UAlg ISE
UNIVERSIDADE DO ALGARVE
INSTITUTO SUPERIOR DE ENGENHARIA

UAlg ESGHT
UNIVERSIDADE DO ALGARVE
ESCOLA SUPERIOR DE GESTÃO, HOTELARIA E TURISMO

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 conceção 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 *Backend* 5

3.1.1. Contexto extra 11

3.2. O *frontend* 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[2] (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 *prompt*s 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 *prompt*s 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 *prompt*s para obter respostas mais naturais e satisfatórias ao contexto do utilizador. Sendo assim, foi escolhido pré-definir os *prompt*s e escolhê-los a partir de duas variáveis essenciais: “subject” (assunto de contacto escolhido pelo usuário) “current_stage” (estágio atual da conversa).

*Prompt*s 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 *prompt*s são as entradas textuais que os usuários fornecem para obter informações, gerar texto, responder perguntas, entre outros.

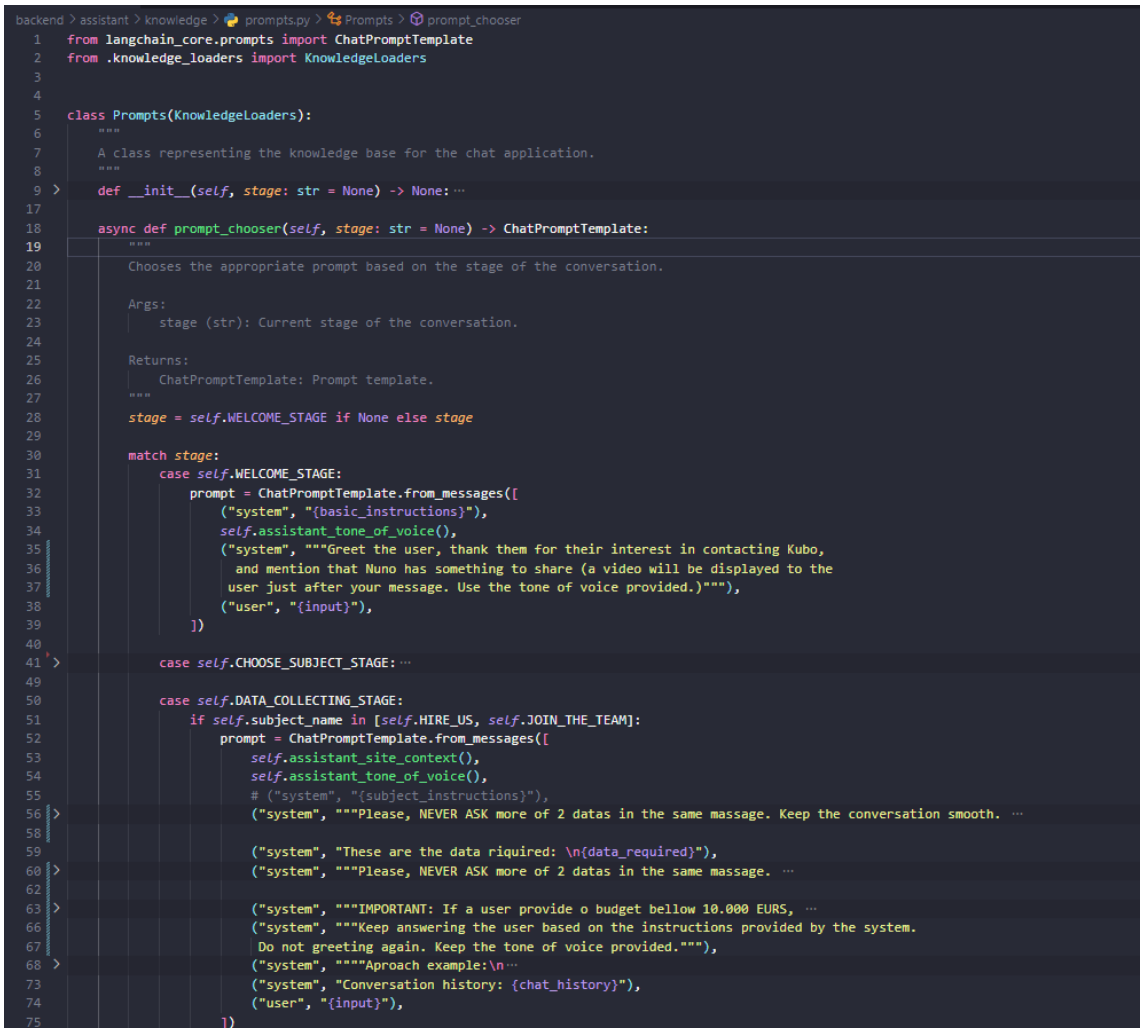


Figura 1 – Método prompt_chooser da classe Prompt, responsável por selecionar o prompt a ser enviado para a LLM. Os prompts são escolhidos de acordo com o estágio da conversa. Os placeholders {} são preenchidos com o conteúdo correspondente, armazenado em arquivos JSON.

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.

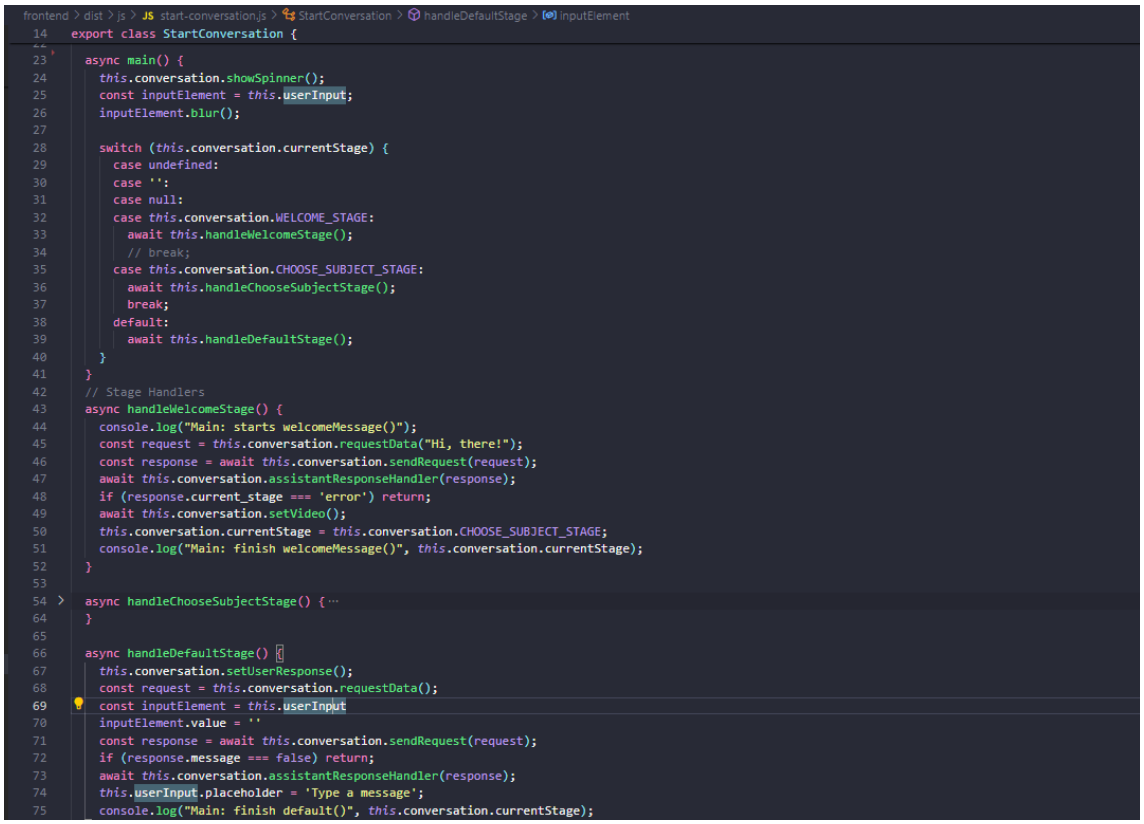


Figura 2 – Classe "StartConversation" do repositório frontend. Na imagem, podemos ver os métodos isolados que tratam diferentes estágios de conversação da interface gráfica.

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 messagem do usuário) e pode ou não ou 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.

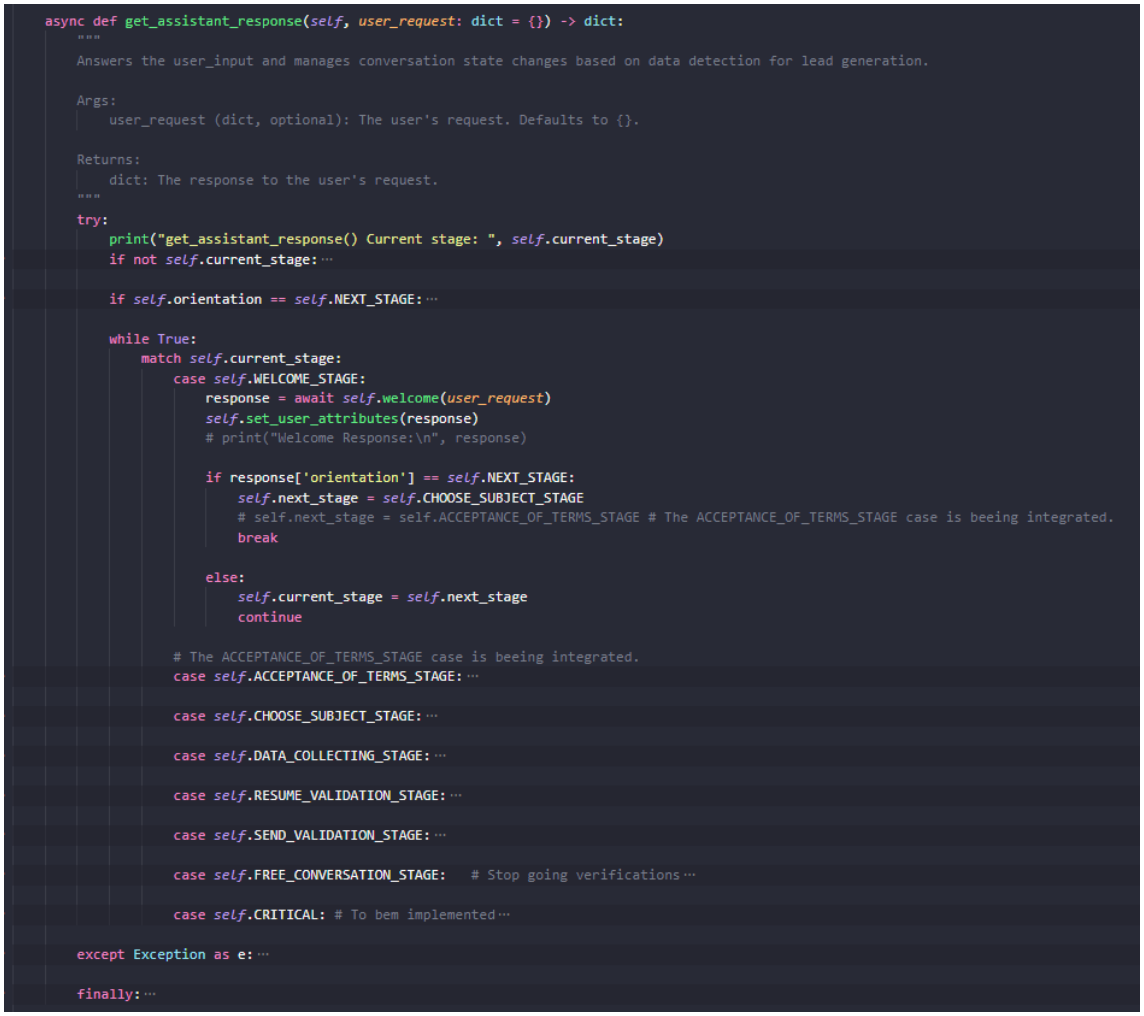
Figura 3 – Classe “ChatConsts” do repositório backend. Nesta classe também ficam salvos as opções passíveis de serem enviadas pelo backend para o frontend. No frontend, essas opções podem ser transformadas em botões dinâmicos.

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 destinta 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.

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.

```python
case self.DATA_COLLECTING_STAGE:
    validation = await asyncio.create_task(self.data_colecting_validation(user_request))
    # response = await asyncio.create_task(self.data_colecting_in_changing(user_request))
    response = await asyncio.create_task(self.data_colecting(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_colecting(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 destintas: uma para caso a validação seja prosseguir e outra para o caso da validação for para pular para o próximo estágio.

```python
189
190     async def data_colecting(self, user_request: dict) -> dict:
191         """Assistant to colect datas from the user."""
192         print("ResponseHandler: data_colecting()")
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_colecting() 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_colecting" concentra a logica 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_colecting_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[1] (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

```python
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 secção "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 = {})
      -> dict:
77        """...
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
              talk another time?"
248           response = {"message": message,'orientation': self.
              orientation, 'current_stage': 'error'}
249
250       finally:
251           # self.debugger_print(response)
252           self.set_user_attributes(response)
253           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 *prompt*s, 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 vectorização 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 vectorização desses dados extraídos.

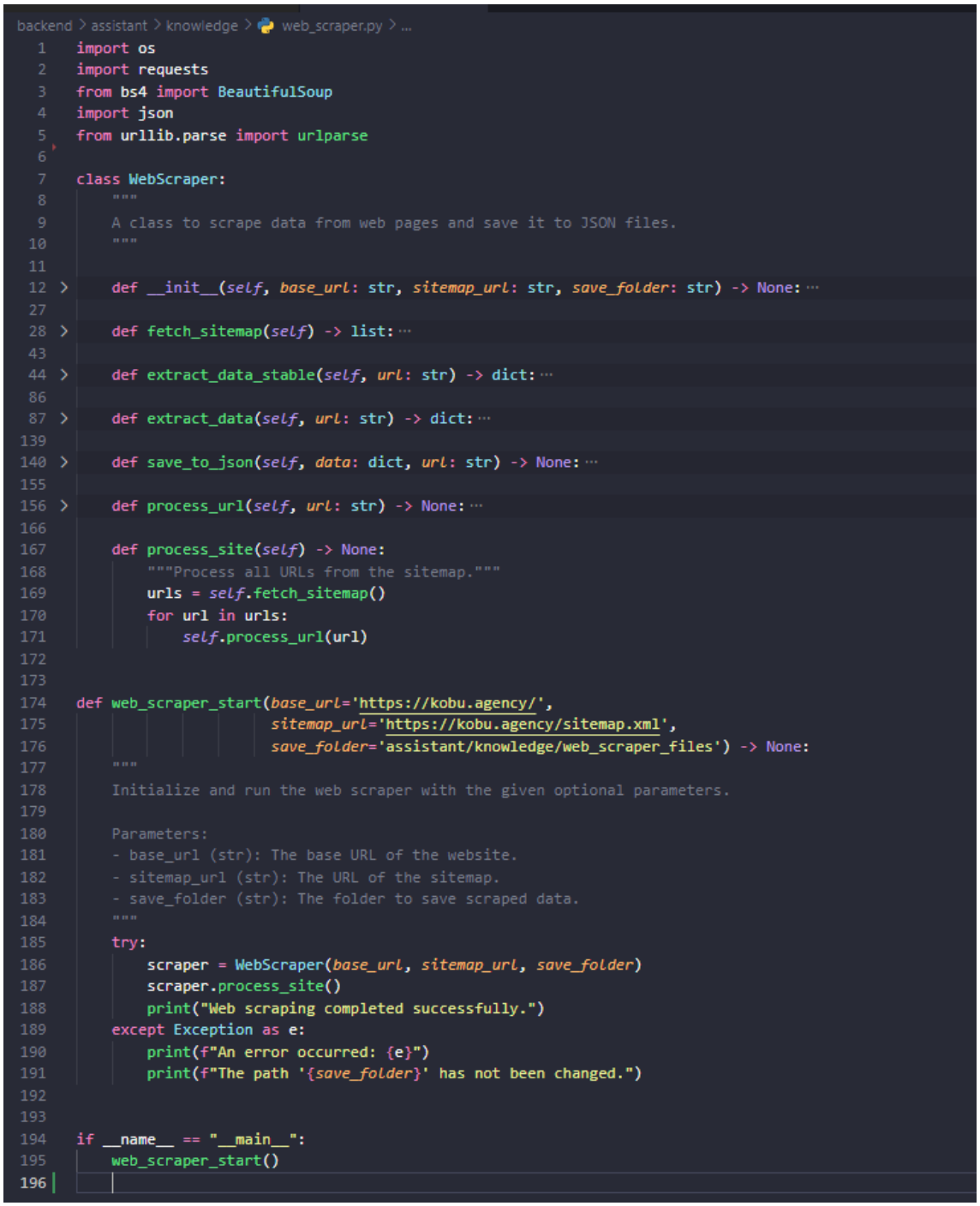


Figura 10 – Ficheiro "web_scraper.py" com a função "web_scraper_start" em evidência. Podemos ver que a segregação do conteúdo gerado é feita através do Site Map.

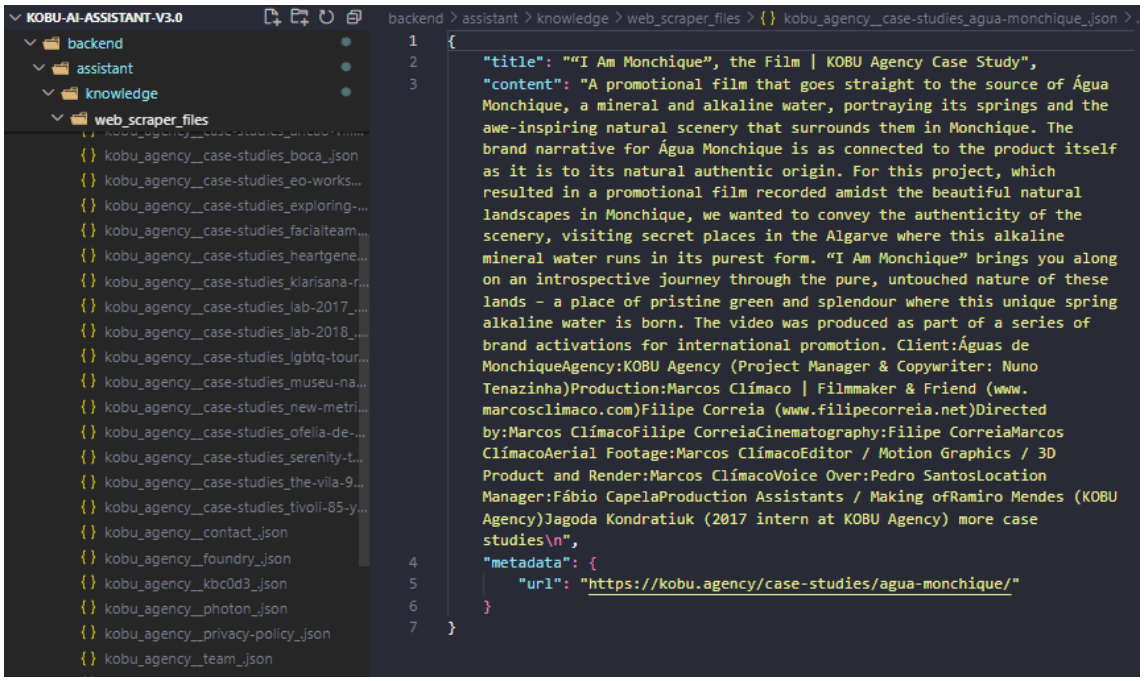


Figura 11 – Exemplo de ficheiro JSON gerado pela classe "WebScraper" com o conteúdo útil extraído da página ""https://kobu.agency/case-studies/agua-monchique/"

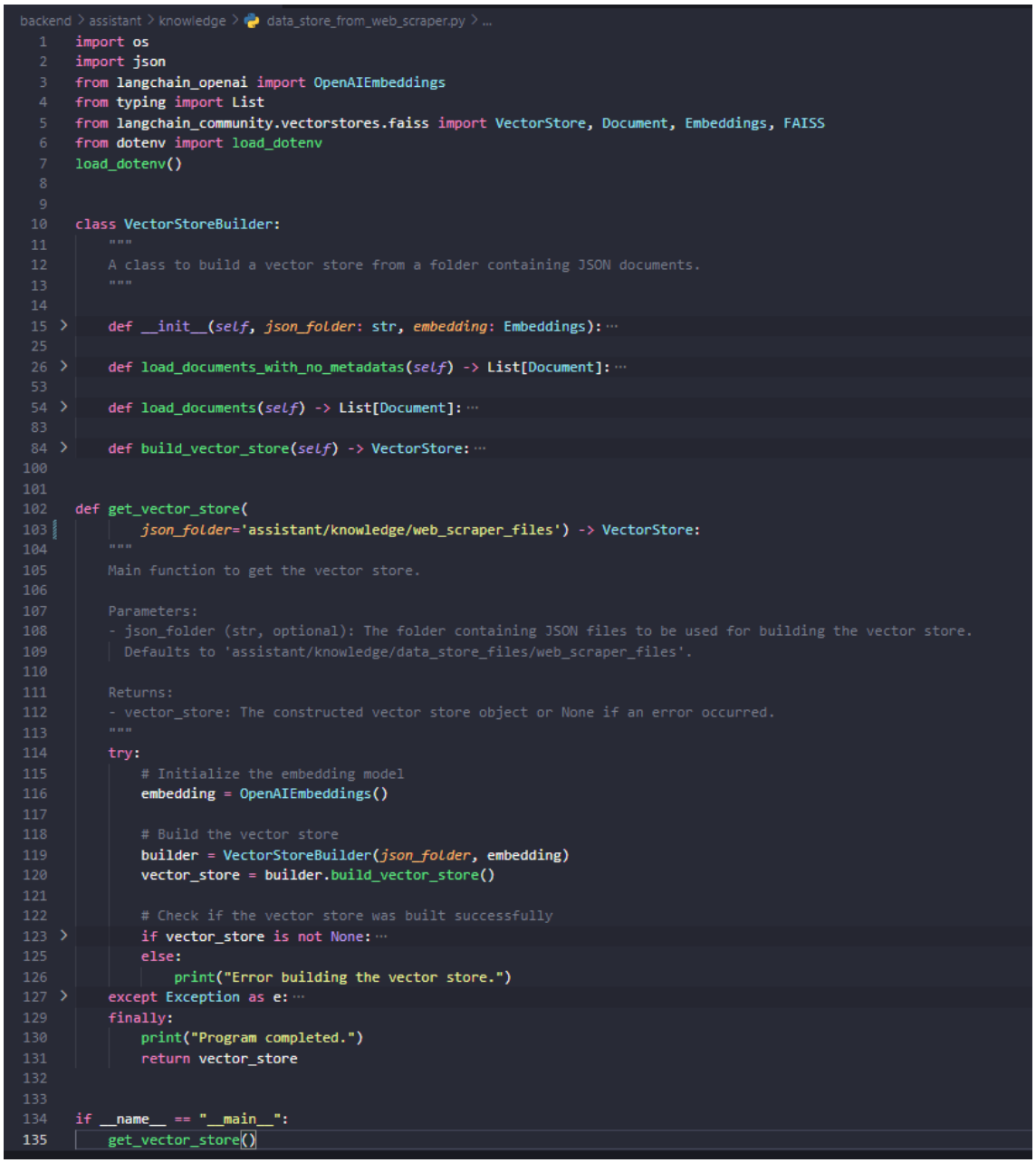


Figura 12 – Ficheiro "data_store_from_web_scraper.py" com a função "get_vector_store" em evidência. Podemos ver que a fonte dos dados são os ficheiros extraídos pela classe "WebScraper".

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*.

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.

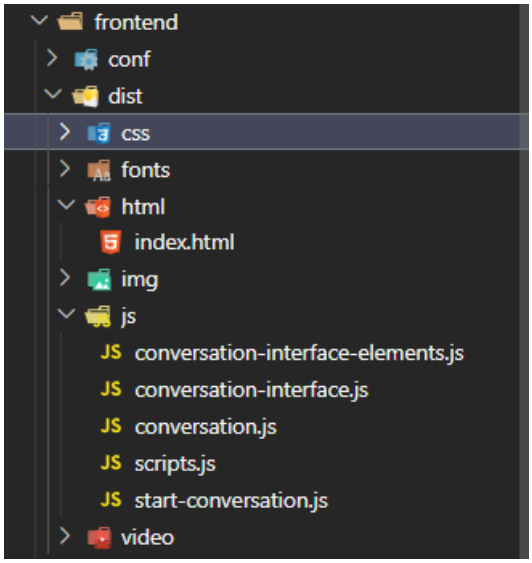


Figura 14 – Esquema do repositório frontend do projeto. Os módulos JavaScript seguem o padrão ES6 (ECMAScript 6).

As classes são descritas abaixo.

- "**InterfaceElements**": Representa os elementos dinamicamente criados da interface gráfica (mensagens do assistente, massagens 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).

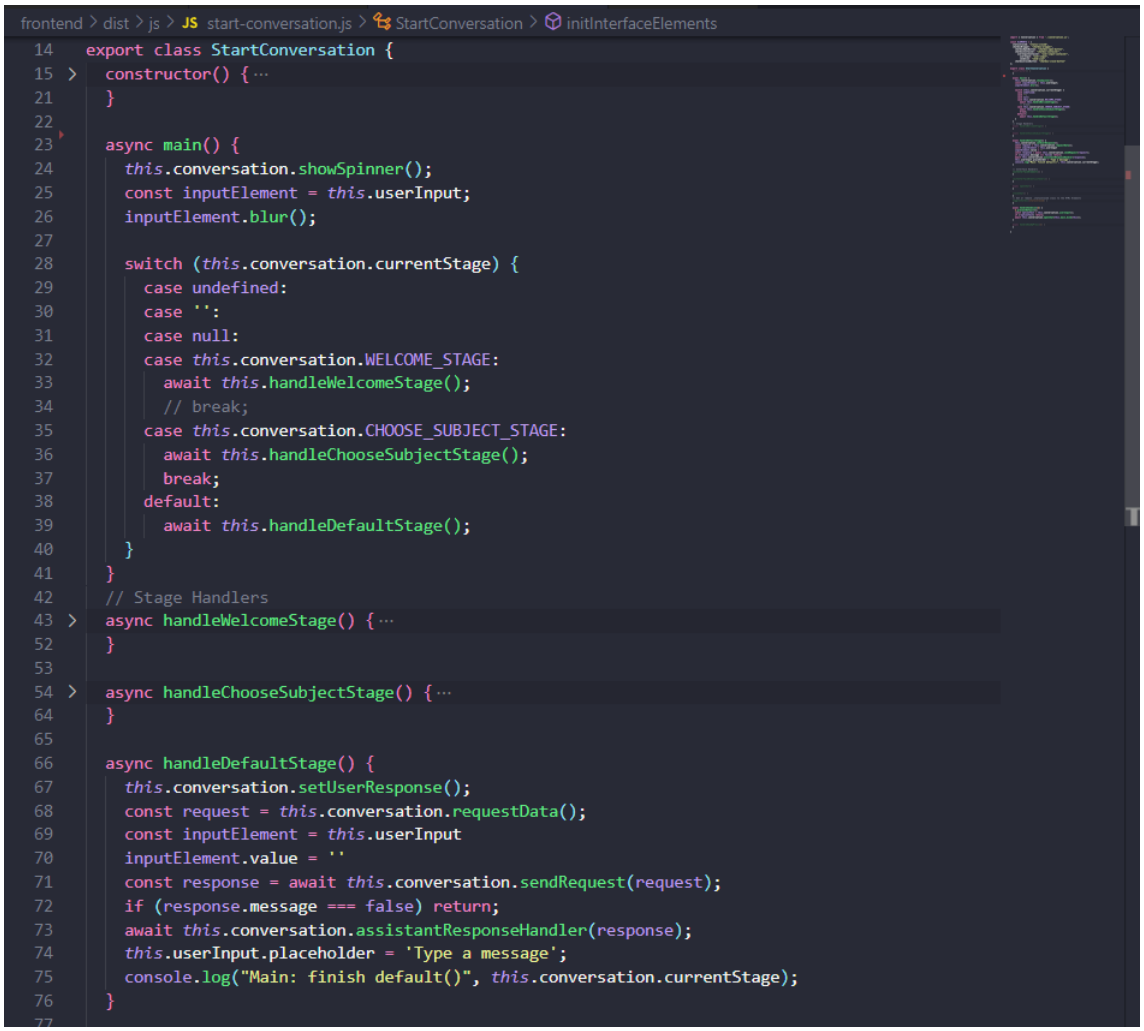


Figura 15 – Classe "StartConversation" do módulo "start-conversation.js". Observe que existem 3 métodos para destintos estágios: "handleWelcomeStage", "handleChooseSubjectStage" e "handleDefaultStage" (para os demais estágios. Note também que a passagem do estágio "WELCOME_STAGE" e "CHOOSE_OBJECT_STAGE" 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 > ...

```
/**
 * Represents elements for the conversation interface of a user and the AI assistant.
 */
export class InterfaceElements {
  constructor() {
      // Possible orientations values
      this.NEXT_STAGE = 'next_stage';
      this.VERIFY_ANSWER = 'verify_answer';
      this.PROCEED = 'proceed';
      // Possible stages values
      this.WELCOME_STAGE = 'welcome';
      this.CHOOSE_SUBJECT_STAGE = 'choose_subject';
      this.DATA_COLLECTING_STAGE = 'data_collecting';
      this.RESUME_VALIDATION_STAGE = 'resume_validation';
      this.SEND_VALIDATION_STAGE = 'send_validation';
      this.FREE_CONVERSATION_STAGE = 'free_conversation';
  }

  // Assistant message element creation
  createMessageAssistantElement(content) {···
  };

  // User Options element creation
  createSuggestionAssistantElement(content) {···
  };

  // User message element creation
  createMessageUserElement(content) {···
  };

  // Scroll to the bottom of the results div
  scrollToBottomOfResults() {···
  };

  // Ascii Spinner
  showSpinner() {···
  };
  hideSpinner() {···
  };
}
```

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

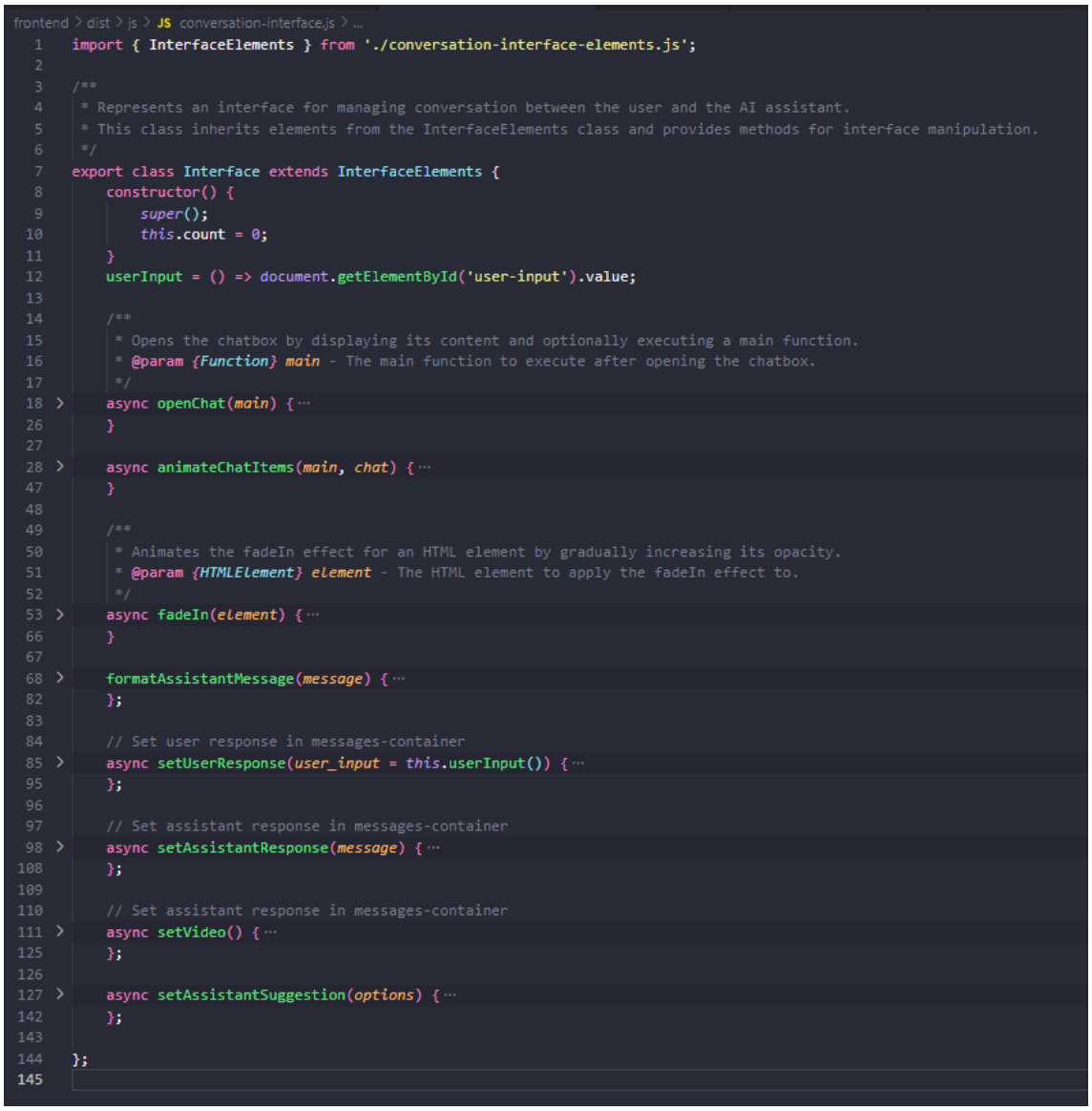


Figura 17 – Classe "Interface" que estende a classe "InterfaceElements".

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

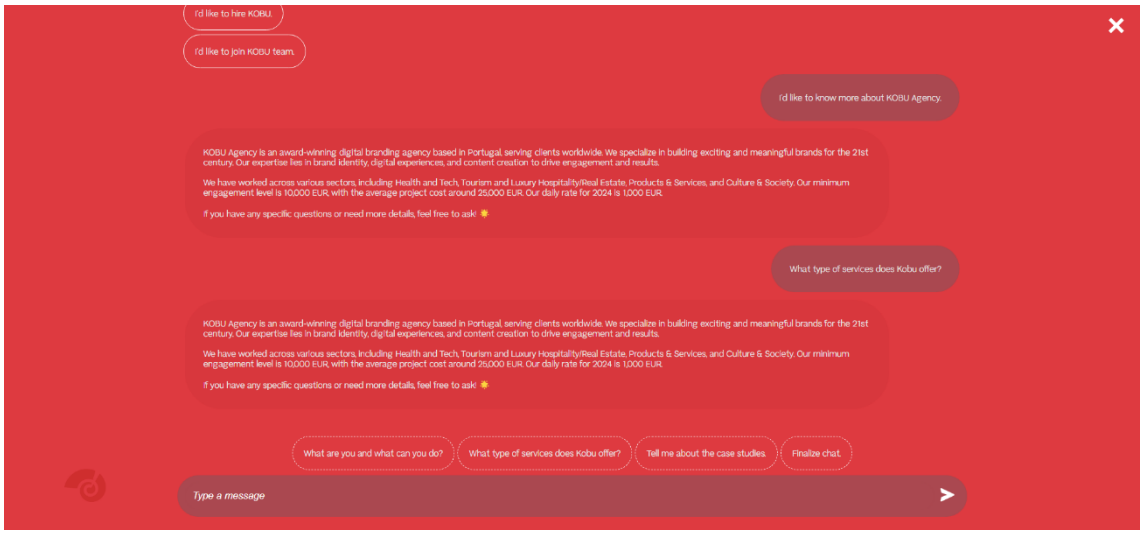
1) LangChain. *LangChain Documentation Introduction*. Disponível em: (https://python.langchain.com/v0.2/docs/introduction/). Consultado em: (20/02/2024)
2) OpenAI. *OpenAI API Reference - Chat*. Disponível em: (https://platform.openai.com/docs/api-reference/chat/create). Consultado em: (20/02/2024)
3)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

6. CONCLUSÃO

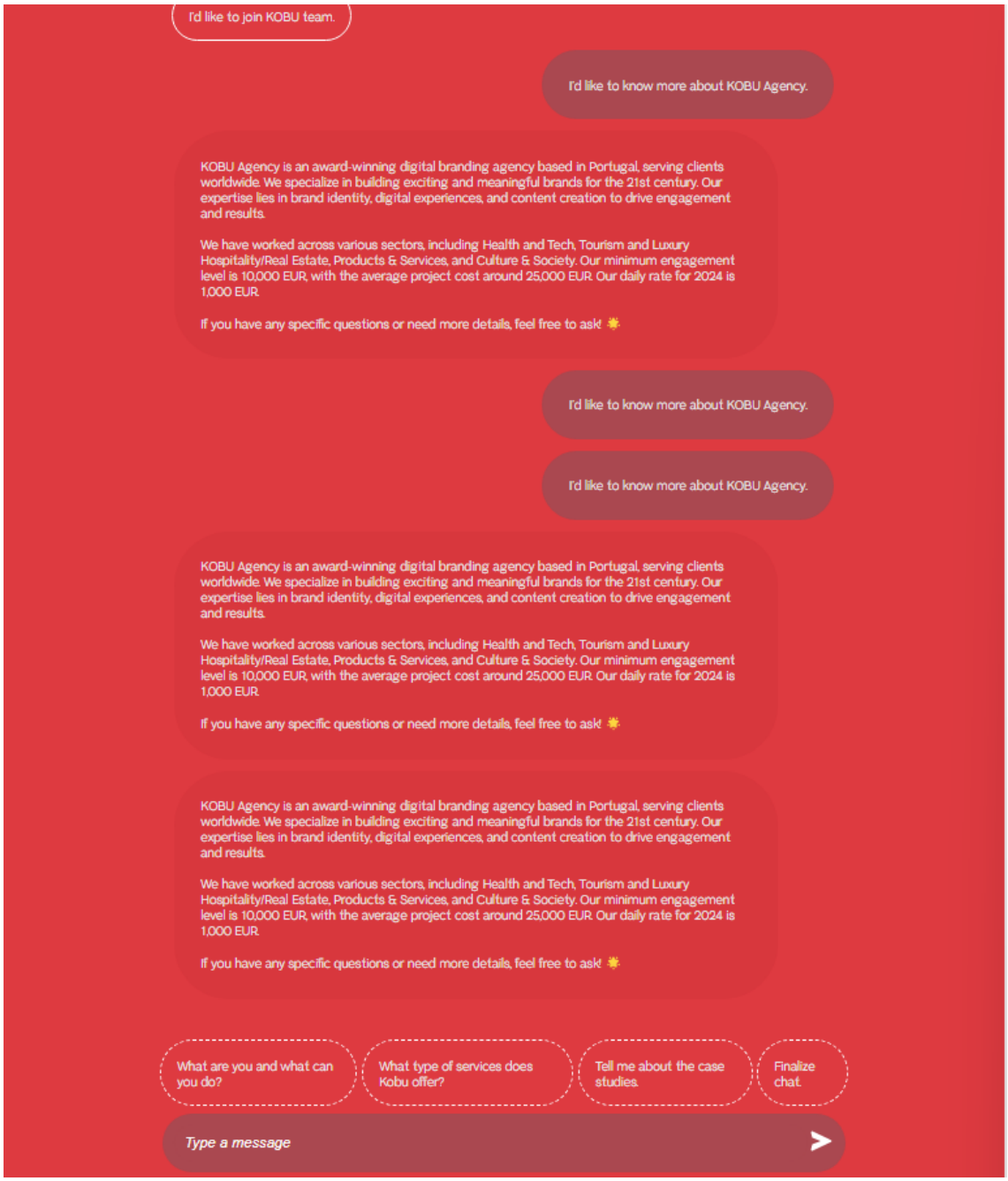
Este relatório apresentou uma visão abrangente do desenvolvimento do assistente virtual para a KOBU Agency. É esperado que o projeto não só aprimore a recepção de contatos da empresa, mas também que o projeto desenvolvido seja depois explorado para uso interno da empresa, uma vez que o mesmo foi estruturado de modo a ser facilmente adaptável a outros fins – por exemplo, o assitente pode ser adaptado para servir como um assistente de marketing ou para qualquer outro fim, uma vez que o contexto adicionado aos *prompt*s 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.

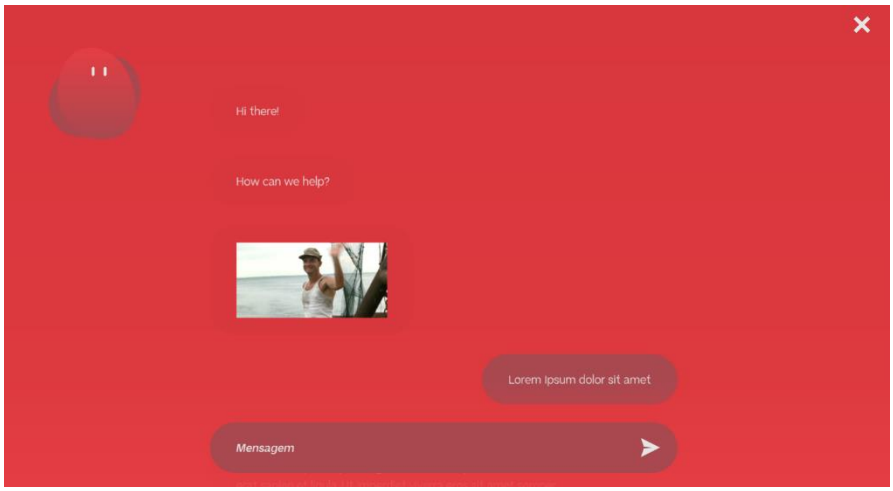
7. ANEXOS



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).