

## **Relatório de Estágio do Curso Técnico Superior Profissional de Sistemas e Tecnologias de Informação**

KOBU Agency, Faro

Marcelo Souza Santos

Orientador: Roberto Lam

Supervisor: Nuno Tenazinha

Faro, 2024

## Resumo

Este relatório descreve o desenvolvimento de um assistente virtual para a KOBU Agency, realizado durante meu estágio. O projeto visou aprimorar a recepção de contatos de possíveis clientes e colaboradores através de um sistema automatizado que utiliza processamento de linguagem natural para entender as intenções dos usuários e fornecer informações relevantes sobre os serviços e projetos da empresa, além de gerar pedidos de contato para o *backoffice*. O assistente virtual foi desenvolvido utilizando uma arquitetura de *frontend* em HTML, JavaScript e CSS/SSS, e um *backend* em Python utilizando a biblioteca Flask para comunicação via API. Este documento detalha as etapas do desenvolvimento, desde a concepção e implementação até a análise crítica dos resultados obtidos, destacando as principais aprendizagens e contribuições para a empresa.

# Índice geral

## Conteúdo

1. INTRODUÇÃO.....	1
2. CARACTERIZAÇÃO DA EMPRESA.....	2
2.1. Caracterização Interna .....	2
2.2. Área de Atuação e Estrutura .....	2
2.3. Contexto Regional, Nacional e/ou Internacional.....	2
2.4. Caracterização Externa .....	2
3. DESCRIÇÃO DAS TAREFAS/ATIVIDADES DESENVOLVIDAS.....	3
3.1. O <i>Backend</i> .....	5
3.1.1. Contexto extra .....	11
3.2. O <i>frontend</i> .....	16
4. ANÁLISE CRÍTICA DO TRABALHO REALIZADO .....	20
5. LISTA DE REFERÊNCIAS .....	21
6. CONCLUSÃO.....	22
7. ANEXOS .....	23

### **Lista de Abreviaturas**

LLM - Large Language Models (Modelos de Linguagem de Grande Escala)

API - Application Programming Interface (Interface de Programação de Aplicação)

JSON - JavaScript Object Notation

## 1. INTRODUÇÃO

Este relatório apresenta as atividades desenvolvidas durante meu estágio na KOBU Agency, focando no projeto de desenvolvimento de um assistente virtual. A introdução do relatório descreve os objetivos do projeto, a metodologia adotada e a relevância do assistente virtual para a empresa.

O projeto não foi desenvolvido dentro de um departamento, mas sim isoladamente dos demais projetos, com a orientação do supervisor do estágio dentro da empresa (Nuno Tenazinha).

### Objetivos do Projeto:

- Desenvolver um assistente virtual capaz de entender e responder às intenções dos usuários utilizando processamento de linguagem natural.
- Melhorar a receção e triagem de contatos de possíveis clientes e colaboradores.
- Integrar tecnologias de *frontend* e *backend* isoladamente para criar uma solução completa e versátil.

### Metodologia:

- Utilização de tecnologias web (HTML, CSS, JavaScript) para o *frontend*.
- Desenvolvimento do *backend* em Python utilizando Flask.
- Implementação de modelos de linguagem da OpenAI<sup>2</sup> (OpenAI 2024) para processamento de linguagem natural.

### Relevância:

A implementação de um assistente virtual automatiza o processo de receção de contatos, tornando-o mais eficiente e permitindo à equipe da KOBU Agency focar em tarefas mais estratégicas. Além disso, o projeto proporciona uma experiência prática valiosa em desenvolvimento de software e integração de tecnologias avançadas.

## **2. CARACTERIZAÇÃO DA EMPRESA**

### **2.1. Caracterização Interna**

A KOBU Agency é um Laboratório para Marcas e Experiências Digitais que opera de Portugal para o mundo. Fundada em outubro de 2014 por Nuno Tenazinha, após quatro anos como desenvolvedor *frontend* freelancer, a agência tem como missão construir marcas do zero com agilidade para se adaptar ao mundo em constante mudança do século XXI. Nuno foi logo acompanhado por Sandra Lopes, ilustradora e amiga de infância, e ao longo dos anos, reuniram uma equipa de seres humanos curiosos com paixão por inovar e explorar maneiras de inspirar e crescer.

A agência foi premiada como a Agência Digital do Ano 2021/2022 em Portugal, pelos Prémios Lusófonos da Criatividade. Durante o IX Festival Anual dos Prémios Lusófonos da Criatividade, realizado em Lisboa em julho de 2022, a agência recebeu esse reconhecimento extraordinário, solidificando seus esforços contínuos para criar produtos digitais que proporcionam experiências de marca significativas.

### **2.2. Área de Atuação e Estrutura**

A agência adota uma abordagem estratégica e experimental, buscando insights de negócios para alimentar seu trabalho criativo. Eles ajudam clientes de diversos mercados, desde marcas sustentáveis e de bem-estar até startups de tecnologia, cultura e entretenimento, moda e estilo de vida, além de marcas de hospitalidade de luxo e imobiliárias.

### **2.3. Contexto Regional, Nacional e/ou Internacional**

A KOBU Agency atua principalmente no mercado português, mas também possui uma presença significativa no cenário internacional, trabalhando com clientes de várias partes do mundo.

### **2.4. Caracterização Externa**

A empresa mantém relacionamentos estratégicos com outras empresas e entidades, colaborando em projetos que exigem expertise multidisciplinar.

### 3. DESCRIÇÃO DAS TAREFAS/ATIVIDADES DESENVOLVIDAS

Primeiramente, foi idealizado o modelo de comunicação entre os 3 diferentes *endpoints*: o servidor do assistente em Python, a API da OpenAI e o *frontend*.

Foi decidido concentrar as requisições à API da OpenAI a partir do servidor Python e realizar a comunicação do *frontend* somente com o servidor Python através de API.

Outra questão essencial seria estabelecer uma forma de não sobrecarregar os *prompts* enviados à OpenAI com demasiada informação. Uma vez que o servidor da OpenAI não funciona com um histórico de mensagens, nem existe uma maneira de pré-carregar as instruções diretamente no servidor da OpenAI, foi necessário desenvolver uma lógica para criar os *prompts* dinamicamente para adaptar ao contexto das mensagens (ver Figura 1).

Ainda a falar das limitações da API da OpenAI, durante o desenvolvimento ficou claro a necessidade de sintetizar os *prompts* para obter respostas mais naturais e satisfatórias ao contexto do utilizador. Sendo assim, foi escolhido pré-definir os *prompts* e escolhê-los a partir de duas variáveis essenciais: “subject” (assunto de contacto escolhido pelo usuário) “current\_stage” (estágio atual da conversa).

*Prompts* são instruções ou perguntas fornecidas a um sistema de inteligência artificial (IA), como um modelo de linguagem, para gerar uma resposta ou realizar uma tarefa específica. Eles servem como diretrizes que orientam o comportamento e a produção do sistema. No contexto de I.A. (Inteligência Artificial), especialmente com modelos como o ChatGPT, os *prompts* são as entradas textuais que os usuários fornecem para obter informações, gerar texto, responder perguntas, entre outros.

```

backend > assistant > knowledge > prompts.py > Prompts > prompt_chooser
1 from langchain_core.prompts import ChatPromptTemplate
2 from .knowledge_loaders import KnowledgeLoaders
3
4
5 class Prompts(KnowledgeLoaders):
6     """
7     A class representing the knowledge base for the chat application.
8     """
9     > def __init__(self, stage: str = None) -> None: ...
10
11
12
13
14
15
16
17 async def prompt_chooser(self, stage: str = None) -> ChatPromptTemplate:
18     """
19     Chooses the appropriate prompt based on the stage of the conversation.
20
21     Args:
22         stage (str): Current stage of the conversation.
23
24     Returns:
25         ChatPromptTemplate: Prompt template.
26     """
27     stage = self.WELCOME_STAGE if None else stage
28
29     match stage:
30         case self.WELCOME_STAGE:
31             prompt = ChatPromptTemplate.from_messages([
32                 ("system", "{basic_instructions}"),
33                 self.assistant_tone_of_voice(),
34                 ("system", ""Greet the user, thank them for their interest in contacting Kubo,
35                     and mention that Nuno has something to share (a video will be displayed to the
36                     user just after your message. Use the tone of voice provided.)""),
37                 ("user", "{input}"),
38             ])
39
40
41 > case self.CHOOSE_SUBJECT_STAGE: ...
42
43
44
45
46
47
48
49 case self.DATA_COLLECTING_STAGE:
50     if self.subject_name in [self.HIRE_US, self.JOIN_THE_TEAM]:
51         prompt = ChatPromptTemplate.from_messages([
52             self.assistant_site_context(),
53             self.assistant_tone_of_voice(),
54             # ("system", "{subject_instructions}"),
55             ("system", ""Please, NEVER ASK more of 2 datas in the same message. Keep the conversation smooth. ...
56
57
58
59             ("system", "These are the data required: \n{data_required}"),
60             ("system", ""Please, NEVER ASK more of 2 datas in the same message. ...
61
62
63             ("system", ""IMPORTANT: If a user provide o budget bellow 10.000 EURS, ...
64             ("system", ""Keep answering the user based on the instructions provided by the system.
65             Do not greeting again. Keep the tone of voice provided.""),
66             ("system", ""Approach example:\n...
67             ("system", "Conversation history: {chat_history}"),
68             ("user", "{input}"),
69         ])
70
71
72
73
74
75

```

Figura 1 – Método `prompt_chooser` da classe `Prompt`, responsável por selecionar o prompt a ser enviado para a LLM. Os prompts são escolhidos de acordo com o estágio da conversa. Os placeholders `{}` são preenchidos com o conteúdo correspondente, armazenado em arquivos JSON.

A mudança dos estágios da conversa é feita pelo *backend*, porém, para que a interface gráfica possa proporcionar uma melhor experiência ao usuário, alguns estágios têm tratamento diferente por parte do *frontend*, como vê-se na Figura 2 abaixo.



```

frontend > dist > js > JS start-conversation.js > StartConversation > handleDefaultStage > InputElement
14 export class StartConversation {
15
16
17
18   async main() {
19     this.conversation.showSpinner();
20     const inputElement = this.userInput;
21     inputElement.blur();
22
23     switch (this.conversation.currentStage) {
24       case undefined:
25       case '':
26       case null:
27         case this.conversation.WELCOME_STAGE:
28           await this.handleWelcomeStage();
29           // break;
30         case this.conversation.CHOOSE_SUBJECT_STAGE:
31           await this.handleChooseSubjectStage();
32           break;
33       default:
34         await this.handleDefaultStage();
35     }
36   }
37
38   // Stage Handlers
39   async handleWelcomeStage() {
40     console.log("Main: starts welcomeMessage()");
41     const request = this.conversation.requestData("Hi, there!");
42     const response = await this.conversation.sendRequest(request);
43     await this.conversation.assistantResponseHandler(response);
44     if (response.current_stage === 'error') return;
45     await this.conversation.setVideo();
46     this.conversation.currentStage = this.conversation.CHOOSE_SUBJECT_STAGE;
47     console.log("Main: finish welcomeMessage()", this.conversation.currentStage);
48   }
49
50   async handleChooseSubjectStage() { ...
51   }
52
53   async handleDefaultStage() {
54     this.conversation.setUserResponse();
55     const request = this.conversation.requestData();
56     const inputElement = this.userInput;
57     inputElement.value = '';
58     const response = await this.conversation.sendRequest(request);
59     if (response.message === false) return;
60     await this.conversation.assistantResponseHandler(response);
61     this.userInput.placeholder = 'Type a message';
62     console.log("Main: finish default()", this.conversation.currentStage);
63   }
64 }

```

Figura 2 – Classe “StartConversation” do repositório frontend. Na imagem, podemos ver os métodos isolados que tratam diferentes estágios de conversação da interface gráfica.

### 3.1. O Backend

O *backend* do projeto consiste em uma API Python que trata o conteúdo das requisições do *frontend*, instancia-se os usuários (cria-se o usuário em memória), salva o ambiente de conversação, realiza a lógica para gerar uma resposta e a retorna ao *frontend*.

As requisições recebidas pelo *backend* têm em seu corpo as seguintes chaves: “user\_id” (obrigatório), “user\_input” (obrigatório), “orientation”, “subject”, “next\_stage” e “current\_stage”. O retorno das requisições contém, além das mesmas chaves da solicitação recebida, a chave “message” (com o texto de resposta final do assistente à última mensagem do usuário) e pode ou não conter uma chave “options”.

A chave “options”, por sua vez, é uma lista que contém opções pré-definidas de *inputs* a serem escolhidos pelo usuário, a depender do estágio atual da conversa. A implementação dessa chave foi feita para diminuir o fator aleatoriedade do modelo de linguagem, uma vez que o fluxo da conversa requer decisões (respostas) precisas em determinadas alturas – por exemplo, para escolher o tema da conversa (contratar serviço, conhecer a agência ou envio de candidaturas de trabalho).

Essas variáveis estáticas estão concentradas na classe “ChatConsts”, como mostra a Figura 3 abaixo.

```
backend > assistant > consts.py > ChatConsts
1 from .tools.subjects_lead_extractor import HireUs, GeneralContact, JoinTheTeam
2
3
4 class ChatConsts:
5     """
6     Utility class containing constant values used throughout the chat application.
7     """
8     # SUBJECTS POSSIBLES
9     GENERAL_CONTACT = 'general_contact'
10    HIRE_US = 'hire_us'
11    JOIN_THE_TEAM = 'join_the_team'
12
13    # Initialize instances of subjects
14    CLASS_GENERAL_CONTACT = GeneralContact(GENERAL_CONTACT)
15    CLASS_HIRE_US = HireUs(HIRE_US)
16    CLASS_JOIN_THE_TEAM = JoinTheTeam(JOIN_THE_TEAM)
17
18    # ORIENTATIONS POSSIBLES
19    NEXT_STAGE = 'next_stage' # Proceed to the next stage
20    VERIFY_ANSWER = 'verify_answer' # Verify the answer
21    PROCEED = 'proceed' # Proceed with the current stage
22    CRITICAL = 'critical' # Critical mode for error handling
23
24    # STAGES POSSIBLES
25    WELCOME_STAGE = 'welcome' # Welcome stage
26    ACCEPTANCE_OF_TERMS_STAGE = 'acceptance_of_terms'
27    CHOOSE_SUBJECT_STAGE = 'choose_subject' # Stage to choose a subject
28    DATA_COLLECTING_STAGE = 'data_collecting' # Data collecting stage
29    DATA_COLLECTING_VALIDATION_STAGE = 'data_collecting_validation' # Data collecting validation stage
30    RESUME_VALIDATION_STAGE = 'resume_validation' # Resume validation stage
31    SEND_VALIDATION_STAGE = 'send_validation' # Send validation stage
32    FREE_CONVERSATION_STAGE = 'free_conversation' # Free conversation stage
33
34    # OPTIONS POSSIBLES BY STAGES
35    ACCEPTANCE_OF_TERMS_STAGE_OPTIONS = [
36        "I agree to the terms and conditions.",
37        "I do not agree to the terms and conditions."
38    ] # Options for the acceptance of terms stage
39
40    CHOOSE_SUBJECT_STAGE_OPTIONS = [
41        "I'd like to know more about KOBU Agency.",
42        "I'd like to hire KOBU.",
43        "I'd like to join KOBU team.",
44        # "I'd like to talk to somebody in KOBU Agency."
45    ] # Options for the choose subject stage
46
47    RESUME_VALIDATION_STAGE_OPTIONS = [
48        "It looks fine!",
49        "Actually I'd like to change something, if you don't mind."
50    ] # Options for the resume validation stage
51
52    SEND_VALIDATION_STAGE_OPTIONS = [
53        "Yes, you may send!",
54        "Wait. Do not send it yet, please."
55    ] # Options for the send validation stage
56
57
```

Figura 3 – Classe “ChatConsts” do repositório backend. Nesta classe também ficam salvos as opções passíveis de serem enviadas pelo backend para o frontend. No frontend, essas opções podem ser transformadas em botões dinâmicos.

A API (módulo “api”) possui um endpoint POST “/kobu-assistant” onde são rececionadas as requisições da interface do *frontend* utilizando a biblioteca Flask. Em seguida, a classe “RequestHandler” trata o JSON da requisição, identificando o “user\_id”, instanciando o novo usuário caso este não esteja ativo, salvando-o em um

dicionário e então executando a função “main” da classe “Conversation” do módulo “assistant”, passando como argumento a requisição recebida.

A função “main” da classe “Conversation” é responsável por tratar os atributos da classe, nomeadamente salvar o histórico da conversa, além de solicitar, salvar e retornar para a API a resposta final do assistente. A resposta do assistente é obtida pela função “get\_assistant\_response”, que recebe a mesma requisição que a função “main” recebe.

A função “get\_assistant\_response” tem duas responsabilidades principais: atualizar o estado da conversação (ao avaliar os valores contidos nas chaves “current\_stage”, “next\_stage” e “orientation” contidas na requisição) e invocar um dos assistentes pré-configurados para cada estágio da conversa, ilustrada na Figura 4.

```
async def get_assistant_response(self, user_request: dict = {}) -> dict:
    """
    Answers the user_input and manages conversation state changes based on data detection for lead generation.

    Args:
        user_request (dict, optional): The user's request. Defaults to {}.

    Returns:
        dict: The response to the user's request.
    """
    try:
        print("get_assistant_response() Current stage: ", self.current_stage)
        if not self.current_stage: ...

        if self.orientation == self.NEXT_STAGE: ...

        while True:
            match self.current_stage:
                case self.WELCOME_STAGE:
                    response = await self.welcome(user_request)
                    self.set_user_attributes(response)
                    # print("Welcome Response:\n", response)

                    if response['orientation'] == self.NEXT_STAGE:
                        self.next_stage = self.CHOOSE_SUBJECT_STAGE
                        # self.next_stage = self.ACCEPTANCE_OF_TERMS_STAGE # The ACCEPTANCE_OF_TERMS_STAGE case is beeing integrated.
                        break

                    else:
                        self.current_stage = self.next_stage
                        continue

            # The ACCEPTANCE_OF_TERMS_STAGE case is beeing integrated.
            case self.ACCEPTANCE_OF_TERMS_STAGE: ...

            case self.CHOOSE_SUBJECT_STAGE: ...

            case self.DATA_COLLECTING_STAGE: ...

            case self.RESUME_VALIDATION_STAGE: ...

            case self.SEND_VALIDATION_STAGE: ...

            case self.FREE_CONVERSATION_STAGE: # Stop going verifications ...

            case self.CRITICAL: # To bem implemented ...

        except Exception as e: ...

    finally: ...
```

Figura 4 – Método “get\_assistant\_response” da classe “Conversation”. No screenshot, é possível ver que cada estágio dentro do loop “while True” é tratado de uma maneira distinta e que a lógica aceita mudança de estágio dentro do próprio looping, o que adiciona mais uma cada a evitar respostas duplicadas.

A resposta final provida pelo assistente é obtida dentro de um *looping* que, através do estado atual (atributo “current\_stage” da conversa instanciada), aplica diferentes lógicas

conforme necessário. As lógicas para cada estado são métodos da classe “Assistant”, representada na Figura 5.

```
backend > assistant > assistant.py > ...
1  import json
2  from .tools.utils import Utils
3
4
5  class Assistant(Utils):
6      """Assistant class handles various stages of user interaction,
7         including welcoming the user, choosing a subject, collecting and validating data,
8         and managing free conversation after lead generation."""
9
10 >     async def welcome(self, user_request: dict) -> dict: ...
31
32 >     async def choose_subject(self, user_request: dict) -> dict: ...
64
65 >     async def data_collecting_validation(self, user_request: dict) -> dict: ...
146
147 >     async def data_collecting(self, user_request: dict) -> dict: ...
164
165 >     async def resume_validation(self, user_request: dict) -> dict: ...
198
199 >     async def send_validation(self, user_request: dict) -> dict: ...
231
232 >     async def free_conversation(self, user_request: dict) -> dict: ...
266
```

Figura 5 – Classe “Assistant” onde cada método concentra uma lógica diferente para obter uma mensagem coerente para o input do usuário. Cada método retorna um dicionário Python que fará parte do corpo da resposta da API do assistente para o frontend.

Por exemplo, no estado “DATA\_COLLECTING\_STAGE” (Figura 6), o assistente solicita os dados em falta para gerar uma Lead e verifica se no histórico da conversa foram providos esses dados. Para isso, o retorno da função “data\_collecting\_validation” é armazenado na variável “validation” enquanto o retorno da função “data\_collecting” é armazenado na variável “response”.

A função “data\_collecting\_validation” (Figura 7) condensa a lógica para verificar se no histórico da conversa há dados suficientes para gerar uma Lead de contato. A extração dos dados da conversa é feita através da API da OpenAI, onde é enviada uma requisição com um *prompt* específico que contém o histórico da conversa, as instruções, além de outros atributos que especificam o formato do retorno. A função “data\_collecting\_validation” possui outras camadas para assegurar que os dados extraídos fazem sentido, mas estamos interessados na chave “orientation” contida no dicionário retornado.

```

case self.DATA_COLLECTING_STAGE:
    validation = await asyncio.create_task(self.data_collecting_validation(user_request))
    # response = await asyncio.create_task(self.data_collecting_in_changing(user_request))
    response = await asyncio.create_task(self.data_collecting(user_request))
    self.set_user_attributes(response)
    # print("Data Collecting Response:\n", response)

    if validation['orientation'] == self.PROCEED:
        self.next_stage = self.DATA_COLLECTING_STAGE
        self.orientation = self.VERIFY_ANSWER
        break
    # del response['options']

    elif validation['orientation'] == self.NEXT_STAGE:
        # response = await self.data_collecting(user_request)
        self.current_stage = self.next_stage = self.RESUME_VALIDATION_STAGE
        self.orientation = self.PROCEED

        # self.lead = self.subject_instance.get_leads_info(self.chat_history)
        await asyncio.create_task(self.chat_buffer(system_message=f"Datas detected: {self.lead}"))

    continue

```

Figura 6 – Tratamento da requisição para o estágio “DATA\_COLLECTING”. Observe que há duas lógicas distintas: uma para caso a validação seja prosseguir e outra para o caso da validação for para pular para o próximo estágio.

```

189
190 async def data_collecting(self, user_request: dict) -> dict:
191     """Assistant to collect datas from the user."""
192     print("ResponseHandler: data_collecting()")
193     self.current_stage = self.DATA_COLLECTING_STAGE
194
195     try:
196         user_input = user_request.get('user_input')
197         chain = await self.chain_builder(self.DATA_COLLECTING_STAGE)
198         message = self.chain_invoker(chain=chain, user_input=user_input)
199         self.orientation = self.PROCEED
200
201     except Exception as e:
202         print(f"ResponseHandler: data_collecting() Error {e}")
203
204     finally:
205         response = {"message": message, 'orientation': self.orientation, 'current_stage': self.DATA_COLLECTING_STAGE}
206         return response
207

```

Figura 7 – Método “data\_collecting” da classe “Assistant”.

Se a chave “orientation” do dicionário armazenado na variável “validation” tiver o valor “PROCEED”, a resposta final do assistente a ser devolvida para a API será o dicionário armazenado na variável “response”, que recebeu o retorno da função “data\_collecting”, encerrando o *looping*. Caso contrário, outra lógica para atualização dos atributos acontece, e o *looping* continua, passando para o próximo estágio da conversação (por exemplo, “RESUME\_VALIDATION\_STAGE”).

A função “data\_collecting” concentra a lógica de solicitar a resposta ao assistente, utilizando o *prompt* pré-definido que será utilizado como corpo da chamada à API da OpenAI (similar a lógica implementada na função “data\_collecting\_validation”). A chamada à API é feita pela função “chain\_invoker”

A maior parte das funções que retornam o texto da mensagem final do assistente utilizam uma lógica de “chains”, importada da biblioteca LangChain<sup>1</sup> (LangChain, 2024). Esta lógica permite montar uma sequência de funções onde o output de uma função serve como input para outra. No caso, uma chain precisa de um *prompt* que contenha as instruções para o assistente, o modelo de linguagem (LLM da OpenAI) e, opcionalmente, um *retriever* de contexto extra. A Figura 8 ilustra parte desta lógica

```
▼ async def chain_builder(self, stage: str = '') -> object:
▼     """
    Build the main chain that will answer the user_input.

    Args:
        stage (str): Stage of the conversation.

    Returns:
        object: Main chat chain.
    """
    print("chain_builder starts")

    try:
        # prompt = await asyncio.create_task(self.prompt_chooser(stage=stage))
        prompt = await asyncio.create_task(self.prompt_chooser(stage=stage))

        if self.extra_context == False:
            chain = prompt | self.llm_conversation
            return chain

        elif self.extra_context == True:          # Add a retriever to the chain with
            the extra context obtained...

    except Exception as e:
        print(f"chain_builder Error {e}")
        return False
```

Figura 8 – Método “chain\_builder” da classe “Utils”. Observe que há uma lógica para adição do retriever à chain a ser retornada pelo método (ver seção “3.1.1. Contexto extra” do documento).

Uma vez que o *looping* é quebrado, a função retorna o dicionário response à função “main”, que por sua vez salva a resposta do assistente e retorna para a API a resposta à solicitação recebida pelo *frontend* (Figura 9).

```

76  ✓   async def get_assistant_response(self, user_request: dict = {})
77  >   -> dict:
86  >   try:|...
243
244  ✓   except Exception as e:
245       print(f"Chat get_assistant_response() Error {e}")
246
247       message = "I'm not feeling ok... Would you mind if we
248       talk another time?"
249       response = {"message": message, 'orientation': self.
250       orientation, 'current_stage': 'error'}
251
252   finally:
253       # self.debugger_print(response)
254       self.set_user_attributes(response)
255       return response

```

Figura 9 – Método “get\_assistant\_response” da classe “Conversation”. Observe que este método, assim como outros do projeto, é uma função assíncrona, o que permite implementar uma lógica de paralelismo a fim de aumentar a performance do projeto.

### 3.1.1. Contexto extra

Dentro dos desafios com a criação dinâmicas de *prompts*, surge a necessidade de segregar as informações usadas pela LLM para gerar respostas naturais ao usuário.

Neste sentido, a lógica foi estruturada de modo a poder adicionar informações dinamicamente às *chains* a serem invocadas nos diferentes estágios. Sendo assim, foram desenvolvidas duas classes para extração e salvaguarda dos dados a partir do site da agência e uma segunda classe para vetorização dos dados.

A classe “WebScraper” (Figura 10) faz uma varredura do site da agência e organiza por página em ficheiros JSONs (veja o exemplo de *output* na Figura 11). Esses ficheiros depois serão usados pela classe “VectorStoreBuilder” (Figura 12) para vetorização desses dados extraídos.

```

backend > assistant > knowledge > 📄 web_scraper.py > ...
1  import os
2  import requests
3  from bs4 import BeautifulSoup
4  import json
5  from urllib.parse import urlparse
6
7  class WebScraper:
8      """
9      A class to scrape data from web pages and save it to JSON files.
10     """
11
12 >     def __init__(self, base_url: str, sitemap_url: str, save_folder: str) -> None: ...
13
14 >     def fetch_sitemap(self) -> list: ...
15
16 >     def extract_data_stable(self, url: str) -> dict: ...
17
18 >     def extract_data(self, url: str) -> dict: ...
19
20 >     def save_to_json(self, data: dict, url: str) -> None: ...
21
22 >     def process_url(self, url: str) -> None: ...
23
24     def process_site(self) -> None:
25         """Process all URLs from the sitemap."""
26         urls = self.fetch_sitemap()
27         for url in urls:
28             self.process_url(url)
29
30
31 def web_scraper_start(base_url='https://kobu.agency/',
32                      sitemap_url='https://kobu.agency/sitemap.xml',
33                      save_folder='assistant/knowledge/web_scraper_files') -> None:
34     """
35     Initialize and run the web scraper with the given optional parameters.
36
37     Parameters:
38     - base_url (str): The base URL of the website.
39     - sitemap_url (str): The URL of the sitemap.
40     - save_folder (str): The folder to save scraped data.
41     """
42     try:
43         scraper = WebScraper(base_url, sitemap_url, save_folder)
44         scraper.process_site()
45         print("Web scraping completed successfully.")
46     except Exception as e:
47         print(f"An error occurred: {e}")
48         print(f"The path '{save_folder}' has not been changed.")
49
50
51 if __name__ == "__main__":
52     web_scraper_start()
53
54

```

Figura 10 – Ficheiro “web\_scraper.py” com a função “web\_scraper\_start” em evidência. Podemos ver que a segregação do conteúdo gerado é feita através do Site Map.



```

1 {
2   "title": "'I Am Monchique', the Film | KOBU Agency Case Study",
3   "content": "A promotional film that goes straight to the source of Água Monchique, a mineral and alkaline water, portraying its springs and the awe-inspiring natural scenery that surrounds them in Monchique. The brand narrative for Água Monchique is as connected to the product itself as it is to its natural authentic origin. For this project, which resulted in a promotional film recorded amidst the beautiful natural landscapes in Monchique, we wanted to convey the authenticity of the scenery, visiting secret places in the Algarve where this alkaline mineral water runs in its purest form. 'I Am Monchique' brings you along on an introspective journey through the pure, untouched nature of these lands - a place of pristine green and splendour where this unique spring alkaline water is born. The video was produced as part of a series of brand activations for international promotion. Client:Águas de MonchiqueAgency:KOBU Agency (Project Manager & Copywriter: Nuno Tenazinha)Production:Marcos Clímaco | Filmmaker & Friend (www.marcosclimaco.com)Filipe Correia (www.filipecorreia.net)Directed by:Marcos ClímacoFilipe CorreiaCinematography:Filipe CorreiaMarcos ClímacoAerial Footage:Marcos ClímacoEditor / Motion Graphics / 3D Product and Render:Marcos ClímacoVoice Over:Pedro SantosLocation Manager:Fábio CapelaProduction Assistants / Making ofRamiro Mendes (KOBU Agency)Jagoda Kondratiuk (2017 intern at KOBU Agency) more case studies\n",
4   "metadata": {
5     "url": "https://kobu.agency/case-studies/agua-monchique/"
6   }
7 }

```

Figura 11 – Exemplo de ficheiro JSON gerado pela classe “WebScraper” com o conteúdo útil extraído da página “<https://kobu.agency/case-studies/agua-monchique/>”

```

backend > assistant > knowledge > data_store_from_web_scraper.py > ...
1  import os
2  import json
3  from langchain_openai import OpenAIEmbeddings
4  from typing import List
5  from langchain_community.vectorstores.faiss import VectorStore, Document, Embeddings, FAISS
6  from dotenv import load_dotenv
7  load_dotenv()
8
9
10 class VectorStoreBuilder:
11     """
12     A class to build a vector store from a folder containing JSON documents.
13     """
14
15 > def __init__(self, json_folder: str, embedding: Embeddings): ...
25
26 > def load_documents_with_no_metadata(self) -> List[Document]: ...
53
54 > def load_documents(self) -> List[Document]: ...
83
84 > def build_vector_store(self) -> VectorStore: ...
100
101
102 def get_vector_store(
103     json_folder='assistant/knowledge/web_scraper_files') -> VectorStore:
104     """
105     Main function to get the vector store.
106
107     Parameters:
108     - json_folder (str, optional): The folder containing JSON files to be used for building the vector store.
109     Defaults to 'assistant/knowledge/data_store_files/web_scraper_files'.
110
111     Returns:
112     - vector_store: The constructed vector store object or None if an error occurred.
113     """
114     try:
115         # Initialize the embedding model
116         embedding = OpenAIEmbeddings()
117
118         # Build the vector store
119         builder = VectorStoreBuilder(json_folder, embedding)
120         vector_store = builder.build_vector_store()
121
122         # Check if the vector store was built successfully
123 > if vector_store is not None: ...
125     else:
126         print("Error building the vector store.")
127 > except Exception as e: ...
129     finally:
130         print("Program completed.")
131         return vector_store
132
133
134 if __name__ == "__main__":
135     get_vector_store()

```

Figura 12 – Ficheiro “data\_store\_from\_web\_scraper.py” com a função “get\_vector\_store” em evidência. Podemos ver que a fonte dos dados são os ficheiros extraídos pela classe “WebScraper”.

Quando a API está em funcionamento, existe uma variável “vector\_store” da classe “KnowledgeLoaders” que fica disponível na memória para ser usado pelo método “chain\_builder” da classe “Utils” (Figura 13). Vamos verificar como o vetor de dados torna-se um *retriever* a ser invocado na *chain*.

```

backend > assistant > tools > 🛠️ utils.py > 📁 Utils > 📄 chain_builder
13 class Utils(Prompts):
14
15     async def chain_builder(self, stage: str = '') -> object:
16         """
17         """
18         print("chain_builder starts")
19
20         try:
21             # prompt = await asyncio.create_task(self.prompt_chooser(stage=stage))
22             prompt = await asyncio.create_task(self.prompt_chooser(stage=stage))
23
24             if self.extra_context == False:
25                 chain = prompt | self.llm_conversation
26                 return chain
27
28             elif self.extra_context == True: # Add a retriever to the chain with the extra context obtained
29                 print("self.extra_context == True")
30
31                 chain = create_stuff_documents_chain(
32                     llm=self.llm_retriever,
33                     prompt=prompt
34                 )
35
36                 retriever = self.vector_store.as_retriever(search_kwargs={"k": self.search_kwargs})
37
38                 retriever_prompt = ChatPromptTemplate.from_messages([
39                     # MessagesPlaceholder(variable_name="chat_history"),
40                     ("human", """"Given the above conversation, generate a search query to look up
41                     in order to get information relevant to the conversation"""),
42                     ("system", "Messages History: {chat_history}"),
43                     ("human", "{input}")
44                 ])
45
46                 history_aware_retriever = create_history_aware_retriever(
47                     llm=self.llm_retriever,
48                     retriever=retriever,
49                     prompt=retriever_prompt
50                 )
51
52                 # history_aware_retriever_response = self.chain_invoker(history_aware_retriever) # Verify context added
53                 # print("History_aware_retriever:\n", history_aware_retriever_response)
54
55                 retrieval_chain = create_retrieval_chain(
56                     # retriever,
57                     history_aware_retriever,
58                     chain
59                 )
60
61                 return retrieval_chain
62
63             except Exception as e:
64                 print(f"chain_builder Error {e}")
65                 return False

```

Figura 13 – Método “chain\_builder” da classe “Utils”.

Observe que quando o atributo “extra\_context” tem valor “True”, a *chain* a ser retornada passa a conter em um *retriever*, que nada mais é que outra *chain* a ser invocada. Ao executarmos esse objeto pela função “as\_retriever” da biblioteca LangChain, o vetor de dados passa a se comportar como um objeto do tipo “VectorStoreRetriever”, passível de ser executado dentro de uma *chain* pelos métodos competentes da biblioteca LangChain.

Neste sentido, observe também que a informação selecionada é selecionada por uma LLM que seleciona “k” (valor padrão 3) documentos contidos no vetor de dados a serem usado como contexto extra pela *chain* principal.

### 3.2. O *frontend*

O *frontend* foi desenvolvido em HTML e JavaScript. A comunicação entre e *backend* acontece via requisições de API, e o tratamento da interface gráfica é todo concentrado no *frontend*.

Os *scripts* foram segregados em módulos, pelo que cada módulo possui uma responsabilidade no que desrespeito à estrutura da interface, conforme mostra a Figura 14.

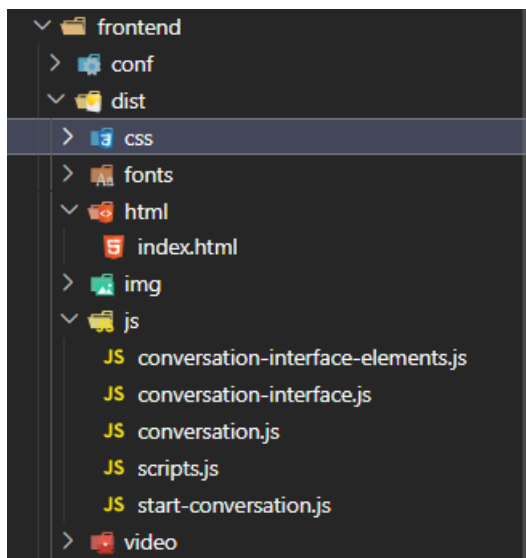


Figura 14 – Esquema do repositório frontend do projeto. Os módulos JavaScript seguem o padrão ES6 (ECMAScript 6).

As classes são descritas abaixo.

- **“InterfaceElements”**: Representa os elementos dinamicamente criados da interface gráfica (mensagens do assistente, mensagens do usuário, sugestões de input etc.);
- **“Interface”**: Controla a exibição dos elementos dinâmico na página;
- **“Conversation”**: Representa os controladores das interações do usuário com a interface depois da primeira interação com a plataforma. Possui métodos para, por exemplo, tratar as respostas recebidas pela API, enviar requisições à API, entre outras.
- **“StartConversation”**: Responsável por inicializar o usuário e criar uma instância da conversa. Também contém o método *main* que concentra toda lógica para reação da interface, tratamento e obtenção da resposta ao *input* do usuário (*screenshot* da classe na Figura 15).

```

frontend > dist > js > JS start-conversation.js > StartConversation > initInterfaceElements
14 export class StartConversation {
15 >   constructor() { ...
21   }
22
23   async main() {
24     this.conversation.showSpinner();
25     const inputElement = this.userInput;
26     inputElement.blur();
27
28     switch (this.conversation.currentStage) {
29       case undefined:
30       case '':
31       case null:
32         case this.conversation.WELCOME_STAGE:
33           await this.handleWelcomeStage();
34           // break;
35         case this.conversation.CHOOSE_SUBJECT_STAGE:
36           await this.handleChooseSubjectStage();
37           break;
38         default:
39           await this.handleDefaultStage();
40       }
41     }
42     // Stage Handlers
43 >   async handleWelcomeStage() { ...
52   }
53
54 >   async handleChooseSubjectStage() { ...
64   }
65
66   async handleDefaultStage() {
67     this.conversation.setUserResponse();
68     const request = this.conversation.requestData();
69     const inputElement = this.userInput
70     inputElement.value = ''
71     const response = await this.conversation.sendRequest(request);
72     if (response.message === false) return;
73     await this.conversation.assistantResponseHandler(response);
74     this.userInput.placeholder = 'Type a message';
75     console.log("Main: finish default()", this.conversation.currentStage);
76   }
77 }

```

Figura 15 – Classe “StartConversation” do módulo “start-conversation.js”. Observe que existem 3 métodos para distintos estágios: “handleWelcomeStage”, “handleChooseSubjectStage” e “handleDefaultStage” (para os demais estágios). Note também que a passagem do estágio “WELCOME\_STAGE” e “CHOOSE\_OBJECT\_STAGE” é automática.

A comunicação do *frontend* com a API do *backend* do é feito por meio de requisição HTTP. Ao carregar a página, a classe “StartConversation” é instanciada e passam a ficar ativo os escutadores de eventos da interface gráfica. Ao abrir o chat, é enviado uma requisição à API.

As requisições recebidas são tratadas pelo *frontend* a depender da chave “current\_stage” recebida.

Parte do código das classes do *frontend* estão ilustrados pelas Figuras 16 e 17 abaixo.

```

frontend > dist > js > JS conversation-interface-elements.js > ...
1  /**
2   * Represents elements for the conversation interface of a user and the AI assistant.
3   */
4  export class InterfaceElements {
5      constructor() {
6          // Possible orientations values
7          this.NEXT_STAGE = 'next_stage';
8          this.VERIFY_ANSWER = 'verify_answer';
9          this.PROCEED = 'proceed';
10         // Possible stages values
11         this.WELCOME_STAGE = 'welcome';
12         this.CHOOSE_SUBJECT_STAGE = 'choose_subject';
13         this.DATA_COLLECTING_STAGE = 'data_collecting';
14         this.RESUME_VALIDATION_STAGE = 'resume_validation';
15         this.SEND_VALIDATION_STAGE = 'send_validation';
16         this.FREE_CONVERSATION_STAGE = 'free_conversation';
17     }
18
19     // Assistant message element creation
20 > createMessageAssistantElement(content) { ...
28     };
29
30     // User Options element creation
31 > createSuggestionAssistantElement(content) { ...
39     };
40
41     // User message element creation
42 > createMessageUserElement(content) { ...
50     };
51
52     // Scroll to the bottom of the results div
53 > scrollToBottomOfResults() { ...
59     };
60
61     // Ascii Spinner
62 > showSpinner() { ...
70     };
71 > hideSpinner() { ...
77     };
78 }

```

Figura 16 – Classe “InterfaceElements” de onde são gerados os elementos de mensagem do usuário, do assistente, além da aparição e ocultação do spinner.

```

frontend > dist > js > JS conversation-interface.js > ...
1  import { InterfaceElements } from './conversation-interface-elements.js';
2
3  /**
4   * Represents an interface for managing conversation between the user and the AI assistant.
5   * This class inherits elements from the InterfaceElements class and provides methods for interface manipulation.
6   */
7  export class Interface extends InterfaceElements {
8      constructor() {
9          super();
10         this.count = 0;
11     }
12     userInput = () => document.getElementById('user-input').value;
13
14     /**
15      * Opens the chatbox by displaying its content and optionally executing a main function.
16      * @param {Function} main - The main function to execute after opening the chatbox.
17      */
18     > async openChat(main) { ...
26     }
27
28     > async animateChatItems(main, chat) { ...
47     }
48
49     /**
50      * Animates the fadeIn effect for an HTML element by gradually increasing its opacity.
51      * @param {HTMLElement} element - The HTML element to apply the fadeIn effect to.
52      */
53     > async fadeIn(element) { ...
66     }
67
68     > formatAssistantMessage(message) { ...
82     };
83
84     // Set user response in messages-container
85     > async setUserResponse(user_input = this.userInput()) { ...
95     };
96
97     // Set assistant response in messages-container
98     > async setAssistantResponse(message) { ...
108     };
109
110     // Set assistant response in messages-container
111     > async setVideo() { ...
125     };
126
127     > async setAssistantSuggestion(options) { ...
142     };
143
144 };
145

```

Figura 17 – Classe “Interface” que estende a classe “InterfaceElements”.

#### 4. ANÁLISE CRÍTICA DO TRABALHO REALIZADO

Durante o desenvolvimento do assistente virtual, foram identificados vários desafios e oportunidades de aprendizagem.

É evidente que as unidades curriculares (U.C.) ligadas à programação em Python (Programação e, principalmente, Complementos de Programação) foram essenciais como base de conhecimento para o desenvolvimento do projeto. O uso da biblioteca Flask para a produção da API, por exemplo, foi baseado nas aulas da U.C. de Complementos de Programação, ministradas durante o segundo semestre do CTesp (Curso Técnico Superior Profissional) de Sistemas e Tecnologias de Informação. Além disso, os conhecimentos adquiridos na disciplina de Ambientes de Desenvolvimento Colaborativos foram fundamentais para a criação, alimentação e manutenção do repositório Git do projeto.

A integração entre *frontend* e *backend* exigiu uma coordenação precisa para garantir a eficiência da comunicação via API. Além disso, o uso de modelos de linguagem da OpenAI e a necessidade de formatar e validar dados com performance satisfatória exigiram a exploração de formas avançadas de programação orientada a objetos.

A experiência proporcionou uma compreensão aprofundada das tecnologias utilizadas e espera-se que a implementação contribua significativamente para as operações da KOBU Agency. Como o projeto foi desenvolvido de maneira independente pelo estagiário, sem orientação prática a nível de programação, é de se esperar que algumas construções lógicas tenham espaço para melhorias.

Apesar de a implementação prática do projeto ter sido ultrapassada por outras questões prioritárias da agência, é esperado que o assistente virtual desenvolvido integre ao lançamento do novo site da KOBU Agency.



## 5. LISTA DE REFERÊNCIAS

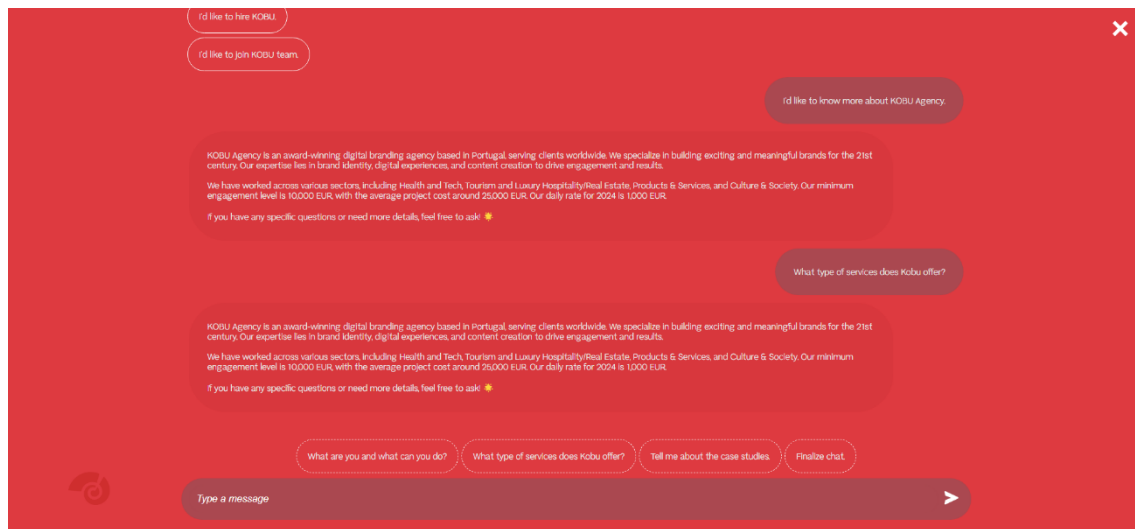
- 1) LangChain. *LangChain Documentation Introduction*. Disponível em:  
(<https://python.langchain.com/v0.2/docs/introduction/>). Consultado em:  
(20/02/2024)
- 2) OpenAI. *OpenAI API Reference - Chat*. Disponível em:  
(<https://platform.openai.com/docs/api-reference/chat/create>). Consultado em:  
(20/02/2024)
- 3) Raj Kapadia. *Connect Openai [ChatGPT] to your Website*. Disponível em:  
(<https://github.com/RajKKapadia/YouTube-Openai-Website>). Consultado em:  
(20/02/2024)

## 6. CONCLUSÃO

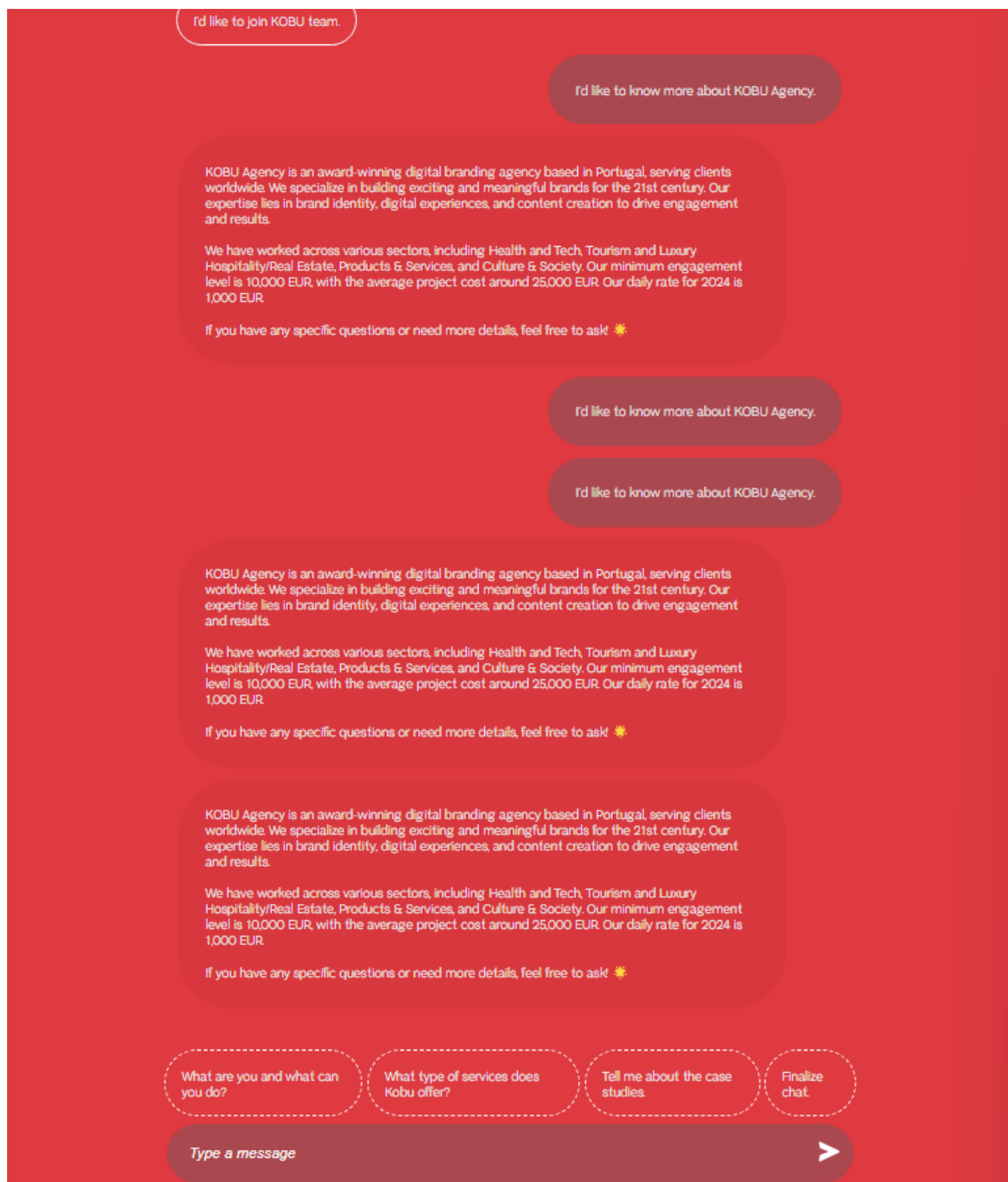
Este relatório apresentou uma visão abrangente do desenvolvimento do assistente virtual para a KOBU Agency. É esperado que o projeto não só aprimore a recepção de contatos da empresa, mas também que o projeto desenvolvido seja depois explorado para uso interno da empresa, uma vez que o mesmo foi estruturado de modo a ser facilmente adaptável a outros fins – por exemplo, o assistente pode ser adaptado para servir como um assistente de marketing ou para qualquer outro fim, uma vez que o contexto adicionado aos *prompts* são obtidos dinamicamente com base no conteúdo dos arquivos JSONs.

Além disso o estágio proporcionou uma valiosa experiência prática em desenvolvimento de software e integração de tecnologias avançadas. A análise crítica destacou os principais desafios e aprendizados, consolidando a importância do estágio para a formação profissional.

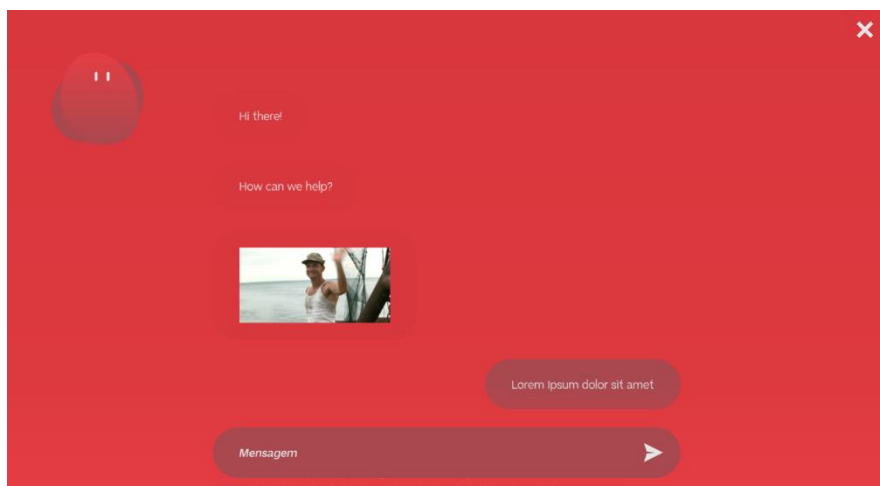
## 7. ANEXOS



Anexo 1 - Screenshot da interface gráfica desenvolvida (versão computador).



Anexo 2 - Screenshot da interface gráfica desenvolvida (versão computador).



*Anexo 3 - Screenshot da referência de interface gráfica para desenvolvida (versão computador).*