

QAjuda - Plataforma web de voluntariado

Documento de Arquitetura do Sistema

Histórico da Revisão

Data	Versão	Descrição	Autoria
01/11/2023	1.0	Criação inicial do documento	Ana Célia, Felipe Xavier, José Vilanir, Matheus Duarte e Yuri Thairony
02/01/2024	1.1	Revisão e verificação do documento	Ana Célia, Felipe Xavier, José Vilanir, Matheus Duarte e Yuri Thairony

1. Introdução

Este documento refere-se ao sistema em desenvolvimento “QAjuda - Plataforma web de voluntariado” e utilizou como ponto de partida o Documento de Visão do Projeto, que detalha o problema, perfis, ambiente e principais necessidade dos usuários, concorrentes, visão geral do projeto e a lista de requisitos funcionais e não funcionais, modelo de domínio e entidade-relação (ver Anexos). Neste documento estão descritos os principais aspectos do sistema, definindo o projeto, implementação da arquitetura além dos objetivos, limitações, decisões e justificativas e detalhamento da visão arquitetural do Projeto em questão.

O PDSD consiste na segunda etapa do processo continuado dos PDS, que é realizado em grupos ao longo de três semestres, durante a formação no curso de TADS do IFRN. Cada semestre possui uma ênfase: começando com PDS Web, depois PDSD e, por fim, o PDS Corporativo (PDSC). Este documento refere-se ao desenho arquitetural do grupo QAjuda durante o desenvolvimento de sistemas distribuídos e dá continuidade ao que foi trabalhado no PDS Web: a criação de uma plataforma web com o objetivo de conectar usuários interessados em realizar ações de trabalho voluntário.

Contudo, os padrões definidos para o desenvolvimento desta plataforma aqui apresentados poderão ser futuramente revisitados e, conforme a necessidade e a busca pela melhoria contínua, poderão ser modificados.

Em suma, neste documento apresentam-se as tecnologias, ferramentas e frameworks que serão utilizados no desenvolvimento da segunda versão do QAjuda, durante o PDSD, e que poderão continuar a dar suporte nas etapas futuras de desenvolvimento da plataforma no PDSC. Nesse sentido, o objetivo arquitetural deste projeto é listar as decisões sobre tecnologias, ferramentas e frameworks que deverão ser utilizados e darão suporte para o desenvolvimento da plataforma QAjuda.

2. Termos e Abreviações

PDS - Projeto de Desenvolvimento de Sistema;

PDSD – Projeto de Desenvolvimento de Sistema Distribuído;

TADS – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas;

IFRN - Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte.

3. Requisitos Significantes

A partir de aplicação da metodologia do Mini-QAW de levantamento de Requisitos Arquiteturalmente Significantes (RAS), referentes a atributos da qualidade, serão apresentados na sequência os RAS prioritários, enumerados de 1 a 6, que tiveram sua aplicação implementada na fase atual do sistema QAjuda.

- Atributo de qualidade “Modificabilidade”: RAS 1 - “Implementação de testes”;
- Atributo de qualidade “Usabilidade”: RAS 2 - “Design de interface sóbrio”; RAS 3 - “Esclarecimento participação em cada ação (ajudar ou participar?)”;
- Atributo de qualidade “Desempenho”: RAS 4 - “Realização da integração do back e front”; 5 - RAS “Testes de carga e estresse”;
- Atributo de qualidade “Segurança”: RAS 6 - “Proteger dados pessoais”.

Seguindo a mesma ordem de numeração dos RAS de 1 a 6, na sequência estão detalhados no Quadro 1 os cenários para cada requisito elencado como prioritário.

Parte do Cenário	RAS 1	RAS 2	RAS 3	RAS 4	RAS 5	RAS 6
Source (Origem)	Desenvol-v edor	Desenvol-v edor	500 Usuários	Usuário	Usuário	Invasor
Stimulus (Estímulo)	Submete requisição de teste parâmetro s errados	Cria requisi- ções do front para o back	Inicia 1000 requests em 20 segundos	Utiliza a platafor- ma	Usuário tipo "Colabora- dor" clica em participar de uma ação	Ataque para coleta de dados indevidos
Artifact (Artefato)	Api	Sistema	Sistema	Sistema	Sistema	Database
Environment (Ambiente)	Desenvol- vimento	Desenvol- vimento	Ambiente de teste	Interface gráfica como ambiente de testes	Interface gráfica como ambiente de testes	Operacio- nal
Response (Resposta)	Teste unitário ou funcional informa erro	Back responde as solicita- ções do front	Processa todos os requests	Re-design da interface gráfica	Retorna mensagem informand o que está incluído na ação como participant e ou como voluntário	O sistema deve estar apto a identificar e remover o acesso não autorizado através de código automati- zados
Response Measure (Medição da resposta)	< 10 erros / mês	Retorno em < 1 segundo	Retorno em no máximo 3 segundos	Pesquisa de satisfação com usuários (>= 50%)	Retorno em no máximo 1 segundo	O acesso não autorizado deve ser removido e o usuário afetado notificado

						em no máximo 5 minutos
--	--	--	--	--	--	------------------------------

4. Restrições Arquiteturais

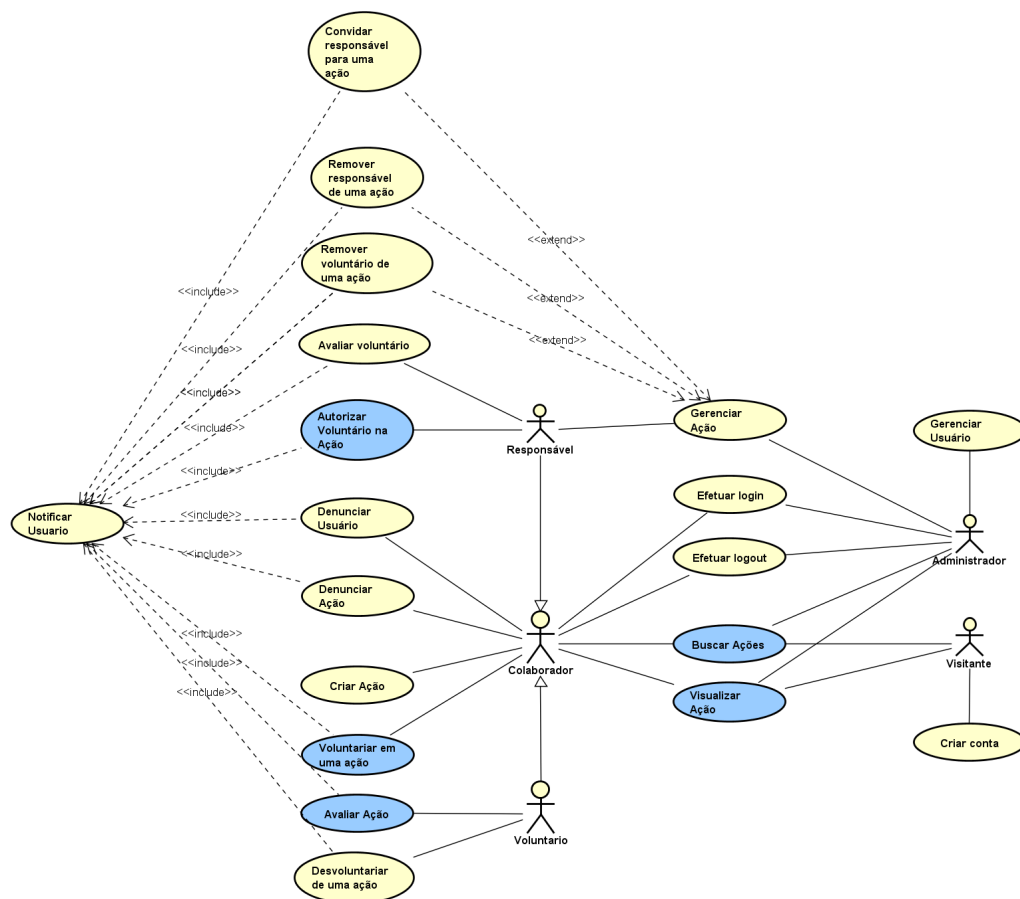
4.1. Restrições técnicas

	Restrição	Contexto e/ou Motivação
Restrição de software e programação		
RT1	Uso do framework Django para backend	Decidiu-se pelo uso do framework Django pois é um framework muito utilizado no mercado mundial (grandes empresas utilizam como Disqus, Instagram, Youtube, entre outras), possui boa comunidade e documentação (livros, vídeos e documentos); e a equipe de desenvolvimento trabalhou no semestre anterior (PDS Web) com esta tecnologia, o que diminuiu a curva de aprendizado de um novo framework.
RT2	Uso do framework Angular para frontend	Decidiu-se pela utilização do framework Angular pois a equipe desenvolvedora não possuía experiência com esse tipo de tecnologia, e como teria que aprender um framework, optou-se pelo acompanhamento do conteúdo da disciplina "Interfaces Ricas", que priorizou a formação no uso do Angular e, assim, foi determinante para a escolha dessa tecnologia para o frontend.
Restrição de sistema operacional		
RT3	Uso do sistema operacional Windows	Os ambientes de desenvolvimento, homologação e produção foram executados e pensados para o ambiente Windows.

5. Escopo do Sistema e Contexto

5.1. Diagrama de Casos de Uso

Diagrama de CDU (destaque em azul para os principais)



Para a atual atual de construção do sistema QAjuda, foram eleitos quatro principais casos de uso, que foram implementados e integrados com a API, conforme detalhados a seguir. Destaca-se também que esses CDU prioritários levaram em consideração os atributos de qualidade e os os RAS eleitos como prioritários e apresentados na seção 3 - Requisitos Significantes deste documento.

- **CDU001. Autorizar voluntário na ação**

- **Ator principal:** Responsável
- **Atores secundários:** Voluntário, colaborador
- **Resumo:** Responsável da ação poderá autorizar voluntários em uma ação.
- **Pré-condição:** 1 - ação existir; 2 - colaborador ser o responsável da ação; 3 - colaborador ter solicitado ser voluntário da ação.
- **Pós-Condição:** Colaborador se torna voluntário da ação.

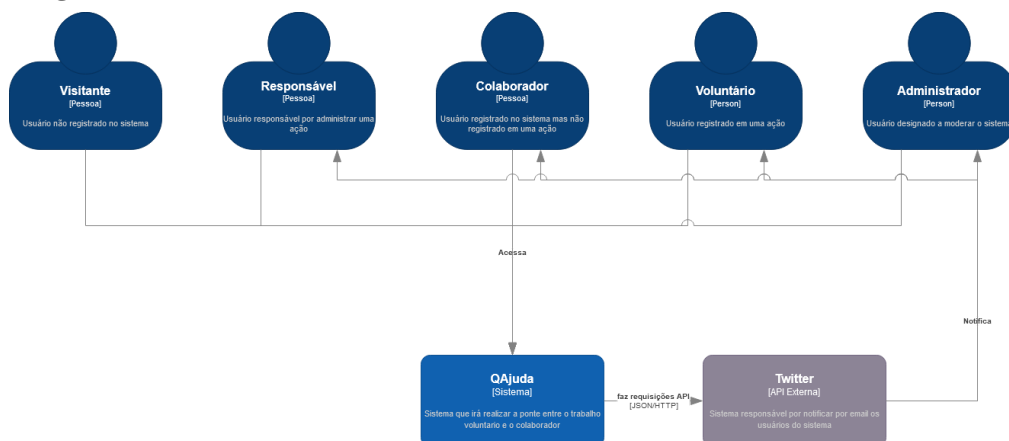
- **CDU002. Voluntariar em uma ação**

- **Ator principal:** Colaborador
- **Atores secundários:** Responsável, voluntário

- **Resumo:** Colaborador cadastra-se para se tornar voluntário de uma ação
- **Pré-condição:** 1 - Existir uma ação; 2 - Ação estar ativa; 3 - Colaborador estar logado.
- **Pós-Condição:** Ser aceito pelo responsável e se tornar um voluntário
- **CDU003. Visualizar Ação**
 - **Ator principal:** Visitante
 - **Atores secundários:** Colaborador, voluntário, responsável
 - **Resumo:** Colaborador pode visualizar os detalhes de uma ação.
 - **Pré-condição:** 1 - ação existir; 2 - ação estar ativa.
 - **Pós-Condição:** Caso visitante queira inscrever-se na ação.
- **CDU004. Buscar ações**
 - **Ator principal:** Visitante
 - **Atores secundários:** Colaborador, responsável, voluntário
 - **Resumo:** Colaborador pesquisa por ação por nome que contenha no título.
 - **Pré-condição:** 1 - existir ação cadastrada.
 - **Pós-Condição:** não há.

5.2. Diagrama de Contexto

Diagrama de Contexto

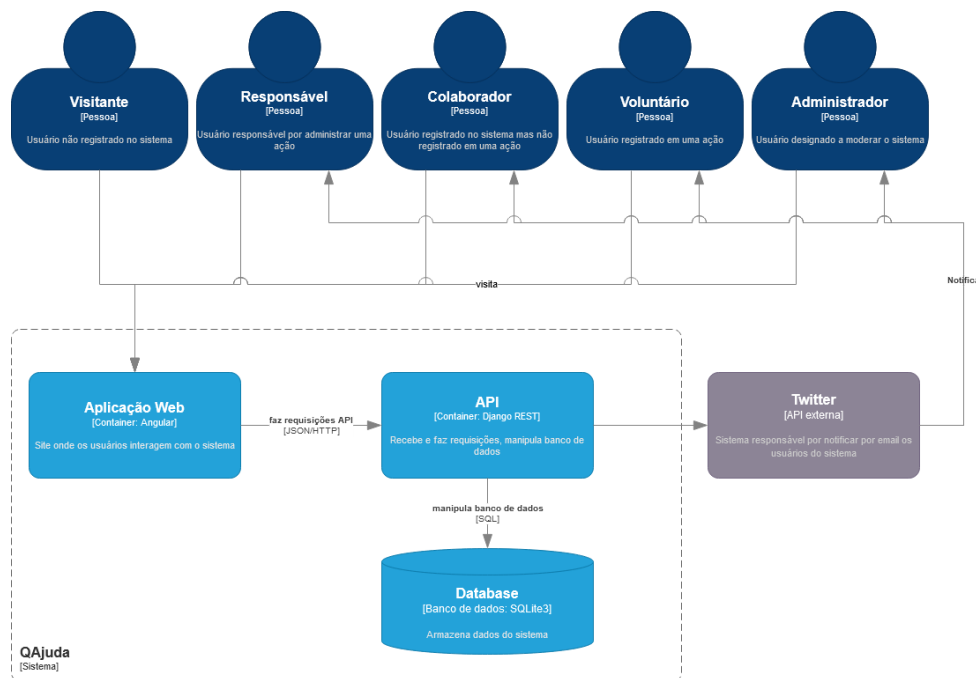


A partir do Diagrama de Contexto observa-se que existem cinco perfis de usuários para o sistema QAjuda (Visitante, Responsável, Colaborador, Voluntário e Administrador), e que o sistema interage com todos atores e com os sistemas

externos, porém o sistema externo, “Twitter” apenas não interage com os Visitantes, usuários não cadastrados.

5.3. Diagrama de Containers

Diagrama de Containers

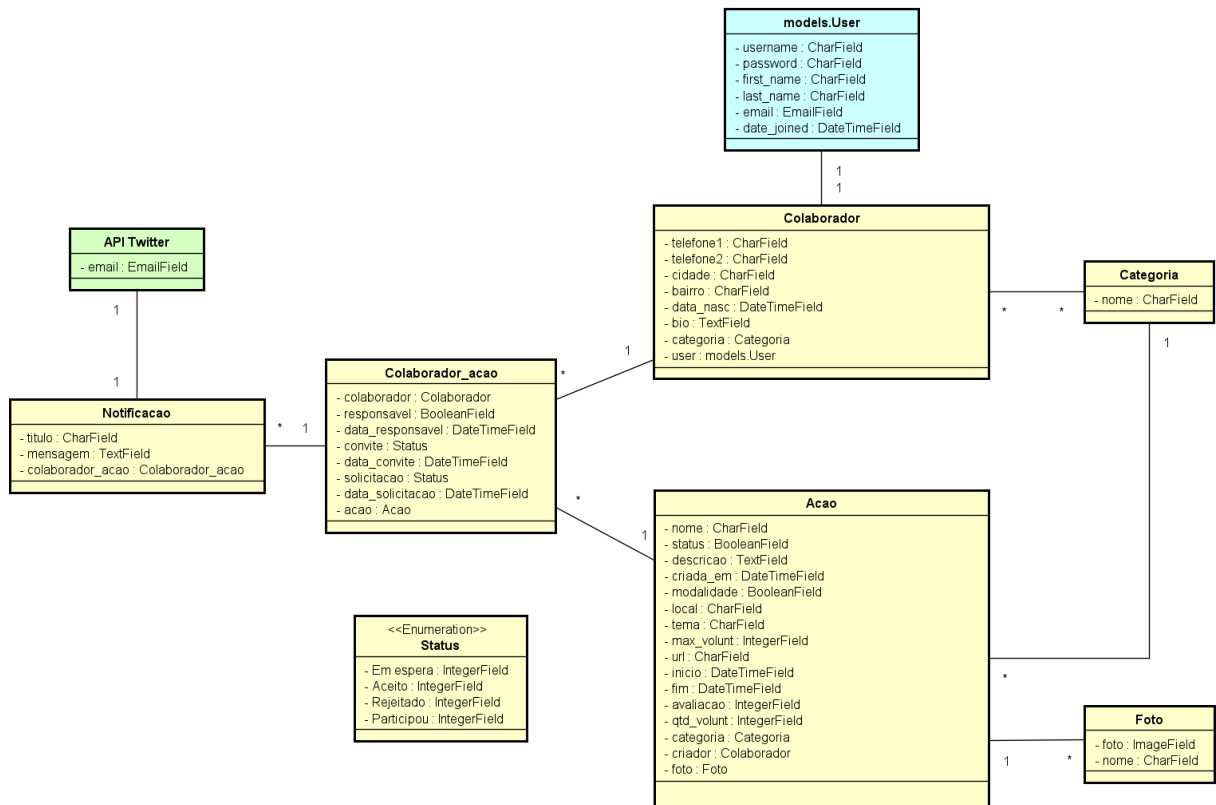


Neste Diagrama de Containers é possível observar que o sistema divide suas responsabilidades pela interface do usuário com o container “Aplicação Web”, que diretamente através de requisições HTTP realiza operações básicas de CRUD com a “API” desenvolvida para tratar pelo backend os dados armazenados e gerenciados pelo container “Database” e para enviar as notificações através da API externa “Twitter”. Todos os perfis de usuários se relacionam com os containers, com exceção do “Visitante”, que por não estar cadastrado no sistema não recebe notificações através da API externa “Twitter”.

6. Diagramas Conceituais

6.1. Visão Lógica

Na sequência, o Diagrama de Classes de Domínio do sistema QAJuda.



As classes de domínio apresentadas nesse Diagrama podem ser compreendidas no Quadro sobre conceito e descrição referentes ao domínio.

Conceito	Descrição
Acao	Contém informações de ações de voluntariado cadastradas
models.User	Model de usuário do Django
Colaborador	Mantém informações de usuários cadastrados na plataforma
Categoria	Ajuda a classificar o tipo de cada ação
Colaborador_ação	Relaciona quando um colaborador se torna voluntario para uma ação
Status	Enumeração de possíveis estados de uma solicitação
Foto	Arquivos de imagens
Notificação	Comunicação do sistema com um voluntário de uma ação

API Twitter	Relacionado ao sistema externo para notificações de voluntários
-------------	---

7. Detalhamento da Implementação e Ambiente Físico

7.1. Visão de Implementação

Backend

Componente	Descrição
Gerenciamento da API	Recebe e faz requisições, manipula banco de dados
Solicitações	Gerência de solicitações
Notificações	Gerência de mensagens
Ação	Gerência de ação
Usuários	Gerência de usuários (responsáveis, colaboradores, voluntários, administradores, visitantes)

Destaca-se que o padrão de projeto Gang of Four (Gof) da categoria “State” foi escolhido e foi implementado no projeto, tendo em vista que o usuário do sistema, no contexto da plataforma “QAjuda”, tem comportamentos/permisões/visões de tela diferentes de acordo com o CDU e, assim, poderá ter seu estado alterado, conforme permissões e comportamentos necessários em cada situação.

Frontend

Componente	Descrição
Services	Gerência de métodos das classe (models) no frontend referentes às views/templates
Notificações	Gerência de mensagens que interage com o backend e indiretamente com a API externa
Usuário	Gerência de usuários (responsáveis, colaboradores, voluntários, administradores, visitantes)
Solicitações	Gerência de solicitações

7.3. Persistência

Para o sistema QAjuda, a escolha de tecnologia para implementar a persistência elegeu o sistema de gerenciamento de banco de dados (SGDB) SQLite3, tendo em vista os requisitos específicos do projeto, as habilidades da equipe de desenvolvimento trabalhadas em consonância à disciplina “Programação e Administração de Banco de Dados”, as metas de desempenho e as regras de negócios.

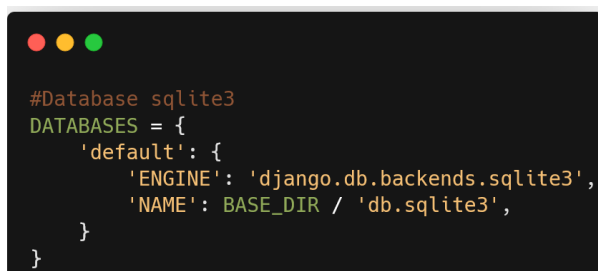
O SQLite3 foi escolhido pois é recomendado em muitos cenários de testes e/ou de pouca carga (o que se adequa ao projeto) com bom desempenho, é um SGDB confiável e fácil de usar, leve, não exige uma configuração complicada ou administração de servidor, não requer um processo de servidor separado, é gratuito e, além de ser compatível com SQL, em caso de necessidade, possibilita

a migração dos dados para outro sistema com facilidade.

Para sua implementação, a tecnologia aplicada do DjangoREST utilizou o pacote “django.db” no arquivo models.py, as configurações “DATABASES” no arquivo settings.py e,

para atualizações, os comandos “makemigrations” e “migrate” do python admin através do “manage.py” registram as modificações no SGDB após alterações em models.py.

Na API RESTful implementada do QAjuda, os módulos implementados que solicitam a persistência utilizam os “serializers” para converter objetos do modelo Django em JSON e vice-versa. O relacionamento entre os dados do banco de dados, a partir dos models.py e serializers.py, podem ser observados nos Diagramas de Entidade-Relação e Esquema Relacional, na sequência.



```
#Database sqlite3
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Diagrama Entidade-Relação

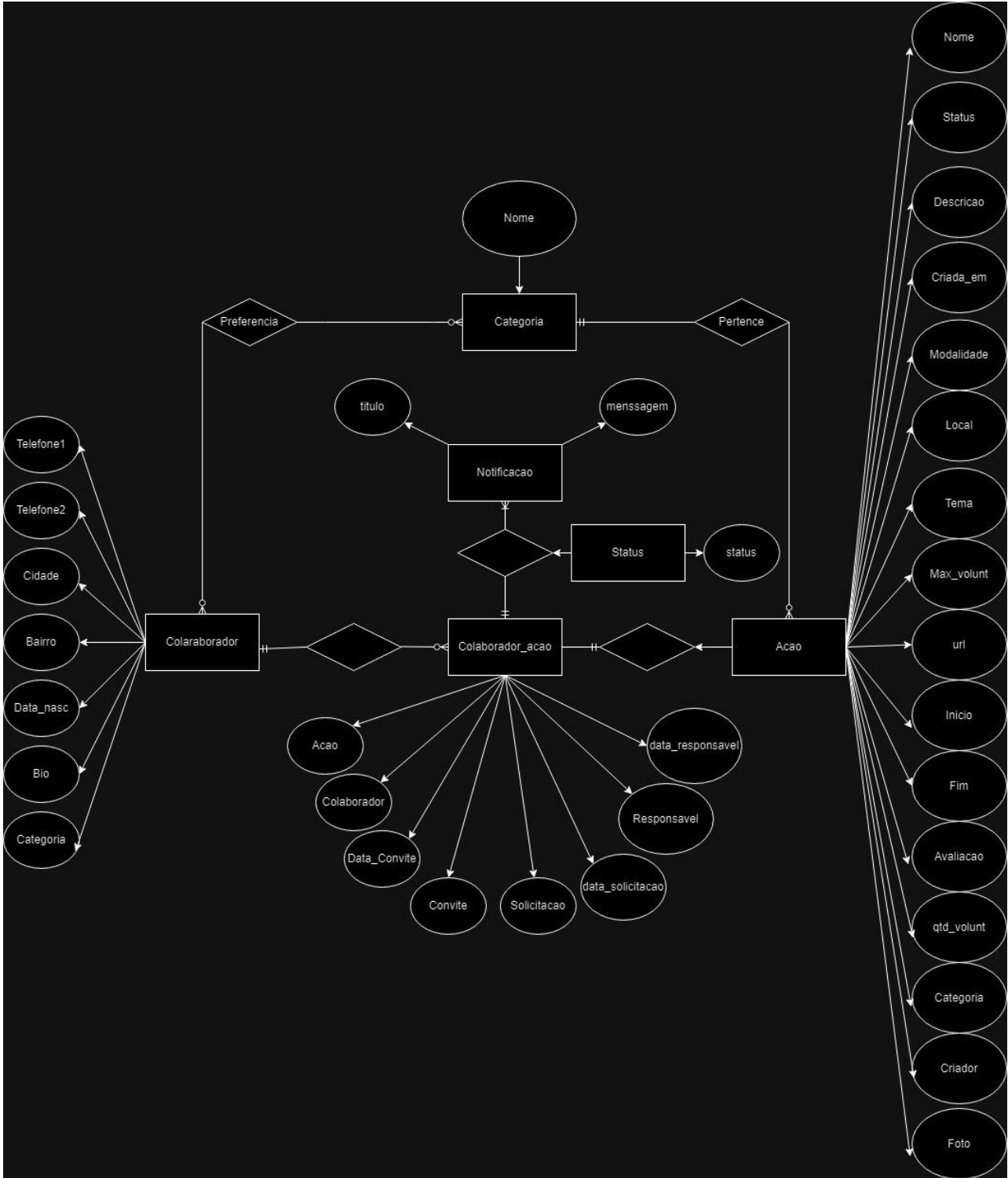
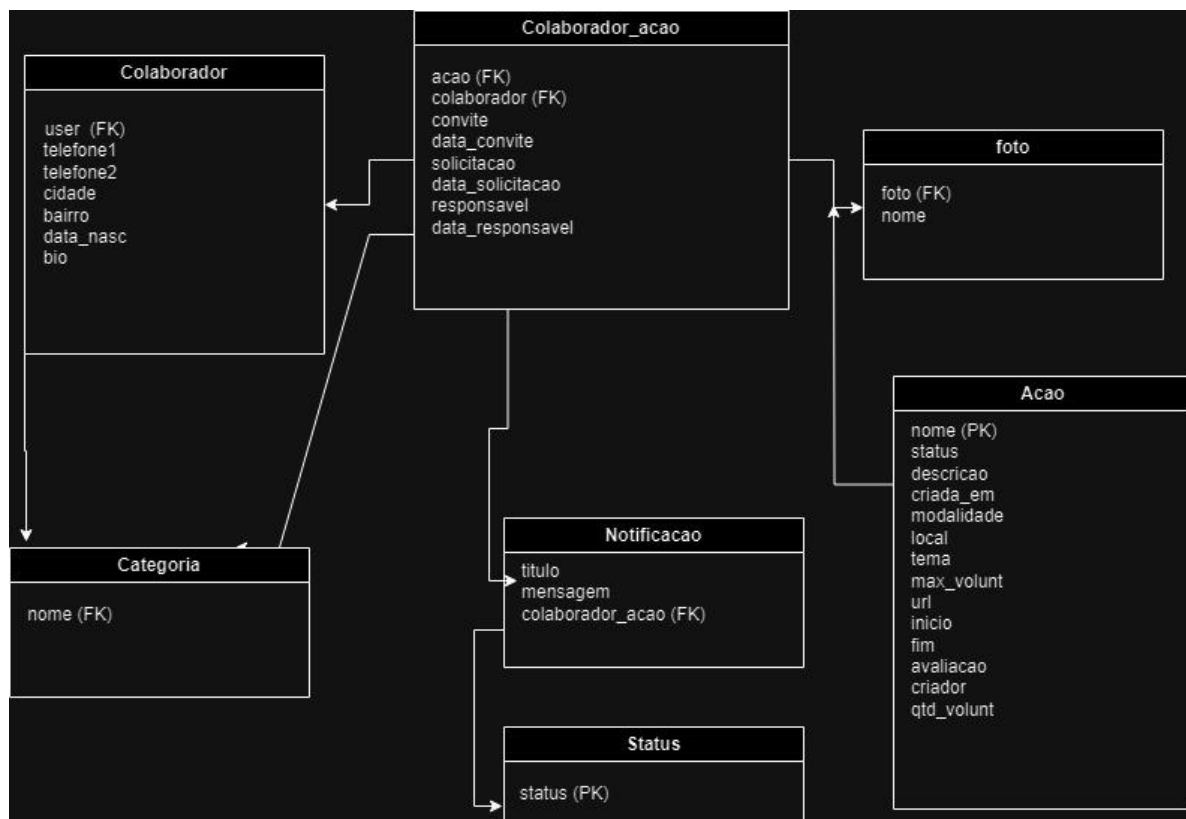


Diagrama Esquema Relacional



7.4. Interface de Usuário

Ao implementar uma interface de usuário para a plataforma web de trabalho voluntário, como é a proposta do QAJuda, alguns aspectos de usabilidade, acessibilidade e facilidade no uso foram priorizadas. O projeto do QAJuda no frontend, desenvolvido a partir do Angular, tem utiliza bootstrap para garantir uma interface mais profissional e design simples. Sobre a identidade visual, a plataforma QAJuda possui uma logo, porém ainda está aperfeiçoando seu padrão visual e ainda não foi decidida uma identidade. Para esta versão do projeto, mantém-se a logo e a tonalidade de azul e optou-se por desenvolver uma interface simples e amigável, com predominância das cores branco e azul, referentes à logo do Projeto.

Nas próximas fases de desenvolvimento, deverão ser aperfeiçoados o design e as ferramentas de acessibilidade, mantendo o uso do framework Angular para que a plataforma possa ser facilmente integrada com backend, além de garantir mais dinamicidade na atualização de informações/tela (modelo SPA).

Por fim, tendo em vista que um dos objetivos da plataforma QAJuda é ser inclusiva, está sendo estudada a possibilidade de integrar ao projeto o uso de

APIs externas que facilitem a implementação desse objetivo, à exemplo de uma API que realiza leitura de texto convertendo para áudio, em português.

8. Anexos

8.1. API

Foi utilizada a aplicação Swagger para descrever os recursos que a API do QAjuda possui (endpoints, dados recebidos, dados retornados, códigos HTTP e métodos de autenticação, entre outros). Abaixo, detalharemos os endpoints e suas finalidades. Para conferir a documentação total da API no Swagger, rodar o servidor localmente e checar no navegador web o endereço <http://127.0.0.1:8000/swagger/> ou clicar no link: <http://127.0.0.1:8000/swagger/?format=openapi>

A documentação do Swagger detalha cada endpoint, assim, neste documento serão apresentados apenas os objetivos dos endpoints referentes a: ação (ações de voluntariado cadastradas no sistema), categoria (tipos de classificação para ações, por exemplo, mutirões, aulas, etc), colaborador (usuários cadastrados no sistema), colaborador_ação (no âmbito de determinada ação, um colaborador que solicitou participação torna-se voluntário) e solicitação (solicitação para ser voluntário em uma ação, que deverá ser aprovada pelos responsáveis da ação).

“Acao”

A seguir serão apresentados os detalhes dos seguintes endpoints:

GET	/acao/	acao_list	▼	🔒
POST	/acao/	acao_create	▼	🔒
GET	/acao/busca/{nome}/	acao_busca_read	▼	🔒
GET	/acao/{id}/	acao_read	▼	🔒
PUT	/acao/{id}/	acao_update	▼	🔒
PATCH	/acao/{id}/	acao_partial_update	▼	🔒
DELETE	/acao/{id}/	acao_delete	▼	🔒

- GET/acao/: o objetivo desse endpoint é listar as ações (método acao_list), sendo uma requisição GET que listará todas as ações de voluntariado cadastradas;
- POST/acao/ : abre um formulário para criar ações (método acao_create), sendo uma requisição POST que enviará todas as informações de uma ação de voluntariado para o sistema;
- GET/acao/busca/{nome}/: realiza a pesquisa (busca) de ações cadastradas através de um parâmetro string “nome” (método acao_busca_read), requisição GET que retornará todas as ações cadastradas no banco de dados que possuem o parâmetro passado;
- GET/acao/{id}/: endpoint que direciona a tela para detalhe de uma ação (método acao_read), requisição GET cujo parâmetro passado é a id de uma ação cadastrada;
- PUT/acao/{id}/: direciona a tela para detalhe de uma ação (método acao_update), requisição PUT cujo parâmetro passado é a id de uma ação cadastrada e permite que seus atributos sejam editados;
- PATCH/acao/{id}/: endpoint que direciona a tela para detalhe de uma ação (método acao_partial_update), requisição PATCH cujo parâmetro passado é a id de uma ação cadastrada e permite a edição parcial do objeto;
- DELETE/acao/{id}/: endpoint que direciona a tela para detalhe de uma ação (método acao_delete), requisição DELETE cujo parâmetro passado é a id de uma ação cadastrada e apaga o objeto selecionado na pesquisa.

“Categoria”

A seguir serão apresentados os detalhes dos seguintes endpoints:

GET	/categoria/	categoria_list	▼	🔒
POST	/categoria/	categoria_create	▼	🔒
GET	/categoria/{id}/	categoria_read	▼	🔒
PUT	/categoria/{id}/	categoria_update	▼	🔒
PATCH	/categoria/{id}/	categoria_partial_update	▼	🔒
DELETE	/categoria/{id}/	categoria_delete	▼	🔒

- GET/categoria/: o objetivo desse endpoint é listar as categorias (método categoria_list), sendo uma requisição GET que listará todas as categorias de voluntariado cadastradas;
- POST/categoria/: o objetivo desse endpoint é listar as ações (método categoria_create), sendo uma requisição POST que irá criar categorias para ações no sistema;
- GET/categoria/{id}/: endpoint que direciona a tela para detalhe de uma categoria (método categoria_read), requisição GET cujo parâmetro passado é a id de uma categoria cadastrada;
- PUT/categoria/{id}/: direciona a tela para detalhe de uma ação (método categoria_update), requisição PUT cujo parâmetro passado é a id de uma categoria cadastrada e permite que seja editada;
- PATCH/categoria/{id}/: endpoint que direciona a tela para detalhe de uma ação (método categoria_partial_update), requisição PATCH cujo parâmetro passado é a id de uma categoria cadastrada e permite a edição parcial do objeto;
- DELETE/categoria/{id}/: endpoint que direciona a tela para detalhe de uma ação (método categoria_delete), requisição DELETE cujo parâmetro passado é a id de uma categoria cadastrada e apaga o objeto selecionado na pesquisa.

“Colaborador”

A seguir serão apresentados os detalhes dos seguintes endpoints:

GET	/colaborador/	colaborador_list	▼	🔒
POST	/colaborador/	colaborador_create	▼	🔒
GET	/colaborador/{id}/	colaborador_read	▼	🔒
PUT	/colaborador/{id}/	colaborador_update	▼	🔒
PATCH	/colaborador/{id}/	colaborador_partial_update	▼	🔒
DELETE	/colaborador/{id}/	colaborador_delete	▼	🔒

- GET/colaborador/: o objetivo desse endpoint é listar os colaboradores (método colaborador_list), sendo uma requisição GET que listará todos os colaboradores cadastrados;
- POST/colaborador/: o objetivo desse endpoint é criar usuários (método colaborador_create) no sistema, sendo uma requisição POST que irá criar formulário do Django novos usuários cadastrados;
- GET/colaborador/{id}/: endpoint que direciona a tela para detalhe de um colaborador (método colaborador_read), requisição GET cujo parâmetro passado é a id de um colaborador cadastrado;
- PUT/colaborador/{id}/: direciona a tela para detalhe de uma ação (método colaborador_update), requisição PUT cujo parâmetro passado é a id de um colaborador e permite que seja editado/atualizado;
- PATCH/colaborador/{id}/: endpoint que direciona a tela para detalhe de um perfil de colaborador (método colaborador_partial_update), requisição PATCH cujo parâmetro passado é a id de um colaborador cadastrado na plataforma e permite a edição parcial do objeto;
- DELETE/colaborador/{id}/: endpoint que permite que um colaborador seja deletado do sistema (método colaborador_delete), requisição DELETE cujo parâmetro passado é a id de um colaborador.

“Colaborador_acao”

A seguir serão apresentados os detalhes dos seguintes endpoints:

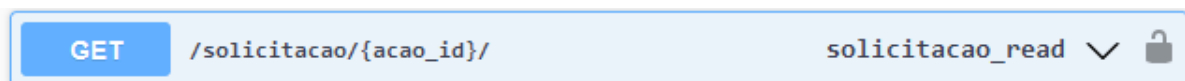
GET	/colaborador_acao/	colaborador_acao_list	▼	🔒
POST	/colaborador_acao/	colaborador_acao_create	▼	🔒
GET	/colaborador_acao/{id}/	colaborador_acao_read	▼	🔒
PUT	/colaborador_acao/{id}/	colaborador_acao_update	▼	🔒
PATCH	/colaborador_acao/{id}/	colaborador_acao_partial_update	▼	🔒
DELETE	/colaborador_acao/{id}/	colaborador_acao_delete	▼	🔒

- GET/colaborador_acao/: o objetivo desse endpoint é listar os colaboradores (método colaborador_acao_list), sendo uma requisição GET que listará todos os colaboradores cadastrados;

- POST/colaborador_acao/: o objetivo desse endpoint é criar usuários (método colaborador_acao_create) no sistema, sendo uma requisição POST que irá criar formulário do Django novos usuários cadastrados;
- GET/colaborador_acao/{id}/: endpoint que direciona a tela para detalhe de um colaborador (método colaborador_acao_read), requisição GET cujo parâmetro passado é a id de um colaborador cadastrado;
- PUT/colaborador_acao/{id}/: direciona a tela para detalhe de uma ação (método colaborador_acao_update), requisição PUT cujo parâmetro passado é a id de um colaborador e permite que seja editado/atualizado;
- PATCH/colaborador_acao/{id}/: endpoint que direciona a tela para detalhe de um perfil de colaborador (método colaborador_acao_partial_update), requisição PATCH cujo parâmetro passado é a id de um colaborador cadastrado na plataforma e permite a edição parcial do objeto;
- DELETE/colaborador_acao/{id}/: endpoint que permite que um colaborador seja deletado (método colaborador_acao_delete), requisição DELETE cujo parâmetro passado é a id de um colaborador voluntário em determinada ação.

“Solicitacao”

A seguir serão apresentados os detalhes do seguinte endpoint:



- GET/solicitacao/{acao_id}/: o objetivo desse endpoint é listar as solicitações referentes ao id de determinada ação (método solicitacao_read), sendo uma requisição GET que listará todas as solicitações de uma ação.

Referências

- [1] <https://www.django-rest-framework.org/>
- [2] <https://angular.io/>
- [3] <https://developer.mozilla.org/en-US/docs/Glossary/SPA?ref=jerrynsh.com>
- [4] <https://c4model.com/>