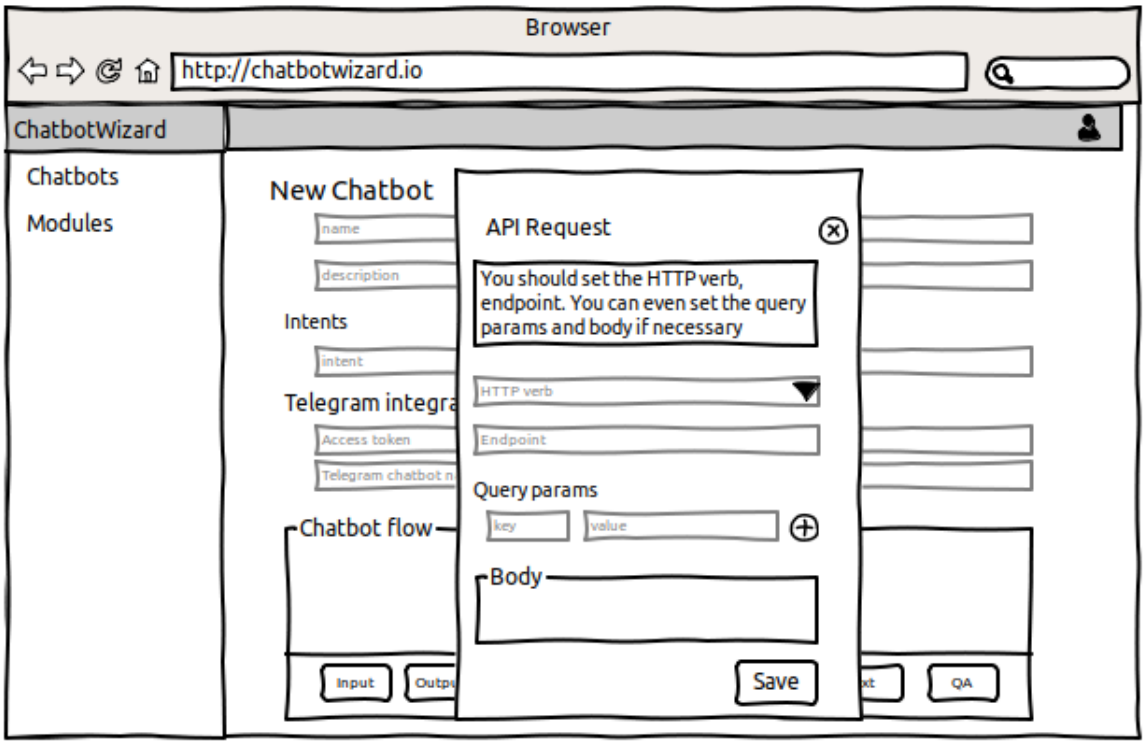


Figura 3.16: Mockup para configurar o módulo API request

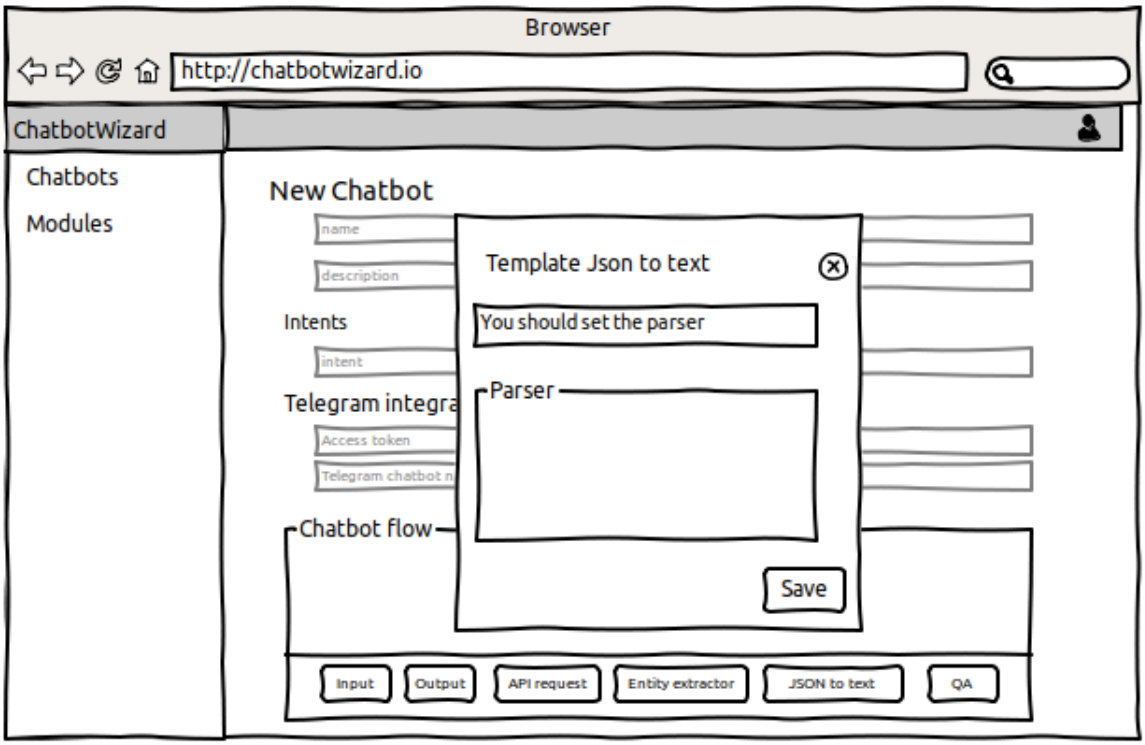


Figura 3.17: Mockup para configurar o módulo de template para construir texto a partir de JSON

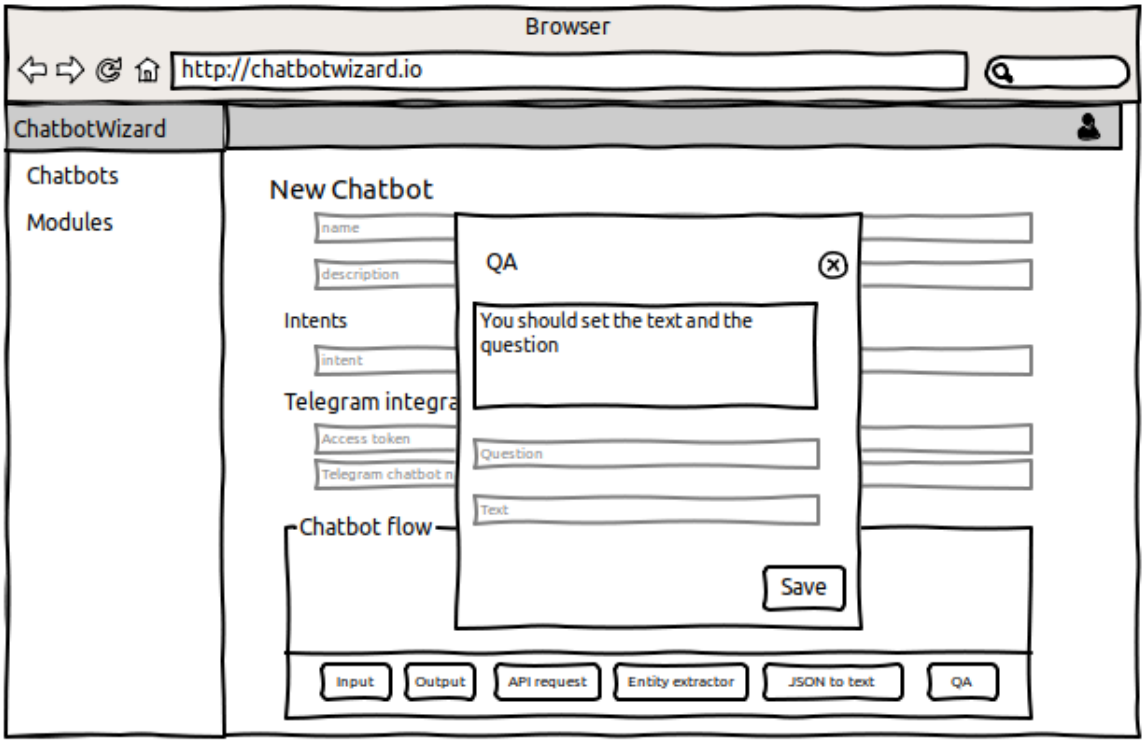


Figura 3.18: Mockup para configurar o módulo de QA

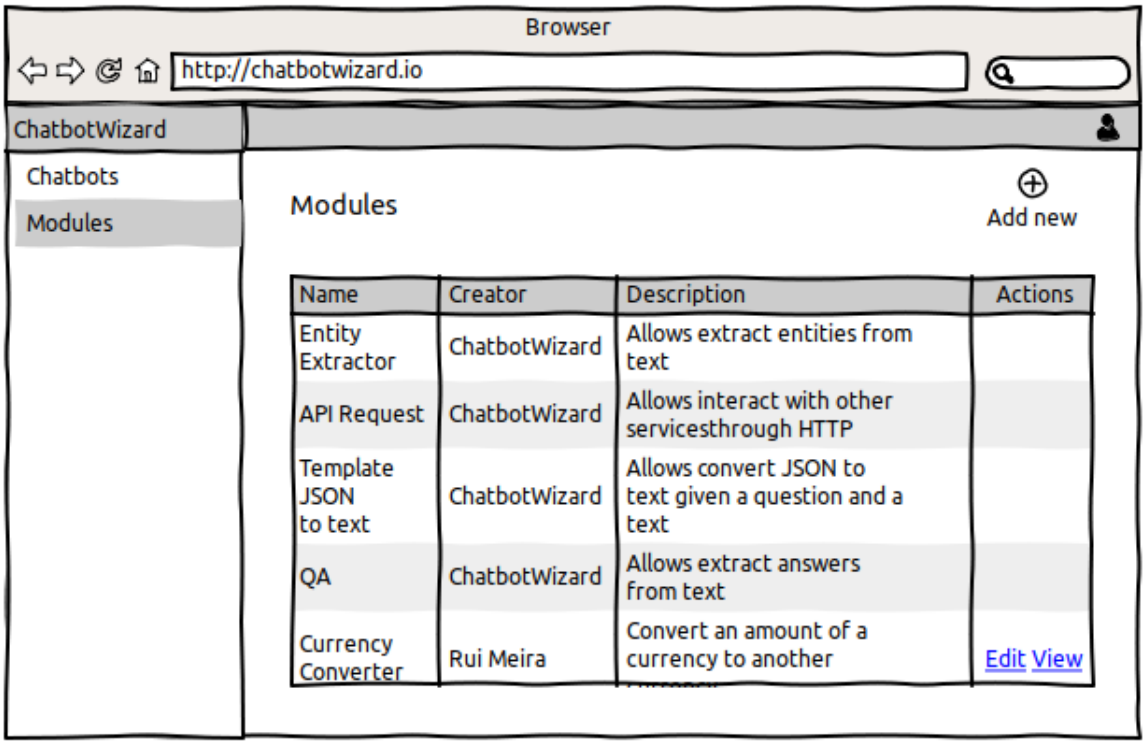


Figura 3.19: Mockup da página com a lista de módulos

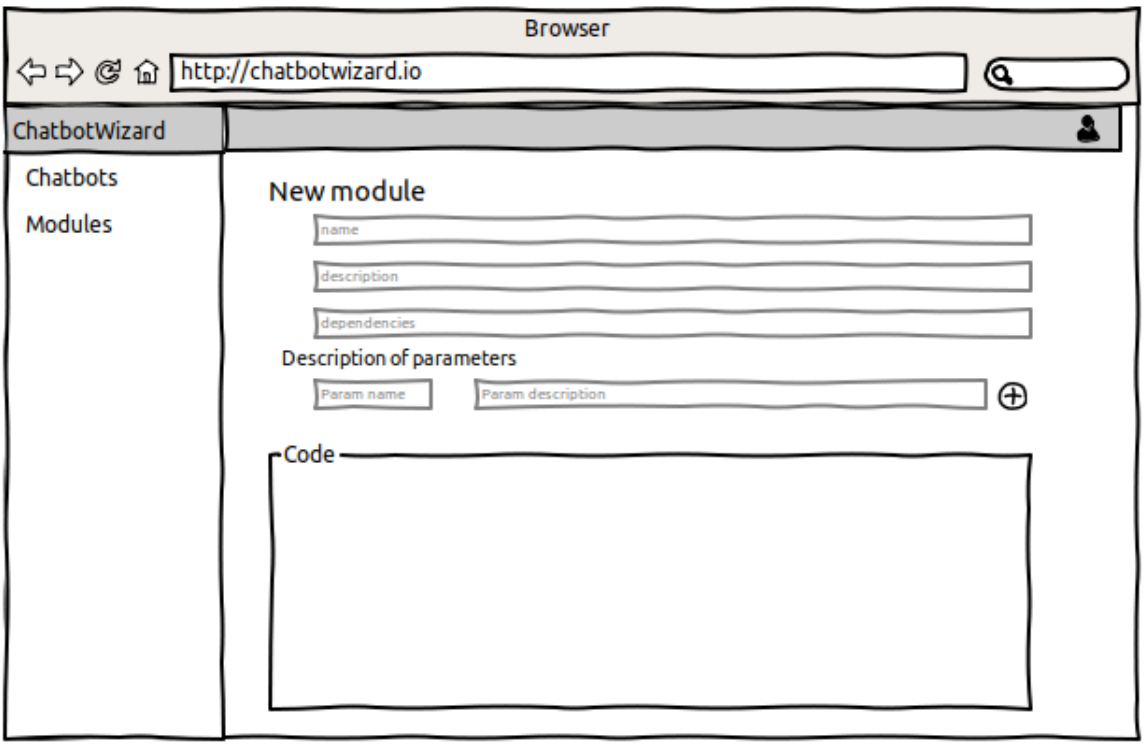


Figura 3.20: Mockup da página que permite a criação de um novo módulo

Capítulo 4

Implementação

O desenvolvimento do ChatbotWizard é separado em dois blocos: o *backend* e o ChatbotWizard web, tal como proposto no capitulo anterior.

O *backend* é responsável pela estrutura lógica do chatbot. O *backend* do ChatbotWizard assegura a criação de novos chatbots assim como a sua consulta. Esta camada permite a criação de novos chatbots com uso do Rasa como sistema de dialogo e a integração dos módulos no fluxo do chatbot.

A aplicação *web* do ChatbotWizard é de fácil interação para o utilizador e permite criar novos chatbots. A aplicação *web* comunica com o *backend* para criar novos chatbots.

A arquitetura da implementação está presente na figura 4.1. Como se pode ver por esta figura, o utilizador interage com o ChatbotWizard através da aplicação web e a mesma comunica com o *backend* para a criação de novos chatbots. De salientar que alguém pode criar a sua aplicação *web* e fazer apenas uso do *backend* do ChatbotWizard devido ao *backend* ser um web service exposto na internet.

4.1 Backend do ChatbotWizard

O *backend* do ChatbotWizard contém toda a lógica de negócio para a criação de chatbots. Para o desenvolvimento do *backend* escolheu-se usar Python, pois é um das linguagens de programação que tem mais integrações de bibliotecas de *machine learning*, é a única linguagem que dá para fazer uso do Rasa e por último é a linguagem em que me sinto mais à vontade e em que trabalho profissionalmente.

Do *backend* fazem parte três componentes:

- Rasa - responsável pela gestão de dialogo
- Módulos do ChatbotWizard - servem para ser adicionados ao fluxo do chatbot
- API - valida os dados enviados pelo ChatbotWizard web

O principal propósito do Rasa é servir como sistema de dialogo. Além disso o Rasa permite mais funcionalidades de que fizemos uso, tais como:

- Integrar com redes sociais
- Criar ações dinâmicas, ou seja, permite inserir lógica para *intents* à nossa escolha com blocos de código. As ações dinâmicas permitem integrar os módulos.
- Criar o fluxo que o agente virtual deve seguir para um tema particular

Os módulos permitem integrar as funcionalidades de extração de entidades, pedidos a *web services* expostos na internet, *template* para construir texto a partir de JSON e QA no chatbot com parâmetros de entrada que as mesmas necessitam e a conexão entre os diversos módulos. Na figura 4.1 os módulos são representados no interior do Rasa, pois o Rasa vai-lhes dar vida, integrando-os em ações dinâmicas de que iremos falar mais à frente.

A API tem como objetivo receber os dados para a criação do chatbot, validar-los e invocar uma função que crie o novo chatbot.

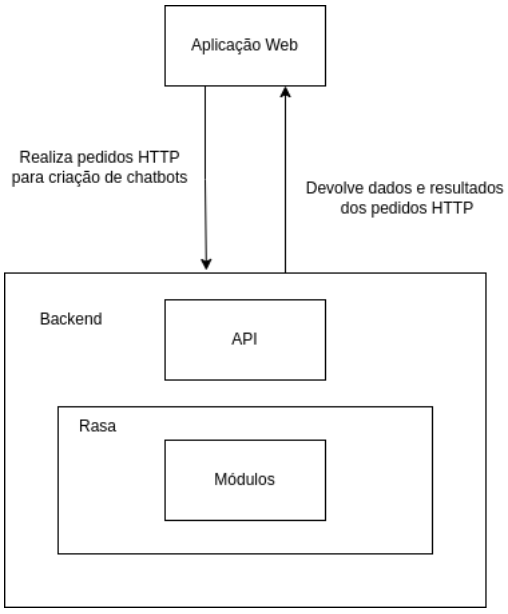


Figura 4.1: Arquitetura da implementação do ChatbotWizard

4.1.1 Módulos do ChatbotWizard

O objetivo desta subsecção é apresentar as tecnologias que permitem o desenvolvimento dos quatro módulos: módulo de extração de entidades, módulo para pedidos a API, módulo de *template* para construir texto a partir de JSON e o módulo de *Question and Answering*. Os módulos permitem ao utilizador criar um chatbot definindo os parâmetros de cada módulo que queira usar assim como a ligação entre os mesmos.

4.1.1.1 Extrator de entidades do ChatbotWizard

A extração de entidades, NER, é um tópico de NLP que consiste reconhecer entidades num texto ou áudio. Para o propósito de NLP, uma entidade é essencialmente uma palavra ou termo que define um individuo, grupo de indivíduos ou um objeto reconhecido como uma organização. Em python existem duas alternativas principais para extração de entidades: spaCy[1] e Natural Language Toolkit (NLTK)[2]. É possível optar por uma terceira alternativa de fazer pedidos ao *web service thetextapi*[3], sem ter que instalar pacotes externos ou modelos de *machine learning*. Apesar de este *web service* devolver resultados mais precisos do que as duas bibliotecas mencionadas ele não é *open source*, e por isso foi excluído para a implementação do extrator de entidades. As duas bibliotecas, spaCy e NLTK, são *open source* embora que NLTK foi desenvolvido com o propósito de investigação cientifica e SpaCy para se fazer uso em produção. A tabela abaixo apresenta o nome das entidades mais comuns e respetivas descrições que ambas as bibliotecas podem extrair.

Nome da entidade	Descrição
PERSON	Uma pessoa, usualmente reconhecida pelo seu primeiro e/ou último nome
NORP	Nacionalidades ou grupos religiosos/políticos
ORG	O nome de uma organização
GPE	O nome de uma entidade geopolítica
LOC	Uma localização
PRODUCT	O nome de uma produto
EVENT	O nome de um evento
WORK OF ART	O nome de um trabalho artístico
LAW	Uma lei que tenha sido publicada exclusivamente aos EUA
LANGUAGE	O nome de uma língua
DATE	Uma data, não precisa de ser uma data precisa, pode ser uma data relativa tal como “a day ago”
TIME	Um tempo, não precisa de ser uma tempo preciso, pode ser uma tempo relativa tal como “middle of the day”
PERCENT	Uma percentagem
MONEY	Uma quantidade de dinheiro, por exemplo “100 euros”
QUANTITY	Medições de distância ou peso
CARDINAL	Um número, similar a QUANTITY mas que não seja uma medida
ORDINAL	Um número com significado sobre a posição tal como “first”

A biblioteca spaCy pode ser instalada com o comando `pip install spacy`, e como a mesma precisa de um modelo para ser treinada, devemos fazer o download do modelo `en_code_web_sm` (modelo de treino

[1] Ver em: https://spacy.io/
[2] Ver em: https://www.nltk.org/
[3] Ver em: https://www.thetextapi.com/

com textos da internet) com o comando `python -m spacy download en_core_web_sm`. Por exemplo para se obter todas as entidades que são cidades tem-se o exemplo da listagem 4.1. Na primeira linha da listagem importamos a biblioteca, na linha 3 criamos a instância do extrator com o modelo `en_code_web_sm`, na linha 5 temos o texto que queremos extrair as entidades e na linha 7 temos o nome da entidade a extrair, neste caso é "GPE" porque queremos extrair entidades geográficas. De seguida na linha 9 identificamos todas as entidades presentes no texto e na linha 11 filtramos por todas as entidades que são geográficas. Por último na linha 13 imprimimos o resultado que neste caso é `["Braga"]`.

Listagem 4.1: Código que permite extrair todas as entidades que são cidades usando SpaCy

```
1  import spacy
2
3  nlp = spacy.load("en_core_web_sm")
4
5  text = "Braga é uma cidade no norte de Portugal com 193333 habitantes"
6
7  entity_name = "GPE"
8
9  doc = nlp(text)
10
11 entities = [e.text for e in doc.ents if e.label_ == entity_name]
12
13 print(entities)
```

A biblioteca NLTK pode ser instalada com o comando `pip install nltk`. Esta biblioteca depende depende de dados externos para funcionar, e é ideal iniciar uma consola python com `python` e de seguida executar as linhas de código da listagem 4.2, sendo que *Punkt* são os dados do *tokenizer* que reconhece pontuação, *Averaged Perceptron Tagger* cria *tags* do discurso, *Maxen NE Chunker* define as entidades no texto e *words* é o corpus de texto.

Listagem 4.2: Instalação de extensões necessárias à biblioteca NLTK

```
1 import nltk
2 nltk.download("punkt")
3 nltk.download("averaged_perceptron_tagger")
4 nltk.download("maxent_ne_chunker")
5 nltk.download("words")
```

A listagem 4.3 mostra um exemplo de extrair entidades com NLTK. Na primeira linha importamos a biblioteca necessária, na terceira linha e quinta linha definimos o texto e o nome da entidade a extrair. De seguida nas linhas 7, 8 e 9 definimos o *tokenizer* para reconhecer a pontuação do texto, com o texto com *tokens* de pontuação definidos processa-se a sequência de palavras e atribui-se uma *tags* a cada palavra com `nltk.pos_tag(tokenized)` e por último atribui-se nomes de entidades a cada palavra ou sequência de palavras se as mesmas forem de facto entidades.

Listagem 4.3: Código que permite extrair todas as entidades que são cidades usando SpaCy

```
import nltk

text = ”Braga é uma cidade no norte de Portugal com 193333 habitantes”

entity_name = ”GPE”

tokenized = nltk.word_tokenize(text)
pos_tagged = nltk.pos_tag(tokenized)
chunks = nltk.ne_chunk(pos_tagged)
for chunk in chunks:
   if hasattr(chunk, 'label'):
      print(chunk)
```

Apesar de ambas as bibliotecas realizarem o propósito deste método, escolheu-se spaCy devido a necessitar de menos dependências e menor manutenção de bibliotecas necessárias para extrair entidades do que NLTK e também porque spaCy foi desenvolvida para se usar em produção enquanto que NLTK foi desenvolvida para fins de investigação académica.

4.1.1.2 Pedidos a API do ChatbotWizard

Há uma grande quantidade de informação espalhada pela internet. Muitos *web services*, como YouTube e Github, disponibilizam os seus dados a terceiros através de uma *Application Programming Interface* (API). Atualmente a forma mais usada para construir API é através do protocolo e arquitetura REST. REST significa *representational state transfer* e é uma arquitetura de software que define um padrão para comunicação entre cliente e servidor através da internet. REST define um conjunto de restrições para a arquitetura de software de forma a promover o desempenho, escalabilidade, simplicidade e segurança do sistema. REST define as seguintes restrições de arquitetura:

- *Stateless* - o servidor não deve manter um estado entre pedidos do cliente
- Cliente e servidor - O cliente e o servidor são unidades separas, permitindo cada um se desenvolver independentemente
- Sistema em camadas - O Cliente pode aceder aos recursos do servidor indiretamente através de outras camadas como uma *proxy* ou um *load balancer*
- *Code on demande* - Apesar de esta ser uma restrição opcional, o servidor pode transferir para o cliente código executável, tal como JavaScript para uma aplicação de *single-page*

Um REST *web service* é um serviço *web* que subscreve as restrições da arquitetura REST. Estes serviços web expõem os seus dados na internet através de uma API. As REST API disponibilizam acesso a um *web service* através de URLs públicos, como por exemplo:

```
https://api.github.com/users/<username>
```

que permite aceder à informação sobre um utilizador especifico do github. As REST API respondem a métodos/verbos HTTP tais como GET, POST e DELETE que identificam qual o tipo de operação deve ser realizada sobre os recursos de um *web service*. Um recurso é qualquer informação ou dados disponíveis no serviço web que podem ser acedidos e manipulados através de pedidos HTTP à REST API. A tabela 4.1 informa sobre os possíveis métodos HTTP e respetiva descrição.

Método HTTP	Descrição
GET	Devolve um recurso existente
POST	Cria um novo recurso
PUT	Edita um recurso
PATCH	Edita um recurso parcialmente
DELETE	Elimina um recurso

Tabela 4.1: Métodos HTTP e respetiva descrição

Após a REST API receber e processar o pedido HTTP, vai devolver uma resposta HTTP. Nesta resposta está incluído um HTTP status *code*. Este código devolve informação sobre o resultado do pedido. Uma aplicação pode obter informação sobre um pedido que enviou à REST API e através do resultado da mesma realizar a operação ideal. Esta ações incluem tratar erros ou imprimir a informação se o pedido foi realizado com sucesso. Os HTTP status *code* são numerados com base na categoria do resultado tal como presente na tabela 4.2. Estes status *code* são essenciais para aplicar diferente lógica consoante o seu resultado.

Código HTTP	Descrição
2xx	Operação bem sucedida
3xx	Redirecionado
4xx	Erro do cliente
5xx	erro do servidor

Tabela 4.2: Códigos HTTP e respetiva descrição

Em python existem várias bibliotecas para se realizar pedidos a REST API tais como:

- urllib3 - https://pypi.org/project/urllib3
- Requests - https://pypi.org/project/requests
- aiohttp - https://pypi.org/project/aiohttp
- GRequests - https://pypi.org/project/grequests
- HTTPX - https://pypi.org/project/httpx
- Uplink - https://pypi.org/project/uplink

A biblioteca requests é a que tem mais downloads, com um total de 236 milhões de downloads mensais[4], sendo que a biblioteca da lista acima com o segundo maior número de downloads é aiohttp com 74 milhões de downloads mensais[5]. Dado que requests é a biblioteca desta lista que mais usei e também a mais usada na lista acima foi a escolhida para o módulo de API request. Para se instalar esta biblioteca usa-se o comando `pip install requests` e pode-se fazer um simples pedido a uma API e imprimir o resultado tal como na listagem 4.4. Na primeira linha importa-se a biblioteca requests. Na segunda linha define-se o URL, sendo este um URL de uma API que disponibiliza *endpoints* falsos e devolve respostas que podem ser processadas. Na terceira linha realizamos o pedido GET e na quarta linha imprime-se o resultado em formato JSON que é `{'userId': 1, 'id': 1, 'title': 'delectus aut autem', 'completed': False}`

Listagem 4.4: Código que faz um pedido HTTP a uma API usando a biblioteca requests

```
1 import requests
2 api_url = "https://jsonplaceholder.typicode.com/todos/1"
3 response = requests.get(api_url)
4 print(response.json())
```

4.1.1.3 Template para construir texto a partir de JSON do ChatbotWizard

O módulo de *template* para construir texto a partir de JSON, dado um JSON como da listagem 4.5 e um *template* igual a `$name is a $profession that studied at $university and he's favorite hobby is $hobby` deve converter o JSON para o texto. A implementação deste módulo está presente na listagem 4.6. Na primeira linha importa-se a biblioteca `re` para fazer manipulações do texto com expressões regulares. Na linha 3 define-se o exemplo de um JSON. Na linha 10 está presente o template em que cada palavra com o prefixo `$` deve ser substituída pelo valor presente no JSON das linhas 3 a 7. A função da linha 13 vai substituir pelo valor do `json_example` para cada palavra prefixada com `$`. Por fim na linha 18 é aplicada a substituição com a função `convert_match_obj_to_json_properties` para a pattern `r"$([^\s]+)` no texto da variavel template e imprime "Rui Meira is a Software Engineer that studied at Universidade do Minho and he's favorite hobby is Surf" tal como esperado.

Listagem 4.5: Exemplo de JSON

```
1 {
2    'name': 'Rui Meira',
3    'profession': 'Software Engineer',
4    'university': 'Universidade do Minho',
5    'hobby': 'Surf'
6 }
```

Listagem 4.6: Código que converte JSON em texto

[4]Datado a 17 de Novembro de 2022 com informação obtida através da página https://pepy.tech/project/requests
[5]Datado a 17 de Novembro de 2022 com informação obtida através da página https://pepy.tech/project/aiohttp

```
import re

json_example = {
   'name': 'Rui Meira',
   'profession': 'Software Engineer',
   'university': 'Universidade do Minho',
   'hobby': 'Surf'
}

template = f"$name is a $profession that studied at $university and he's favorite hobby is
    ↪ $hobby"

def convert_match_obj_to_json_properties(match_obj):
   json_keyword = match_obj.group(1)
   return json_example[json_keyword]

print(re.sub(r"$([^\s]+)", convert_match_obj_to_json_properties, template))
```

4.1.1.4 Question and Answering do ChatbotWizard

Na implementação do módulo de QA decidiu-se usar o estado de arte da área que é Bert, tal como referido ao longo da dissertação. Para se fazer uso dos transformers recorreu-se ao HuggingFace usando um BERT com um modelo já treinado. Neste caso escolheu-se o modelo *BERT large model (uncased) whole word masking finetuned on SQuAD (Stanford Question Answering Dataset)*. Para se fazer uso da biblioteca do HuggingFace de transformers faz-se uso do comando `pip install transformers`. Um exemplo do uso de BERT e resposta a questões presentes num texto está presente na listagem 4.7. Na primeira linha importa-se o *tokenizer* e a função que vai aplicar QA ao modelo. Na linha 3 e 6 são aplicados o *tokenizer* e a operação de QA ao modelo *BERT large model (uncased) whole word masking finetuned on SQuAD*. Na linha 10 e 11 estão definidos a questão e o texto a aplicar o BERT.

Listagem 4.7: Código que implementa a funcionalidade de QA

```
from transformers import AutoTokenizer, AutoModelForQuestionAnswering

tokenizer = AutoTokenizer.from_pretrained(
   "bert-large-uncased-whole-word-masking-finetuned-squad"
)
model = AutoModelForQuestionAnswering.from_pretrained(
   "bert-large-uncased-whole-word-masking-finetuned-squad"
)
inputs = tokenizer.encode_plus(
   "Where's the University of Minho",
   "The University of Minho is located in Braga",
   add_special_tokens=True,
```

```
13      return_tensors="pt"
14  )
15  input_ids = inputs["input_ids"].tolist()[0]
16  answer_start_scores, answer_end_scores = model(**inputs, return_dict=False)
17  answer_start = torch.argmax(answer_start_scores)
18  # Get the most likely beginning of answer with the argmax of the score
19  answer_end = torch.argmax(answer_end_scores) + 1
20  return tokenizer.convert_tokens_to_string(
21      tokenizer.convert_ids_to_tokens(input_ids[answer_start:answer_end]
22  )
```

4.1.2 Rasa

Escolheu-se o Rasa como sistema de gestão de diálogo para o ChatbotWizard das opções discutidas ao longo da dissertação, pois é a única alternativa que possibilita a criação de ações dinâmicas (necessário para integrar os módulos e a sua interconexão), ser *open source* não necessitando de licenças, usa-se como uma biblioteca e permite integrar com redes sociais.

4.1.2.1 Arquitetura do Rasa

É possível instalar Rasa com o comando `pip3 install Rasa`, após a instalação pode-se fazer a inicialização da estrutura do chatbot Rasa com o comando `Rasa init`. Após se inicializar tem-se a estrutura do projeto Rasa da figura 4.2 que nos permite usar como base para gerar o chatbot dinamicamente. Da estrutura que nos é presente tem que se configurar os seguintes ficheiros:

- `actions/actions.py`
- `data/nlu.yml`
- `domain.yml`
- `stories.yml`
- `credentials.yml`

O ficheiro `data/nlu.yml` que é responsável por definir o nome de `intents` e respetivos exemplos tal como na listagem 4.8. Na primeira linha da listagem está definida a versão `yaml` do ficheiro. Na linha 3 é definida a lista de *intents* a que o chatbot deve reconhecer. Na linha 4 está definido o *intent* `greet` e nas linhas seguintes (5 a 8) exemplos para este *intent*. Na linha 10 está definido o *intent* `goodbye` e nas linhas seguintes (11 a 14) exemplos para este *intent*. Ao definirmos um *intent* e os respetivos exemplos permite ao Rasa quando instanciado treinar este *intent* com os exemplos e permitir realizar ações consequentes quando o *intent* é identificado.

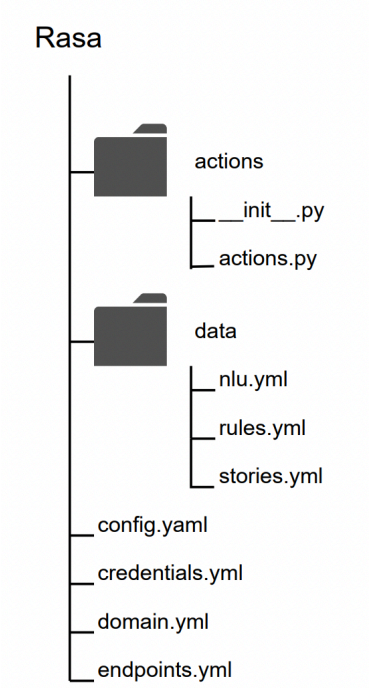


Figura 4.2: Estrutura do projeto Rasa

Listagem 4.8: Exemplo de configuração do ficheiro nlu.yml

```
nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi

- intent: goodbye
  examples: |
    - good night
    - see you later
    - bye bye
```

O domínio especifica o universo no qual o chatbot opera. Ele especifica os *intents*, entidades, *slots*, repostas, *forms* e ações que o chatbot deve reconhecer e o domínio é configurado no ficheiro `domain.yaml`.

A chave `intents` lista os intents presentes no ficheiro `data/nlu.yml`

As secção das entidades (`entities`) lista todas as entidades que podem ser extraídas por um extrator de entidades. Os *slots* são a memória do chatbot, agem como *key-value* que pode guardar informação providenciada pelo utilizador (por exemplo a cidade natal do utilizador) ou guardar informação obtida através de um serviço externo (resultado de uma *query* a uma base de dados).

Respostas (`responses`) são ações que enviam uma mensagem ao utilizador sem executar código. As `forms` são um tipo especial de ação que têm como objetivo ajudar o assistente a obter informação do

utilizador.

As ações (`actions`) são processamentos que o bot realiza como por exemplo: responder ao utilizador, realizar uma pedido a uma API, *query* uma base de dados, etc. Dentro das *actions* temos as *custom actions* que permitem criar ações dinâmicas, as quais nos permitem integrar os módulos e sua lógica, que iremos detalhar mais à frente.

A listagem 4.9 demonstra um exemplo da configuração de `domain.yaml`. Na primeira linha está definida a versão do ficheiro yaml. Em `intents`, entre as linhas 3 e 5, estão presente os nomes dos *intents* Nas linhas 7 e 8 temos definida a ação que este chatbot deve realizar quando, neste caso é uma *custom action* denominada `action_greet` que será explicada em detalhe mais à frente.

Listagem 4.9: Exemplo da configuração do ficheiro domain.yml

```
1  intents:
2     - greet
3     - goodbye
4
5  actions:
6     - action_greet
7
8  responses:
9     utter_goodbye:
10    - text: "Bye"
```

As histórias (`stories`) configuradas no ficheiro `stories.yaml` são um tipo de dados de treino usados para treinar o modelo de gestão de dialogo. As histórias podem ser usadas para treinar modelos que são capazes de generalizar para possíveis caminhos de conversação.

Uma história é uma representação da conversa entre o utilizador e o chatbot, convertida num formato especifico em que os inputs do utilizador são expressos como *intents* (e entidades quando necessário), enquanto que as respostas do *bot* são ações ou nomes de ações.

No exemplo da listagem 4.10 temos presente apenas uma história na linha 4 em que o *intent* `greet` é o primeiro input do utilizador e a resposta do utilizador é dada através da *action* `action_greet` e de seguida o utilizador deve dar como input uma palavra ou frase que corresponda ao *intent* `goodbye` e o *bot* responde através da *action* `utter_goodbye`.

Listagem 4.10: Exemplo da configuração do ficheiro stories.yml

```
1  stories:
2  - story: greet and goodbye path
3    steps:
4    - intent: greet
5    - action: action_greet
6    - intent: goodbye
7    - action: utter_goodbye
```

Um chatbot pode ter mais que uma *storie* tal como na listagem 4.11

Listagem 4.11: Exemplo da configuração do ficheiro stories.yml

```
stories:

- story: greet path
  steps:
  - intent: greet
  - action: action_greet

- story: goodbye path
  steps:
  - intent: goodbye
  - action: utter_goodbye
```

O ficheiro de ações define tudo o que deve ser executado pelo chatbot. Todas as ações dinâmicas devem estar presentes no ficheiro `actions/action.py`. Com a ação `greet_action` listada no domínio deve-se implementar a mesma tal como no exemplo da listagem 4.12. Esta ação deve herdar de `Action` e o nome da ação é reconhecível através da propriedade `name` presente na linha 4. A lógica da ação é implementada na função `run` na linha 7. A lógica desta função é obter a mensagem do utilizador na linha 12 e se a mensagem do utilizador for uma expressão em Português, o *chatbot* responde com a mensagem "Olá amigo! Saudades de falar consigo", se não o chatbot responde com "Hello".

Listagem 4.12: Exemplo da configuração do ficheiro actions/actions.py

```
from rada_sdk import Action, Tracker

class ActionGreet(Action):

   def name(self) -> Text:
       return "action_greet"

   def run(self,
         dispatcher: CollectingDispatcher,
         tracker: Tracker,
         domain: Dict[Text, Any]
   ) -> List[Dict[Text, Any]]:
      message: str = tracker.latest_message['text']

      if is_a_portuguese_expression(message):
         message = "Olá amigo! Saudades de falar consigo"
      else:
         message = "Hello"

      dispatcher.utter_message(text=message)

      return []
```

Por último para se integrar com redes sociais deve-se configurar o ficheiro `credentials.yml`. A listagem 4.13 demonstra a configuração da integração do chatbot com o telegram disponibilizando o mesmo na aplicação de *chat*.

Listagem 4.13: Exemplo da configuração do ficheiro credentials para integrar com o Telegram

```
1 access_token: "490161424:AAGlRxinBRtKGb21_rlOEMtDFZMXBl6EC0o"
2 verify: "your_bot"
3 webhook_url: "https://your_url.com/webhooks/telegram/webhook"
```

4.1.2.2 Configuração dinâmica

Agora que já foram explicados todos os ficheiros que precisam de ser configurados de forma ao chatbot funcionar é necessário preencher as configurações desses ficheiros dado as configurações do utilizador. Além de ser necessário configurar os ficheiros também é necessário integrar os módulos que o utilizador escolheu e a sua interconexão, que deve estar presente em `actions/actions.py`.

A melhor abordagem para se preencher os ficheiros é usando um *template engine*. O *template engine* gera o código para os módulos que o utilizador pretende usar no seu chatbot. A biblioteca que se escolheu para *template engine* foi o Jinja2[6] pois é a biblioteca de *template engine* que estou mais familiarizado e também a mais usada no ecossistema python.

Jinja2 é um *template engine* com ótimo desempenho que disponibiliza *placeholders* no *template* para se escrever código com sintaxe idêntica à do python. Sendo apenas necessário passar os dados ao *template* para construir o documento final. O Jinja2 inclui:

- Herança de *templates*
- Definir e importar macros dentro dos *templates*
- Os *templates* HTML podem usar *autoescaping* para prevenir *Cross-Site Scripting* (XSS) de *input* dos utilizadores
- Ambiente isolado para renderizar *templates* não confiáveis
- Suporte para gerar *templates* assincronamente que suporta funções síncronas e assíncronas
- *Templates* são compilados para otimizar código python
- Exceções para corrigir linhas nos *templates* e facilitar o *debug*
- Filtros, testes, funções e sintaxe extensível

O Jinja2 tem como filosofia que a lógica da aplicação pertença ao python sempre que possível sem dificultar o design do *template*.

[6] Ver em: https://jinja.palletsprojects.com

Jinja2 suporta renderizar *templates* com base em ficheiros ou com base em simples strings. Dado o *template* 4.15 localizado em `templates/template.yaml`, pode-se gerar o *template* com o simples excerto 4.14, obtendo o documento renderizado 4.16. Se por outro lado se quiser renderizar o *template* com base em string pode-se gerar o *template* com o excerto do código 4.17 tendo como output o ficheiro 4.18.

Listagem 4.14: Exemplo do uso do Jinja2 a partir de um ficheiro template

```
from Jinja2 import Environment, FileSystemLoader

file_loader = FileSystemLoader('templates')
env = Environment(loader=file_loader)

template = env.get_template('template.yaml')

template.render(
   intent_name = 'greet',
   examples = ['hey', 'hello', 'hi']
)
```

Listagem 4.15: Exemplo de ficheiro de template

```
nlu:
- intent: {{ intent_name }}
  examples: |
  {% for example in examples %}
    - {{ example }}
  {% endfor %}
```

Listagem 4.16: Resultado de preencher o ficheiro de template usando o Jinja2

```
nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
```

Listagem 4.17: Exemplo do uso do Jinja2 a partir de uma string

```
from Jinja2 import Template

data = """
def multiply_a(a: int) -> int:
   return a * {{ b }}
"""

```

```
8 tm = Template(data)
9 tm.render(b=4)
```

Listagem 4.18: Resultado de preencher a string usando o Jinja2

```
1 def multiply_a(a: int) -> int:
2    return a * 4
```

Assim, é necessário definir um *template* para cada um dos ficheiros a editar: `data/nlu.yml`, `domain.yml`, `stories.yml` e `credentials.yml`.

O código para preencher o ficheiro `data/nlu.yml` está presente na listagem 4.19 e o respetivo *template* na listagem 4.20. Para preencher este ficheiro os dados a enviar são um dicionário, em que a chave é o nome do *intent* e o valor uma lista de exemplos do *intent* tal como se pode ver nas linha 6 a 9 da listagem 4.19.

Listagem 4.19: Código que preenche o ficheiro nlu.yml

```
1  from Jinja2 import Environment, FileSystemLoader
2  file_loader = FileSystemLoader('templates')
3  env = Environment(loader=file_loader)
4  template = env.get_template('template.yaml')
5
6  data = {
7     'greet': ['Hello', 'Hi'],
8     'goodbye': ['Bye', 'See you']
9  }
10
11 template.render(data=data)
```

Listagem 4.20: Template do ficheiro nlu.yml a preencher

```
1 nlu:
2 {% for name, examples in data.items() %}
3 - {{ name }}
4    {% for example in examples %}
5    - {{ example }}
6    {% endfor %}
7 {% endfor %}
```

Para preencher o ficheiro `domain.yml` com o *template* da listagem 4.22 é necessário enviar um dicionário com a lista dos nomes de *intents* e a lista de *actions* tal como o da listagem 4.21

Listagem 4.21: Código para preencher o ficheiro domain.yml

```
1 data = {
2    'intents': ['greet','goodbye']
3    'actions': ['action_greet']
4 }
```

```
template.render(data)
```

Listagem 4.22: Template do ficheiro domain.yml a preencher

```
intents:
{% for intent_name in data.intents %}
   - {{ intent_name }}
{% endfor %}

actions:
{% for action in data.actions %}
   - {{ action }}
{% endfor %}
```

No caso de configurar *stories* o ChatbotWizard de momento só aceita um *intent* e respetiva ação, assim deve-se enviar o nome do *intent*, o nome da *action* e o nome do chatbot que o utilizador definiu tal como na listagem 4.23. O *template* para preencher este ficheiro está presente na listagem 4.24

Listagem 4.23: Código para preencher o ficheiro stories.yml

```
data = {
   'story_name': 'Greet story',
   'intent_name': 'greet',
   'action': 'action_greet'
}
template.render(data)
```

Listagem 4.24: Template do ficheiro stories.yml a preencher

```
stories:
- story: {{ data.story_name }}
steps:
   - intent: {{ data.intent_name }}
   - action: {{ data.action_name }}
```

O último ficheiro que necessita de ser configurado dinamicamente é `credentials.yml` que permite integrar redes sociais. Neste caso é preciso enviar os dados necessários (listagem 4.25) à integração com o Telegram que é a integração que o ChatbotWizard suporta de momento. O template para este ficheiro está presente na listagem 4.26.

Listagem 4.25: Código para preencher o ficheiro credentials.yml

```
data = {
   access_token: '490161424:AAGlRxinBRtKGb21_rlOEMtDFZMXBl6EC0o',
   verify: 'your_bot',
   url: 'https://your_url.com/webhooks/telegram/webhook',
}
template.render(data)
```

Listagem 4.26: Template do ficheiro credentials.yml a preencher

```
access_token: {{ data.access_token }}
verify: {{ data.verify }}
webhook_url: {{ data.url }}
```

Em código cada módulo deve ser definido por:

- O nome do módulo, atributo estático que define o nome do módulo, tais como `entity_extractor` para o módulo de extração de entidades, `json_to_text` para o *template* para construir texto a partir de JSON, `api_request` para o módulo de API request e `qa` para módulo de QA.
- ID que identifica a instância do módulo e permite integrar e identificar o módulo no fluxo que o utilizador desenhou.
- Nome da função que permite invocar a instância do módulo
- Método abstrato `to_func` que deve devolver o código gerado para o módulo

De forma a assegurar que todos os módulos implementem este contrato devem herdar a classe `Module` presente na listagem 4.27. Como se pode utilizar o output de um módulo como input de outro, qualquer propriedade que o módulo necessita, por exemplo nome da entidade e texto para extrair entidades no caso do módulo de extração de entidades, pode ser obtida por `kwargs` ou definida estaticamente através da função na linha 20. Quando o valor da propriedade for apenas constituído por dígitos significa que advém de outro módulo (esses dígitos identificam o `id` do módulo que serve de input) deve-se obter o valor dos `kwargs`, se não deve-se usar o valor em si. Sendo que tal deve definir-se no *template*.

Listagem 4.27: Superclass Module que deve ser herdada por todos os módulos dinâmicos

```
import re
from typing import List

class Module:
    id: str
    name: str
    input_actions: List[str]

    @property
    def func_name(self) -> str:
        raise NotImplementedError()

    @property
    def param_name(self) -> str:
        return f"input_{self.id}"

    def to_func(self):
        raise NotImplementedError()
```

```
19
20    @staticmethod
21    def resolve_property(p: str) -> str:
22        if re.match(r"^$", p):
23            return f"kwargs.get('{p}')"
24        return p
```

O módulo de extrair entidades (listagem 4.28) deve herdar a classe `Module` que deve definir as propriedades do nome da entidade (`entity` na linha 3) e o texto para extrair essa entidade (`input` na linha 2) além das propriedades herdadas de `Module`. Para se usar este módulo e gerar-se o código para extrair uma entidade com o nome "GPE" do texto "Braga é uma cidade no norte de Portugal com 193333 habitantes", instancia-se a classe `EntityExtractor` na primeira linha e invoca-se o método `to_func` na quinta linha da listagem 4.29 e obtém-se o código gerado presente na listagem 4.30.

Listagem 4.28: Módulo que gera o código para extrair entidades

```
1  class EntityExtractor(Module):
2     input: str
3     entity: str
4     name: "entity_extractor"
5
6     def to_func(self):
7        self.__replace_input()
8        tm = Template("""
9     def {{ func_name }}(self, **kwargs):
10       print(f"extracting {{ entity_type }} for the input { {{ input }} }")
11       doc = nlp( {{ input }} )
12       entities = [x.text for x in doc.ents if x.label_ == "{{ entity_type }}" ]
13       if not entities:
14          raise Exception("Not able to extract the entity {{ entity_type }}")
15       else:
16          return entities[0]
17       """)
18       func = tm.render(
19          func_name=self.func_name,
20          entity_type=self.resolve_property(self.entity),
21          input=self.resolve_property(self.input)
22       )
23       return func
24
25    def __replace_input(self):
26       if len(self.input_actions):
27          for input_action in self.input_actions:
28             self.input = re.sub(f'\${input_action}', f"kwargs.get('input_{input_action}')",
                   ↪ self.input)
```

Listagem 4.29: Código que executa EntityExtractor

```
entity_extractor = EntityExtractor(
   input="Braga é uma cidade no norte de Portugal com 193333 habitantes",
   entity="GPE"
)
entity_extractor.to_func()
```

Listagem 4.30: Código gerado do EntityExtractor

```
def entity_extractor(self, **kwargs):
   print(f"extracting GPE for the input Braga é uma cidade no norte de Portugal com 193333
       ↪ habitantes")
   doc = nlp("Braga é uma cidade no norte de Portugal com 193333 habitantes")
   entities = [x.text for x in doc.ents if x.label_ == "GPE" ]
   if not entities:
      raise Exception("Not able to extract the entity GPE")
   else:
      return entities[0]
```

O módulo de template JSON para texto além de herdar as propriedades de `Module` deve definir o json presente na segunda linha da listagem 4.31 e o *template* na terceira linha. Pode-se usar este módulo para gerar o código da listagem 4.33 instanciando o módulo `TemplateJsonText` com a propriedade `template` com o valor `$name is a $profession that studied at $university` e a propriedade `json` com o valor `{"name": "Rui Meira", "profession": "Software Engineer", "university": "Universidade do Minho"}` tal como na listagem 4.35.

Listagem 4.31: Módulo que gera o código para o template para construir texto a partir de JSON

```
class TemplateJsonText(Module):
   json: dict
   template: str
   name = "template_json_text"

   def to_func(self):
      parser = self.__parse_input_action()

      tm = Template("""
   def {{ func_name }}(self, **kwargs):
      return f"{{ parser }}"
         """)
      func = tm.render(
         func_name=self.func_name,
         json=self.resolve_property(json),
         template=self.resolve_property(template)
      )
      return func
```

```
19
20    def __parse_input_action(self):
21       string = parse_string_action(self.parser)
22       print("string before", string)
23       return re.sub(
24          pattern="(kwargs\.get\('input_\d+'\))\.([^\s]+)",
25          repl=self.__replace_func,
26          string=string
27       )
28
29    def __replace_func(self, match_obj):
30       object_accessor = ''.join([
31          f"['{k}']" for k in match_obj.group(2).split('.')
32       ])
33       return "{" + f"{match_obj.group(1)}{object_accessor}" + "}"
```

Listagem 4.32: Código que executa TemplateJsonText

```
1 template = TemplateJsonText(
2    json = {
3       "name": "Rui Meira",
4       "profession": "Software Engineer",
5       "university": "Universidade do Minho"
6    },
7    template="$name is a $profession that studied at $university"
8 )
9 template.to_func()
```

Listagem 4.33: Código gerado do TemplateJsonText

```
1 def template_json_text(self, **kwargs):
2    return "Rui Meira is Software Engineer that studied at Universidade do Minho"
```

A implementação do módulo de API request define as propriedades `endpoint` e `query_parameters` nas linhas 2 e 3 da listagem 4.34. Ao usar este módulo com as propriedades `endpoint` e `query_parameters` respetivamente com os valores `https://jsonplaceholder.typicode.com/todos` e `{"page": 1, "per_page" ↪ : 15 }` tal como na listagem 4.35 obtém-se o código gerado da listagem 4.36.

Listagem 4.34: Módulo que gera o código para o módulo API Request

```
1 class ApiRequest(Module):
2    endpoint: str
3    query_parameters: dict
4    name: "api_request"
5
6    def to_func(self):
7       endpoint = parse_string_action(self.endpoint)
```

```
8        query_params_str = self.__parse_query_params()
9
10       if query_params_str:
11          endpoint += f'?{query_params_str}'
12
13       tm = Template("""
14    def {{ func_name }}(self, **kwargs):
15       return requests.get(f"{{ endpoint }}").json()
16       """)
17       func = tm.render(func_name=self.func_name, endpoint=endpoint)
18       return func
19
20    def __parse_query_params(self):
21       parsed_query_parameters = {
22          parse_string_action(key): parse_string_action(value)
23          for key, value in self.queryParameters.items()
24       }
25       return "&".join([
26          f"{'{' + key + '}' if key.startswith('kwargs') else key}={'{' + value + '}' if value.
             ↪ startswith('kwargs') else value}"
27          for key, value
28          in parsed_query_parameters.items()
29       ])
```

Listagem 4.35: Código que executa ApiRequest e gera o código

```
1  api_request = ApiRequest(
2     endpoint="https://jsonplaceholder.typicode.com/todo",
3     query_params={
4        "page": 1,
5        "per_page": 15
6     }
7  )
8  api_request.to_func()
```

Listagem 4.36: Código gerado do módulo API Request

```
1  def api_request(self, **kwargs):
2     return requests.get(
3        "https://jsonplaceholder.typicode.com/todo?page=1&per_page=15"
4     ).json()
```

A implementação do módulo de QA exige que o utilizador defina as propriedades `question` e `text` presentes nas linhas 2 e 3 da listagem 4.37. Definindo as propriedades `question` e `text` com os valores "The University of Minho is located in Braga" e "Where's the University of Minho", pode-se usar o módulo `QA` tal como na listagem 4.38 gerando o código 4.39.

Listagem 4.37: Módulo que gera o código para o módulo de QA

```
class QuestionAnswer(Module):
   question: str
   text: str
   name: "question_answer"


   def to_func(self):
      tm = Template("""
def {{ func_name }}(self, **kwargs):
   tokenizer = AutoTokenizer.from_pretrained(
      "bert-large-uncased-whole-word-masking-finetuned-squad"
   )
   model = AutoModelForQuestionAnswering.from_pretrained(
      "bert-large-uncased-whole-word-masking-finetuned-squad"
   )
   inputs = tokenizer.encode_plus(
      {{ question }},
      {{ text }},
      add_special_tokens=True,
      return_tensors="pt"
   )
   input_ids = inputs["input_ids"].tolist()[0]
   answer_start_scores, answer_end_scores = model(**inputs, return_dict=False)
   answer_start = torch.argmax(answer_start_scores)
   # Get the most likely beginning of answer with the argmax of the score
   answer_end = torch.argmax(answer_end_scores) + 1
   return tokenizer.convert_tokens_to_string(
      tokenizer.convert_ids_to_tokens(input_ids[answer_start:answer_end])
   )
      """)
      func = tm.render(
         question=self.resolve_property(self.question),
         text=self.resolve_property(self.text)
      )
      return func
```

Listagem 4.38: Código que executa QuestionAnswer

```
qa = QuestionAnswer(
   text="The University of Minho is located in Braga"
   question="Where's the University of Minho"
)
qa.to_func()
```

Listagem 4.39: Código gerado do módulo de QA

```
tokenizer = AutoTokenizer.from_pretrained(
   ”bert-large-uncased-whole-word-masking-finetuned-squad”
)
model = AutoModelForQuestionAnswering.from_pretrained(
   ”bert-large-uncased-whole-word-masking-finetuned-squad”
)
inputs = tokenizer.encode_plus(
   ”Where's the University of Minho”,
   ”The University of Minho is located in Braga”,
   add_special_tokens=True,
   return_tensors=”pt”
)
input_ids = inputs[”input_ids”].tolist()[0]
answer_start_scores, answer_end_scores = model(**inputs, return_dict=False)
answer_start = torch.argmax(answer_start_scores)
# Get the most likely beginning of answer with the argmax of the score
answer_end = torch.argmax(answer_end_scores) + 1
return tokenizer.convert_tokens_to_string(
   tokenizer.convert_ids_to_tokens(input_ids[answer_start:answer_end]
)
```

4.1.2.3 Execução

Assim que um chatbot é criado e todos os ficheiros foram devidamente preenchidos é necessário executar alguns comandos para ser possível interagir com o mesmo.

Inicialmente é preciso treinar o modelo usado os dados dos ficheiros `data/nlu.yml` e `data/stories.yml` com o comando `Rasa train`.

Após o modelo estar treinado pode-se executar o servidor do modelo com o comando `Rasa run`. Como queremos integrar o chatbot com o Telegram devemos adicionar a *flag* `--enable-api`, que expõe o chatbot a poder ser invocado como uma API. No entanto, executando `Rasa run --enable-api` localmente o Telegram ou qualquer serviço externo não consegue chegar ao IP, pois é um IP local. Desta forma, recorreu-se ao **ngrok**[7] para expor o nosso chatbot à internet através do comando `ngrok http 5005`.

As ações dinâmicas funcionam como um servido à parte, sendo que é necessário executar o comando `Rasa run actions`, permitindo o servidor do modelo consiga invocar as ações.

4.1.3 API

O *backend* do CahtbotWizard é um *web service* que segue o protocolo REST. A API deve receber os dados de clientes, validar e processar os mesmos. Neste caso, a API do ChatbotWizard deve receber dados da aplicação web para criar novos chatbots.

[7] Ferramenta de linha de comandos que permite expor serviços locais à Internet (https://ngrok.com/)

Dado que o ecossistema que estamos a usar no *backend* do ChatbotWizard é Python, temos três possíveis *frameworks* que podemos fazer uso para criar a API: Django, Flask e FastAPI.

Django[8] é uma *web framework* de alto nível criada por Adrian Holovaty e Simon Willison em 2003. Django segue o o padrão *Model-View-Template* (MVT), foi desenvolvida com o propósito de responder a requisitos complexos que equipas de desenvolvimento enfrentavam na altura. A ideia era permitir desenvolvimento menos complexo. Os autores queriam permitir aos programadores criar a aplicações web rapidamente e lança-las em horas. As vantagens do Django são:

- Permite desenvolvimento rápido
- Bom desempenho
- Estrutura de código eficiente
- Diversas integrações
- *Toolkit* para desenvolver aplicações web - Django Rest Framework
- Segura, ajuda os programadores a evitar SQL *injections*, *cross-site request forgery*, *cross-site scripting* e *clickjacking*
- Escalável

As desvantagens do Django são:

- Demasiados módulos reutilizáveis
- Curva de aprendizagem acentuada
- Sem convenções
- Não é ideal para projetos pequenos

Flask[9] foi desenvolvido por Armin Ronacher, que tinha anteriormente desenvolvido duas outras soluções, Werkzeug e Jinja2, e decidiu juntar as duas soluções que deu origem ao Flask. Flask é categorizado como uma *microframework* porque não depende de bibliotecas externas para desempenhar tarefas de uma *framework*. Flask tem as próprias ferramentas, bibliotecas e tecnologias que ajuda a suportar todas as funcionalidades do desenvolvimento. As vantagens do Flask são:

- Flexibilidade
- Escalabilidade
- Compacto

[8]Ver em: https://www.djangoproject.com
[9]Ver em: https://flask.palletsprojects.com/

- Promove a experimentação
- Ideal para projetos pequenos
- Pequena curva de aprendizagem

As desvantagens do Flask são:

- Algumas falhas de segurança como não ter proteção de *cross-site request forgery*.
- Não é indicado para projetos grandes
- Pequena comunidade de suporte
- Não tem muitas ferramentas

Fast API[10] é um *framework* web com alto desempenho para construir API com Python3.6+ baseado no padrão de *type hints*. FastAPI é das 3 a *framework* mais rápida. Foi construida para otimizar a experiência de desenvolvimento, permitindo que uma equipa possa desenvolver código simples e construir API efetivas. FastAPI suporta código assíncrono e tem uma excelente documentação. As vantagens da FastAPI são:

- A mais rápida das 3 na execução de código
- Suporta código assíncrono
- Baseada em padrões
- Documentação automática.

As desvantagens da FastAPI são:

- Pequena comunidade

Dado que a nossa API é de um projeto pequeno e permitir executar código assíncrono beneficia a nossa API escolheu-se a FastAPI para a *framework* da API.

4.2 Aplicação Web

Uma aplicação web é uma aplicação que é executada no browser, ao contrário de aplicações que correm nativamente num sistema operativo. As aplicações web são distribuídas na web para os utilizadores através da conexão à internet sem necessidade de instalar a aplicação no seu dispositivo.

Para ajudar no desenvolvimento da aplicação web do ChatbotWizard é útil usar uma *framework* ou biblioteca que permita acelerar o seu desenvolvimento. Apesar da existência de imensas *frameworks* e bibliotecas que permitem acelerar o desenvolvimento a nossa escolha restringiu-se a escolher entre três: Angular, React e Vue.

[10]Ver em: https://fastapi.tiangolo.com

Angular

Angular[11] foi desenvolvido em 2009 pela Google. A primeira versão foi Angular.JS. Todos os projetos significativos da Google foram desenvolvidos com Angular. Angular é uma *framework Model-View-Controller* (MVC). Angular é ideal para desenvolver uma aplicação complexa e multifuncional. A maior desvantagem de Angular é que usa o DOM normal, assim, toda a estrutura da árvore de HTML é atualizada o que tem um impacto no tempo de carregamento. As vantagens de Angular são:

- Permite arquitetura MVC
- Boa capacidade de manutenção
- Permite gerir uma arquitetura de *microfrontends*
- Disponibiliza uma *framework* simples para desenvolver aplicações web e sem a necessidade de bibliotecas adicionais
- Fácil de realizar testes unitários e testes *end-to-end*

As desvantagens do Angular são:

- Atualiza toda a estrutura HTML
- Tempo de carregamento demorado
- Devido a ser uma *framework*, é relativamente inflexível

React

React[12] foi lançado em Março de 2013 pelo Facebook como uma biblioteca de JavaScript. React não é apropriado para desenvolver com foco numa arquitetura MVC. React é ideal para desenvolver aplicações modernas num curto espaço de tempo, soluções mobile para desenvolvimento web, aplicações *cross-platform*, aplicações mobile *single-page* e adicionar características a uma aplicação existente. A principal vantagem do React é usar um DOM virtual que apenas compara diferenças com o código HTML anterior e assim só atualiza as novas partes. Esta vantagem tem um grande impacto nos tempos de carregamento. As vantagens do React são:

- Rápido a carregar novos dados
- Um ficheiro contem linguagem de *markup* e a lógica (JSX)
- Permite a separação de dados e apresentação

As desvantagens do React são:

[11] Ver em: https://angular.io
[12] Ver em: https://reactjs.org

- É apenas uma biblioteca JavaScript e não uma *framework*
- Não é possível implementar a arquitetura MVC
- Frequentemente é insuficiente para o desenvolvimento de uma aplicação web, necessitando de outras bibliotecas

Vue.js

Vue.js[13] é uma *framework* baseada em JavaScript para criar aplicações *single-page*. Foi criada com a escalabilidade em mente, assim como com a facilidade de integrar com outras *views* baseadas noutras *frameworks*. O uso de Vue.js é ideal para aplicações *single-page*, aplicações dinâmicas e que necessitam de um elevado desempenho. As vantagens do Vue são:

- Um conjunto de ferramentas e bibliotecas
- Flexibilidade e simplicidade na utilização
- Documentação completa

As desvantagens do Veu são:

- Comunidade limitada em relação a Angular e React
- Número de *plugins* limitados
- Demasiado complicação no que diz respeito à flexibilidade

Dado que se quis desenvolver a aplicação web do ChatbotWizard moderna e num curto espaço de tempo, escolheu-se React. Além de se usar React como a base para a nossa aplicação web, adicionou-se a *framework* de React denominada NextJS. Tal como referido anteriormente, React é uma biblioteca de JavaScript que permite construir interface gráficas interativas. Por interface gráfica quer-se dizer o que os utilizadores vêm e interagem no ecrã, tal como na figura 4.3. Parte do sucesso de React é que não define uma opinião sobre outros aspetos de construir a aplicação web. Isto resultou num ecossistema de ferramentas e soluções de *third-party*. React disponibiliza um conjunto de ferramentas e funções para construir a interface gráfica, mas deixa o programador escolher como deseja utilizar essas ferramentas e funções.

NextJS

O NextJS[14] é uma framework de React que nos dá blocos de construção para criar as aplicações web. Por framework, quer-se dizer que NextJS trata das ferramentas e configurações precisas para o react,

[13] Ver em: https://vuejs.org
[14] Ver em: https://nextjs.org

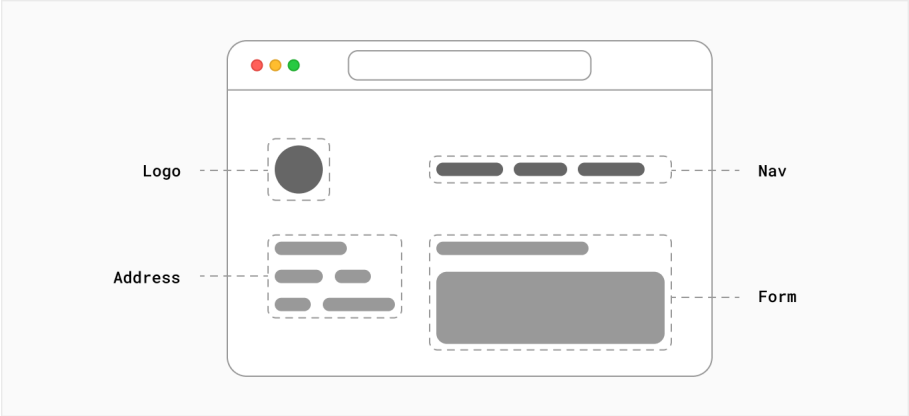


Figura 4.3: Funcionalidades e funções do react

e disponibiliza estrutura adicional, funcionalidades adicionais e otimizações para a aplicação, tal como na figura 4.4. O NextJS permite usar React para construir a interface gráfica, e então incrementalmente adicionar funcionalidades desta *framework* para resolver requisitos comuns duma aplicação como *routing*, obter dados e integrações, melhorando a experiência do utilizador e facilidade em desenvolver a aplicação para o programador.

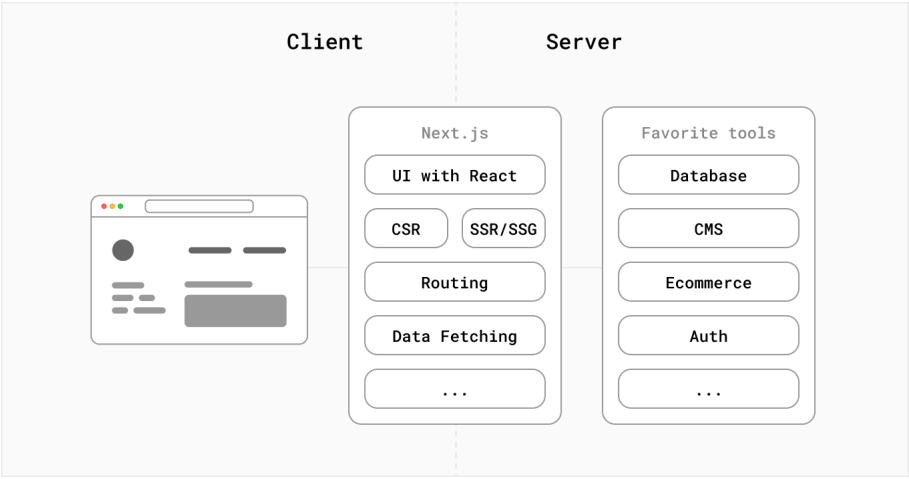


Figura 4.4: Funcionalidades e funções do NextJS

Após se escolher React e utilizar NextJS como *framework* para o desenvolvimento da aplicação web, foi necessário adicionar uma biblioteca para permitir o utilizador desenhar o fluxo do chatbot.

A nossa ideia foi ter nós e ligações entre os mesmos, em que um nó representa a instância de um módulo. O objetivo passou por imaginando que temos duas instâncias de módulos, A e B, estando eles conectados em que a direção é de A para B, significa que B pode usar o output de A nos seus parâmetros.

A biblioteca escolhida para representar o fluxo do chatbot é React Flow. React Flow é uma biblioteca para construir aplicações baseadas em nós, estes podem ser simples diagramas estáticos ou fluxos complexos. Nas figuras 4.5 e 4.6 demonstram exemplos do que se pode desenvolver com o auxílio desta aplicação.

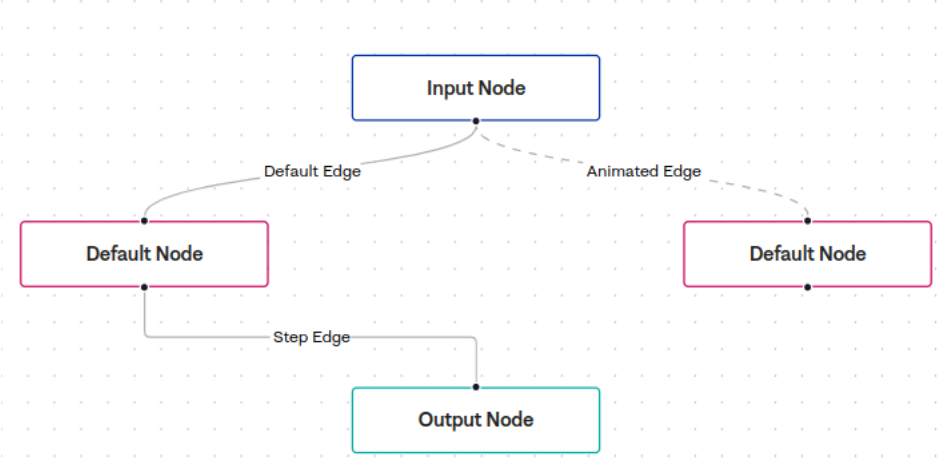


Figura 4.5: Exemplo de fluxo em React Flow (1)

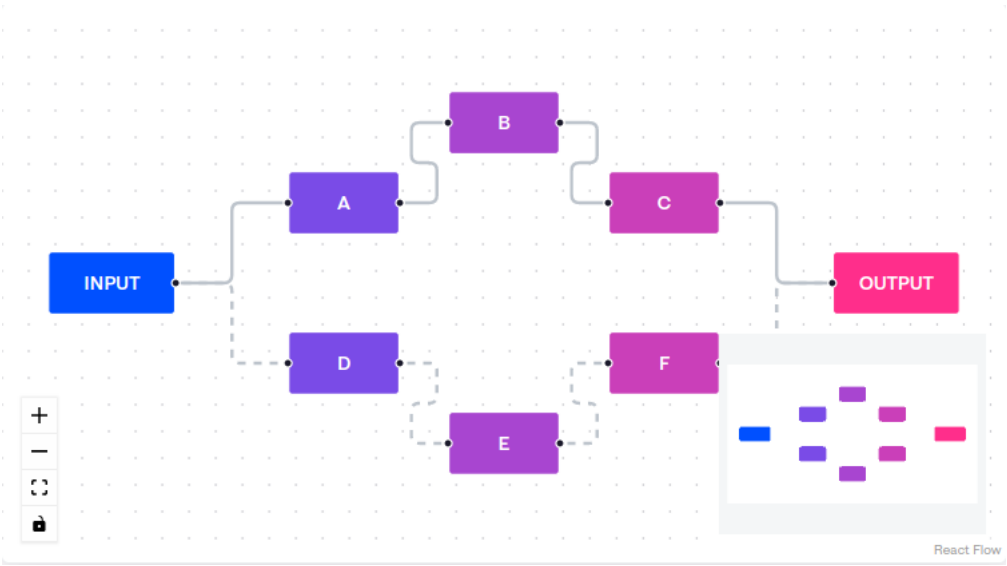


Figura 4.6: Exemplo de fluxo em React Flow (2)

Neste capítulo descreveram-se os detalhes de implementação do ChatbotWizard. Sendo que no capítulo seguinte será demonstrado a criação de um *chatbot* através do ChatbotWizard. Além disso também será apresentado como fica cada ficheiro configurado automaticamente através da definição do chatbot no ChatbotWizard web.

Capítulo 5

Caso de Estudo

O propósito deste capitulo é demonstrar a utilização do ChatbotWizard. Para isso definiu-se o caso de uso como sendo criar um chatbot para responder a questões meteorológicas de uma localização.

O fluxo deste chatbot deve seguir as seguintes etapas:

1. Extrair a entidade geográfica que identifique a localização da questão do utilizador. Para tal usa-se o módulo de extração de entidades em que as entidades a identificar são "países, cidades, estados e distritos".

2. Para se obter dados meteorológicos será usada a API `http://api.weatherapi.com/v1/forecast.json` que permite obter dados sobre temperatura, velocidade do vento, humidade, precipitação, entre outros. Com este objetivo deve-se usar o módulo de pedidos a API em que o parâmetro de *query* "q" é o output da etapa anterior que identifica a localização.

3. Após se ter os dados meteorológicos da localização é necessário usar o módulo de *template* para construir texto a partir da resposta do serviço web.

4. Por último, tendo toda a informação meteorológica da localização em texto através da etapa anterior, usa-se o módulo de QA para dada a questão do utilizador, responder com o texto que contém informações meteorológicas da localização.

Para se realizar as etapas anteriores acedemos à aplicação web do ChatbotWizard presente na figura 5.1 e fica preenchido tal como na figura 5.7.

Em que o fluxo do chatbot para o caso de uso descrito anteriormente está presente na figura 5.2.

De notar que no ChatbotWizard web cada instância de um módulo é identificada por um código numérico gerado pela aplicação precedido pelo carácter "$", como por exemplo "$9999". Foi desenvolvido desta forma para facilitar a lógica de integrar os diversos módulos no *backend* do ChatbotWizard.

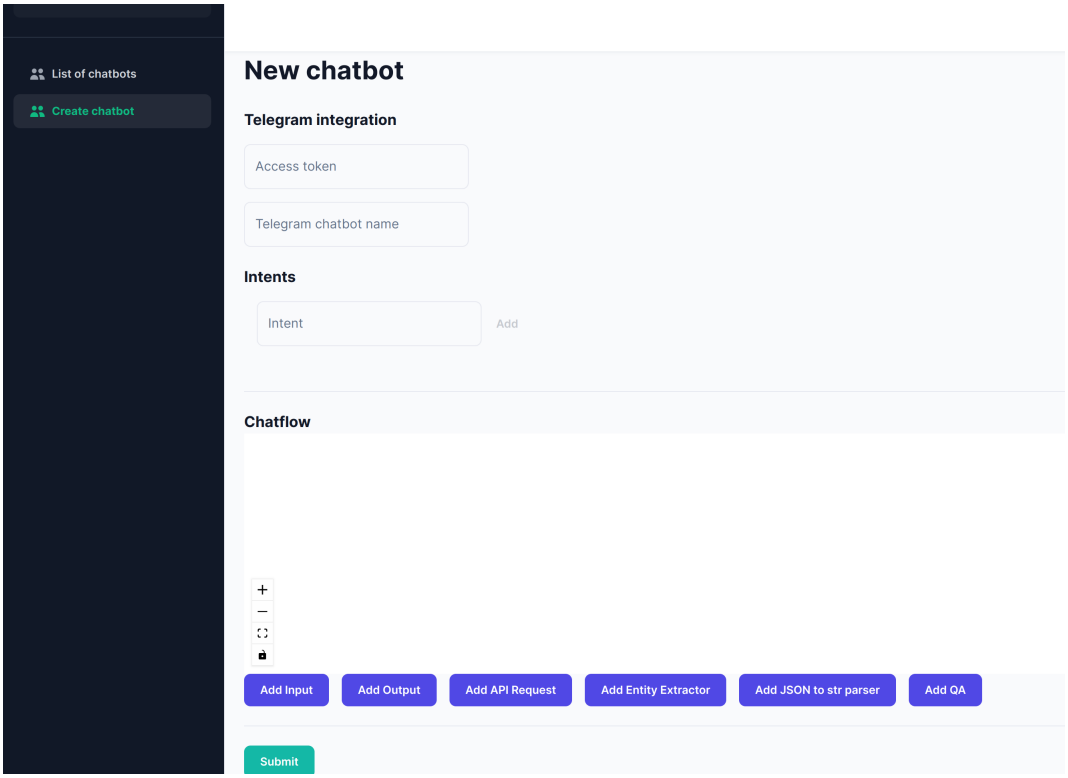


Figura 5.1: Página inicial da aplicação Web

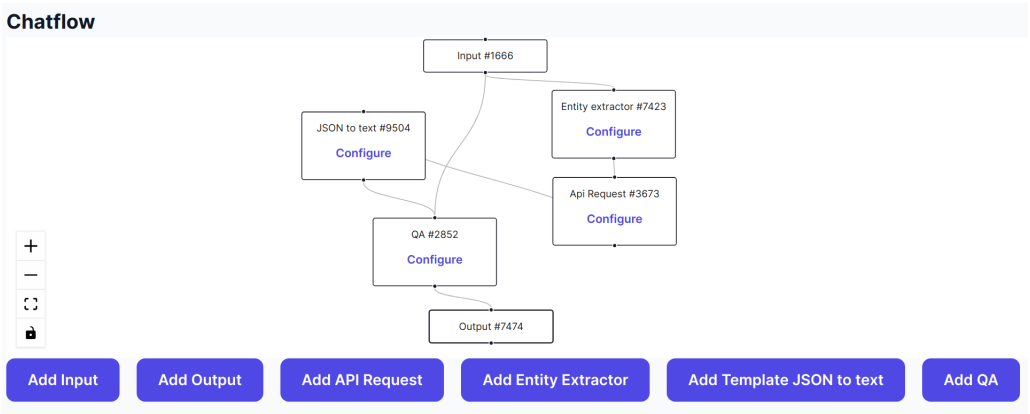


Figura 5.2: Fluxo do chatbot de meteorologia

Para definir o fluxo do chatbot tal como descrito nas etapas anteriores, começa-se por usar a opção "Add input" que define onde o fluxo do chatbot começa.

A etapa seguinte é adicionar o módulo de extração de entidades, clicando em "Add Entity Extractor". Na configuração deste módulo o input é "$1666" que identifica a questão do utilizador e a entidade a extrair é "Countries, cities, states" (figura 5.3).

A segunda etapa é clicar em "Add API Request" e ligar o módulo de extração de entidades a este, configurando este módulo tal como na figura 5.4, o verbo HTTP é o "GET", o *endpoint* é `http://api.weatherapi.com/v1/forecast.json` e a localização identificada com a chave "q" que tem o valor "$7423" que identifica o output do extrator de entidades.

Obtendo-se a informação meteorológica através da etapa anterior em formato JSON, e sabendo que

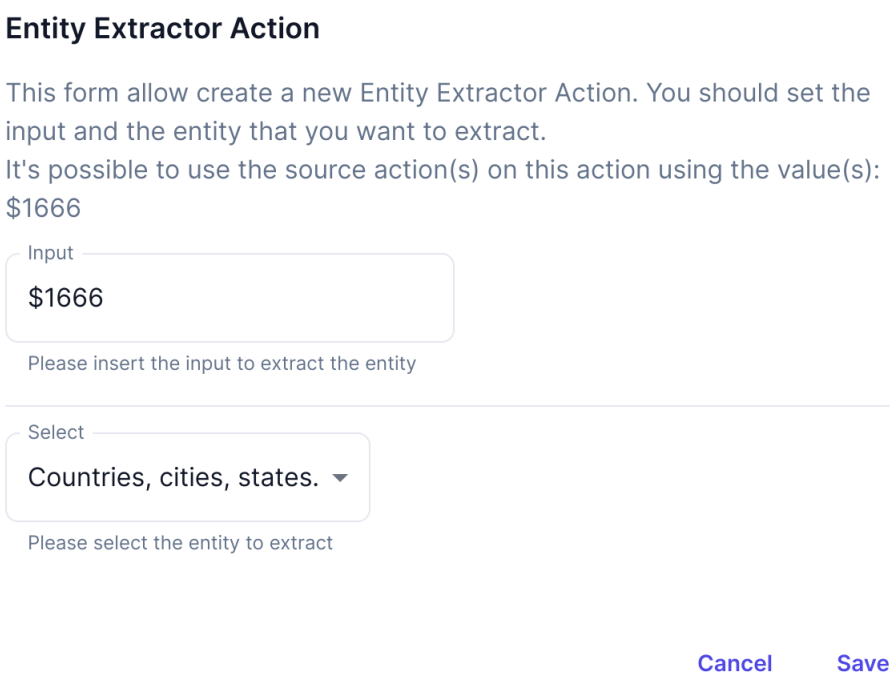


Figura 5.3: Configuração do extrator de entidades para o chatbot de meteorologia

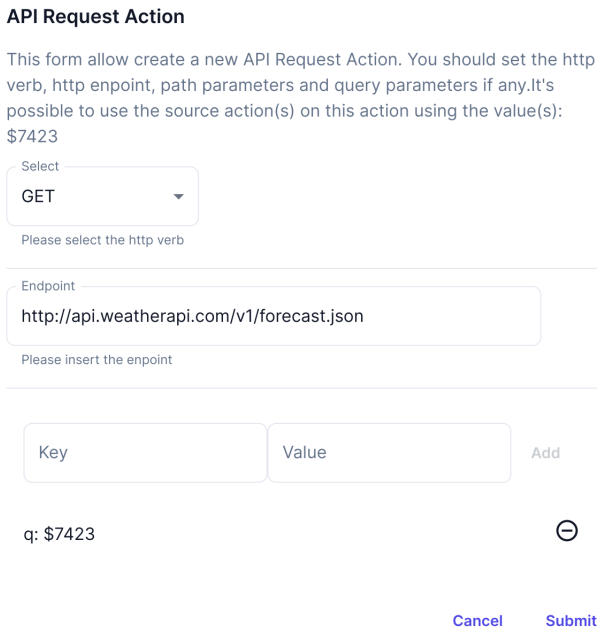


Figura 5.4: Configuração do módulo de pedidos a API para o chatbot de meteorologia

um exemplo deste JSON está presente na listagem 5.2, converte-se este texto com o *template* da listagem 5.1. Para se realizar esta tarefa clica-se em "Add Template JSON to Text" e conecta-se o módulo anterior a este. Sendo que a configuração deste módulo fica tal e qual a figura 5.5.

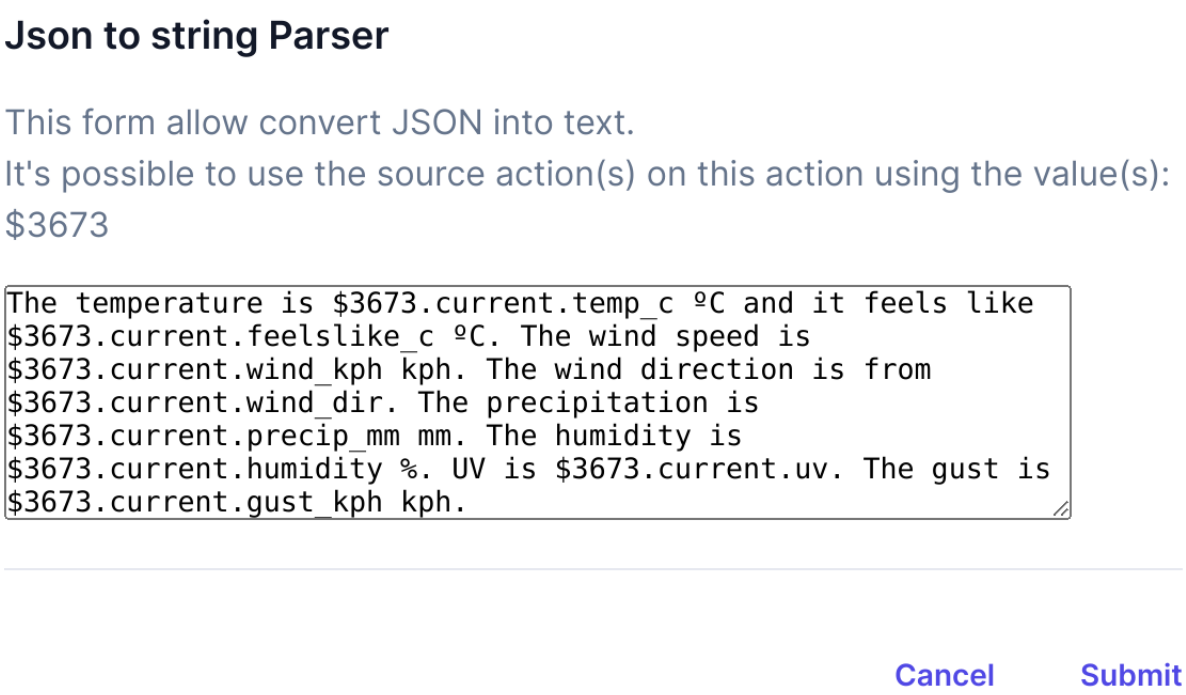


Figura 5.5: Configuração do módulo de template para construir texto a partir de JSON para o chatbot de meteorologia

Listagem 5.1: Template para converter o JSON da resposta de weatherapi em texto

```
The temperature is $3673.current.temp_c ºC and it feels like $3673.current.feelslike_c ºC. The
    ↪  wind speed is $3673.current.wind_kph kph. The wind direction is from $3673.current.
    ↪ wind_dir. The precipitation is $3673.current.precip_mm mm. The humidity is $3673.
    ↪ current.humidity %. UV is $3673.current.uv. The gust is $3673.current.gust_kph kph.
```

Listagem 5.2: Exemplo de pedido à API de meteorologia

```
{
   "location": {
      "name":"Lisbon",
      "region":"Lisboa",
      "country":"Portugal",
      "lat":38.72,
      "lon":-9.13,
      "tz_id":"Europe/Lisbon",
      "localtime_epoch":1667217152,
      "localtime":"2022-10-31 11:52"
   },
   "current": {
      "last_updated_epoch":1667216700,
      "last_updated":"2022-10-31 11:45",
      "temp_c":21.0,
      "temp_f":69.8,
      "is_day":1,
```

```
      "condition": {
         "text":"Partly cloudy",
         "icon":"//cdn.weatherapi.com/weather/64x64/day/116.png",
         "code":1003
      },
      "wind_mph":8.1,
      "wind_kph":13.0,
      "wind_degree":250,
      "wind_dir":"WSW",
      "pressure_mb":1016.0,
      "pressure_in":30.0,
      "precip_mm":0.7,
      "precip_in":0.03,
      "humidity":83,
      "cloud":75,
      "feelslike_c":21.0,
      "feelslike_f":69.8,
      "vis_km":10.0,
      "vis_miles":6.0,
      "uv":5.0,
      "gust_mph":16.6,
      "gust_kph":26.6
   }
}
```


A última etapa é adicionar o módulo de QA, clicando em "add QA", e ligando o módulo de *template* para construir texto a partir de JSON e o input do utilizador a este módulo de QA. A configuração deste módulo fica tal como na figura 5.6, em que o "$1666" é o input do utilizador e "$9504" é o texto para se obter a resposta à questão do utilizador.

Para se concluir o fluxo do chatbot deve-se adicionar o bloco de output, clicando em "Add output", e conectar o módulo de QA a este. Esta operação permite identificar o fim do fluxo do chatbot.

Agora que o fluxo está definido deve-se definir os *intents* e as credenciais para integrar com o Telegram. Os *intents* são exemplo de perguntas que o utilizador vai fazer a este chatbot, servindo de exemplos de treino para o modelo do Rasa. Dado o *template* definido no fluxo anterior, pode-se definir questões como:

- What's the temperature in Braga?
- Precipitation in Porto?
- Wind speed in Barcelos?

Dados estes exemplos tem-se a configuração completa do chatbot tal como na figura 5.7.

Após o utilizador clicar em "Submit" para criar o novo chatbot, os dados da listagem 5.3 são enviados para o *backend*. Estes dados permitem criar o chatbot que é gerado modificando automaticamente os

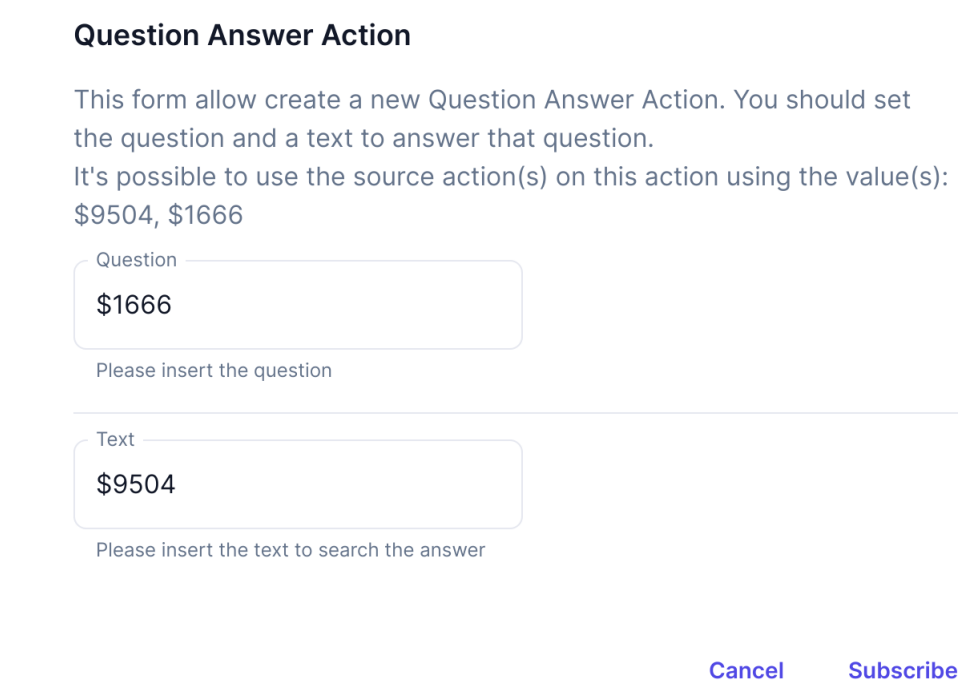


Figura 5.6: Configuração do módulo de QA para o chatbot de meteorologia

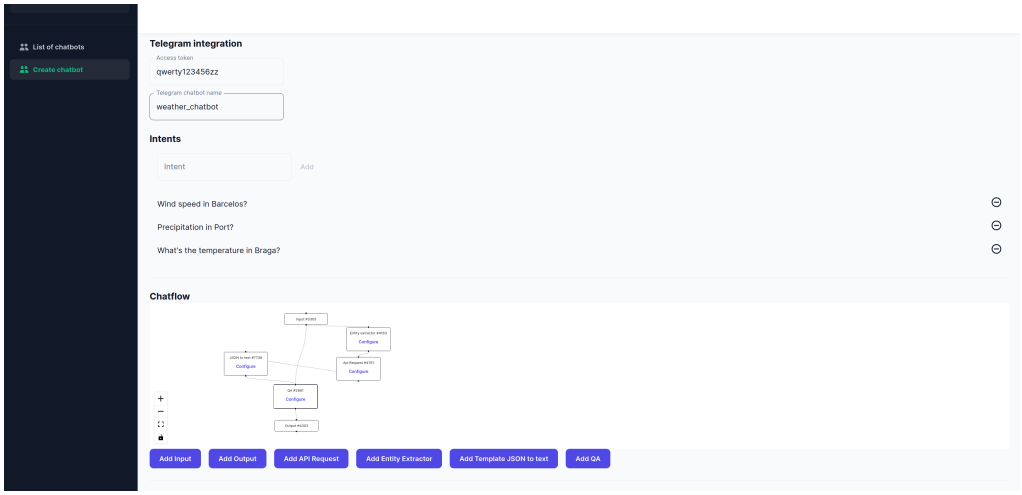


Figura 5.7: Configuração completa do chatbot de meteorologia

ficheiros: `data/nlu.yml`, `domain.yml`, `stories.yml`, `credentials.yml` e `actions/actions.py`, através dos dados enviados da listagem 5.3.

Listagem 5.3: Body do POST para a criação de um novo chatbot

```
{
   "telegram_access_token": "qwerty123456zz",
   "telegram_chatbot_name": "weather_chatbot",
   "intent_examples": [
      "What's the temperature in Braga?",
      "Precipitation in Porto?",
      "Wind speed in Barcelos?"
```

```
8     ],
9     "actions": [
10       {
11          "id": "$7423",
12          "input": "$1666",
13          "entity": "GPE",
14       },
15       {
16           "id": "$3673",
17           "httpVerb": "GET",
18           "endpoint": "http://api.weatherapi.com/v1/current.json",
19           "queryParameters": {
20             "q": "$7423"
21           },
22       },
23       {
24          "id": "$9504",
25          "template": "The temperature is $3673.current.temp_c ºC and it feels like $3673.
                ↪ current.feelslike_c ºC. The wind speed is $3673.current.wind_kph kph. The wind
                ↪  direction is from $3673.current.wind_dir. The precipitation is $3673.current.
                ↪ precip_mm mm. The humidity is $3673.current.humidity %. UV is $3673.current.uv
                ↪ . The gust is $3673.current.gust_kph kph.",
26       },
27       {
28          "id": "$2852",
29          "question": "$1666",
30          "text": "$9504",
31       }
32    ]
33 }
```

O ficheiro `data/nlu.yml` fica configurado tal como na figura 5.4 com os exemplos de intents ("What's the temperature in Braga?", "Precipitation in Porto?" e "Wind speed in Barcelos?") presentes nas linhas 6 a 8 da listagem 5.4.

Listagem 5.4: Configuração do ficheiro nlu.yml para o chatbot de meteorologia

```
1 nlu:
2 - intent: custom
3 examples: |
4    - What's the temperature in Braga?
5     - Precipitation in Porto?
6     - Wind speed in Barcelos?
```

Para este chatbot, o ficheiro `domain.yml` apenas contem o *intent* `custom` e a *action* `custom_action`, que será definida mais à frente, tal como na listagem 5.5.

Listagem 5.5: Configuração do ficheiro domain.yml para o chatbot de meteorologia

```
intents:
   - custom

actions:
   - custom_action
```

Este chatbot também só tem uma historia, assim a definição do ficheiro `stories.yml` fica tal como na figura 5.6

Listagem 5.6: Configuração do ficheiro stories.yml para o chatbot de meteorologia

```
stories:
- story: Custom story
  steps:
  - intent: custom
  - action: custom_action
```

O ficheiro `credentials.yml` contém as credenciais necessárias para a integração com o Telegram (figura 5.7)

Listagem 5.7: Configuração do ficheiro credentials.yml para o chatbot de meteorologia

```
telegram:
   access_token: "qwerty123456zz"
   verify: "weather_chatbot"
```

Por fim o ficheiro `actions/actions.py` fica definido tal como na listagem 5.8.

Listagem 5.8: Configuração do ficheiro actions.py para o chatbot de meteorologia

```
class CustomAction(Action):

   def name(self) -> Text:
      return "custom_action"

   def entity_extractor_7423(self, **kwargs):
      print(f"extracting GPE for the input {kwargs.get('input_1666')}")
      doc = nlp(kwargs.get('input_1666'))
      entities = [x.text for x in doc.ents if x.label_ == "GPE"]
      if not entities:
         raise Exception("Not able to extract the entity GPE")
      else:
         return entities[0]

   def api_request_3673(self, **kwargs):
      return requests.get(
         f"http://api.weatherapi.com/v1/current.json?q={kwargs.get('input_7423')}").json()

```

```
    def template_json_text_9504(self, **kwargs):
        return f"""
The temperature is $3673.current.temp_c ºC and it feels like $3673.current.feelslike_c ºC. The
    ↪  wind speed is $3673.current.wind_kph kph. The wind direction is from $3673.current.
    ↪ wind_dir. The precipitation is $3673.current.precip_mm mm. The humidity is $3673.
    ↪ current.humidity %. UV is $3673.current.uv. The gust is $3673.current.gust_kph kph.
        """

    def question_answer_2852(self, **kwargs):
        tokenizer = AutoTokenizer.from_pretrained(
            "bert-large-uncased-whole-word-masking-finetuned-squad"
        )
        model = AutoModelForQuestionAnswering.from_pretrained(
            "bert-large-uncased-whole-word-masking-finetuned-squad"
        )
        inputs = tokenizer.encode_plus(
            kwargs.get("input_1666"),
            kwargs.get("input_9504"),
            add_special_tokens=True,
            return_tensors="pt"
        )
        input_ids = inputs["input_ids"].tolist()[0]
        answer_start_scores, answer_end_scores = model(**inputs, return_dict=False)
        answer_start = torch.argmax(answer_start_scores)
        # Get the most likely beginning of answer with the argmax of the score
        answer_end = torch.argmax(answer_end_scores) + 1
        return tokenizer.convert_tokens_to_string(
            tokenizer.convert_ids_to_tokens(input_ids[answer_start:answer_end])
        )

    def run(
        self,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: Dict[Text, Any]
    ) -> List[Dict[Text, Any]]:
        input_1234 = tracker.latest_message['text']

        input_1666 = self.entity_extractor_2345(**{'input_1234': input_1234})

        input_3673 = self.api_request_3456(**{'input_1666': input_1666})

        input_9504 = self.template_json_text_4567(**{'input_3673': input_3673})

        input_2852 = self.question_answer_5678(
```

```
61          **{'input_1666': input_1666, 'input_3673': input_3673}
62      )
63
64      dispatcher.utter_message(text=input_2852)
65      return []
```

Por fim na figura 5.8 temos a interação com este chatbot criado através do Telegram.

Figura 5.8: Interação com o chatbot de meteorologia através do Telegram

Capítulo 6

Conclusão e trabalho futuro

O objetivo desta dissertação passou pela criação de uma ferramenta que permita orquestrar *chatbots* , ou seja, criar e gerir *chatbots* baseados em módulos.

Para o desenvolvimento de um *chatbot* é necessário que o mesmo reconheça como associar a questão de um utilizador a uma fonte de dados que possa providenciar a resposta. Este reconhecimento é realizado por um sistema de diálogo em que associa a questão a um processo que providencia a resposta. Como sistema de diálogo escolheu-se o Rasa pois permite definir exemplos de *intents* e respetivas ações (processo de lógica para obter a resposta). O Rasa também permite facilmente integrar com redes sociais.

No ChatbotWizard o desenvolvimento do processo de lógica consiste em ter-se quatro módulos que se podem integrar para a criação de um *chatbot* : extração de entidades, pedidos a API, converter JSON em texto por template e módulo de QA. Na aplicação web, o utilizador pode arrastar os módulos, configurar e conectar os mesmos para definir o fluxo de resposta às questões dos utilizadores.

No capítulo dos objetivos e funcionalidades foi demonstrada grande ambição para se implementar diversas características de um grau de complexidade elevada, tendo sido implementado o que é mais relevante para as funcionalidades chave do ChatbotWizard.

Conseguiu-se integrar o *chatbot* no Telegram, pretendendo-se integrar com as redes sociais Facebook, Messenger e o Whatshapp no futuro.

No entanto algumas funcionalidades não foram possíveis de implementar e fazem parte do trabalho futuro. De entre as quais se encontra a edição de um *chatbot* já existente e permitir criar novos módulos dinamicamente.

Por último faz também parte do trabalho futuro adicionar uma base de dados para se poder gerir os *chatbots* e respetivos módulos.

Bibliografia

Abu-Salih, B. (2021). Domain-specific knowledge graphs: A survey. *Journal of Network and Computer Applications*, *185*, 103076. https://doi.org/https://doi.org/10.1016/j.jnca.2021.103076

Adamopoulou, E. & Moussiades, L. (2020a). Chatbots: History, technology, and applications. *Machine Learning with Applications*, *2*, 100006. https://doi.org/https://doi.org/10.1016/j.mlwa.2020.100006

Adamopoulou, E. & Moussiades, L. (2020b). An overview of chatbot technology. Em I. Maglogiannis, L. Iliadis & E. Pimenidis (Eds.), *Artificial intelligence applications and innovations* (pp. 373–383). Springer International Publishing.

Akbik, A., Blythe, D. & Vollgraf, R. (2018). Contextual string embeddings for sequence labeling. *Proceedings of the 27th international conference on computational linguistics*, 1638–1649.

Alexei, B., Sergey, E., Yinhan, L. & Michael, A. (2019). Cloze-driven pretraining of self-attention networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 5360–5369.

Al-Moslmi, T., Gallofré Ocaña, M., L. Opdahl, A. & Veres, C. (2020). Named entity extraction for knowledge graphs: A literature overview. *IEEE Access*, *8*, 32862–32881. https://doi.org/10.1109/ACCESS.2020.2973928

Ando, R. K., Zhang, T. & Bartlett, P. (2005). A framework for learning predictive structures from multiple tasks and unlabeled data. *Journal of Machine Learning Research*, *6*(11).

Brandtzaeg, P. B. & Følstad, A. (2017). Why people use chatbots. *International conference on internet science*, 377–392.

Braşoveanu, A. M. P. (2021). Integrating machine learning techniques in semantic fake news detection. *Neural Processing Letters*, *53*, 103076. https://doi.org/https://doi.org/10.1007/s11063-020-10365-x

Colby, K. M., Weber, S. & Hilf, F. D. (1971). Artificial paranoia. *Artificial Intelligence*, *2*(1), 1–25.

Cucerzan, S. (2007). Large-scale named entity disambiguation based on wikipedia data. *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, 708–716.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Etzioni, O., Cafarella, M., Downey, D., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S. & Yates, A. (2005). Unsupervised named-entity extraction from the web: An experimental study. *Artificial intelligence*, *165*(1), 91–134.

Ferragina, P. & Scaiella, U. (2011). Fast and accurate annotation of short texts with wikipedia pages. *IEEE software*, *29*(1), 70–75.

Goyal, A., Gupta, V. & Kumar, M. (2018). Recent named entity recognition and classification techniques: A systematic review. *Computer Science Review*, *29*, 21–43.

Green Jr, B. F., Wolf, A. K., Chomsky, C. & Laughery, K. (1961). Baseball: An automatic question-answerer. *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, 219–224.

Grishman, R. & Sundheim, B. (1996). Message understanding conference-6: A brief history. *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, 466–471. https://doi.org/10.3115/992628.992709

Heller, B., Proctor, M., Mah, D., Jewell, L. & Cheung, B. (2005). Freudbot: An investigation of chatbot technology in distance education. *EdMedia+ Innovate Learning*, 3913–3918.

Hien, H. T., Cuong, P.-N., Nam, L. N. H., Nhung, H. L. T. K. & Thang, L. D. (2018). Intelligent assistants in higher-education environments: The fit-ebot, a chatbot for administrative and learning support. *Proceedings of the ninth international symposium on information and communication technology*, 69–76.

Hoy, M. B. (2018). Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical reference services quarterly*, *37*(1), 81–88.

Jurafsky, D. & Martin, J. H. (2000). *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition* (1st). Prentice Hall PTR.

Kucherbaev, P., Bozzon, A. & Houben, G.-J. (2018). Human-aided bots. *IEEE Internet Computing*, *22*(6), 36–43.

Li, B. Z., Min, S., Iyer, S., Mehdad, Y. & Yih, W.-t. (2020). Efficient one-pass end-to-end entity linking for questions. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6433–6441. https://doi.org/10.18653/v1/2020.emnlp-main.522

Luong, M.-T., Pham, H. & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *CoRR*, *abs/1508.04025*. http://arxiv.org/abs/1508.04025

Mauldin, M. L. (1994). Chatterbots, tinymuds, and the turing test: Entering the loebner prize competition. *AAAI*, *94*, 16–21.

Mihalcea, R. & Csomai, A. (2007). Wikify! linking documents to encyclopedic knowledge. *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, 233–242.

Milne, D. & Witten, I. H. (2008). Learning to link with wikipedia. *Proceedings of the 17th ACM conference on Information and knowledge management*, 509–518.

Molnár, G. & Szüts, Z. (2018). The role of chatbots in formal education. *2018 IEEE 16th International Symposium on Intelligent Systems and Informatics (SISY)*, 000197–000202.

Nadeau, D. & Sekine, S. (2007). A survey of named entity recognition and classification. *Lingvisticae Investigationes*, *30*(1), 3–26.

Nimavat, K. & Champaneria, T. (2017). Chatbots: An overview types, architecture, tools and future possibilities. *Int. J. Sci. Res. Dev*, *5*(7), 1019–1024.

Passos, A., Kumar, V. & McCallum, A. (2014). Lexicon infused phrase embeddings for named entity resolution. *arXiv preprint arXiv:1404.5367*.

Ramesh, K., Ravishankaran, S., Joshi, A. & Chandrasekaran, K. (2017). A survey of design techniques for conversational agents. *International conference on information, communication and computing technology*, 336–350.

Richardson, L. & Ruby, S. (2007). *Restful web services*. O'Reilly. https://www.safaribooksonline.com/library/view/restful-web-services/9780596529260/

Roberts, A., Raffel, C. & Shazeer, N. (2020). How much knowledge can you pack into the parameters of a language model? *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 5418–5426. https://doi.org/10.18653/v1/2020.emnlp-main.437

Sharnagat, R. (2014). Named entity recognition: A literature survey. *Center For Indian Language Technology*, 1–27.

Shawar, B. A. & Atwell, E. (2007). Chatbots: Are they really useful? *Ldv forum*, *22*(1), 29–49.

Singh, A., Ramasubramanian, K. & Shivam, S. (2019). Introduction to microsoft bot, rasa, and google dialogflow. *Building an enterprise chatbot: Work with protected enterprise data using open source frameworks* (pp. 281–302). Apress. https://doi.org/10.1007/978-1-4842-5034-1_7

Singhal, A. et al. (2012). Introducing the knowledge graph: Things, not strings. *Official google blog*, *5*, 16.

Sreeharsha, A. S. S. K. & Kesapragada, S. (2022). Building chatbot using amazon lex and integrating with a chat application. *INTERANTIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT*, *06*. https://doi.org/10.55041/IJSREM12145

Sreeram, J. G., Luo, X. & Tian, R. (2021). Contextual and behavior factors extraction from pedestrian encounter scenes using deep language models. Em M. Golfarelli, R. Wrembel, G. Kotsis, A. M. Tjoa & I. Khalil (Eds.), *Big data analytics and knowledge discovery* (pp. 131–136). Springer International Publishing.

TURING, A. M. (1950). I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, *59*(236), 433–460. https://doi.org/10.1093/mind/LIX.236.433

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017). Attention is all you need.

Weizenbaum, J. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45.

Yadav, V. & Bethard, S. (2019). A survey on recent advances in named entity recognition from deep learning models. *arXiv preprint arXiv:1910.11470*.

Yan, J., Wang, C., Cheng, W., Gao, M. & Zhou, A. (2018). A retrospective of knowledge graphs. *Frontiers of Computer Science*, *12*(1), 55–74.

Zhang, S. & Elhadad, N. (2013). Unsupervised biomedical named entity recognition: Experiments with clinical and biological texts. *Journal of biomedical informatics*, *46*(6), 1088–1098.

Zheng, S., Xu, J., Zhou, P., Bao, H., Qi, Z. & Xu, B. (2016). A neural network framework for relation extraction: Learning entity semantic and relation pattern. *Knowledge-Based Systems*, *114*, 12–23.