Ferramenta conversacional para captura de *design rationale* em diálogos sobre processos de desenvolvimento

Relatório de Projeto Final de Programação

Gabriel Diniz Junqueira Barbosa
(2020951)

Orientadores:
Hélio Côrtes Vieira Lopes
Clarisse Sieckenius de Souza

Sumário

1. Descrição do Projeto

Esse projeto consiste na concepção e implementação de uma ferramenta de conversa para capturar raciocínios de design ao longo de diálogos entre membros da equipe de desenvolvimento. A fim de ter uma captura mais estruturada, mensagens da conversa serão categorizadas de acordo com algum arcabouço organizacional.

Dada a importância da comunicação nos processos de desenvolvimento, logs de conversas podem ser valiosos para análises sobre o processo em si. Normalmente, estas mensagens são capturadas de forma pouco estruturada, impedindo uma categorização fácil. O objetivo do sistema construído neste projeto é permitir que tenhamos conversas estruturadas de acordo com alguma categorização (teoria) prévia, garantindo que os dados coletados já tenham algum nível de organização.

Uma dificuldade presente nesta proposta é o aspecto de interação da conversa. Como as categorias dependem de interações dos usuários, a qualidade dos dados resultantes pode ser muito variada. Basta um usuário selecionar uma única categoria e continuar a conversa alheio à categorização. Outra possibilidade seria também o esquecimento deste aspecto do sistema, fazendo com que mensagens que deveria pertencer a categorias diferentes permanecessem em uma única categoria devido à falta de categorização por parte do usuário. Tentamos limitar essa possibilidade através de indícios na interface de que esse aspecto da interação está presente e é crucial.

O código do projeto está disponível em https://github.com/GabrielDJB/INF2102-PFP, em um repositório aberto *Git*.

2. Especificação do Projeto

2.1. Especificação de Requisitos

Antes de começar a conceber a ferramenta implementada, construí uma especificação de requisitos do sistema. Estes foram separados em requisitos funcionais e requisitos não funcionais, devidamente codificados de acordo com a categoria a que pertencem.

2.1.1. Requisitos Funcionais

Estes são os requisitos relacionados a funcionalidades que deverão estar disponíveis no sistema final. Para este projeto, os requisitos foram:

[*FR1*] – Usuários podem enviar mensagens em uma conversa com outros usuários.

[*FR2*] – Usuários podem definir o tópico (categoria) que está sendo discutido no momento.

[*FR3*] – Usuários devem conseguir identificar quem enviou uma certa mensagem.

[*FR4*] – Usuários devem conseguir identificar qual é o tópico relacionado a cada mensagem.

[*FR5*] – Usuários devem conseguir identificar quando uma mensagem foi enviada.

[*FR6*] – Usuários devem conseguir acessar os chats através de "*magic links*" que fazem uma autenticação automática.

[*FR7*] – Usuários podem acessar diferentes conversas que participam, através de "*magic links*."

2.1.2. Requisitos Não Funcionais

[*NFR1*] – Informações persistentes do sistema devem ser armazenadas em um banco de dados não relacional.

[*NFR2*] – Sistema deve ser acessível pela internet, na forma de uma aplicação web.

[*NFR3*] – Usuários devem ser autenticados através de um *magic link*.

[*NFR4*] – Interface *front-end* deve ser separada das funcionalidades de *back-end*.

[*NFR5*] – Sistema deveria ter um tempo de resposta abaixo de 1 segundo, por interação.

Todos esses requisitos foram satisfeitos no artefato final. Um usuário é capaz de acessar uma conversa, de forma autenticada, através de um *magic link*, onde ele pode engajar com outros usuários em uma conversa categorizada de acordo com uma categorização prévia. Utilizando diferentes links, consegue acessar diferentes conversas.

Requisitos não funcionais também foram respeitados, com uma implementação que separa a interface de *front-end* dos serviços de *back-end*, com resposta abaixo de 1 segundo para qualquer interação. O sistema, em seu modo de desenvolvimento, pode ser ativado através de um *batch file* (*deploy.bat*) e acessado através de qualquer browser, na porta 3000. É importante lembrar que o acesso precisa ser feito através de um *magic link*, de tal forma que o sistema possa autenticar o usuário e preparar o ambiente para a conversa estruturada.

2.2. Tecnologias Utilizadas

Em termos de tecnologias utilizadas, o sistema foi implementado em duas partes: um sistema *front-end* e um servidor *back-end*, seguindo uma lógica de serviços.

A implementação em *front-end* foi feita em *JavaScript*, usando a biblioteca *React*. O servidor de *front-end* foi configurado em *Node.js*. A fim de facilitar o processo de desenvolvimento da interface, utilizei uma biblioteca de componentes prontos, que seguem os princípios de *Material Design* da *Google*. Assim, questões estéticas podem ser resolvidas com pequenas alterações.

A implementação do servidor *back-end* se baseou em *Flask*, biblioteca da linguagem de programação *Python*. Em termos de persistência de dados, o sistema utiliza um banco de dados não relacional *MongoDB*. O controle programático desse banco, feito em *Python*, se deu através da biblioteca *Pymongo*.

2.3. Modelagem de Domínio

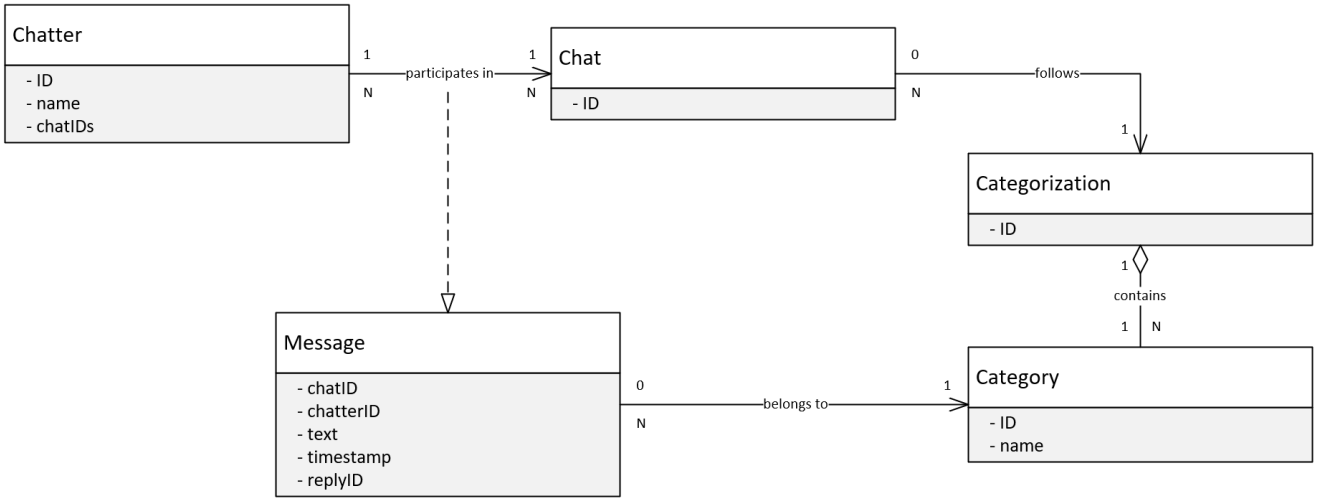


Figura 1 - Modelo de Domínio

Em termos do domínio da aplicação, podemos identificar as entidades e os relacionamentos descritos no modelo acima. Cada participante pode fazer parte de uma ou mais conversas, que por sua vez podem ter um ou mais participantes. Nestas conversas, participantes podem enviar mensagens que pertencem a uma categoria da categorização atribuída à conversa. Categorias fazem parte de categorizações. Cada conversa está relacionada a uma categorização.

Cada entidade possui também um conjunto de atributos que a caracterizam. Nessa fase, mapeamos cada um deles, sem nos preocuparmos com questões de formato ou tipo. Esses atributos também são essenciais para caracterizar as relações entre as entidades, que serão representadas de diferentes formas no banco de dados.

Tendo esse entendimento do domínio, podemos passar a modelar e planejar a aplicação em si, focando em aspectos mais técnicos da implementação.

2.4. Modelagem de Banco de Dados

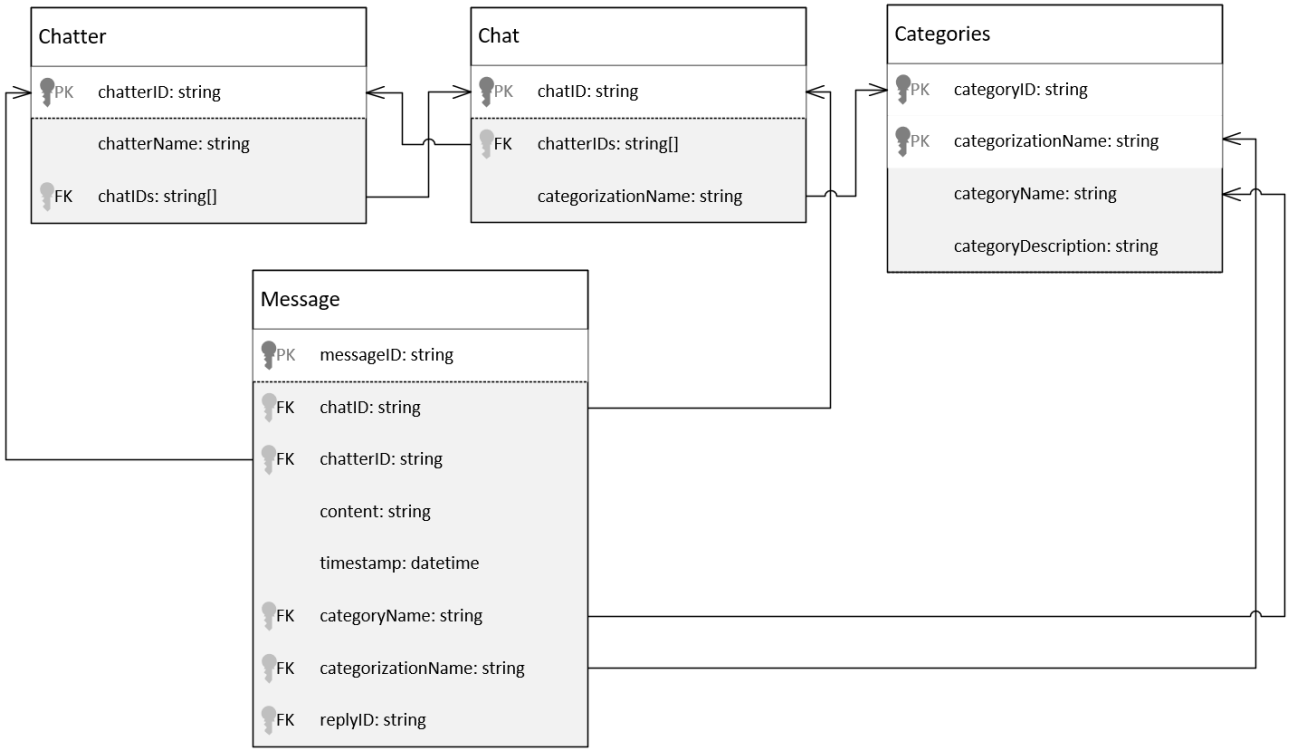


Figura 2 - Modelo do Banco MongoDB

O modelo acima descreve o esquema do banco de dados utilizado. Como usamos o *MongoDB*, não teremos uma estrutura relacional, mas mesmo assim precisamos estar cientes das relações entre chaves e atributos de cada uma dessas coleções, a fim de garantir a consistência dos dados.

Comparando com o modelo de domínio, podemos observar que há uma grande semelhança, como normalmente ocorre entre esses tipos de modelagem. Participantes (conhecidos como *chatters*) possuem seus identificadores e estão relacionados a múltiplos *chats*, que por sua vez estão ligados às categorizações (*categories*) que disponibilizam aos usuários. A principal entidade dessas conversas é a mensagem (*message*), que depende de todas as outras entidades. Elas são enviadas por usuários, pertencem a chats e estão associadas a uma das categorias da categorização disponível.

Cada uma dessas coleções deve ser manipulada através de simples operações *CRUD* (*Create*, *Retrieve*, *Update*, *Delete*). Essas são as operações básicas para todas as funcionalidades descritas na seção de requisitos (seção 2.2.). Como mencionado na seção de tecnologias utilizadas (seção 2.1.), essas operações serão acessíveis pelo servidor *Flask*, utilizando a biblioteca *PyMongo* para interagir com o banco de dados.

Além de operações *CRUD* básicas, o sistema também possui dois outros serviços para coleta de diversos documentos de uma mesma classe: *retrieve_chat_messages* e *retrieve_categorization*. O

primeiro, ao receber um identificador de *chat*, retorna todas as mensagens dessa conversa. Esse é o serviço utilizado para manter o display de mensagens atualizado. O segundo serviço recebe como *input* um *categorizationName* e retorna todas as categorias que pertencem a essa categorização, de forma a compor as categorias que serão atribuídas às mensagens nas conversas.

Além da estrutura descrita no modelo, temos também uma coleção auxiliar de autenticação, que indexa instâncias de *chatters* em determinados *chats* com uma chave *hash*. Essa chave é a utilizada nos *magic links* de acesso, uma vez que a autenticação busca por uma entidade com essa chave e obtém os identificadores de *chat* e do *chatter* corrente, ou seja, o usuário. Com essas informações, podemos construir todas as interações do ponto de vista do usuário que está acessando o sistema, reconhecendo quais foram as mensagens que ele enviou e quais foram enviadas por outros *chatters*.

2.5. Protótipo de Baixa Fidelidade

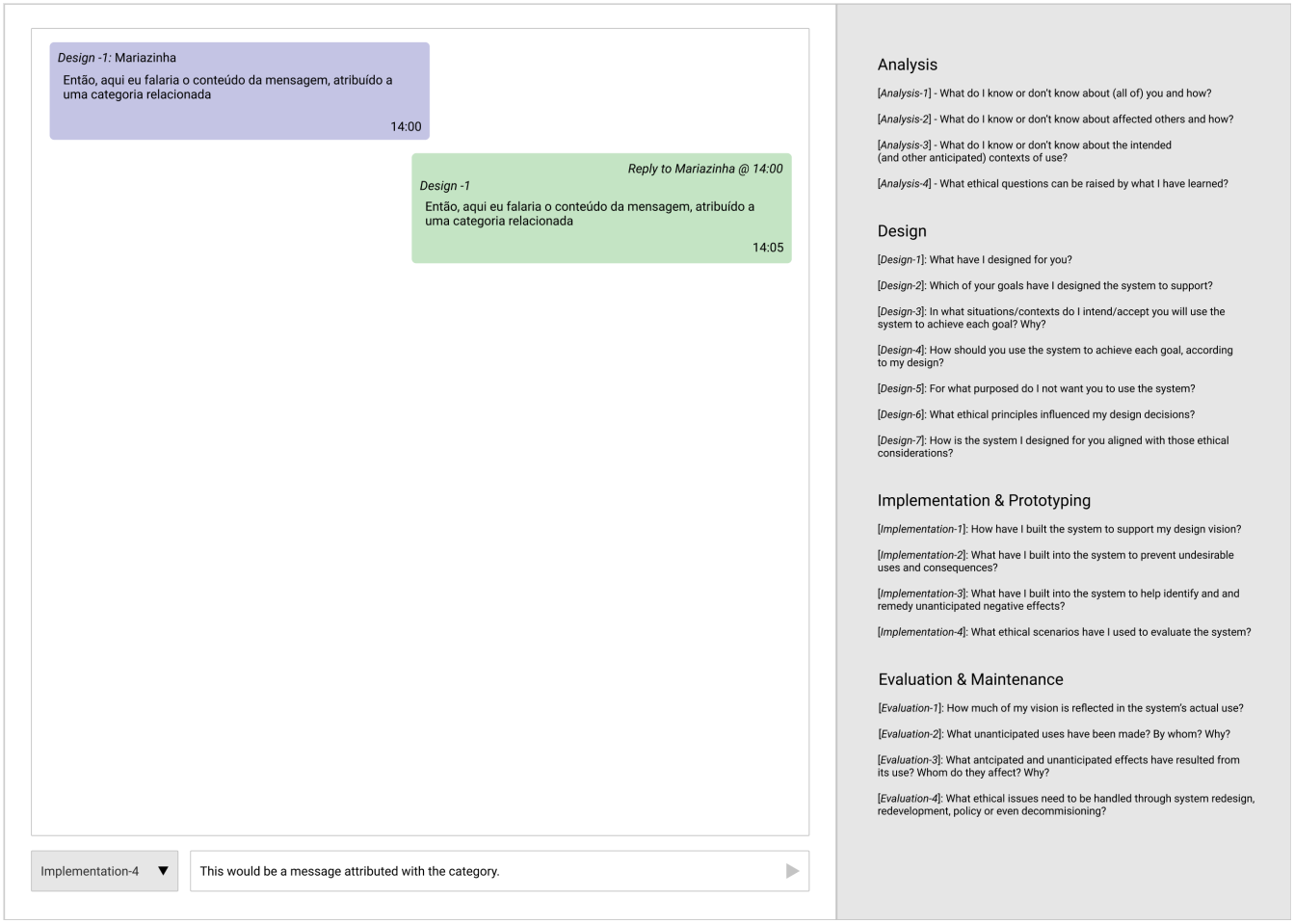


Figura 3 – Protótipo de baixa fidelidade

Além de modelar e planejar o *back-end* da aplicação, também temos que ter algum nível de planejamento para o *front-end*. Temos então o protótipo de baixa fidelidade que nos permite ter uma ideia inicial de como deveria ficar a interface da aplicação (sem considerar as interações possíveis).

Com esse protótipo, podemos pensar nos posicionamentos dos elementos na tela, além de aspectos informacionais como textos de *labels*, que indicam as *interaction affordances* de elementos da interface.

Temos duas regiões principais da interface: a região de mensagens e a região de categorias. Na questão de categorias, temos apenas uma listagem de categorias que podem ser selecionadas para as mensagens. Na seção de mensagens, temos um display das mensagens enviadas e uma seção de input de mensagens, com um seletor de categorias e um input de texto.

O resultado final ficou esteticamente diferente dado o uso de uma biblioteca de componentes pré-construídos com seus devidos estilos, seguindo os padrões de *Material Design* da *Google*. Algumas das realidades desses componentes pré-construídos mudaram a aparência de certas partes da interface.

2.6. Modelagem de Classes (Componentes)

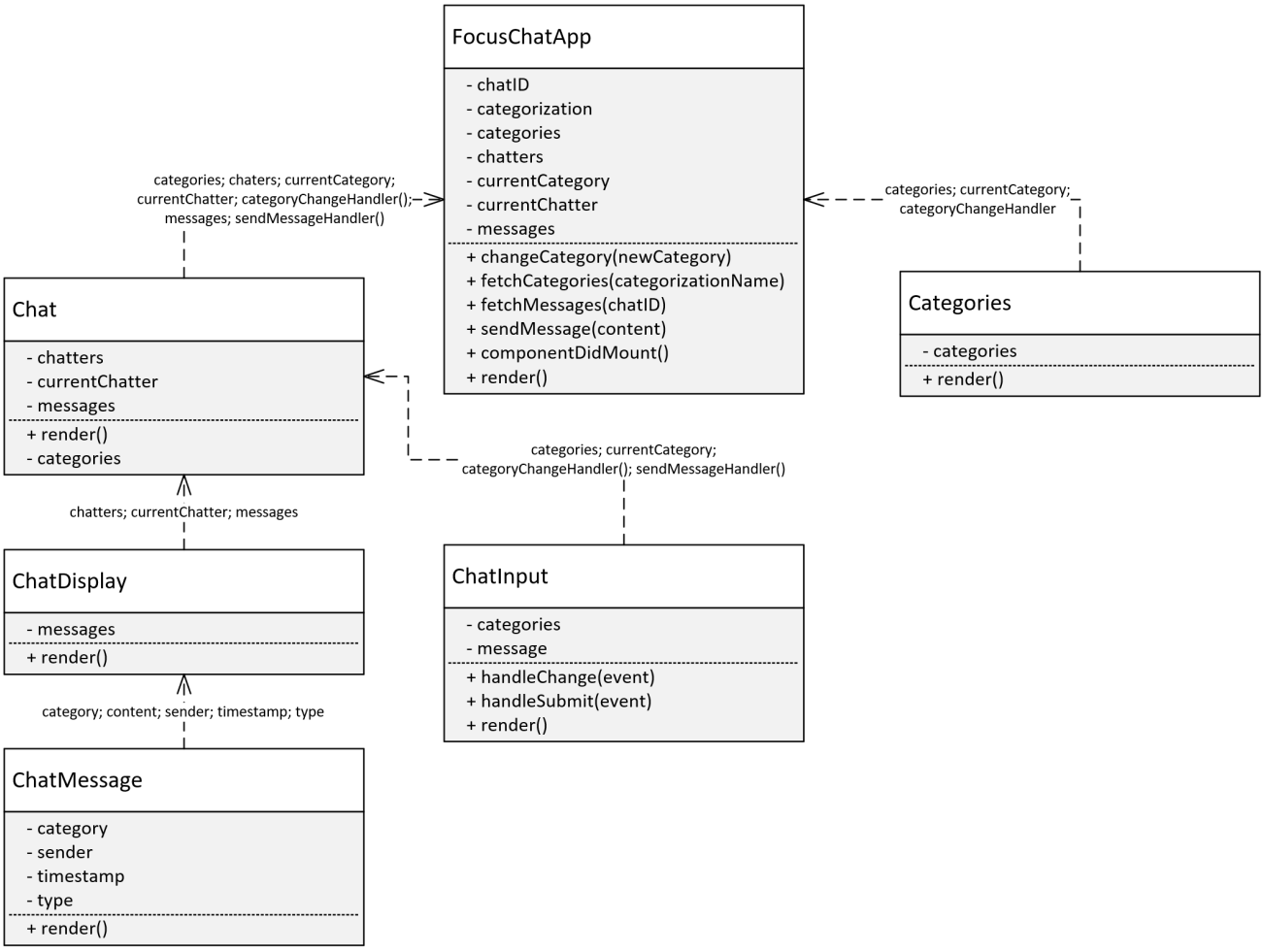


Figura 4 - Modelo de componentes (UML Class adaptado para considerar fluxos de atributos e métodos em classes React)

Juntando questões de programação de interações com os elementos de interface em si, podemos modelar as classes do *front-end*, que seriam os componentes apresentados na tela para o usuário. Essa análise estrutural nos permite analisar não só as funcionalidades atribuídas a cada um desses elementos, mas também o fluxo de informação entre essas classes (componentes). Este último ponto é uma parte crucial de desenvolvimento em *React*, onde o fluxo de informações é mais controlado.

Podemos observar que o componente primário (*FocusChatApp*) é responsável pela maioria das interações com o *back-end*. A partir das requisições, esse componente é responsável por controlar o contexto geral da aplicação, além de direcionar as informações para seus componentes-filhos.

Além de passar informações para contextualizar os componentes filhos através de suas *props*, podemos também passar *event handlers* para que os filhos possam modificar os estados de seus pais. Como as informações contextuais ficam contidas no componente pai, é nele que ficam os métodos que modificam o estado da aplicação como um todo. Ele passa para seus filhos os *handlers categoryChangeHandler()* (que modifica a categoria da aplicação) e *sendMessageHandler()*, que pede que o componente pai envie uma mensagem para o *back-end*.

Estruturamos então o nosso *front-end* de acordo com esses componentes e com esses fluxos de informações e funções.

3. Implementação e Testes

3.1. Instalação e Configuração

Em termos de instalação, temos processos diferentes para o *front-end* e para o *back-end*. Na parte de *front-end* temos que instalar *React.js* e *Node.js*. Na parte de *back-end* temos que instalar *Flask* e as bibliotecas associadas.

O código fonte pode ser encontrado no repositório *Git* disponível em https://github.com/GabrielDJB/INF2102-PFP. Esse repositório está aberto.

3.1.1. Instalação de *front-end*

Dada a presença de um arquivo *package.json* do projeto, a instalação de todas as dependências pode ser feita a partir do seguinte comando, na pasta em que o arquivo se encontra:

```
-- NPM INSTALL
```

Esse comando requer a instalação de Node.js, que pode ser feita baixando e executando os instaladores disponíveis em https://nodejs.org/en/download/. Uma vez instalados os pacotes, a execução do servidor de *front-end* através do arquivo *deploy.bat*.

3.1.2. Instalação de *back-end*

Para instalar as dependências de *back-end*, é necessário ter instalada alguma distribuição Python, dentre as disponíveis em https://www.python.org/downloads/. Uma vez instalada alguma distribuição de Python, recomendamos a criação de um ambiente virtual e a instalação de pacotes nesse ambiente através do comando:

```
-- (VENV) PIP INSTALL -R REQUIREMENTS.TXT
```

Uma vez instaladas as dependências, o arquivo *deploy.bat* deve ser usado para ativar o ambiente virtual e iniciar o servidor de *back-end*.

3.2. Estrutura de pastas

O código deste projeto está dividido em duas pastas, uma de *front-end*, e uma de *back-end*, conforme a separação da nossa aplicação.

3.2.1. Arquivos de *front-end*

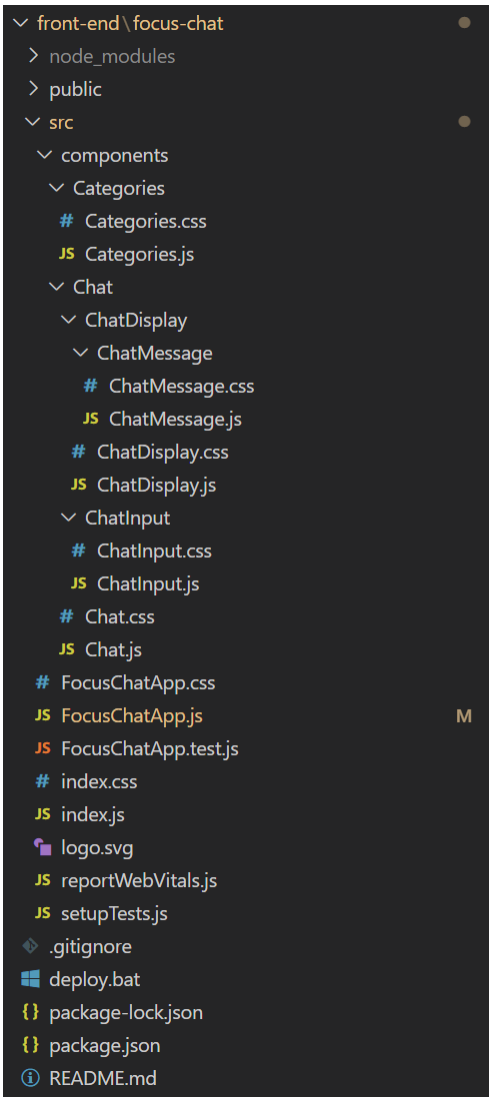


Figura 5 - Arquivos de front-end

Os arquivos de *front-end* seguem um padrão de projetos de *React Native*. O arquivo base do projeto que gera a aplicação é o arquivo *index.js*. Esse arquivo importa o componente primário *FocusChatApp* e renderiza no domínio.

Em *React*, todos os componentes são estruturados como classes em *JavaScript* com seus estilos definidos por um arquivo CSS com o mesmo nome. Observando a estrutura da pasta *components* e de suas subpastas, conseguimos ver os componentes *Chat*, *Categories*, *ChatDisplay*, *ChatInput* e *ChatMessage*.

O arquivo *deploy.bat* executa as chamadas necessárias para instanciar o servidor de desenvolvimento, de acordo com as dependências descritas em *package.json*. As bibliotecas das dependências podem ser encontradas na pasta *node_modules*. A pasta *public* contém arquivos que ficam disponíveis para os browsers onde a aplicação será renderizada.

3.2.2. Arquivos de *back-end*

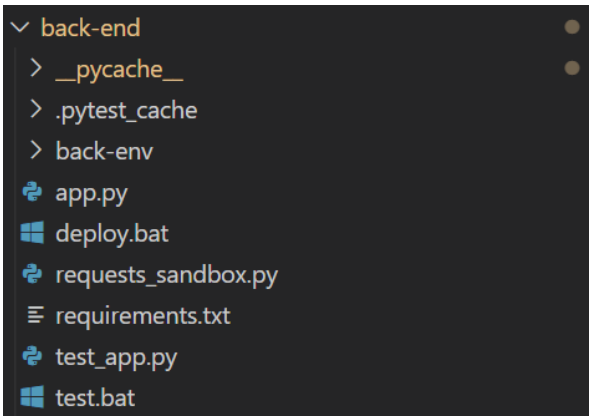


Figura 6 - Arquivos de back-end

Arquivos de *back-end* são estruturados de forma mais simples. O código base do servidor está localizado no arquivo *app.py*. Nele podemos encontrar as funções associadas às rotas nas quais o servidor irá responder as requisições feitas a ele. O arquivo *deploy.bat* executa os comandos necessários para gerar o servidor.

A pasta *back-env* contém o ambiente virtual que é instanciado para a construção do servidor de desenvolvimento. Como as dependências são um pouco diferentes de acordo com o sistema operacional que está sendo utilizado, esses ambientes virtuais devem ser reconstruídos em cada computador em que o servidor deverá estar disponível. Para isso, podemos utilizar o arquivo *requirements.txt*, como descrito na seção de instalação.

Em termos de arquivos de teste, temos como principal o arquivo *test_app.py*. Nele podemos encontrar cada um dos testes automatizados a serem executados caso o código seja alterado. Para facilitar a inicialização dos procedimentos de teste, construímos um arquivo *test.bat* que executa os comandos de terminal necessários para isso.

3.3. Documentação de Código

Em termos de documentação do código, temos comentários pontuais sobre a execução e cabeçalhos de funções e métodos. Todas as funções e os métodos possuem cabeçalhos contendo o nome da função e uma descrição do que ela faz. Funções e métodos que requerem parâmetros de input também possuem descrições de cada um desses inputs.

Ao início de arquivos de *front-end*, possuímos cabeçalhos contendo uma descrição do que está presente nesse arquivo e qual foi o seu autor.

```
// ---------------------------------------------------------------------
// FILE FOCUSCHATAPP.JS
//
// Author: Gabriel D. J. Barbosa
//
// Description:
//   File containing React Javascript code for the application's top
//   component. Renders to HTML via front-end Node server.
// ---------------------------------------------------------------------
```

Figura 7 - Exemplo de cabeçalho de arquivo

```
// ---------------------------------------------------------------------------------------
// METHOD constructor()
//
// Description:
//   Constructor method for FocusChatApp class. Instantiates class and defines
//   initial state and available properties.
//
// Inputs:
//   props - Dictionary (object) with properties provided to component (initial variables)
// ---------------------------------------------------------------------------------------
```

Figura 8 - Exemplo de cabeçalho de método

Além de cabeçalhos de funções e métodos, temos comentários em quaisquer blocos de código coesos, descrevendo sua execução.

```
async componentDidMount() {

  var categorizationName = ''
  var chatID = ''
  var chatterID = ''
  const instanceID = window.location.pathname.substring(1)

  // Loading initialization data
  await fetch("http://127.0.0.1:5000/login/?instanceID=" + instanceID, {
    method: 'GET',
  }).then(
    response => {
      // Checking if response is OK and structuring into JSON if so
      if (!response.ok) {
        console.log("There was an error with the login fetch request.")
      } else {
        return response.json()
      }
    }
  ).then(
    // Setting messages in the current state
    data => {
      // Setting function variables for the rest of function
      categorizationName = data['categorizationName']
      chatID = data['chatID']
      chatterID = data['chatterID']
      // Updating state
      var currState = this.state
      currState['categorization'] = data['categorizationName']
      currState['chatID'] = data['chatID']
      currState['currentChatter'] = data['chatterID']
      this.setState(currState)
    }
  )

  // Fetching the rest of the information
  await this.fetchCategories(categorizationName)
  await this.fetchMessages(chatID)

  // Setting message refresh procedure
  setInterval(() => this.fetchMessages(chatID), 500);

}
```

Figura 9 - Exemplo de documentação de código

3.4. Testes Automatizados

Cada um dos serviços de *back-end* possui diferentes critérios de validação de seus parâmetros. A fim de testar cada um desses critérios, utilizamos a biblioteca *PyTest* para executar diferentes testes e asserções para cada ponto de validação.

Testes executam diferentes *requests* para os serviços, esperando receber mensagens de erro. Um exemplo de teste de serviço seria:

```python
# -----------------------
# Testing Update Chatter
# -----------------------
#@pytest.fixture
def test_update_chatter(client):

    # ------------ SETUP ------------ #
    # Successful instance
    success = {'chatterID': 'test_chatter_ID', 'chatterName': 'test_chatter_name', 'chatIDs':['test_chat_ID']}
    # Failed chatterID instances
    fail_chatterID_missing = {'chatterName': 'test_chatter_name', 'chatIDs':['test_chat_ID']}
    fail_chatterID_empty = {'chatterID': '', 'chatterName': 'test_chatter_name', 'chatIDs':['test_chat_ID']}
    # Failed chatterName instances
    fail_chatterName_missing = {'chatterID': 'test_chatter_ID', 'chatIDs': ['test_chat_ID']}
    fail_chatterName_empty = {'chatterID': 'test_chatter_ID', 'chatterName': '', 'chatIDs': ['test_chat_ID']}
    # Failed chatIDs instances
    fail_chatIDs_missing = {'chatterID': 'test_chatter_ID', 'chatterName': 'test_chatter_name'}

    # ------------ TESTING ------------ #
    # Successful instance
    # assert client.post('/chatter/update/', data = success).get_json()['status'] == "Success"
    # Testing chatterID validation
    assert client.post('/chatter/update/', data = fail_chatterID_missing).get_json() == {'status': 'Error', 'error': 'Chatter ID is missing. Request failed.'}
    assert client.post('/chatter/update/', data = fail_chatterID_empty).get_json() == {'status': 'Error', 'error': 'Chatter ID is missing. Request failed.'}
    # Testing chatterName validation
    assert client.post('/chatter/update/', data = fail_chatterName_missing).get_json() == {'status': 'Error', 'error': 'Chatter name is missing. Request failed.'}
    assert client.post('/chatter/update/', data = fail_chatterName_empty).get_json() == {'status': 'Error', 'error': 'Chatter name is missing. Request failed.'}

    # ------------ CLEAN UP ------------ #
    pass
```

Figura 10 - Exemplo de teste de serviço

Para construir casos de teste, partimos de instâncias corretas e sistematicamente fazemos alterações em cada um dos parâmetros para acionar cada critério de validação. Por exemplo, a fim de testar as validações do parâmetro *chatterID* no serviço *update_chatter(),* construímos instâncias problemáticas com esse parâmetro faltando e com ele tendo o valor de uma *string* vazia. Ambos os casos deveriam ser detectados pelo serviço, e uma mensagem de erro deveria ser retornada. Com isso, construímos asserções para cada um desses casos, que serão validados ao rodar os testes.

Ao chamar o comando PYTEST executamos todos os testes contidos no arquivo *test_app.py*. Temos então o seguinte resultado dos dezoito testes de serviços, totalizando 92 asserções.

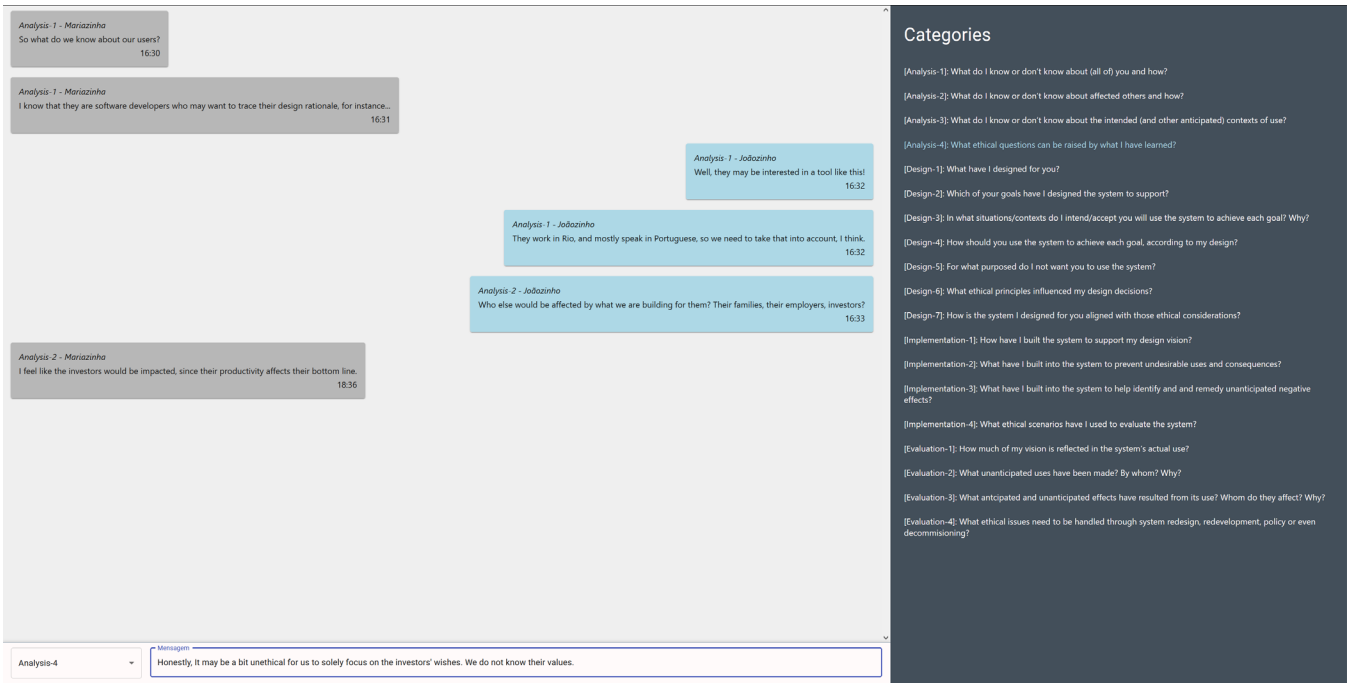
```
test_app.py ..................                                          [100%]

============================================== 18 passed in 1.35s ==============================================
```

Figura 11 - Resultado de execução dos testes automatizados

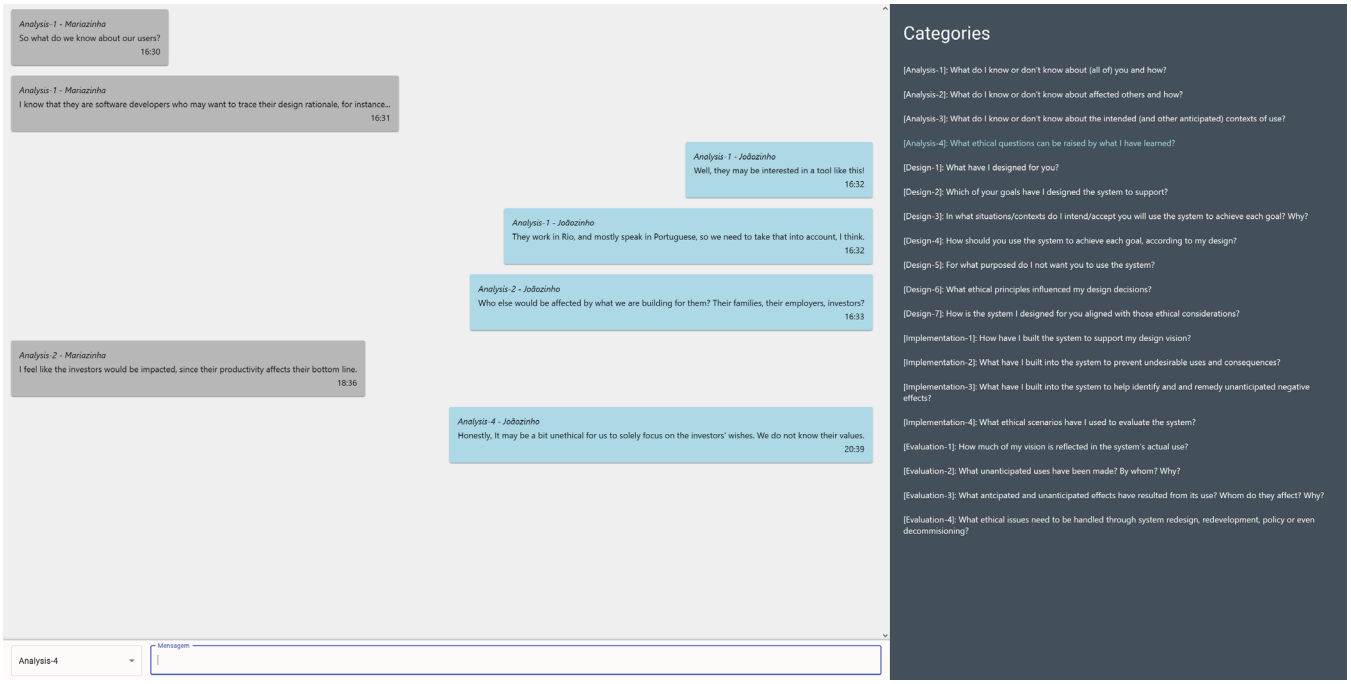
Na medida que os serviços foram avançando, mais e mais casos de teste foram adicionados. No conjunto que temos no momento, todos os pontos de validação de parâmetros são testados, garantindo que chamadas errôneas não serão aprovadas e executadas.

4. Demonstração de Uso

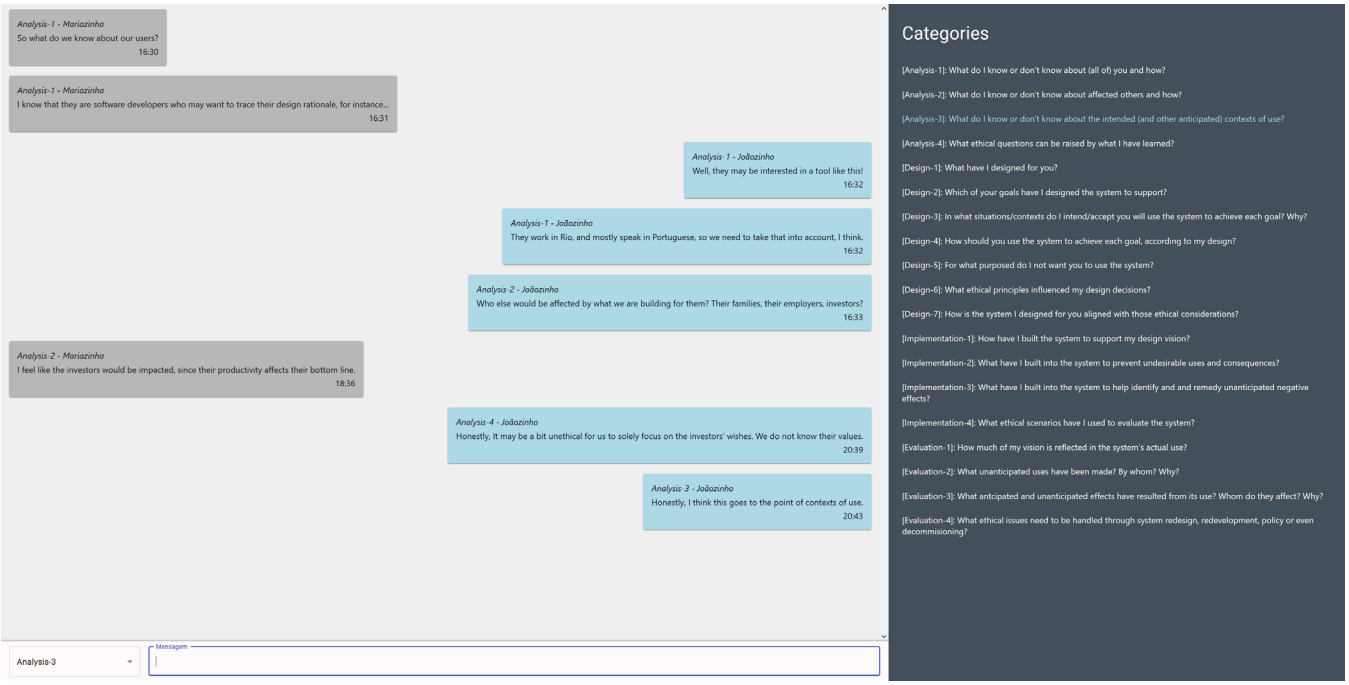
Ao entrar na plataforma com um dos nossos *magic links*, podemos enxergar uma das visões sobre a conversa. Começamos pela visão de Joãozinho, nosso primeiro usuário.



Como podemos enxergar, conseguimos escrever uma mensagem e atribuir uma categoria a ela. Selecionando uma categoria pelo menu da direita, ou através do seletor à esquerda do input, o outro mecanismo se modifica para visualizar qual é a categoria ativa.

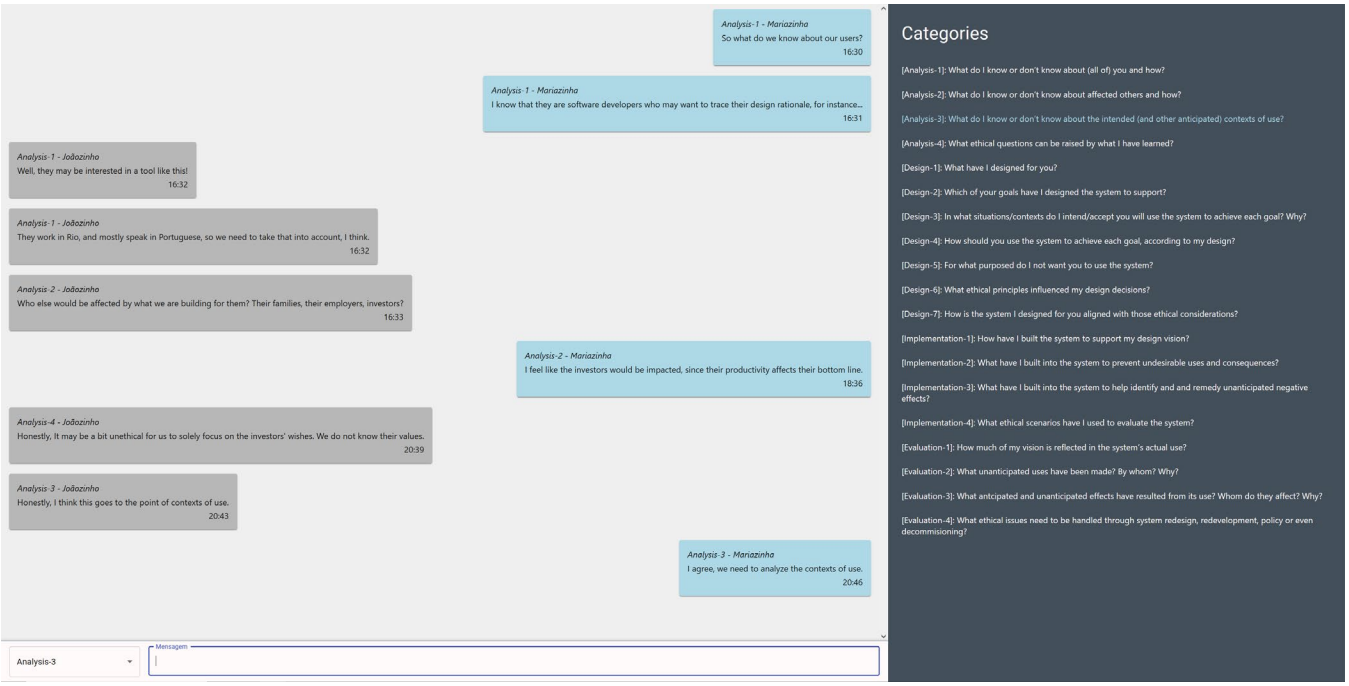


Enviando a mensagem, podemos enxergar que temos uma mensagem adicionada com o conteúdo que escrevemos, com a categoria que selecionamos, listando nosso nome (Joãozinho) e visualizando o horário em que a mensagem foi enviada.

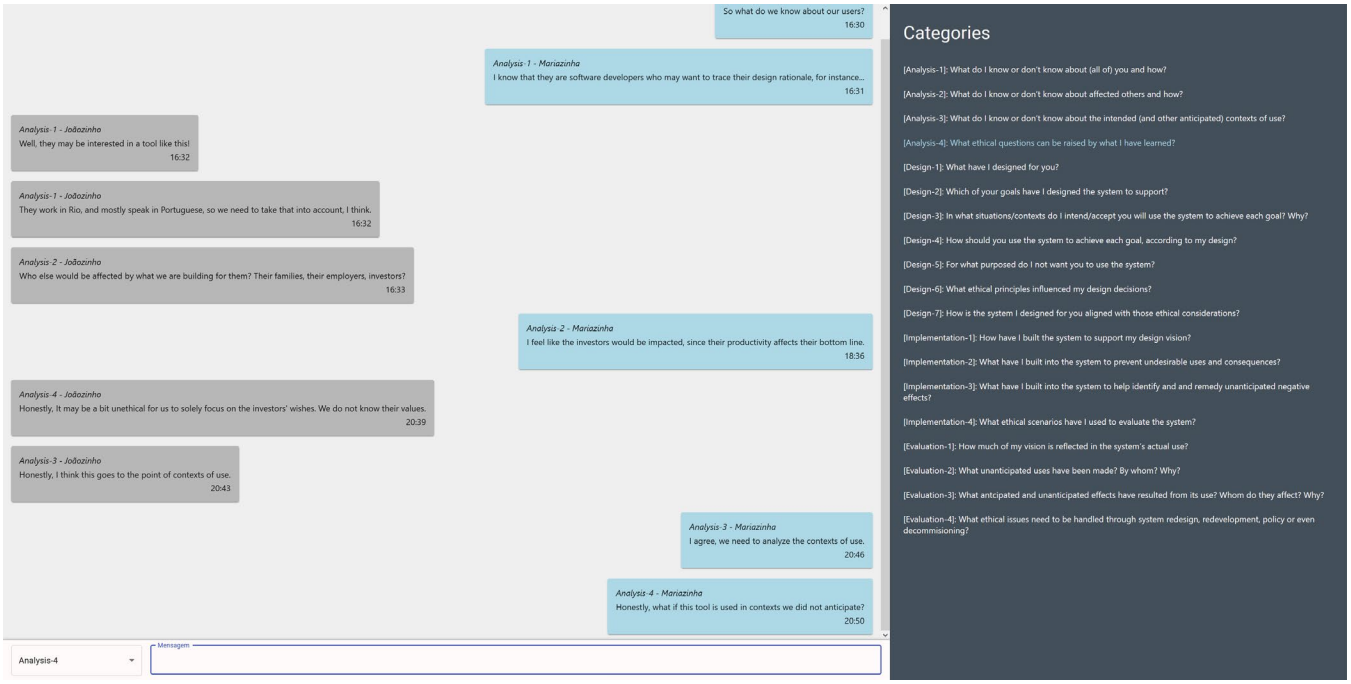


Modificando a categoria corrente para *Analysis-3*, conseguimos ver que a nova mensagem é atribuída a nova categoria. Seguindo esse fluxo, conseguimos construir uma conversa estruturada para fins de análise posterior.

Podemos então enxergar a visão de outra participante dessa conversa, a de Mariazinha.



Como podemos ver, na visão de Mariazinha, as mensagens de Joãozinho constam como mensagens enviadas por outro usuário, sendo visualizadas à esquerda do *display*. Ao enviar uma mensagem respondendo à consideração de Joãozinho, sua mensagem aparece à direita do *display*.



Seguindo a conversa, através dos diferentes *magic links*, conseguimos construir uma conversa estruturada sobre o processo de desenvolvimento. Considerando as categorias, se utilizadas corretamente, podemos ter uma conversa que traça o *design rationale* da equipe envolvida. Passando de tópico em tópico, os usuários consideram diferentes aspectos do desenvolvimento.

Podemos observar na imagem a seguir que as mensagens enviadas ficam estruturadas no banco *MongoDB* para obtenção e análise posterior. Assim, conseguimos ver que as conversas ficam registradas corretamente, realizando a proposta de valor da ferramenta.

```
_id: ObjectId("60d90d38f9c9ffe8d2da12eb")
chatID: "60d7c635e753bbe49edef57a"
chatterID: "60d7c5d3e753bbe49edef578"
content: "Honestly, I think this goes to the point of contexts of use."
timestamp: "1624837432391"
categorizationName: "ExtendedMetacommunicationTemplate"
categoryName: "Analysis-3"
replyID: ""
```

```
_id: ObjectId("60d90dd1f9c9ffe8d2da12ec")
chatID: "60d7c635e753bbe49edef57a"
chatterID: "60d7c5ede753bbe49edef579"
content: "I agree, we need to analyze the contexts of use."
timestamp: "1624837585282"
categorizationName: "ExtendedMetacommunicationTemplate"
categoryName: "Analysis-3"
replyID: ""
```

```
_id: ObjectId("60d90ed9f9c9ffe8d2da12ed")
chatID: "60d7c635e753bbe49edef57a"
chatterID: "60d7c5ede753bbe49edef579"
content: "Honestly, what if this tool is used in contexts we did not anticipate?"
timestamp: "1624837849559"
categorizationName: "ExtendedMetacommunicationTemplate"
categoryName: "Analysis-4"
replyID: ""
```


Figura 12 - Exemplo de instâncias no banco MongoDB

5. Considerações Finais

Apesar de severas limitações de interação, a ferramenta construída se demonstra capaz de construir conversas estruturadas para fins posteriores de análise. Além de viabilizar conversas sobre o processo de desenvolvimento, a ferramenta também estrutura as mensagens em uma coleção de mensagens com suas categorias.

Essa estruturação pode ser utilizada para analisar o processo de desenvolvimento e os raciocínios que aparecem ao longo do processo. A ordenação de categorias, as idas e voltas, todas essas tendências, quando relacionadas com os conteúdos dessas mensagens, podem possuir um certo valor epistêmico.

Um dos maiores riscos da ferramenta é a de usuários esquecerem das categorias e pararem de categorizar suas mensagens corretamente. Nesses casos, o *dataset* resultante não seria adequado para os fins de análise posteriores.

A fim de evitar esse possível cenário, podemos evoluir a interface com certos sinais visuais de quais são as categorias ativas no momento, possivelmente utilizando cores no menu de categorias e possivelmente em uma barra de acompanhamento de mensagens no display.

Atualmente também não é possível corrigir as categorias de mensagens já enviadas. Essa seria outra possibilidade para garantir a consistência dos dados.

Outra informação interessante seria se uma mensagem responde outra anterior. Já possuímos essa possibilidade no banco de dados e no *back-end*, mas ainda não implementamos as interações necessárias no *front-end*.

No final do projeto, temos uma primeira versão de uma ferramenta de conversas estruturadas de acordo com categorizações. A ferramenta é capaz de criar um *dataset* estruturado que pode ser posteriormente utilizado para traçar o raciocínio de design dos desenvolvedores envolvidos. Sendo assim, temos um primeiro passo para a ferramenta que viabilizaria a coleta de raciocínio de design em processos de desenvolvimento.

Referências

Seguem algumas das referências utilizadas para a implementação. Nelas constam a maior parte das bibliotecas utilizadas, além de guias de instalação.

1. https://nodejs.org/en/
2. https://reactjs.org/
3. https://flask.palletsprojects.com/en/2.0.x/
4. https://www.mongodb.com/
5. https://material-ui.com/
6. https://material.io/design
7. https://flask.palletsprojects.com/en/2.0.x/testing/