# 📚 SchoolPass - Sistema de Catraca Inteligente

Esta documentação técnica detalha a arquitetura, os componentes e a lógica interna do projeto SchoolPass, um sistema de catraca com reconhecimento facial.

**GitHub:** SchoolPass (Substitua pelo link real do GitHub)

# 1. Visão Geral da Arquitetura

O SchoolPass é um sistema híbrido que combina uma **aplicação desktop de visão computacional** (para a catraca) com um **backend Django** (para gestão de dados e interface administrativa).

**Componentes Principais:**

- **Catraca (Desktop Application):** Um executável Python que interage com a câmera e o banco de dados localmente para realizar o reconhecimento facial e registrar acessos.
- **Backend Django:** Um projeto web (administrado via navegador) que gerencia o cadastro de usuários, a grade horária e visualiza os registros de acesso.
- **Banco de Dados (SQLite3):** Compartilhado entre a aplicação desktop da catraca e o backend Django.

**Fluxo de Dados Simplificado:**

1. A câmera da catraca captura frames de vídeo.
2. O script Python (`interfaceFullscreen.py` ou `interface padrao.py`) processa os frames para detecção e reconhecimento facial.
3. As aplicações Python interagem com o `db.sqlite3` para buscar dados de usuários e grades, e para registrar acessos.
4. A interface visual da catraca exibe mensagens de status e informações relevantes.
5. O painel Django acessa o mesmo `db.sqlite3` para permitir a gestão dos dados e visualização dos registros.

# 2. Estrutura do Projeto

Com base na estrutura fornecida, o projeto está organizado da seguinte forma:

```
.
├── manage.py
├── db.sqlite3
├── README.md
```

```
├── haarcascade_frontalface_default.xml # Arquivo do classificador Haar Cascade do OpenCV
├── interface padrao.py              # Script da interface padrão da catraca
├── interfaceFullscreen.py            # Script da interface de catraca em tela cheia
├── media/
│   ├── fotos/                # Fotos capturadas pela catraca
│   └── fotos_usuarios/       # Fotos dos usuários autorizados para reconhecimento
├── reconhecimento_facial/    # Pasta raiz do projeto Django (contém settings.py, urls.py, etc.)
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   ├── wsgi.py
│   └── asgi.py
├── cadastro_face/            # App Django de cadastro de faces/usuários
│   ├── migrations/
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── models.py             # Contém CadastroFaceUsuario
│   ├── tests.py
│   └── views.py
└── gestor/                   # App Django de gestão (registros de acesso, grade horária)
    ├── migrations/
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── models.py             # Contém GestorRegistroAcesso, GestorGradeHoraria
    ├── tests.py
    └── views.py
```

# 3. Módulos e Componentes Essenciais

## 3.1. `interfaceFullscreen.py` e `interface padrao.py` (Aplicação Desktop da Catraca)

Esses scripts são o coração da funcionalidade da catraca, rodando como executáveis autônomos. A lógica é similar entre eles, com a principal diferença sendo a configuração da janela (tela cheia vs. padrão).

**Imports Chave:** `cv2`, `face_recognition`, `os`, `sqlite3`, `datetime`, `sys`.

**Configuração de Diretórios e DB:**

- **BASE_DIR:** Dinamicamente determinado para ser o diretório do executável (se congelado) ou do script (se não congelado), garantindo o acesso correto ao `db.sqlite3` e às pastas `media/`.
- **DB_NAME:** Caminho para o banco de dados SQLite.
- **output_dir:** Caminho para salvar as fotos capturadas (`media/fotos`).
- **banco_faces_dir:** Caminho para as fotos dos usuários autorizados (`media/fotos_usuarios`).
- `get_db_connection()`: Estabelece e retorna uma conexão SQLite. Configura `row_factory` para facilitar o acesso a colunas por nome. Inclui tratamento de erro crítico para falha na conexão.
- `ID_USUARIO_DESCONHECIDO_OU_ERRO`: Constante inteira para um `usuario_id` *fallback* em `gestor_registroacesso` quando um usuário não é reconhecido ou um erro ocorre. Deve corresponder a um registro no DB.

**`get_usuario_id_by_name(nome_usuario_completo)`:**

- **Função:** Busca o `id` e `curso_id` de um usuário na tabela `cadastro_face_usuario` (do app `cadastro_face`).
- **Lógica:** Remove o sufixo `_foto` do `nome_usuario_completo` antes de consultar o DB, para corresponder aos nomes da tabela.
- **Retorno:** `(usuario_id, curso_id)` ou `(None, None)`.

**`dia_semana()`:** Retorna o nome do dia da semana atual em português (ex: "Segunda").

**`busca_sala(curso_id)`:**

- **Função:** Consulta a tabela `gestor_gradehoraria` (do app `gestor`) para encontrar a sala com base no `curso_id` e no `dia_semana` atual.
- **Retorno:** O nome da sala (string) ou "Funcionario" (se `curso_id` for `None`) ou "N/A" se não encontrada.

**`inserir_registro_acesso_db(nome_reconhecido, status_acesso_bool, usuario_id, sala_atual)`:**

- **Função:** Insere um registro na tabela `gestor_registroacesso` (do app `gestor`).
- **Colunas:** `usuario_id`, `sala`, `horario_entrada` (usando `DATETIME('now', 'localtime')` do SQLite), e `status` (booleano).
- **Tratamento de Erro:** Captura exceções `sqlite3.Error`.

**`carregar_rostos_autorizados(diretorio)`:**

- **Função:** Carrega as codificações faciais das imagens na pasta `media/fotos_usuarios`.
- **Lógica:** Itera sobre os arquivos, carrega cada imagem, tenta encontrar uma codificação facial. Ignora arquivos sem rostos.
- **Retorno:** Listas de codificações e nomes correspondentes.

**Loop Principal (`while True`):**

- Captura frames da câmera (`cv2.VideoCapture`).
- Detecta rostos (`detect_bounding_box` usando Haar Cascade).
- Se um rosto é detectado continuamente por 5 segundos:
    - Salva a foto em `media/fotos`.
    - Realiza o reconhecimento facial (`face_recognition.compare_faces`).
    - Com base no resultado:
        - **Acesso Liberado:** Busca `usuario_id` e `curso_id`. Se `usuario_id` for válido, busca a sala e insere o registro com `status=True`.
        - **Acesso Liberado (ID não encontrado):** Usa `ID_USUARIO_DESCONHECIDO_OU_ERRO` e `status=True` (com aviso).
        - **Acesso Negado:** Insere registro com `nome="DESCONHECIDO"`, `status=False`, e `ID_USUARIO_DESCONHECIDO_OU_ERRO`.
        - **Erro de Detecção/Inesperado:** Insere registro com detalhes do erro, `status=False`, e `ID_USUARIO_DESCONHECIDO_OU_ERRO`.
- Exibe mensagens visuais na tela (`cv2.putText`) com fundo para melhor legibilidade.
- Atualiza a janela (`cv2.imshow`) e espera pela tecla 'q' para sair.

## 3.2. App `cadastro_face`

**`cadastro_face/models.py`:**

- **`CadastroFaceUsuario`:** Modelo Django para armazenar informações dos usuários (alunos/funcionários).
    - `id` (PrimaryKey, gerado automaticamente)
    - `nome` (CharField)
    - `tipo` (CharField, ex: "aluno", "funcionario")
    - `matricula` (CharField)
    - `foto` (ImageField/CharField - caminho para a imagem da face)
    - `curso_id` (ForeignKey para o modelo de Curso, ou IntegerField simples se não houver modelo de Curso separado).

## 3.3. App `gestor`

**`gestor/models.py`:**

- **`GestorRegistroAcesso`:** Modelo Django para registrar cada evento de acesso.
    - `id` (PrimaryKey, gerado automaticamente)
    - `usuario_id` (ForeignKey para CadastroFaceUsuario, `null=False` requer um ID válido ou o ID do usuário curinga).
    - `sala` (CharField - a sala associada ao acesso, pode ser "N/A").
    - `horario_entrada` (DateTimeField - timestamp do acesso).
    - `status` (BooleanField - `True` para acesso permitido, `False` para negado/erro).
- **`GestorGradeHoraria`:** Modelo Django para armazenar a grade de matérias.
    - `id` (PrimaryKey, gerado automaticamente)
    - `curso_id` (ForeignKey para o modelo de Curso, ou IntegerField).
    - `dia_semana` (CharField - ex: "Segunda", "Terça").
    - `horario_inicio` (TimeField).
    - `horario_fim` (TimeField).
    - `sala` (CharField).
    - `materia` (CharField - nome da disciplina/atividade).

# 4. Banco de Dados (SQLite3)

O `db.sqlite3` é o arquivo de banco de dados que contém as tabelas criadas a partir dos modelos Django (`cadastro_face_usuario`, `gestor_registroacesso`, `gestor_gradehoraria`). É um banco de dados local adequado para fins de desenvolvimento e implantações de pequena escala.

**Estrutura da Tabela `cadastro_face_usuario`:**

- `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
- `nome` (TEXT)
- `tipo` (TEXT)
- `matricula` (TEXT)
- `foto` (TEXT - caminho relativo da foto)
- `curso_id` (INTEGER - chave estrangeira ou ID do curso)

**Estrutura da Tabela `gestor_registroacesso`:**

- `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
- `usuario_id` (INTEGER - FOREIGN KEY para `cadastro_face_usuario.id`, NOT NULL)
- `sala` (TEXT)

- `horario_entrada` (DATETIME)
- `status` (BOOLEAN - 0 para False, 1 para True)

**Estrutura da Tabela `gestor_gradehoraria`:**

- `id` (INTEGER PRIMARY KEY AUTOINCREMENT)
- `curso_id` (INTEGER - chave estrangeira ou ID do curso)
- `dia_semana` (TEXT - ex: "Segunda")
- `horario_inicio` (TEXT - ex: "08:00")
- `horario_fim` (TEXT - ex: "09:00")
- `sala` (TEXT)
- `materia` (TEXT)

# 5. Processo de Execução (Executável)

O script `interfaceFullscreen.py` é transformado em um executável autônomo (`CatracaFacial.exe`) usando PyInstaller.

- **Empacotamento:** O PyInstaller inclui todas as dependências Python (`opencv`, `face_recognition`, `numpy`, etc.) e os arquivos de dados (Haar Cascades do OpenCV, modelos do `face_recognition`, fotos de usuários) dentro do executável ou em um diretório temporário.
- **Caminhos Relativos:** A lógica de `BASE_DIR` garante que o executável encontre o `db.sqlite3` e as pastas `media/fotos` e `media/fotos_usuarios` no mesmo diretório em que o `.exe` é executado.
- **Interface:** A aplicação OpenCV abre uma janela de vídeo não-fullscreen, exibindo o *feed* da câmera e mensagens de status.
- **Interação com DB:** O executável interage diretamente com o `db.sqlite3` (o mesmo arquivo usado pelo Django) para persistir e buscar dados.

# 6. Considerações de Desenvolvimento e Manutenção

- **Sincronização de Dados:** É crucial manter as fotos em `media/fotos_usuarios` e os dados na tabela `cadastro_face_usuario` sincronizados. Adicionar/remover um usuário no Django Admin deve ser acompanhado da adição/remoção da foto correspondente e da reconstrução do executável (se as fotos estiverem empacotadas).
- **Escalabilidade:** Para maior escala ou ambientes multiusuário, o SQLite3 seria substituído por um banco de dados mais robusto (PostgreSQL, MySQL). A comunicação da catraca com o backend poderia evoluir para APIs RESTful (como a ideia original de *upload* de *frames*).

- **Tratamento de Erros:** O código inclui `try-except blocks` para lidar com erros comuns (falha de DB, rosto não detectado), mas um tratamento mais robusto pode ser adicionado.
- **Desempenho:** O reconhecimento facial pode ser intensivo. Otimizações de *hardware* (GPUs) ou algoritmos mais leves podem ser necessárias em ambientes de alta demanda.
- **Segurança:** Para um ambiente real, senhas de acesso ao DB e proteção contra adulteração do executável seriam cruciais.

# 7. Melhorias Futuras Potenciais

- **Interface Web em Tempo Real:** Desenvolver a interface web da catraca (*frontend* JavaScript) que envia *frames* para uma API Django para reconhecimento, permitindo o uso via navegador.
- **Sincronização Automática de Fotos:** Implementar um mecanismo para que o executável da catraca possa sincronizar as fotos dos usuários autorizados diretamente do backend Django, eliminando a necessidade de reconstruir o executável a cada nova foto.
- **Controle da Catraca Física:** Integrar com *hardware* de catraca real para liberação automática (saída de sinal).
- **Relatórios Avançados:** Gerar relatórios mais complexos no painel do gestor (ex: presença por turma, pico de acessos).
- **Detecção de Vida (*Liveness Detection*):** Para prevenir fraudes com fotos ou vídeos.

**Integrantes:**


- Alceu Scandolara - 202403624524
- Antonio Carlos Sena da Conceição Junior - 202102120748
- Gustavo Cerqueira Bonfim Oliveira - 202303392877
- Javier Ferreira dos Santos - 202051415321
- Gabriel Bomfim da Rocha Dias - 202302375057