📚 Documentação Técnica Completa - ChatJovemProgramador

📋 Índice

---

1. Visão Geral do Projeto

1.1 Nome do Projeto

ChatJovemProgramador (também conhecido como "ChatLeo" ou "Leozin")

1.2 Objetivo

Desenvolver um assistente virtual inteligente especializado em responder dúvidas sobre o **Programa Jovem Programador**, utilizando Inteligência Artificial (Google Gemini) para fornecer respostas precisas, baseadas em dados oficiais extraídos do site do programa.

1.3 Descrição do Chatbot

O chatbot é um assistente virtual que:

- Responde perguntas sobre o Programa Jovem Programador 24/7
- Utiliza dados extraídos em tempo real do site oficial do programa
- Possui personalidade amigável e prestativa
- Coleta informações de leads (nome, interesse, cidade, estado, idade) durante a conversa
- Mantém histórico de conversas quando Firestore está habilitado
- Oferece interface web acessível com recursos de acessibilidade (TTS, alto contraste, tamanho de fonte)

1.4 Tecnologias Utilizadas

Backend

- **Python 3.x**: Linguagem principal
- **Flask 2.3.2**: Framework web para API REST
- **Flask-CORS 3.0.10**: Habilita CORS para requisições cross-origin
- **Google Generative AI (Gemini) 0.3.2**: API de IA para processamento de linguagem natural

- **Firebase Admin SDK (>=6.0.0)**: Integração com Firestore para persistência de dados
- **BeautifulSoup4**: Web scraping para extração de dados do site oficial
- **python-dotenv 1.0.0**: Gerenciamento de variáveis de ambiente
- **Werkzeug**: Hashing de senhas para autenticação admin

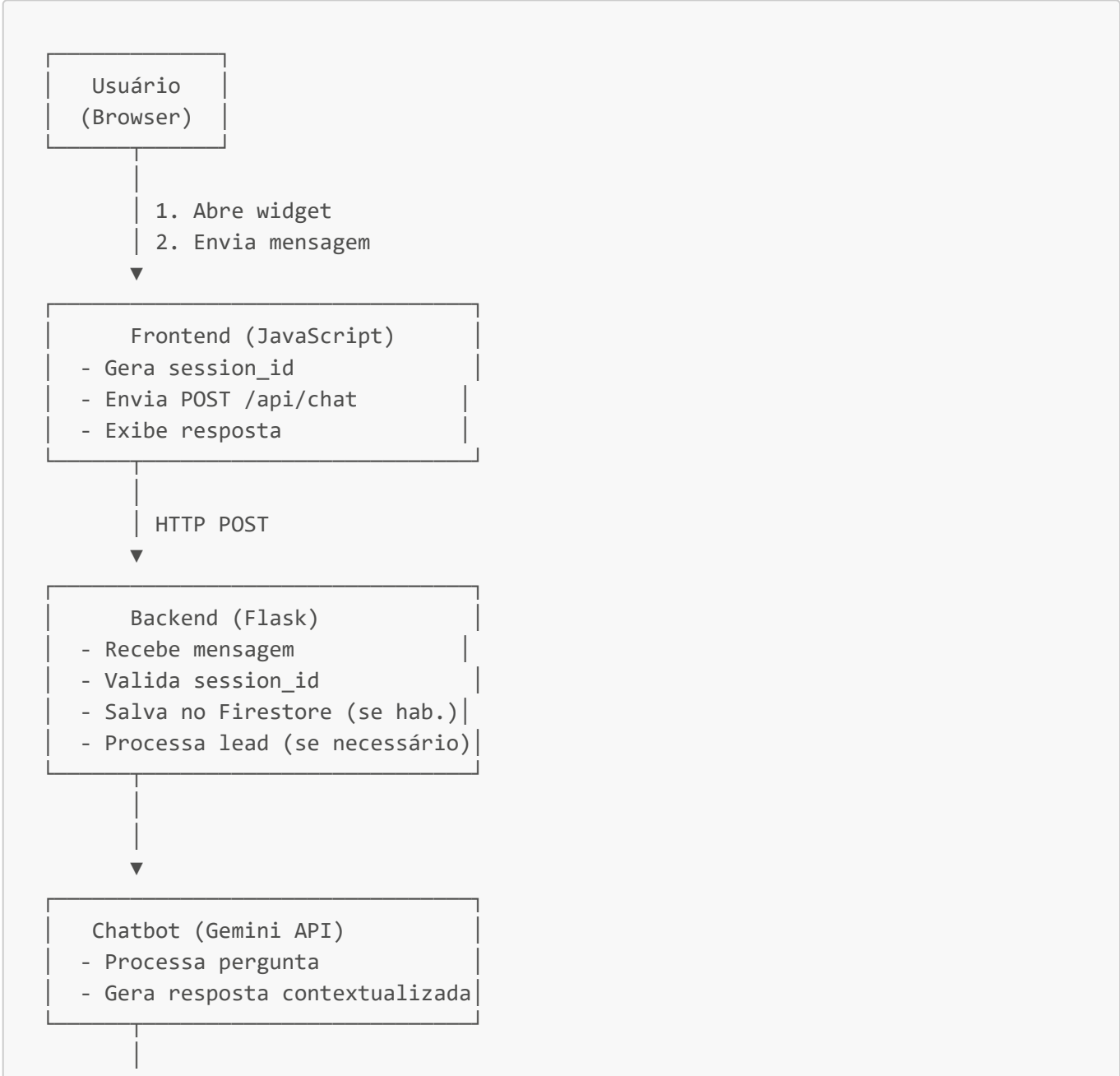
Frontend

- **HTML5**: Estrutura do widget
- **CSS3**: Estilização (com variáveis CSS para personalização)
- **JavaScript (Vanilla)**: Lógica do widget, comunicação com backend, acessibilidade
- **Web Speech API**: Text-to-Speech (TTS) para acessibilidade

Banco de Dados

- **Google Firestore**: Banco NoSQL para persistência de conversas, mensagens, leads e configurações

1.5 Fluxo Geral de Funcionamento



```
        │ Resposta
        ▼
┌──────────────────────────────────┐
│      Firestore                   │
│  - Salva mensagem                │
│  - Atualiza conversa             │
│  - Salva lead (se completo)      │
└──────────────────────────────────┘
```

---

2. Arquitetura e Estrutura de Pastas

2.1 Arquitetura Geral

O projeto segue uma arquitetura em **camadas**:

```
┌─────────────────────────────────────────────┐
│         CAMADA DE APRESENTAÇÃO              │
│  (Templates HTML + CSS + JavaScript)        │
└──────────────────┬──────────────────────────┘
                   │
┌──────────────────▼──────────────────────────┐
│         CAMADA DE APLICAÇÃO                 │
│  (Flask Routes + Blueprints)                │
└──────────────────┬──────────────────────────┘
                   │
┌──────────────────▼──────────────────────────┐
│         CAMADA DE SERVIÇOS                  │
│  (Chatbot + Firestore + Scraper)            │
└──────────────────┬──────────────────────────┘
                   │
┌──────────────────▼──────────────────────────┐
│         CAMADA DE DADOS                     │
│  (Firestore + dados.json)                   │
└─────────────────────────────────────────────┘
```

2.2 Fluxo de Dados (Backend → Firestore → Frontend → Admin)

```
┌──────────┐      ┌──────────┐      ┌──────────┐      ┌──────────┐
│ Backend  │─────▶│Firestore │─────▶│ Frontend │─────▶│  Admin   │
│ (Flask)  │◀─────│          │◀─────│  (Widget)│◀─────│ (Panel)  │
└──────────┘      └──────────┘      └──────────┘      └──────────┘
     │                 │                 │                 │
     │                 │                 │                 │
     ▼                 ▼                 ▼                 ▼
┌──────────┐      ┌──────────┐      ┌──────────┐      ┌──────────┐
│ Gemini   │      │Conversas │      │  JS API  │      │Dashboard │
│   API    │      │ Mensagens│      │  Calls   │      │ Analytics│
└──────────┘      └──────────┘      └──────────┘      └──────────┘
```

```
  │          │      │  Leads   │      │          │      │          │
  └──────────┘      └──────────┘      └──────────┘      └──────────┘
```

2.3 Estrutura de Pastas

```
ChatJovemProgramador-staging/
│
├── app.py                          # Aplicação Flask principal
├── admin.py                        # Blueprint do painel administrativo
├── requirements.txt                # Dependências Python
├── dados.json                      # Base de conhecimento (gerado pelo scraper)
├── README.md                       # Documentação básica
├── DOCUMENTACAO.md                 # Esta documentação
│
├── services/                       # Camada de serviços
│   ├── __init__.py
│   └── firestore.py               # Integração com Firestore
│
├── utils/                          # Utilitários e helpers
│   ├── responder.py               # Classe Chatbot (integração Gemini)
│   ├── scraper.py                 # Web scraping do site oficial
│   ├── menu.py                     # Menu CLI (legado)
│   └── dados.json                  # Backup/cache de dados
│
├── static/                         # Arquivos estáticos
│   ├── assets/                     # Imagens e recursos
│   │   ├── logo.png               # Logo do bot
│   │   ├── logo-user.png          # Avatar do usuário
│   │   ├── jovem_rodape.png       # Logo rodapé
│   │   └── senac-logo.png         # Logo SENAC
│   │
│   ├── css/                        # Folhas de estilo
│   │   ├── style.css              # Estilos gerais da página
│   │   ├── chatleo.css            # Estilos do widget de chat
│   │   └── admin.css              # Estilos do painel admin
│   │
│   └── js/                         # Scripts JavaScript
│       └── script.js              # Lógica do widget de chat
│
├── templates/                      # Templates Jinja2
│   ├── index.html                 # Página principal (com widget)
│   │
│   └── admin/                      # Templates do painel admin
│       ├── base.html              # Template base
│       ├── login.html             # Página de login
│       ├── dashboard.html         # Dashboard principal
│       ├── conversations.html     # Lista de conversas
│       └── settings.html          # Configurações
│
├── tests/                          # Testes (estrutura)
│   ├── unit/                      # Testes unitários
```

```
│   └── e2e/                        # Testes end-to-end
│
└── docs/                           # Documentação adicional
    ├── fluxo-geral-funcionamento.mmd
    └── fluxo-geral-funcionamento.html
```

2.4 Explicação de Cada Pasta/Arquivo

`/app.py`

- **Responsabilidade**: Ponto de entrada da aplicação Flask
- **Funcionalidades**:
  - Inicializa servidor Flask
  - Configura CORS
  - Inicializa Firestore e admin padrão
  - Define rotas principais (`/`, `/api/chat`, `/api/chat-config`, `/health`)
  - Gerencia fluxo de coleta de leads
  - Integra com `utils/responder.py` para processar mensagens

`/admin.py`

- **Responsabilidade**: Blueprint do painel administrativo
- **Funcionalidades**:
  - Autenticação de administradores
  - Dashboard com métricas e gráficos
  - Visualização de conversas e mensagens
  - Configurações do chat
  - Gerenciamento de leads

`/services/firestore.py`

- **Responsabilidade**: Camada de abstração para Firestore
- **Funcionalidades**:
  - Inicialização do Firebase Admin SDK
  - CRUD de conversas e mensagens
  - Gerenciamento de leads
  - Configurações (settings)
  - Autenticação de admins
  - Queries e agregações para analytics

`/utils/responder.py`

- **Responsabilidade**: Lógica do chatbot com Gemini
- **Funcionalidades**:
  - Carrega base de conhecimento (`dados.json`)
  - Cria contexto inicial (prompt de sistema)
  - Inicializa modelo Gemini (com fallback)

    - Processa mensagens e gera respostas
    - Gerencia sessão de chat com histórico

`/utils/scraper.py`

- **Responsabilidade**: Web scraping do site oficial
- **Funcionalidades**:
    - Extrai informações sobre o programa
    - Coleta dúvidas frequentes
    - Raspa notícias completas (título + conteúdo)
    - Extrai dados de apoiadores, patrocinadores, parceiros
    - Coleta links de redes sociais e portais
    - Gera arquivo `dados.json` com tudo estruturado

`/static/`

- **Responsabilidade**: Recursos estáticos (CSS, JS, imagens)
- **Estrutura**:
    - `css/`: Estilos do widget, página e admin
    - `js/script.js`: Lógica completa do widget (estado, eventos, API calls)
    - `assets/`: Imagens e logos

`/templates/`

- **Responsabilidade**: Templates HTML renderizados pelo Flask
- **Estrutura**:
    - `index.html`: Página principal com widget embutido
    - `admin/*.html`: Painel administrativo (dashboard, conversas, settings)

---

3. Backend

3.1 Como o Servidor Flask Funciona

O servidor Flask é inicializado em `app.py`:

```
app = Flask(__name__)
app.config['SECRET_KEY'] = os.getenv('FLASK_SECRET_KEY', 'dev-secret-key-change-in-production')
CORS(app)  # Habilita CORS para requisições cross-origin
```

Características:

- Modo debug habilitado por padrão (desenvolvimento)
- Porta padrão: `5000`
- Host: `0.0.0.0` (aceita conexões externas)
- CORS habilitado para permitir requisições do frontend

3.2 Inicialização do Firebase

A inicialização do Firebase ocorre no início de `app.py`:

```
# Inicializa Firestore (se habilitado)
init_admin()
# Inicializa admin padrão após Firestore estar pronto
init_default_admin()
```

Fluxo de inicialização (`services/firestore.py`):

1. Verifica variável `AI_FIRESTORE_ENABLED`
2. Se `true`, lê `FIREBASE_CREDENTIALS` (JSON string)
3. Faz parse do JSON e cria credenciais
4. Inicializa Firebase Admin SDK
5. Obtém cliente Firestore
6. Cria admin padrão (`admin`/`admin123`) se não existir

Tratamento de erros:

- Se Firestore estiver desabilitado, todas as funções retornam silenciosamente
- Se houver erro na inicialização, o chat continua funcionando (graceful degradation)

3.3 Como Funciona o Firestore e Onde São Salvas as Conversas

Estrutura de Coleções

```
Firestore
│
├── conversations/                    # Coleção de conversas
│   └── {session_id}/                # Documento da conversa
│       ├── session_id: string
│       ├── iniciadoEm: timestamp
│       ├── ultimaMensagemEm: timestamp
│       ├── created_at: timestamp
│       ├── updated_at: timestamp
│       ├── total_user_messages: number
│       ├── total_bot_messages: number
│       ├── channel: "web"
│       ├── status: "open" | "closed"
│       ├── lead_stage: null | "collecting" | "done"
│       ├── lead_done: boolean
│       ├── lead_data: object
│       │   ├── nome: string
│       │   ├── interesse: string
│       │   ├── cidade: string
│       │   ├── estado: string
│       │   └── idade: number
│       │
```

```
│       └── messages/                # Subcoleção de mensagens
│           └── {auto-id}/           # Documento de mensagem
│               ├── role: "user" | "bot"
│               ├── content: string
│               ├── created_at: timestamp
│               ├── papel: "user" | "bot" (legado)
│               ├── texto: string (legado)
│               ├── criadoEm: timestamp (legado)
│               └── metadata: object (opcional)
│
├── leads/                           # Coleção de leads
│   └── {auto-id}/
│       ├── session_id: string
│       ├── nome: string
│       ├── email: string (opcional)
│       ├── cidade: string
│       ├── estado: string
│       ├── idade: number
│       ├── interesse: string
│       └── createdAt: timestamp
│
├── settings/                        # Coleção de configurações
│   ├── global/
│   │   └── admin_theme: "dark" | "light"
│   └── chat_config/
│       ├── chat_title: string
│       ├── welcome_message: string
│       ├── bot_avatar: string
│       ├── user_avatar: string
│       ├── primary_color: string
│       └── secondary_color: string
│
└── admin_users/                     # Coleção de usuários admin
    └── {username}/
        ├── username: string
        ├── password_hash: string
        ├── created_at: timestamp
        └── updated_at: timestamp
```

3.4 Métodos Principais

`init_admin()`

```python
def init_admin():
    """Inicializa Firebase Admin SDK usando FIREBASE_CREDENTIALS."""
```

O que faz:

- Verifica se Firestore está habilitado
- Lê credenciais do ambiente

- Inicializa Firebase Admin SDK
- Cria cliente Firestore global

Quando é chamado: No bootstrap da aplicação (`app.py`)

`get_or_create_conversation(session_id)`

```
def get_or_create_conversation(session_id):
    """Cria ou atualiza documento de conversa."""
```

O que faz:

- Verifica se conversa existe em `conversations/{session_id}`
- Se não existe, cria com campos iniciais
- Se existe, atualiza `ultimaMensagemEm` e `updated_at`

Retorna: `True` se sucesso, `False` caso contrário (silencioso)

`save_message(session_id, role, text, meta=None)`

```
def save_message(session_id, role, text, meta=None):
    """Salva mensagem em conversations/{session_id}/messages."""
```

O que faz:

- Cria documento em subcoleção `messages`
- Normaliza `role` ("assistant" → "bot")
- Salva campos novos e legados (compatibilidade)
- Atualiza contadores da conversa (`total_user_messages` ou `total_bot_messages`)
- Atualiza `ultimaMensagemEm` da conversa

Parâmetros:

- `session_id`: ID da sessão
- `role`: "user" ou "assistant"
- `text`: Texto da mensagem
- `meta`: Dict opcional com metadados

`query_chatbot()` **(método do Chatbot)**

```
def gerar_resposta(self, pergunta: str) -> str:
    """Gera resposta usando Gemini API."""
```

O que faz:

- Valida mensagem não vazia
- Verifica se `chat_session` existe (reinicializa se necessário)
- Envia mensagem para Gemini API
- Retorna resposta processada
- Trata erros com fallback e reinicialização automática

Localização: `utils/responder.py`

3.5 Rotas do Flask

`GET /`

```
@app.route('/')
def index():
    return render_template('index.html')
```

- Renderiza página principal com widget embutido

`GET /api/chat-config`

```
@app.route('/api/chat-config', methods=['GET'])
def api_chat_config():
```

- Retorna configurações do chat (título, mensagem de boas-vindas, avatares, cores)
- Lê de `settings/chat_config` no Firestore
- Retorna defaults se não houver configuração

`POST /api/chat`

```
@app.route('/api/chat', methods=['POST'])
def chat():
```

- **Fluxo principal de chat**
- Recebe `message` e `session_id` (opcional)
- Gera `session_id` se não fornecido
- Se Firestore desabilitado: resposta direta do Gemini
- Se Firestore habilitado:
    1. Cria/atualiza conversa
    2. Salva mensagem do usuário
    3. Verifica comando especial ("apagar dados")
    4. Se lead concluído: resposta normal com IA
    5. Se lead em coleta: processa campo atual
    6. Salva resposta do bot

`GET /health`

```
@app.route('/health')
def health():
```

- Endpoint de health check
- Retorna status do chatbot e modelo em uso

Rotas Admin (`/admin/*`)

- `GET /admin/login`: Página de login
- `POST /admin/api/login`: Autenticação
- `GET /admin/`: Dashboard
- `GET /admin/api/reports`: Métricas e analytics
- `GET /admin/conversations`: Lista de conversas
- `GET /admin/api/conversations`: API de conversas
- `GET /admin/api/conversations/<session_id>/messages`: Mensagens de uma conversa
- `GET /admin/settings`: Página de configurações
- `GET /admin/api/settings`: Ler configurações
- `POST /admin/api/settings`: Salvar configurações
- `POST /admin/api/change-password`: Trocar senha

3.6 Fluxo Completo: Usuário → JS → Flask → Gemini → Firestore → Retorno

```
1. USUÁRIO
   └─> Digita mensagem no widget
       └─> Pressiona Enter ou clica em enviar

2. JAVASCRIPT (script.js)
   └─> sendMessage()
       └─> getOrCreateSessionId() (localStorage ou gera novo)
       └─> showTypingIndicator() (mostra "digitando...")
       └─> sendToBackend(message)
           └─> fetch('/api/chat', {
                 method: 'POST',
                 body: JSON.stringify({ message, session_id })
               })

3. FLASK (app.py)
   └─> @app.route('/api/chat', methods=['POST'])
       └─> Recebe JSON: { message, session_id }
       └─> Valida mensagem não vazia
       └─> Se não tem session_id: gera novo (sess_{timestamp}_{random})

4. FIRESTORE (se habilitado)
   └─> get_or_create_conversation(session_id)
       └─> Cria/atualiza documento em conversations/{session_id}
   └─> save_message(session_id, "user", message)
```

```
        └─> Adiciona documento em conversations/{session_id}/messages

5. FLUXO DE LEAD (se em coleta)
   └─> get_conversation(session_id)
       └─> Verifica lead_stage e lead_data
   └─> Se "collecting":
       └─> get_next_lead_field(lead_data)
       └─> normalize_lead_answer(field, message)
       └─> Valida e salva resposta
       └─> Se todos campos preenchidos:
           └─> save_lead_from_conversation(session_id, lead_data)
           └─> Marca lead_done = True
       └─> Retorna próxima pergunta ou mensagem final

6. GEMINI API (se lead concluído ou Firestore desabilitado)
   └─> chatbot_web.gerar_resposta(user_message)
       └─> Chatbot.gerar_resposta() (utils/responder.py)
           └─> chat_session.send_message(pergunta)
           └─> Gemini API processa com contexto inicial
           └─> Retorna resposta

7. FIRESTORE (salvar resposta)
   └─> save_message(session_id, "assistant", bot_response)
       └─> Adiciona documento em conversations/{session_id}/messages
       └─> Atualiza total_bot_messages

8. FLASK (retorno)
   └─> return jsonify({
         'response': bot_response,
         'session_id': session_id
       })

9. JAVASCRIPT (recebe resposta)
   └─> hideTypingIndicator()
   └─> addMessage(bot_response, 'bot')
       └─> Cria elemento DOM
       └─> Renderiza mensagem
       └─> scrollMessagesToBottom()
   └─> Se TTS habilitado: speakText(bot_response)
   └─> setLocked(false) (libera input)
```

---

4. Frontend (Widget do Chat)

4.1 Como o Widget Funciona

O widget é um componente JavaScript que:

- **Inicialização**: Carrega quando DOM está pronto
- **Estado**: Gerencia estado da aplicação (aberto/fechado, mensagens, preferências)
- **Comunicação**: Envia mensagens via `fetch()` para `/api/chat`
- **Renderização**: Cria elementos DOM dinamicamente para mensagens

- **Acessibilidade**: Suporta navegação por teclado, TTS, alto contraste, tamanho de fonte

4.2 Estrutura HTML

O widget está embutido em `templates/index.html`:

```
<!-- Botão/Bolha Flutuante -->
<div id="chatbot-trigger" class="chatbot-trigger chatleo-bubble">
  <div class="chat-bubble chatleo-bubble__avatar">
    <img src="/static/assets/logo.png" alt="Abrir chat">
    <div class="chat-bubble-notification" id="chatleo-badge" hidden>
      <span>0</span>
    </div>
  </div>
  <div class="chat-bubble-tooltip">
    <span>💬 Precisa de ajuda? Clique aqui!</span>
  </div>
</div>

<!-- Widget do Chatbot (Oculto inicialmente) -->
<div class="chatbot-widget" id="chatbot-widget">
  <!-- Header -->
  <div class="widget-header">
    <!-- Título, status, controles de acessibilidade, minimizar/fechar -->
  </div>

  <!-- Conteúdo -->
  <div class="widget-content">
    <div class="widget-messages" id="widget-messages">
      <!-- Mensagens renderizadas dinamicamente -->
    </div>
  </div>

  <!-- Input -->
  <div class="widget-input-container">
    <div class="widget-quick-actions">
      <!-- Botões de ação rápida -->
    </div>
    <div class="widget-input">
      <input type="text" id="widget-message-input">
      <button id="widget-send-btn">➤</button>
    </div>
  </div>
</div>
```

4.3 CSS Principal

O CSS está em `static/css/chatleo.css` e `static/css/style.css`.

Características:

- **Variáveis CSS** para personalização:

```
.chatbot-widget {
  --chat-primary: #3D7EFF;
  --chat-secondary: #8B5CF6;
}
```

- **Responsividade**: Media queries para mobile
- **Animações**: Transições suaves para abertura/fechamento
- **Acessibilidade**: Suporte a `prefers-reduced-motion`
- **Safe Area**: Suporte a safe area do iOS

4.4 JavaScript: Envio de Mensagens, Recepção, Indicador "Digitando..."

Envio de Mensagens

```
function sendMessage() {
    if (isWaiting) return; // Previne múltiplos envios

    const message = DOMElements.messageInput?.value.trim();
    if (!message) return;

    // Adiciona mensagem do usuário na UI
    addMessage(message, 'user');

    // Limpa input
    DOMElements.messageInput.value = '';

    // Mostra indicador "digitando..."
    showTypingIndicator();

    // Bloqueia input
    setLocked(true);

    // Envia para backend
    (async () => {
        try {
            const botResponse = await sendToBackend(message);
            hideTypingIndicator();
            addMessage(botResponse || 'Desculpe, estou indisponível.', 'bot');
            if (AppState.isTTSEnabled && botResponse) speakText(botResponse);
        } catch (err) {
            hideTypingIndicator();
            addMessage('Erro ao conectar ao assistente. Tente novamente.', 'bot');
        } finally {
            setLocked(false); // Libera input
        }
    })();
}
```

Recepção de Respostas

```
async function sendToBackend(message) {
    const sessionId = getOrCreateSessionId();

    const response = await fetch('/api/chat', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            message,
            session_id: sessionId
        })
    });

    if (response.ok) {
        const data = await response.json();
        // Atualiza session_id se backend retornou um diferente
        if (data.session_id && data.session_id !== sessionId) {
            localStorage.setItem('chat_session_id', data.session_id);
        }
        return data.response;
    }

    return null;
}
```

Indicador "Digitando..."

```
function showTypingIndicator() {
    const indicator = document.getElementById('widget-typing-indicator');
    if (!indicator) return;

    // Adiciona ao container de mensagens
    if (DOMElements.widgetMessages && DOMElements.typingIndicator) {
        DOMElements.widgetMessages.appendChild(DOMElements.typingIndicator);
    }

    // Mostra visualmente
    indicator.classList.add('active');
    indicator.removeAttribute('hidden');
    indicator.setAttribute('aria-hidden', 'false');

    // Anuncia para leitores de tela
    announceToScreenReader('O assistente está digitando');

    // Scroll para baixo
    scrollMessagesToBottom();
}
```

```
function hideTypingIndicator() {
    if (DOMElements.typingIndicator) {
        DOMElements.typingIndicator.classList.remove('active');
    }
    const indicator = document.getElementById('widget-typing-indicator');
    if (indicator) {
        indicator.setAttribute('hidden', 'true');
        indicator.setAttribute('aria-hidden', 'true');
    }
}
```

4.5 Funcionalidades de Acessibilidade

Text-to-Speech (TTS)

```
function speakText(text) {
    if (!AppState.isTTSEnabled || !('speechSynthesis' in window)) return;

    speechSynthesis.cancel(); // Para fala anterior

    const utterance = new SpeechSynthesisUtterance(text);
    utterance.voice = AppState.ttsVoice; // Voz em português
    utterance.rate = 0.9;
    utterance.pitch = 1;
    utterance.volume = 0.8;

    speechSynthesis.speak(utterance);
}
```

Alto Contraste

```
function toggleHighContrast() {
    AppState.isHighContrast = !AppState.isHighContrast;
    document.body.classList.toggle('high-contrast', AppState.isHighContrast);
    saveUserPreferences();
}
```

Tamanho de Fonte

```
function cycleFontSize() {
    const sizes = ['small', 'normal', 'large', 'extra-large'];
    const currentIndex = sizes.indexOf(AppState.currentFontSize);
    const nextIndex = (currentIndex + 1) % sizes.length;
    AppState.currentFontSize = sizes[nextIndex];
```

```
        document.body.className = document.body.className.replace(/font-\w+/g, '');
        document.body.classList.add(`font-${AppState.currentFontSize}`);
        saveUserPreferences();
    }
```

Navegação por Teclado

- **ESC**: Fecha widget
- **Alt + C**: Abre/fecha chat
- **Alt + M**: Minimiza widget
- **Enter**: Envia mensagem
- **Tab**: Navega entre elementos focáveis

---

5. Banco de Dados (Firestore)

5.1 Estrutura das Coleções

`conversations/`

Documento por sessão de conversa.

Campos:

- `session_id` (string): ID único da sessão
- `iniciadoEm` (timestamp): Quando a conversa começou (legado)
- `ultimaMensagemEm` (timestamp): Última atividade (legado)
- `created_at` (timestamp): Criação (padronizado)
- `updated_at` (timestamp): Última atualização (padronizado)
- `total_user_messages` (number): Contador de mensagens do usuário
- `total_bot_messages` (number): Contador de mensagens do bot
- `channel` (string): Canal de origem ("web")
- `status` (string): Status da conversa ("open" | "closed")
- `lead_stage` (string | null): Estágio do lead ("collecting" | "done" | null)
- `lead_done` (boolean): Se lead foi concluído
- `lead_data` (object): Dados do lead em coleta
  - `nome` (string)
  - `interesse` (string)
  - `cidade` (string)
  - `estado` (string)
  - `idade` (number)

Subcoleção: `messages/`

- Documentos com auto-ID
- Campos: `role`, `content`, `created_at`, `papel`, `texto`, `criadoEm` (legado), `metadata`

`leads/`

Documentos de leads completos.

Campos:

- `session_id` (string): ID da conversa origem
- `nome` (string)
- `email` (string, opcional)
- `cidade` (string)
- `estado` (string, UF)
- `idade` (number)
- `interesse` (string)
- `createdAt` (timestamp)

`settings/`

Configurações do sistema.

Documentos:

- `global/`: Configurações globais
  - `admin_theme` (string): "dark" | "light"
- `chat_config/`: Configurações do chat
  - `chat_title` (string)
  - `welcome_message` (string)
  - `bot_avatar` (string)
  - `user_avatar` (string)
  - `primary_color` (string, hex)
  - `secondary_color` (string, hex)

`admin_users/`

Usuários administradores.

Campos:

- `username` (string)
- `password_hash` (string): Hash bcrypt
- `created_at` (timestamp)
- `updated_at` (timestamp)

5.2 Exemplos de Documentos

Exemplo: Conversa

```
{
  "session_id": "sess_1704067200000_abc123",
  "iniciadoEm": "2024-01-01T10:00:00Z",
  "ultimaMensagemEm": "2024-01-01T10:15:00Z",
  "created_at": "2024-01-01T10:00:00Z",
```

```
  "updated_at": "2024-01-01T10:15:00Z",
  "total_user_messages": 5,
  "total_bot_messages": 5,
  "channel": "web",
  "status": "open",
  "lead_stage": "done",
  "lead_done": true,
  "lead_data": {
    "nome": "João Silva",
    "interesse": "Cursos de programação",
    "cidade": "Florianópolis",
    "estado": "SC",
    "idade": 18
  }
}
```

Exemplo: Mensagem

```
{
  "role": "user",
  "content": "Olá, quero saber sobre os cursos",
  "created_at": "2024-01-01T10:00:00Z",
  "papel": "user",
  "texto": "Olá, quero saber sobre os cursos",
  "criadoEm": "2024-01-01T10:00:00Z",
  "metadata": {
    "source": "web"
  }
}
```

Exemplo: Lead

```
{
  "session_id": "sess_1704067200000_abc123",
  "nome": "João Silva",
  "cidade": "Florianópolis",
  "estado": "SC",
  "idade": 18,
  "interesse": "Cursos de programação",
  "createdAt": "2024-01-01T10:10:00Z"
}
```

5.3 Como Salvar Conversas

```
# Criar/atualizar conversa
get_or_create_conversation(session_id)
```

```
# Salvar mensagem
save_message(session_id, "user", "Olá!")
save_message(session_id, "assistant", "Olá! Como posso ajudar?")
```

5.4 Como Salvar Mensagens

```
save_message(
    session_id="sess_123",
    role="user",  # ou "assistant"
    text="Mensagem aqui",
    meta={"source": "web", "type": "lead_question"}  # opcional
)
```

5.5 Como Buscar Histórico

```
# Buscar conversa
conversation = get_conversation(session_id)

# Buscar mensagens de uma conversa
messages = get_conversation_messages(session_id, limit=200)

# Buscar todas as conversas (com filtros)
conversations = get_all_conversations(limit=50, filters={"search": "sess_123"})
```

5.6 Regras de Segurança

Recomendações para Firestore Security Rules:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // Conversas: leitura/escrita apenas autenticada (ou pública se necessário)
    match /conversations/{sessionId} {
      allow read, write: if request.auth != null;
      // OU para permitir escrita pública (com validação de estrutura):
      // allow write: if request.resource.data.keys().hasAll(['session_id',
'created_at']);

      // Mensagens: mesma regra da conversa pai
      match /messages/{messageId} {
        allow read, write: if request.auth != null;
      }
    }

    // Leads: apenas leitura autenticada, escrita pública (com validação)
```

```
    match /leads/{leadId} {
      allow read: if request.auth != null;
      allow create: if request.resource.data.keys().hasAll(['nome', 'cidade',
'estado']);
    }

    // Settings: apenas leitura pública, escrita autenticada
    match /settings/{settingId} {
      allow read: if true;
      allow write: if request.auth != null;
    }

    // Admin users: apenas leitura/escrita autenticada
    match /admin_users/{username} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Nota: Ajuste as regras conforme sua necessidade de segurança.

---

6. Fluxos Importantes

6.1 Fluxo de Criação de Conversa

```
1. Usuário abre widget pela primeira vez
   └─> JavaScript: getOrCreateSessionId()
       └─> Verifica localStorage['chat_session_id']
       └─> Se não existe: gera sess_{timestamp}_{random}
       └─> Salva no localStorage

2. Usuário envia primeira mensagem
   └─> POST /api/chat com { message, session_id }

3. Backend recebe
   └─> Se session_id não fornecido: gera novo
   └─> get_or_create_conversation(session_id)
       └─> Verifica se conversations/{session_id} existe
       └─> Se não existe:
           └─> Cria documento com campos iniciais
           └─> created_at = SERVER_TIMESTAMP
           └─> total_user_messages = 0
           └─> total_bot_messages = 0
           └─> channel = "web"
           └─> status = "open"
       └─> Se existe:
           └─> Atualiza ultimaMensagemEm e updated_at

4. Salva primeira mensagem
   └─> save_message(session_id, "user", message)
```

```
        └─> Adiciona em conversations/{session_id}/messages
        └─> Incrementa total_user_messages
```

6.2 Fluxo de Envio de Mensagem

```
1. Usuário digita e envia
   └─> JavaScript: sendMessage()
       └─> Valida mensagem não vazia
       └─> addMessage(message, 'user') [UI]
       └─> showTypingIndicator()
       └─> setLocked(true)
       └─> sendToBackend(message)

2. Backend processa
   └─> Valida mensagem
   └─> get_or_create_conversation(session_id)
   └─> save_message(session_id, "user", message)

3. Processa resposta
   └─> Se lead em coleta: processa campo
   └─> Se lead concluído: chatbot.gerar_resposta(message)

4. Salva resposta
   └─> save_message(session_id, "assistant", response)

5. Retorna para frontend
   └─> JSON: { response, session_id }

6. Frontend renderiza
   └─> hideTypingIndicator()
   └─> addMessage(response, 'bot')
   └─> setLocked(false)
```

6.3 Fluxo de Resposta do Bot

```
1. Backend recebe mensagem
   └─> chatbot_web.gerar_resposta(user_message)

2. Chatbot processa (utils/responder.py)
   └─> Verifica se chat_session existe
   └─> Se não: reinicializa modelo e envia contexto
   └─> Envia mensagem para Gemini: chat_session.send_message(pergunta)
   └─> Gemini processa com:
       - Contexto inicial (prompt de sistema)
       - Histórico da sessão
       - Base de conhecimento (dados.json)

3. Gemini retorna resposta
   └─> Extrai texto da resposta
```

```
   └─> Retorna string

4. Backend salva resposta
   └─> save_message(session_id, "assistant", response)

5. Retorna para frontend
```

6.4 Fluxo de Salvar no Firestore

```
1. Função save_message() chamada
   └─> Verifica se Firestore habilitado
   └─> Normaliza role ("assistant" → "bot")
   └─> Monta dados da mensagem (novos + legados)
   └─> Adiciona documento em conversations/{session_id}/messages
   └─> Atualiza contadores da conversa:
       - Se user: incrementa total_user_messages
       - Se bot: incrementa total_bot_messages
   └─> Atualiza ultimaMensagemEm e updated_at da conversa

2. Se erro: retorna False (silencioso, não quebra fluxo)
```

6.5 Fluxo do Admin

Login

```
1. Acessa /admin/login
   └─> Renderiza template admin/login.html

2. Usuário preenche credenciais
   └─> POST /admin/api/login
       └─> verify_admin_password(username, password)
           └─> Se Firestore desabilitado: fallback local (admin/admin123)
           └─> Se Firestore habilitado: busca em admin_users/{username}
               └─> Compara hash com check_password_hash()
       └─> Se válido: session['admin_logged'] = True
       └─> Retorna { ok: true }

3. Redireciona para /admin/
```

Dashboard

```
1. Acessa /admin/
   └─> Middleware: require_admin_login()
       └─> Verifica session['admin_logged']
   └─> Renderiza template admin/dashboard.html
```

```
2. Frontend carrega dados
   └─> GET /admin/api/reports?days=7
       └─> get_conversation_counts(days=7)
       └─> get_message_counts_by_role()
       └─> get_daily_conversation_counts(days=7)
       └─> get_recent_conversations(limit=10)
       └─> get_leads_count_by_city()
       └─> get_leads_count_by_state()
       └─> get_leads_count_by_age_range()
   └─> Renderiza gráficos e métricas
```

Visualizar Conversas

```
1. Acessa /admin/conversations
   └─> Renderiza template admin/conversations.html

2. Frontend carrega lista
   └─> GET /admin/api/conversations?search=sess_123
       └─> get_all_conversations(limit=50, filters={"search": "sess_123"})
   └─> Renderiza tabela

3. Usuário clica em conversa
   └─> GET /admin/api/conversations/{session_id}/messages
       └─> get_conversation_messages(session_id, limit=200)
   └─> Renderiza mensagens em modal ou sidebar
```

---

7. Como Rodar o Projeto Localmente

7.1 Pré-requisitos

- **Python 3.8+** instalado
- **pip** (gerenciador de pacotes Python)
- **Conta Google** (para Gemini API e Firebase)
- **Git** (opcional, para clonar repositório)

7.2 Passo a Passo

1. Clone o Repositório (ou baixe os arquivos)

```
git clone https://github.com/moaaskt/ChatJovemProgramador.git
cd ChatJovemProgramador
```

2. Crie e Ative um Ambiente Virtual

Windows:

```
python -m venv venv
.\venv\Scripts\activate
```

macOS/Linux:

```
python3 -m venv venv
source venv/bin/activate
```

3. Instale as Dependências

```
pip install -r requirements.txt
```

Dependências instaladas:

- flask==2.3.2
- flask-cors==3.0.10
- google-generativeai==0.3.2
- python-dotenv==1.0.0
- firebase-admin>=6.0.0
- beautifulsoup4 (implícito, usado no scraper)
- requests (implícito, usado no scraper)

4. Configure as Variáveis de Ambiente

Crie um arquivo `.env` na raiz do projeto:

```
# Chave da API do Google Gemini (OBRIGATÓRIA)
GEMINI_API_KEY="sua_chave_gemini_aqui"

# Configuração do Firestore (OPCIONAL)
# Se false, o chat funciona sem persistência
AI_FIRESTORE_ENABLED=false

# Se AI_FIRESTORE_ENABLED=true, configure abaixo:
FIREBASE_CREDENTIALS='{"type":"service_account","project_id":"seu-
projeto","private_key_id":"...","private_key":"...","client_email":"...","client_i
d":"...","auth_uri":"...","token_uri":"...","auth_provider_x509_cert_url":"...","c
lient_x509_cert_url":"..."}'
FIREBASE_PROJECT_ID="seu-projeto-id"

# Chave secreta do Flask (para sessões)
FLASK_SECRET_KEY="sua-chave-secreta-aqui-mude-em-producao"
```

Como obter GEMINI_API_KEY:

1. Acesse Google AI Studio
2. Crie uma nova API key
3. Copie e cole no `.env`

Como obter FIREBASE_CREDENTIALS (se usar Firestore):

1. Acesse Firebase Console
2. Crie um projeto ou selecione existente
3. Vá em **Project Settings** → **Service Accounts**
4. Clique em **Generate new private key**
5. Baixe o JSON
6. Copie o conteúdo do JSON e cole como string em `FIREBASE_CREDENTIALS` (ou use caminho do arquivo)

5. Execute o Scraper (Atualizar Base de Conhecimento)

```
python utils/scraper.py
```

Isso irá:

- Acessar o site oficial do Jovem Programador
- Extrair todas as informações (sobre, dúvidas, notícias, etc.)
- Salvar em `dados.json` na raiz do projeto

Tempo estimado: 2-5 minutos (depende da quantidade de notícias)

6. Inicie o Servidor Flask

```
python app.py
```

Você verá logs como:

```
🤖 Inicializando o Chatbot com Gemini...
[Gemini] SDK version: ...
[Gemini] Modelo selecionado: models/gemini-pro-latest
✅ Chatbot pronto e online!
[DEBUG] Iniciando Firestore...
[Firestore] Desabilitado (AI_FIRESTORE_ENABLED=false)
 * Running on http://0.0.0.0:5000
```

7. Acesse o Chatbot

Abra seu navegador e acesse:

```
http://localhost:5000
```

O widget de chat estará disponível no canto inferior direito da página.

7.3 Configurações Adicionais

Habilitar Firestore (Persistência)

1. Configure `AI_FIRESTORE_ENABLED=true` no `.env`
2. Configure `FIREBASE_CREDENTIALS` e `FIREBASE_PROJECT_ID`
3. Reinicie o servidor

Acessar Painel Admin

1. Com Firestore habilitado:
    - Acesse `http://localhost:5000/admin/login`
    - Usuário padrão: `admin`
    - Senha padrão: `admin123`
2. Sem Firestore:
    - Mesmo usuário/senha (fallback local)

7.4 Problemas Comuns e Soluções

Erro: "GEMINI_API_KEY ausente no .env"

Solução: Verifique se o arquivo `.env` existe e contém `GEMINI_API_KEY="sua_chave"`

Erro: "Arquivo 'dados.json' não encontrado"

Solução: Execute `python utils/scraper.py` primeiro

Erro: "Firestore não inicializado"

Solução:

- Verifique se `AI_FIRESTORE_ENABLED=true` no `.env`
- Verifique se `FIREBASE_CREDENTIALS` está correto (JSON válido)
- Verifique se o projeto Firebase existe

Erro: "Port 5000 already in use"

Solução:

- Feche outro processo usando a porta 5000
- Ou altere a porta em `app.py`: `app.run(debug=True, port=5001, host='0.0.0.0')`

Widget não aparece

Solução:

- Verifique o console do navegador (F12) para erros JavaScript
- Verifique se os arquivos estáticos estão sendo servidos (`/static/css/chatleo.css`)
- Limpe o cache do navegador

Mensagens não são salvas

Solução:

- Verifique se `AI_FIRESTORE_ENABLED=true`
- Verifique logs do servidor para erros do Firestore
- O chat funciona mesmo sem Firestore (mas não persiste)

---

8. Deploy

8.1 Como Fazer Deploy em Produção

Opções de Deploy

1. **Heroku**
2. **Railway**
3. **Render**
4. **Google Cloud Run**
5. **AWS Elastic Beanstalk**
6. **VPS próprio** (DigitalOcean, Linode, etc.)

Exemplo: Deploy no Heroku

1. Instale Heroku CLI

```
# Windows: baixe do site
# macOS: brew install heroku/brew/heroku
# Linux: snap install heroku
```

2. Crie arquivo `Procfile`

```
web: gunicorn app:app
```

3. Atualize `requirements.txt` (adicione gunicorn)

```
gunicorn==21.2.0
```

4. Configure variáveis de ambiente no Heroku

```
heroku config:set GEMINI_API_KEY="sua_chave"
heroku config:set AI_FIRESTORE_ENABLED="true"
heroku config:set FIREBASE_CREDENTIALS='{"type":"service_account",...}'
heroku config:set FIREBASE_PROJECT_ID="seu-projeto"
heroku config:set FLASK_SECRET_KEY="chave-secreta-forte"
```

5. Faça deploy

```
heroku create seu-app
git push heroku main
heroku run python utils/scraper.py  # Executar scraper uma vez
```

8.2 Configuração de Variáveis de Ambiente em Produção

Variáveis obrigatórias:

- `GEMINI_API_KEY`: Chave da API do Gemini
- `FLASK_SECRET_KEY`: Chave secreta forte (gere com `secrets.token_hex(32)`)

Variáveis opcionais (se usar Firestore):

- `AI_FIRESTORE_ENABLED`: "true" ou "false"
- `FIREBASE_CREDENTIALS`: JSON string das credenciais
- `FIREBASE_PROJECT_ID`: ID do projeto Firebase

Como gerar FLASK_SECRET_KEY:

```
import secrets
print(secrets.token_hex(32))
```

8.3 Dependências Obrigatórias

Certifique-se de que todas as dependências estão em `requirements.txt`:

```
flask==2.3.2
flask-cors==3.0.10
google-generativeai==0.3.2
python-dotenv==1.0.0
firebase-admin>=6.0.0
gunicorn==21.2.0  # Para produção
```

8.4 Observações Importantes

Segurança

- **NUNCA** commite o arquivo `.env` no Git
- Use variáveis de ambiente do provedor de deploy
- Gere `FLASK_SECRET_KEY` forte e único
- Configure Firestore Security Rules adequadamente

Performance

- Execute o scraper periodicamente (cron job ou scheduler) para atualizar `dados.json`
- Considere cachear respostas do Gemini se necessário
- Monitore uso da API do Gemini (limites de quota)

Escalabilidade

- Para múltiplas instâncias, use banco de dados compartilhado (Firestore)
- Considere usar Redis para cache de sessões (opcional)
- Configure CORS adequadamente para seu domínio

Monitoramento

- Configure logs de erro (Sentry, Loggly, etc.)
- Monitore métricas do Firestore (leituras/escritas)
- Monitore uso da API do Gemini

---

9. Checklist Final

✅ Arquitetura

- ☑ Arquitetura em camadas documentada
- ☑ Fluxo de dados explicado (Backend → Firestore → Frontend → Admin)
- ☑ Estrutura de pastas detalhada
- ☑ Responsabilidades de cada módulo definidas

✅ Fluxos

- ☑ Fluxo de criação de conversa documentado
- ☑ Fluxo de envio de mensagem explicado
- ☑ Fluxo de resposta do bot detalhado
- ☑ Fluxo de salvar no Firestore descrito
- ☑ Fluxo do admin documentado

✅ Código Explicado

- ☑ Métodos principais documentados (`init_admin`, `get_or_create_conversation`, `save_message`, `gerar_resposta`)
- ☑ Rotas do Flask explicadas
- ☑ Estrutura do frontend (HTML, CSS, JS) documentada

- ☑ Funcionalidades de acessibilidade explicadas

☑ Banco de Dados Documentado

- ☑ Estrutura das coleções Firestore detalhada
- ☑ Exemplos de documentos fornecidos
- ☑ Como salvar conversas/mensagens explicado
- ☑ Como buscar histórico documentado
- ☑ Regras de segurança sugeridas

☑ Instalação e Deploy

- ☑ Passo a passo para instalação local completo
- ☑ Configuração de variáveis de ambiente explicada
- ☑ Problemas comuns e soluções listados
- ☑ Guia de deploy em produção fornecido
- ☑ Observações importantes documentadas

☑ Tecnologias Usadas

- ☑ Lista completa de tecnologias (Backend, Frontend, Banco de Dados)
- ☑ Versões das dependências especificadas
- ☑ Propósito de cada tecnologia explicado

☑ Explicações Técnicas + Didáticas

- ☑ Linguagem clara e profissional
- ☑ Títulos e subtítulos organizados
- ☑ Exemplos de código fornecidos
- ☑ Diagramas ASCII de fluxos incluídos
- ☑ Explicações passo a passo detalhadas

📝 Notas Finais

Esta documentação cobre todos os aspectos técnicos do projeto **ChatJovemProgramador**. Para dúvidas ou sugestões de melhoria, consulte o código-fonte ou entre em contato com a equipe de desenvolvimento.

Última atualização: Janeiro 2025

Versão do Projeto: Staging

🖇 Links Úteis

- Documentação Flask
- Documentação Google Gemini
- Documentação Firestore
- Documentação Firebase Admin SDK

Desenvolvido com ♡ pela equipe do SENAC Palhoça