UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

LUIZ HENRIQUE ALMEIDA DE ARAUJO

PLATAFORMA *WEB* PARA O GERENCIAMENTO DE TRABALHOS DE CONCLUSÃO DE CURSO

Recife
2025

LUIZ HENRIQUE ALMEIDA DE ARAUJO

PLATAFORMA *WEB* PARA O GERENCIAMENTO DE TRABALHOS DE CONCLUSÃO DE CURSO

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Orientador(a): Prof. Dr. Márcio Rodrigo Santos de Carvalho

Recife
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Araujo, Luiz Henrique Almeida de.
Plataforma web para o gerenciamento de trabalhos de conclusão de curso / Luiz Henrique Almeida de Araujo. - Recife, 2025.
95 p. : il., tab.

Orientador(a): Márcio Rodrigo Santos de Carvalho
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Tecnologia e Geociências, Engenharia Elétrica - Bacharelado, 2025.
Inclui referências.

1. Automação de procedimentos. 2. Plataforma web. 3. Engenharia de software. I. Carvalho, Márcio Rodrigo Santos de. (Orientação). II. Título.

620 CDD (22.ed.)

LUIZ HENRIQUE ALMEIDA DE ARAUJO

PLATAFORMA *WEB* PARA O GERENCIAMENTO DE TRABALHOS DE CONCLUSÃO DE CURSO

Trabalho de Conclusão de Curso apresentado ao Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Bacharel em Engenharia Elétrica.

Aprovado em: 15/12/2025

BANCA EXAMINADORA

_____
Prof. Dr. Márcio Rodrigo Santos de Carvalho (Orientador)
Universidade Federal de Pernambuco

_____
Prof. Dr. Artur Muniz Szpak Furtado (Examinador Interno)
Universidade Federal de Pernambuco

_____
Prof. Msc. Ericles Mauricio Barbosa (Examinador Interno)
Universidade Federal de Pernambuco

RESUMO

O Trabalho de Conclusão de Curso é uma etapa fundamental na formação do aluno, envolvendo requisitos formais, múltiplos participantes, prazos definidos e procedimentos de avaliação sequenciais. Em diversas instituições de ensino superior, a execução dos processos que incluem suas etapas é efetuada manualmente e de forma descentralizada, por meio de ferramentas dispersas em diversos canais, o que compromete a eficiência administrativa e gera dificuldade no monitoramento dos processos. Com o objetivo de superar essas limitações, este trabalho descreve a criação de uma plataforma *web* voltada para a gestão integrada dos Trabalhos de Conclusão de Curso do Departamento de Engenharia Elétrica da Universidade Federal de Pernambuco, projetada para centralizar informações, realizar os procedimentos derivados de cada etapa, padronizar e acompanhar os processos. A solução foi elaborada com base em uma arquitetura cliente-servidor, de modo que o *frontend* segue os paradigmas da *Single Page Application* desenvolvida em React e TypeScript. O *backend* é implementado por meio do Django e Django REST Framework, interconectados por uma API REST com autenticação via JSON *Web Token*. O sistema abrange ainda o controle de acesso baseado em papéis, uma máquina de estados para representação do ciclo de vida do Trabalho de Conclusão de Curso, versionamento de documentos, registro de avaliações, gestão de bancas e um calendário acadêmico configurável. Os resultados evidenciam que a plataforma diminui a dependência de controles manuais, aprimora a rastreabilidade do fluxo, padroniza procedimentos e facilita a administração por parte da coordenação, assegurando maior clareza para alunos e orientadores em relação às etapas, prazos e documentos requeridos.

Palavras-chave: Automação de procedimentos; Plataforma *web*; Engenharia de *software*.

ABSTRACT

This work presents the development of a web platform designed to support the integrated management of the Final Graduation Project (Trabalho de Conclusão de Curso – TCC) of the Department of Electrical Engineering at the Federal University of Pernambuco. In many higher education institutions, TCC-related procedures are still carried out manually and in a decentralized manner, using disparate tools distributed across multiple channels. This fragmentation compromises administrative efficiency and hinders consistent monitoring of the process. This work describes the creation of a web platform aimed at the integrated management of the Final Course Projects of the Electrical Engineering Department at UFPE, designed to centralize information, perform the procedures derived from each stage, standardize and monitor the processes. The solution was developed based on a client-server architecture, so that the frontend follows the paradigms of the Single Page Application developed in React and TypeScript. The backend is implemented through Django and Django REST Framework, interconnected by a REST API with authentication via JSON Web Tokens. The system includes role-based access control, a state machine for representing the thesis lifecycle, document versioning, evaluation registration, panel management, and a configurable academic calendar. The results indicate that the system reduces dependency on manual controls, improves process traceability, standardizes procedures, and facilitates administrative tasks performed by the coordination team, providing greater clarity to students and supervisors regarding required stages, deadlines and documentation.

Keywords: Procedure Automation; Web platform; Software engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
CRUD	*Create, Read, Update, Delete*
CSRF	*Cross-Site Request Forgery*
CORS	*Cross-Origin Resource Sharing*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
DRF	Django REST Framework
FSM	*Finite State Machine*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
JSON	JavaScript *Object Notation*
JWT	JSON *Web Token*
MVC	*Model-View-Controller*
MTV	*Model-Template-View*
ORM	*Object-Relational Mapping*
OWASP	*Open Web Application Security Project*
REST	*Representational State Transfer*
RFC	*Request for Comments*
SMTP	*Simple Mail Transfer Protocol*
SPA	*Single Page Application*
SQL	*Structured Query Language*
TCC	Trabalho de Conclusão de Curso
URL	*Uniform Resource Locator*
URI	*Uniform Resource Identifier*
ACID	*Atomicity, Consistency, Isolation, Durability*
HTML	*HyperText Markup Language*
DB	*Database*

SUMÁRIO

1 INTRODUÇÃO

O Trabalho de Conclusão de Curso (TCC) constitui uma etapa fundamental na graduação, de forma a caracterizar o momento em que o estudante integra os conhecimentos teóricos e práticos assimilados durante a formação acadêmica (11). O TCC representa não apenas um requisito formal, mas também uma ocasião propícia para o aprimoramento do senso crítico, a aplicação de habilidades de pesquisa e o desenvolvimento da autonomia intelectual (16). Dessa forma, a estruturação e o monitoramento dos Trabalhos de Conclusão de Curso alcançam significativa relevância para as instituições de ensino superior e devem ser administrados de maneira eficiente, dado que implicam prazos severos, diversos participantes e procedimentos formais de avaliação. A execução dessa etapa depende de um processo administrativo organizado, composto por etapas sequenciais, prazos definidos e avaliação formal de documentos e procedimentos (3). Cada participante assume responsabilidades específicas, de modo que estudantes realizam entregas e cumprem prazos, orientadores validam o progresso, avaliadores analisam o trabalho apresentado e a coordenação garante que o fluxo siga o regulamento estabelecido.

Em diversas instituições de ensino superior, a gestão dos TCCs é predominantemente realizada de maneira manual, com o auxílio de formulários digitais, planilhas eletrônicas e plataformas como o Google *Classroom.* Embora úteis de forma isolada, essas ferramentas não foram projetadas para controlar todas as etapas do TCC. Essa fragmentação do processo torna-o vulnerável a falhas operacionais, pois as informações ficam dispersas em diversos canais de acesso, gerando uma dificuldade de consolidação da visão do fluxo de trabalho e tornando todos os processos dependentes de controles manuais executados pelo coordenador do curso (12). Essa dinâmica pode ser agravada pelo fato de que um único coordenador pode ter sob sua responsabilidade uma elevada quantidade de fluxos de acompanhamento simultâneos, com cada uma delas encontrando-se em estágios distintos, acompanhadas de documentação e prazos específicos.

A ausência de um sistema integrado para a gestão dos Trabalhos de Conclusão de Curso provoca dificuldades práticas que comprometem tanto a eficiência administrativa quanto a qualidade do processo acadêmico (3). Além disso, a falta de

padronização pode levar a tratamentos distintos para situações semelhantes, visto que as interações ficam distribuídas em diferentes meios de comunicação.

Diante desse cenário, a implementação de um sistema digital integrado configura-se como uma alternativa viável para resolver essas limitações (12). Um portal desenvolvido para gestão dos fluxos de TCC pode centralizar informações, organizar documentos, registrar histórico de ações, automatizar notificações e oferecer visão clara sobre a situação de cada trabalho. Dessa forma, é possível diminuir a dependência de processos manuais e descentralizados, aumentando a eficiência da gestão dos fluxos.

Além de melhorar a organização, a utilização de um sistema integrado reduz retrabalho, evita perda de prazos e facilita a tomada de decisões (15). Desse modo, estudantes passam a saber exatamente o que deve ser feito em cada etapa de forma mais clara, orientadores acompanham seus orientandos de modo mais estruturado e a coordenação obtém visão consolidada do semestre, eliminando grande parte das consultas repetitivas realizadas em diferentes canais.

A solução proposta neste trabalho busca equilibrar as necessidades dos diferentes participantes envolvidos, de modo a implementar um fluxo de trabalho que seja ao mesmo tempo robusto, flexível e intuitivo, contribuindo assim para a melhoria contínua da execução de procedimentos institucionais.

1.1 Objetivos

1.1.1 Geral

Desenvolver uma plataforma *web* destinada à gestão dos fluxos de Trabalhos de Conclusão de Curso, de modo a contemplar desde solicitação de orientação até a conclusão final do trabalho.

1.1.2 Específicos

- Estabelecer um sistema de controle de acesso fundamentado em papéis, a fim de garantir a distinção de permissões e funcionalidades entre estudantes, orientadores, coordenadores e avaliadores externos;
- Criar uma máquina de estados que possa representar formalmente o ciclo de vida do Trabalho de Conclusão de Curso (TCC), assegurando que as transições aconteçam estritamente de acordo com as normas institucionais e com registros de rastreabilidade;
- Estabelecer um sistema de notificações internas na interface e comunicação via e-mail;
- Instituir sistema de versionamento de documentos que preserve o histórico de pareceres, assegurando integridade e auditoria do processo;
- Desenvolver um módulo para a formação de bancas e a realização de avaliações, abordando o cálculo automático das pontuações e a consolidação das notas finais;
- Elaborar um sistema adaptável para a administração de prazos, baseado em um calendário acadêmico personalizável;
- Implementar medidas de segurança e privacidade em múltiplas camadas, utilizando autenticação por *tokens*, permissões granulares e proteções em conformidade com às recomendações da OWASP.

1.2 Organização do Trabalho

Este trabalho está estruturado em quatro capítulos principais:

- Fundamentação teórica: aborda os conceitos utilizados no desenvolvimento, abrangendo arquitetura para aplicações *web*, API *RESTful*, autenticação *stateless* e segurança (JWT, RBAC, OWASP), além de princípios de engenharia de *software* aplicados ao domínio da solução.
- Desenvolvimento: descreve a solução proposta, contemplando a visão geral da arquitetura e os módulos *backend*, *frontend* e banco de dados implementados, além dos aspectos de segurança e principais linhas de fluxos do sistema
- Resultados: apresenta o modo de funcionamento a partir das linhas de fluxo.
- Conclusões e propostas de continuidade: sintetiza as contribuições do trabalho e direções de evolução institucional.

2 FUNDAMENTAÇÃO TEÓRICA

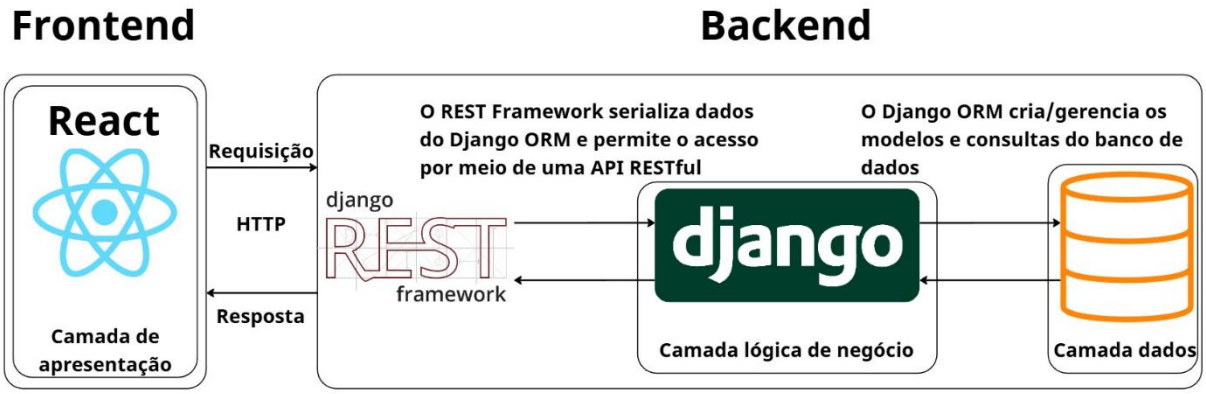
2.1 Arquiteturas de *software* para aplicações *web*

2.1.1 Arquitetura em camadas

A arquitetura de três camadas constitui um padrão amplamente utilizado em aplicações empresariais devido à implementação da divisão de responsabilidades do sistema em módulos autônomos, como exemplificado na Figura 1. Esse modelo favorece a manutenção, realização de testes e a evolução do sistema durante seu ciclo de vida (15), organizando o *software* em três componentes fundamentais:

- A camada de apresentação é responsável pela interação com o usuário. Tem o foco exclusivo nos elementos referentes à interface, evitando a inclusão de lógica de negócio ou acesso direto à base de dados (17). Essa segmentação permite a execução de modificações visuais sem impacto no domínio da aplicação, possibilitando manter a base lógica na elaboração de interfaces para diferentes tipos de dispositivos.
- A camada de lógica de negócio gerencia a implementação das regras, validações e o processamento dos dados. Ela organiza as normas e os procedimentos que definem o domínio da aplicação de forma a assegurar a uniformidade na implementação das diretrizes, independentemente da interface empregada (7).
- A camada de persistência (ou camada de dados) é encarregada pelo armazenamento e recuperação de informações. Ela mantém a independência em relação a sistemas gerenciadores de banco de dados e assegura a portabilidade entre diferentes tecnologias por meio da abstração de informações (15).

Figura 1 – Arquitetura em três camadas de uma aplicação *web*.



Fonte: autor, 2025.

2.1.2 Arquitetura REST *(Representational State Transfer)*

O modelo REST é um estilo arquitetural proposto por Roy Thomas Fielding em sua tese de doutorado em 2000 (5), com o objetivo de orientar o desenvolvimento de sistemas distribuídos baseados em serviços *web* escaláveis e desacoplados. Nesse sistema, a identificação dos recursos ocorre por meio de *URIs* únicas e a manipulação é realizada por operações padronizadas sobre o protocolo HTTP, que é a base para a comunicação entre cliente e servidor, dinâmica ilustrada na Figura 2. Seis restrições fundamentais são estabelecidas pelo autor:

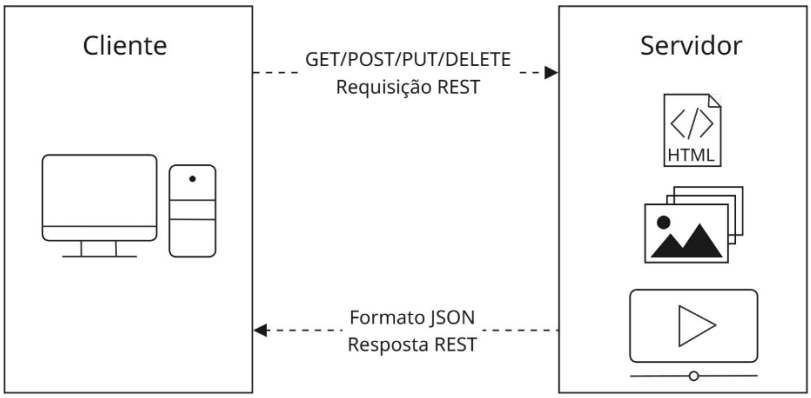
- Cliente-servidor;
- Comunicação *stateless*;
- Uso de *cache*;
- Interface uniforme;
- Arquitetura em camadas;
- Código sob demanda.

A aplicação dessas restrições tem como consequência a geração de sistemas que apresentam simplicidade, elevado desempenho, maior escalabilidade e confiabilidade operacional.

O princípio de comunicação *stateless* é fundamental na arquitetura, pois dispensa a necessidade de que o servidor mantenha o estado atual da sessão entre

requisições consecutivas por intermédio da determinação de que cada requisição contenha todas informações necessárias para seu processamento (5). Como consequência, o processamento torna-se independente de contexto e as requisições podem ser distribuídas para qualquer instância disponível, de forma a favorecer balanceamento de carga e escalabilidade. Além disso, respostas REST podem ser cacheadas quando acompanhadas de metadados apropriados, reduzindo a quantidade de requisições e a latência percebida. O princípio da restrição de interface uniforme resulta na simplificação da arquitetura, independentemente da tecnologia utilizada no cliente ou no servidor à medida que padroniza a dinâmica de interação dos componentes (5).

Figura 2 – Estrutura cliente-servidor adotada em aplicações *web*.



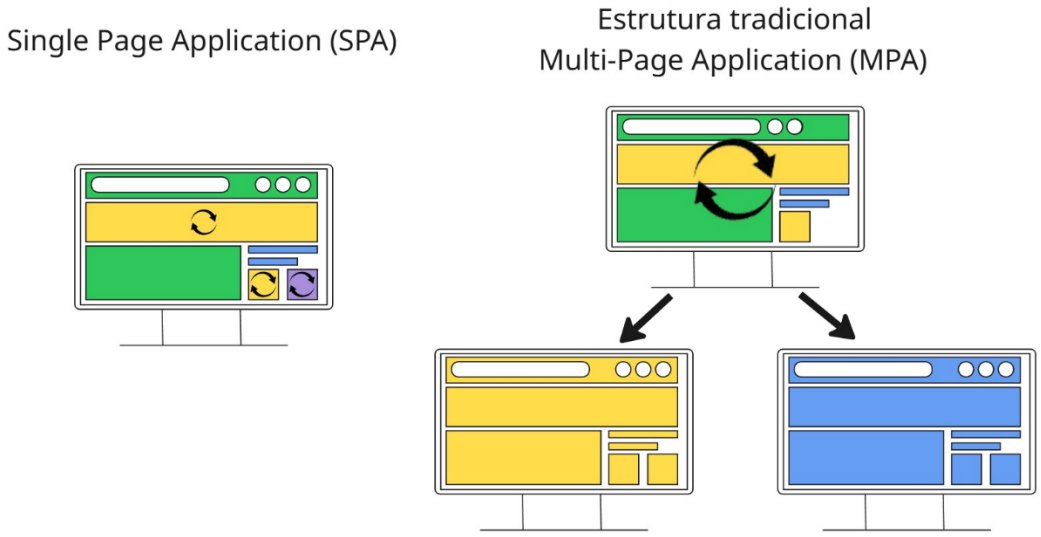
Fonte: autor, 2025.

*2.1.3 Single Page Applications (*SPA*)*

As Aplicações de Página Única (*Single Page Applications* – SPA) constituem um modelo de desenvolvimento *web* baseado no carregamento em um único documento HTML da totalidade da aplicação, o qual se mantém inalterado durante toda a interação com o usuário. São baseadas na manipulação do *Document Object Model* (DOM) e realização de requisições assíncronas ao servidor, de forma a possibilitar que a atualização dos elementos de interface ocorra dinamicamente (15), como demonstrado na Figura 3. Essa dinâmica possibilita uma experiência mais fluida e responsiva, semelhantes a aplicações *desktop*, pois as transições entre telas são renderizadas diretamente no navegador (1).

A divisão clara entre o *frontend* e o *backend* favorece a evolução independente dos componentes e permite que diferentes *clientes* consumam a mesma API de forma padronizada, de modo que o cliente é responsável pela apresentação, reservando ao servidor apenas a disponibilização da API *RESTful*, sem necessidade da lógica de interface (17).

Figura 3 – Funcionamento de uma *Single Page Application* (SPA).



Fonte: autor, 2025.

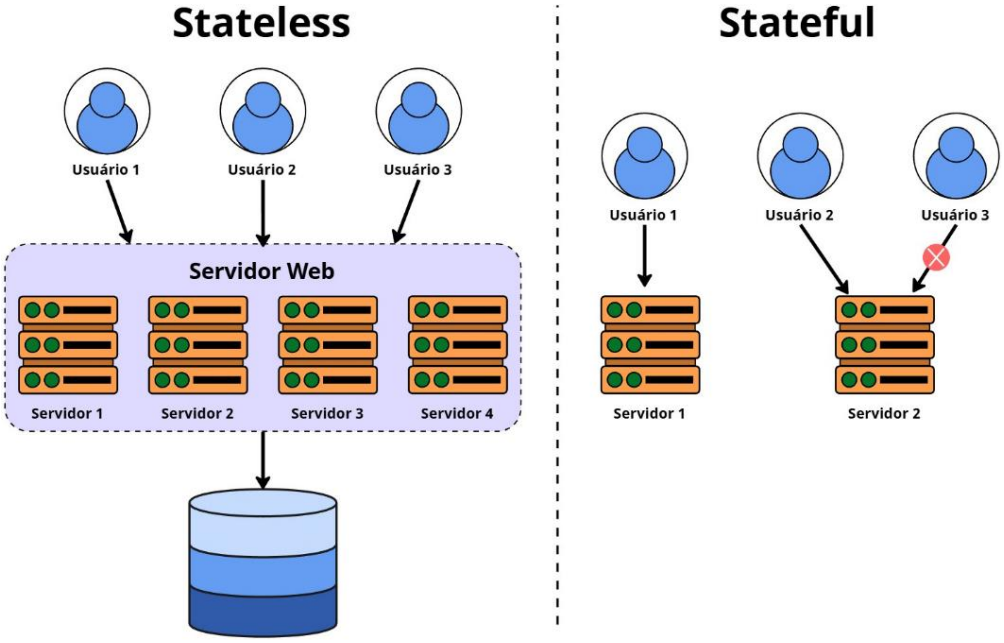
2.1.4 Comunicação Stateless e Autenticação

Em arquiteturas REST, a autenticação necessita de execução sem dependência de sessão no servidor, visto que o modelo *stateless* não preserva contexto entre requisições (5), diferente da comunicação *stateful* que permite sessões persistentes e continuidade entre ações do usuário, porém dificulta a escalabilidade horizontal, possui maior custo de infraestrutura e aumenta o acoplamento servidor-cliente. Diante desse cenário, uma abordagem amplamente adotada consiste na utilização de *tokens* autossuficientes que realizam a incorporação de informações sobre a autenticação e autorização do usuário, dessa forma os JSON *Web Tokens* (JWT) tornaram-se o modelo predominante. A comparação entre as dinâmicas está ilustrada na Figura 4.

O JWT é formado por três partes, cabeçalho (indica o algoritmo de assinatura), *Payload* (contém as informações do usuário, como identificação e data de expiração) e assinatura criptográfica (gerada com uma chave secreta), todas codificadas em

base64 e funcionamento é baseado na verificação local da assinatura, com o servidor realizando a validação do *token* sem a necessidade de acesso ao banco de dados, desse modo contribui-se para redução da latência e maior escalabilidade do sistema, mesmo sob alto volume de requisições (10).

Figura 4 – Processamento de requisições em arquiteturas *Stateless* e *Stateful*.



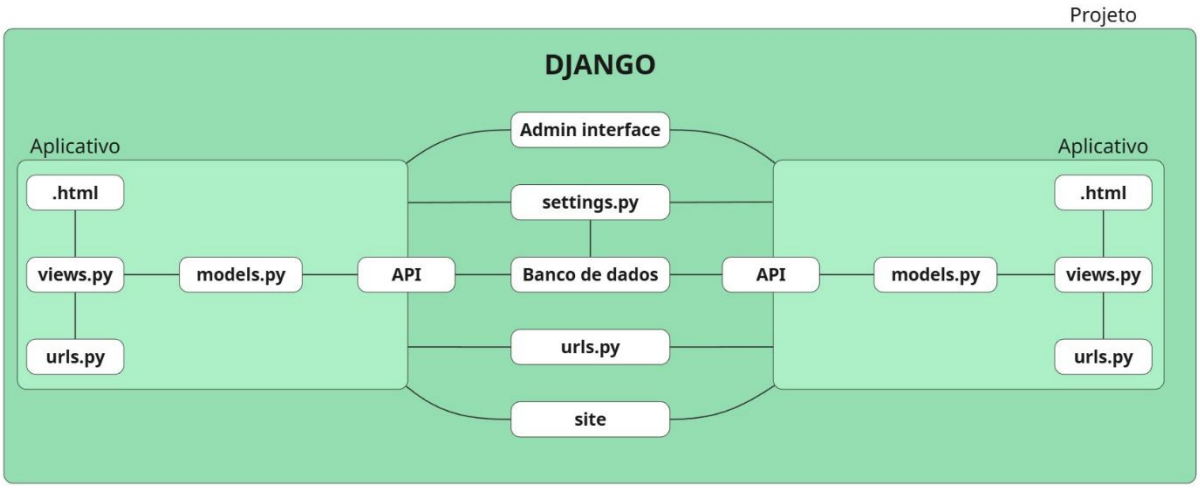
Fonte: autor, 2025.

2.2 *Framework* Django

2.2.1 Framework Django e *padrão* MTV *(Model-Template-View)*

Django é um *framework web* desenvolvido em linguagem Python e oferece um conjunto de funcionalidades nativas, como ORM, sistema de autenticação, painel administrativo e mecanismos de segurança contra as vulnerabilidades mais conhecidas. Ele estabelece convenções para a estrutura de projeto, administra o código e fluxo de requisições, além disso também reduz a quantidade de decisões arquiteturais e promove consistência entre aplicações desenvolvidas com Django, o que favorece a manutenção a longo prazo (1).

Figura 5 – Estrutura do padrão MTV no *framework* Django.



Fonte: autor, 2025.

A sua estrutura arquitetônica adota o padrão MTV, que é uma versão adaptada do modelo MVC para o ambiente *web*. Nesse modelo, representado na Figura 5, o *Model* elucida a organização dos dados e a lógica de negócio por meio de classes em Python que são mapeadas para tabelas no banco de dados. O *Template* especifica a apresentação das informações mediante uma linguagem de marcação com capacidade para conteúdo dinâmico. A *View* desempenha a função de componente de controle, além de processar solicitações HTTP e coordenando a interação entre *Models* e *Templates* (1). O *framework* também oferece *views* genéricas fundamentadas em classes, que sistematizam operações frequentes, como a listagem, criação e modificação de registros, essa dinâmica contribui para diminuição da quantidade de código redundante exigida para a implementação das funcionalidades de CRUD.

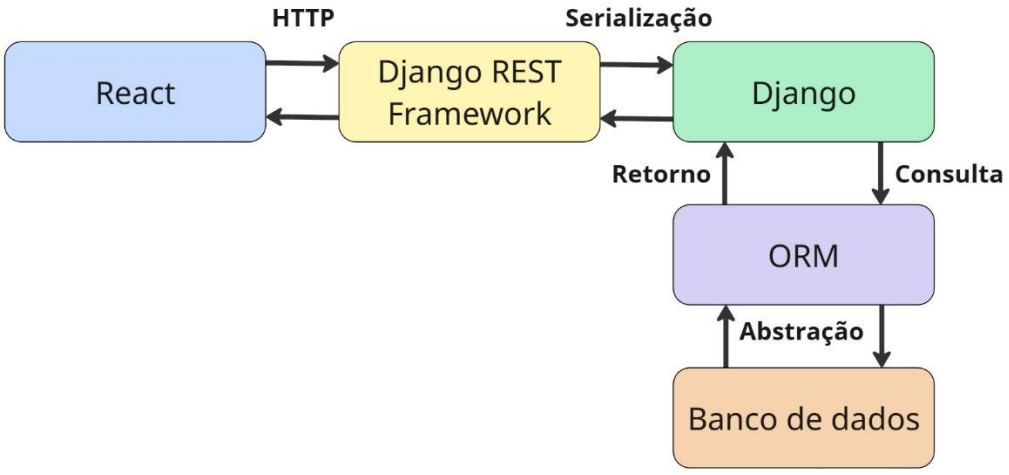
2.2.2 Django REST Framework

Desenvolvido por Christie em 2011, o Django REST Framework amplia as funcionalidades do Django, promovendo a criação de *APIs web* que aderem aos princípios REST. Este *framework* oferece componentes especializados que tornam mais simples a serialização de dados, a validação de entradas, a autenticação e autorização, a paginação de resultados e a documentação automática dos *endpoints*.

Como resultado, ele se firmou como o padrão estabelecido para o desenvolvimento de *APIs* dentro do ecossistema Django, graças à sua abordagem equilibrada entre produtividade e flexibilidade.

Serializadores são elementos fundamentais que realizam a tradução bidirecional entre representações complexas de dados, englobando instâncias de modelos Django e tipos de dados nativos da linguagem Python, os quais podem ser facilmente convertidos para o formato JSON (2). Os *ViewSets* constituem uma abstração que integra a lógica de diversas *views* relacionadas em uma única classe, geralmente implementando operações completas de CRUD por meio de métodos que se correspondem a ações padrão do REST, como listar, recuperar, criar, atualizar e destruir. Simultaneamente, os roteadores criam automaticamente a configuração de *URLs*, dispensando a necessidade de definir manualmente cada *endpoint* e assegurando que as *URLs* estejam em conformidade com convenções REST consistentes (2).

Figura 6 – Integração entre React, Django REST Framework e Django ORM.



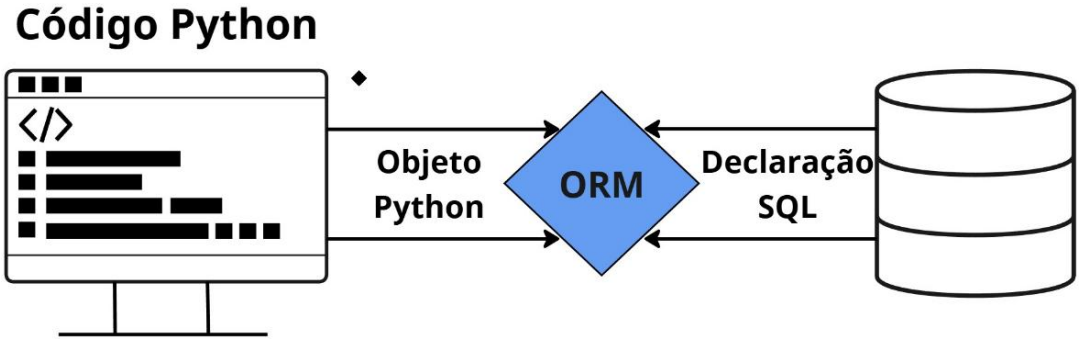
Fonte: autor, 2025.

*2.2.3 Mapeamento Objeto-Relacional (*ORM*)*

O mapeamento objeto-relacional (ORM) cria uma equivalência entre as classes da aplicação e as estruturas do banco de dados, de modo a possibilitar a execução de operações de persistência através do paradigma orientado a objetos. Essa dinâmica permite dispensar a necessidade de elaboração de consultas SQL

manualmente. Essa abstração, representada na Figura 7, permite o desenvolvimento de um código mais simples e favorece a portabilidade, de forma a evitar a dependência de dialetos específicos de SQL (7). Um benefício de utilização é a segurança, pois as consultas parametrizadas reduzem as vulnerabilidades associadas à injeção de comandos.

Figura 7 – Conversão de objetos Python em declarações SQL por meio do ORM.



Fonte: autor, 2025.

No Django, o ORM representa as entidades do domínio por meio de classes Python que são derivadas da classe *Model*, cujos atributos especificam os campos da tabela, incluindo os tipos de dados, as restrições e os relacionamentos. A estrutura é elaborada de forma singular no código e empregada tanto para a geração automatizada do esquema, por meio das migrações, quanto para a validação durante a execução. A interface de programação de aplicações (API) para consultas possibilita a criação de operações sofisticadas por meio do encadeamento de métodos, incluindo filtros, ordenações e agregações. A segurança contra injeções SQL é implementada automaticamente por meio da utilização de parâmetros, assegurando que os dados inseridos pelos usuários sejam processados de maneira distinta da estrutura da consulta (1). As tabelas 1 e 2 apresentam exemplos de mapeamentos e operações ORM nas duas formas de representação.

Tabela 1 – Exemplo de mapeamento ORM.

PYTHON (Orientado a Objetos)	SQL (Relacional)
class TCC(*models.Model*)	*CREATE TABLE* tccs
titulo = CharField	titulo VARCHAR(300)
aluno = ForeignKey	aluno_id INTEGER FK
nota = DecimalField	nota DECIMAL(9,7)

Fonte: autor, 2025.

Tabela 2 – Exemplo de operações.

PYTHON (Orientado a Objetos)	SQL (Relacional)
TCC.*objects.filter*(...)	*SELECT * FROM* tccs *WHERE*...
tcc.*save*()	*INSERT INTO* tccs / *UPDATE*
tcc.*delete*()	*DELETE FROM* tccs *WHERE*...
TCC.*objects.create*(...)	*INSERT INTO* tccs *WHERE*...

Fonte: autor, 2025.

2.2.4 Sistema de sinais e padrão observer

O Padrão *Observer* estabelece uma relação do tipo um-para-muitos entre os objetos, na qual qualquer modificação de estado no objeto que está sendo observado notifica automaticamente todos os objetos que atuam como observadores registrados. Tal dinâmica possibilita que múltiplos componentes reagem a eventos sem um acoplamento direto entre o emissor do evento e os receptores que a ele respondem, facilitando a extensibilidade por meio da inclusão de novos comportamentos que respondem a eventos preexistentes, sem a exigência de modificar o código responsável pela geração dos eventos.

O Django incorporou um sistema de sinais que segue o padrão *Observer*, permitindo que os componentes da aplicação recebam notificações quando certas ações acontecem, particularmente durante o ciclo de vida dos modelos, momento em que os sinais são ativados antes e depois de operações de salvamento e exclusão (1). As aplicações comuns incluem auditoria por meio da geração automática de registros de *log*, os quais identificam quem modificou os dados e em que momento, sincronização, mediante a propagação de alterações para sistemas interconectados, notificações, por meio do envio automático de e-mails quando ocorrem eventos

significativos e inicialização, através da criação automática de objetos relacionados quando um novo objeto principal é estabelecido (1).

2.3 Biblioteca React

2.3.1 React *e estrutura fundamentada em* componentes

React é uma biblioteca JavaScript voltada para o desenvolvimento de interfaces de usuário e é baseada no paradigma de componentes, que reúnem estrutura, lógica e estado em unidades que podem ser reutilizadas e compostas. O React, criado pelo Facebook e lançado como código aberto em 2013, permite que os desenvolvedores criem interfaces complexas por meio da composição hierárquica de componentes mais simples. Esses componentes têm responsabilidades bem definidas e podem ser desenvolvidos, testados e mantidos de maneira independente (4).

O Virtual DOM é uma otimização que funciona com React mantendo uma representação em memória da estrutura de DOM desejada e realizando a comparação da versão atual com a versão anterior, sempre que um componente é *re-renderizado*. Por meio de um algoritmo de reconciliação eficiente, esse processo possibilita a determinação do conjunto mínimo de operações necessárias para atualizar o DOM real. Essa metodologia permite que os desenvolvedores escrevam código como se a interface inteira fosse *re-renderizada* a cada mudança de estado, um paradigma que é significativamente mais fácil de compreender. Ao mesmo tempo, o React garante que apenas as partes que realmente mudaram sejam atualizadas no DOM real (1).

Nesse cenário, a composição surge como um princípio essencial, no qual elementos complexos são criados por meio da combinação de elementos mais simples, ao invés de recorrer à herança.

2.3.2 TypeScript *e tipagem estática em* JavaScript

TypeScript constitui um superconjunto de JavaScript, elaborado pela Microsoft, que introduz um sistema de tipos estático opcional. Este sistema possibilita que os

desenvolvedores especifiquem tipos para variáveis, parâmetros de funções e valores de retorno, que são verificados durante o tempo de compilação. Desenvolvido por Anders Hejlsberg e disponibilizado em 2012, o TypeScript identifica erros frequentes, como o acesso a propriedades ausentes ou a passagem de argumentos de tipos inadequados, antes da execução do código, por meio de uma análise estática que complementa a natureza dinâmica do JavaScript, oferecendo uma rede de segurança durante o processo de desenvolvimento (13).

O sistema de tipos do TypeScript é caracterizado como estrutural, em contraste com o nominal, o que implica que a compatibilidade é definida pela estrutura ao invés de uma declaração explícita de tipo. Essa abordagem está em conformidade com a natureza do *duck typing* presente no JavaScript (1). A inferência de tipos possibilita ao TypeScript deduzir automaticamente tipos com base na forma como são utilizados, de modo a possibilitar a redução da quantidade de anotações explícitas requeridas. O desenvolvedor precisa anotar tipos apenas em pontos de fronteira, como nos parâmetros de função, enquanto o TypeScript dissemina as informações ao longo do código.

2.3.3 Gerenciamento de estado

Um desafio essencial em aplicações *frontend* complexas é a gestão do estado, na qual vários componentes precisam compartilhar e sincronizar dados, responder de forma consistente a mudanças e manter a interface atualizada para refletir o estado atual da aplicação. O React disponibiliza primitivas essenciais para o gerenciamento de estado de forma local em componentes, além de possibilitar o compartilhamento desse estado através da hierarquia por meio da *Context* API. Isso possibilita a implementação de padrões de gerenciamento que se adaptam a aplicações que variam de simples a complexas (1).

Hooks consistem em funções singulares apresentadas no React 16.8 que permitem a componentes funcionais acessar estado e outros recursos que anteriormente eram restritos a componentes de classe, oferecendo uma interface de programação mais ergonômica para a gestão de comportamentos por meio de funções simples (4). O *hook* useEffect possibilita a execução de efeitos colaterais,

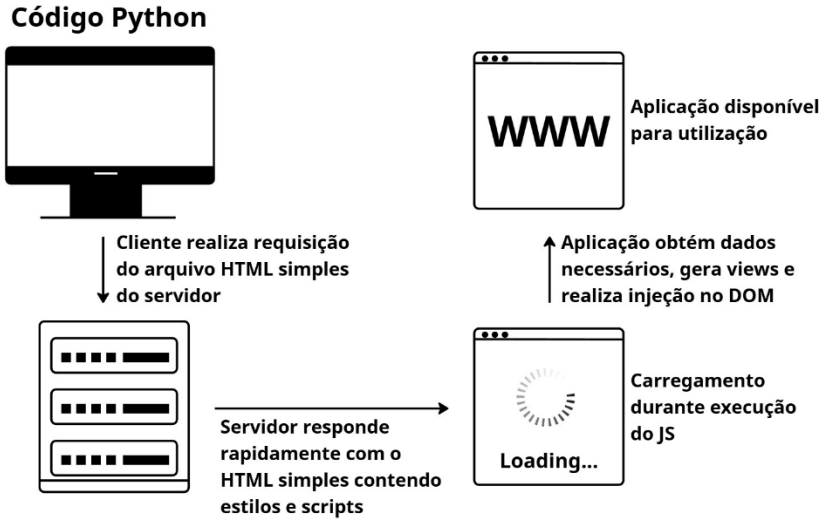
como a inscrição em fontes de dados externas ou a realização de operações que devem ser efetuadas em resposta a uma renderização. Ele aceita uma função de efeito, a qual é chamada após a renderização e um *array* de dependências, que define o momento em que o efeito deve ser reexecutado. A *Context* API disponibiliza uma ferramenta para disseminar valores ao longo da hierarquia de componentes, o que elimina a exigência de transmitir *props* em todos os níveis, solucionando, dessa forma, a questão do *prop drilling* (1).

2.3.4 Roteamento client-side

O roteamento no lado do cliente possibilita que aplicações de página única realizem a navegação entre distintas visualizações sem a necessidade de recarregar o documento HTML completo do servidor, de modo a gerenciar a sincronização entre a URL apresentada no navegador e os componentes renderizados através da manipulação da History API. Essa abordagem facilita transições instantâneas, além de permitir que o estado de aplicações que seriam comprometidas em um recarregamento total possa ser mantido e possibilitar implementar animações de transição entre visões (1).

O React *Router* configura uma biblioteca padrão que disponibiliza componentes declarativos, de modo a permitir estabelecer a correspondência entre *URLs* e os componentes que devem ser exibidos. Essa ferramenta administra a sincronização bidirecional, na qual alterações na URL ocasionam a renderização correspondente, enquanto a navegação programática altera a URL sem a necessidade de recarregar a página (1). A proteção de rotas é realizada através de componentes *wrapper* que realizam a verificação de autorização antes de exibir a rota protegida, de forma a redirecionar usuários não autenticados, enquanto o carregamento sob demanda de rotas possibilita a carga do código apenas no momento em que a rota é acessada, como demonstrado na Figura 8, diminuindo o tamanho do *bundle* inicial por meio da divisão de código.

Figura 8 – Roteamento *client-side* em aplicações SPA.



Fonte: autor, 2025.

2.4 Segurança em aplicações *web*

2.4.1 OWASP *Top 10 e vulnerabilidades críticas*

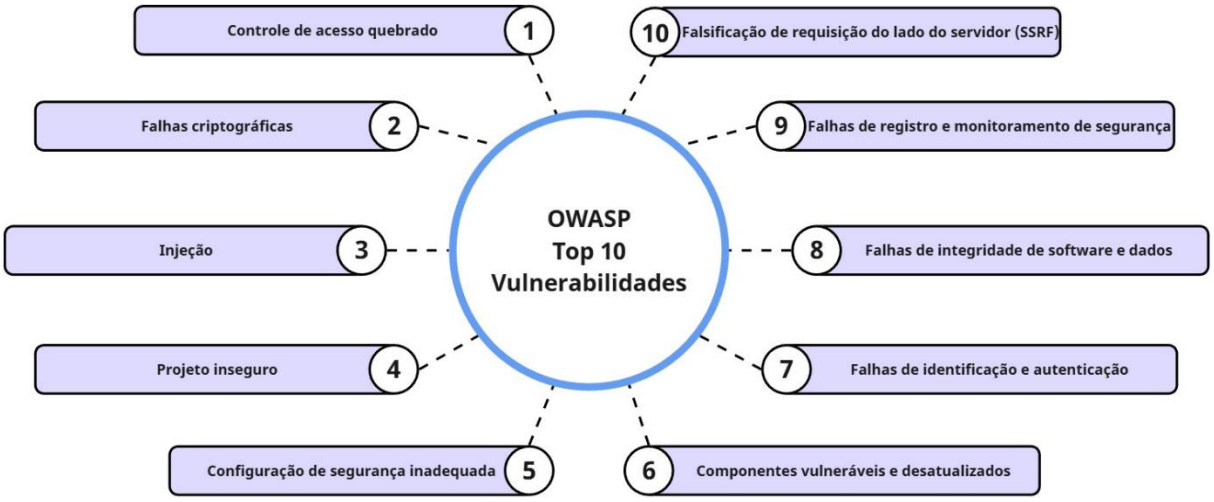
O OWASP Top 10 é uma lista que inclui as vulnerabilidades de segurança mais críticas e comuns em aplicações *web* e é atualizada regularmente. Essa compilação é feita pelo *Open Web Application Security Project*, que utiliza informações sobre a ocorrência de vulnerabilidades identificadas em aplicações reais, combinadas com estimativas de exploitabilidade, detectabilidade e impacto técnico (14). Este conjunto de diretrizes atua como um referencial essencial para que desenvolvedores e instituições direcionem suas iniciativas de segurança, concentrando-se nos riscos mais relevantes que impactam as aplicações *web*, como representado na Figura 9.

Alguns exemplos de vulnerabilidades comuns são:

- Injeção: uma categoria em que informações não confiáveis, fornecidas pelo usuário, são enviadas ao interpretador como parte de um comando ou consulta. Desse modo um invasor pode executar comandos não previstos ou acessar dados sem autorização através de uma entrada maliciosa que modifica a lógica do comando;

- *Cross-Site Scripting* (XSS): classificado como um tipo de injeção, possibilita que invasores insiram *scripts* prejudiciais em páginas da *web* acessadas por outros usuários e execute JavaScript dentro do contexto da sessão da vítima, o que possibilita o furto de cookies ou a realização de ações em nome do usuário;
- *Cross-Site Request Forgery* (CSRF): considerado como categoria falhas de identificação e autenticação, nesse caso é explorada a confiança que um site deposita no navegador de um usuário autenticado, induzindo a vítima a enviar uma solicitação maliciosa, resultando na execução de uma ação não intencional (14).

Figura 9 – Principais vulnerabilidades de segurança em aplicações *web*.



Fonte: autor, 2025.

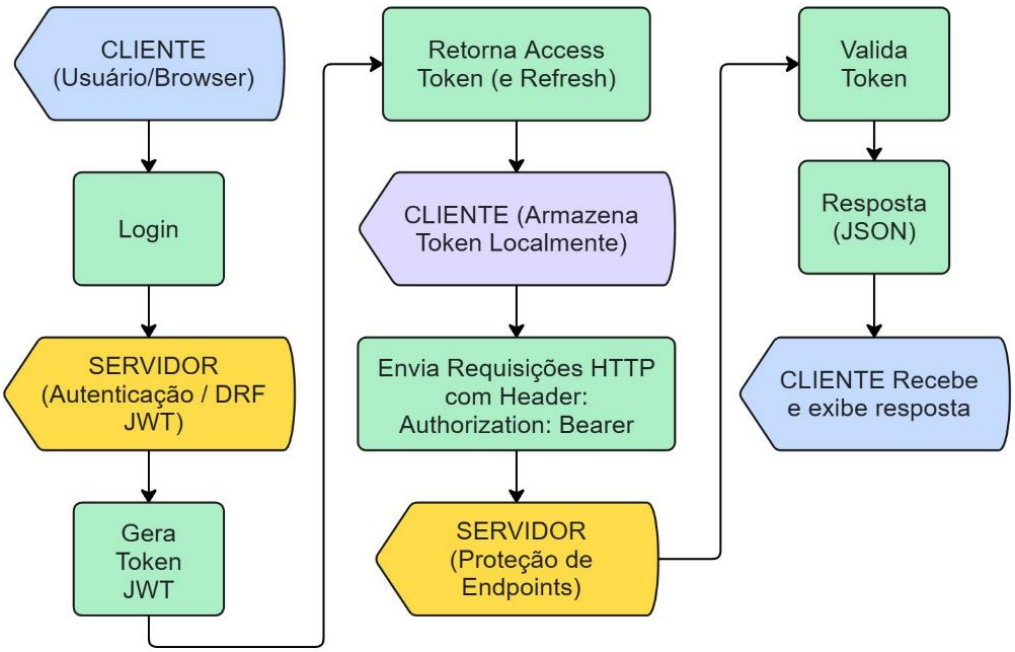
2.4.2 Autenticação JWT*: tokens, renovação e revogação*

Os JWT configuram um padrão aberto, de acordo com o estipulado pela RFC 7519, que define um formato conciso e autossuficiente para a troca de informações entre as partes, apresentando-se na forma de um objeto JSON assinado digitalmente, sendo utilizados predominantemente para fins de autenticação e troca de informações. A configuração de um JWT é formada por três partes codificadas em *Base64*, divididas por pontos: um cabeçalho que define o algoritmo de assinatura, um *payload* que abriga as *claims* sobre o usuário, incluindo seu identificador e o

timestamp de expiração e uma assinatura que é criada através da utilização de um algoritmo criptográfico com uma chave secreta (10).

A validação do JWT no servidor está representada na Figura 11 e envolve a verificação da assinatura ao recalcular o *hash* do cabeçalho e do corpo (*payload*), de forma a utilizar a mesma chave secreta e comparação com a assinatura que acompanha o *token*. Esse processo garante que o *token* foi gerado por um servidor confiável e que não foi alterado. Subsequentemente, realiza-se a verificação das *claims*, com destaque para a *claim* "exp", a fim de assegurar que o *token* não tenha perdido a validade. A renovação de *tokens* trata do desafio de equilibrar a segurança, por meio de *tokens* de curta duração e a usabilidade, realizada com a inclusão de *access tokens* de curta duração nas requisições de API e *refresh tokens* de longa duração, os quais são utilizados unicamente para a obtenção de novos *access tokens*. Por outro lado, a revogação é realizada por meio de uma *blacklist* mantida em *cache*, que contém identificadores de *tokens* revogados com um tempo de vida (TTL) igual ao tempo restante para a expiração natural (10).

Figura 10 – Fluxo de autenticação JWT.

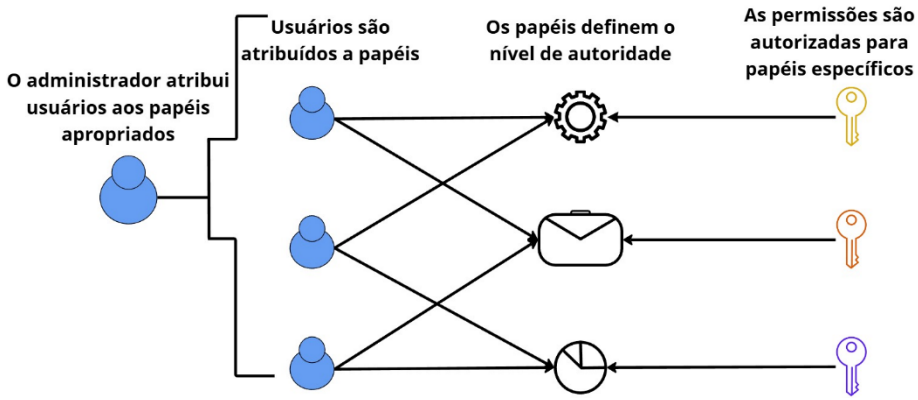


Fonte: autor, 2025.

*2.4.3 Controle de Acesso Baseado em Papéis (*RBAC*)*

O controle de acesso fundamentado em papéis constitui uma estratégia para administrar autorizações em que as permissões são vinculadas a papéis, em vez de a usuários específicos. Os usuários, por sua vez, são designados a papéis compatíveis com suas funções, como ilustrado nas figuras 11 e 12. Uma forma de gerenciar o processo é a utilização de um modelo padrão do NIST para o Controle de Acesso Baseado em Papéis (RBAC), o qual facilita a administração das permissões em sistemas que possuem um grande número de usuários, por meio da gestão de um grupo reduzido de papéis, em vez de configurar as permissões individualmente para cada usuário. Esse modelo reflete a estrutura organizacional, na qual usuários que desempenham funções semelhantes costumam necessitar do mesmo conjunto de permissões.

Figura 11 – Modelo de atribuição de papéis e permissões no modelo RBAC.

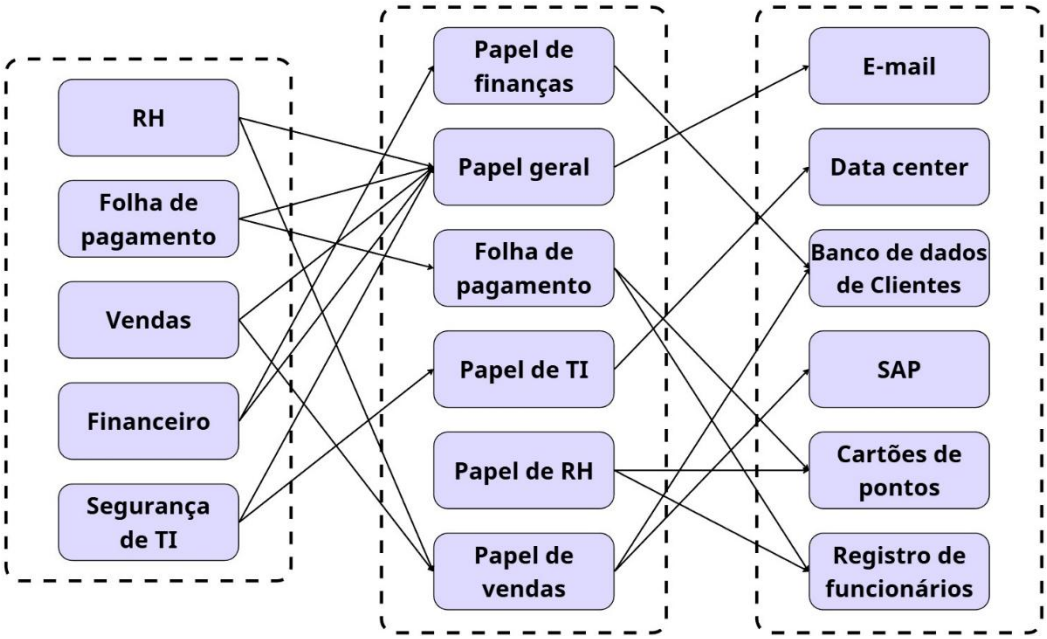


Fonte: autor, 2025.

Os papéis são definidos de acordo com as funções laborais, cada um reunindo um conjunto de permissões relacionadas que representam as atividades que os usuários atribuídos a esse papel devem ser aptos a executar, apresentando uma granularidade que varia desde papéis abrangentes, como o de administrador, até aqueles específicos de domínio. A verificação de autorização ocorre em diversas camadas: a camada de apresentação é responsável por ocultar funcionalidades que o usuário não está autorizado a acessar, o que melhora a usabilidade, a camada de lógica de negócios analisa as permissões antes de efetuar operações essenciais,

garantindo a segurança e a camada de dados pode filtrar as consultas para disponibilizar apenas os registros que possuem autorização, implementando, assim, a segregação de dados.

Figura 12 – Exemplo de relações de acesso no modelo RBAC.



Fonte: autor, 2025.

2.4.4 Proteções CORS*,* CSRF *e validação de dados*

A Política de Mesma Origem constitui uma importante restrição de segurança aplicada por navegadores, a qual impede que *scripts* em execução em uma página de uma origem acessem dados de uma página de origem distinta. Nesse contexto, a origem é definida como a combinação de esquema, domínio e porta. Essa política tem como objetivo proteger contra ataques nos quais um site malicioso tenta obter informações confidenciais de outros sites, aproveitando-se do fato de que o navegador insere cookies automaticamente nas solicitações (14). O mecanismo de Cross-Origin *Resource* Sharing (CORS) proporciona uma forma controlada de flexibilizar essa política, de modo a permitir que os servidores especifiquem quais origens têm permissão para acessar seus recursos por meio de cabeçalhos HTTP.

A validação de dados em várias camadas é uma técnica de defesa em profundidade, na qual os dados passam por processos de validação no cliente,

servidor e banco de dados. A validação feita no lado do cliente pode ser contornada por invasores, apesar de proporcionar um retorno imediato, melhorando assim a experiência do usuário (15). Por outro lado, a validação no lado do servidor é fundamental para a segurança, uma vez que não pode ser comprometida e deve rejeitar toda entrada inválida. Além disso, as restrições impostas pelo banco de dados constituem a última linha de defesa, assegurando a integridade dos dados, mesmo que as validações anteriores não tenham sido efetivas.

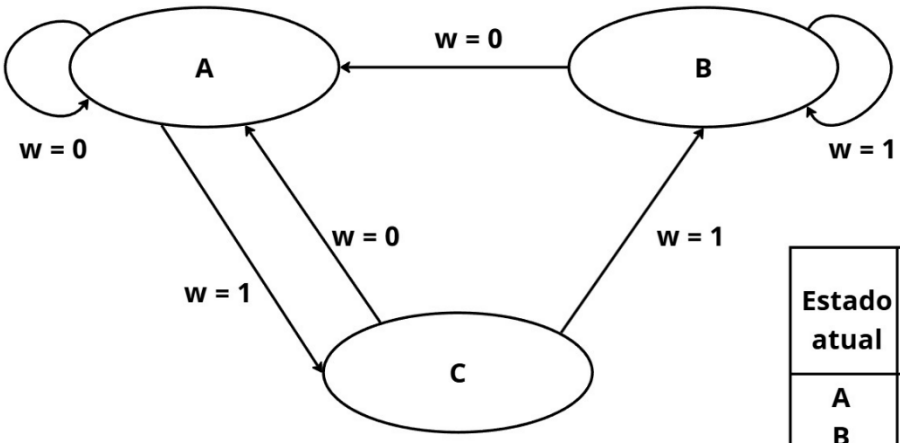
2.5 Aplicação da engenharia de *software* em sistemas de gestão

*2.5.1 Máquinas de Estado Finito (*FSM*)*

As máquinas de estados finitos constituem um modelo computacional que abrange um conjunto limitado de estados, transições entre esses estados fundamentadas em eventos ou condições, além das regras que definem o estado inicial e os estados finais, como exemplificado na Figura 13. Esse formalismo é especialmente efetivo na modelagem de processos de negócios e fluxos de trabalho em que uma entidade avança por uma sequência de estados claramente definidos, desde a sua criação até a conclusão. As transições representam as ações que deslocam a entidade entre os estados, enquanto as regras de transição asseguram que apenas progressões válidas sejam permitidas (17).

Uma definição precisa dos estados exige a explicitação de todas as condições possíveis que uma entidade pode assumir ao longo de seu ciclo de vida, de modo a evitar estados ambíguos que podem resultar em comportamentos inconsistentes. Além disso, a implementação geralmente se utiliza de um campo de estado em um modelo de dados, que armazena o estado atual como um *enum* ou *string*, contando com métodos de transição que validam se a troca de estado é permitida, executam a lógica de negócios necessária, atualizam o campo de estado e registram a transição em um *log* de auditoria (15).

Figura 13 – Exemplo de máquina de estados finita (FSM).



Estado atual	Próximo estado w = 0	Próximo estado w = 1	Saída
A	A	C	0
B	A	B	1
C	A	B	0

Fonte: autor, 2025.

2.5.2 Validação em múltiplas camadas

A validação em múltiplas camadas aplica o princípio de defesa em profundidade, no qual os dados são conferidos em cada camada da aplicação que manipula ou armazena informações. Essa estratégia reconhece que nenhuma camada isolada é totalmente segura e que múltiplas camadas independentes proporcionam redundância, essa dinâmica torna significativamente mais difícil a inserção de dados inválidos ou maliciosos. Dessa forma, procura-se um equilíbrio entre as preocupações com a experiência do usuário, por meio da validação no lado do cliente, que oferece retorno imediato, a segurança, por meio da validação no lado do servidor, que é incontestável e a integridade, por meio de restrições no banco de dados (15).

A validação no lado do servidor estabelece uma camada fundamental que deve tratar toda a entrada recebida como não confiável, a despeito das validações executadas no lado do cliente. Nesta fase, tipos, formatos, intervalos e regras de negócio complexas são verificados antes que os dados sejam processados ou armazenados. A possibilidade de acessar um contexto amplo da aplicação permite executar validações mais sofisticadas, como a checagem da unicidade de valores ou a confirmação de permissões (15). Restrições em bancos de dados representam a última linha de defesa ao implementar regras de integridade de forma direta no esquema, por meio de restrições como NOT NULL, UNIQUE, CHECK e chaves estrangeiras, que são aplicadas independentemente do código da aplicação. Tais

restrições são especialmente relevantes para evitar a corrupção de dados ocasionada por problemas de controle de ordem (17).

2.5.3 Padrões de projeto em aplicações web

Os padrões de projeto constituem soluções reaproveitáveis para questões recorrentes em design de *software*, as quais são identificadas e registradas a partir da experiência coletiva de programadores. Esses padrões oferecem um vocabulário compartilhado para transmitir arquiteturas e decisões de design, ao mesmo tempo em que codificam práticas recomendadas que previnem armadilhas reconhecidas. As aplicações *web* contemporâneas frequentemente implementam múltiplos padrões que operam em conjunto, sendo que cada um deles contempla um aspecto específico do design (8).

O padrão *Observer* estabelece um sistema de publicação e assinatura no qual objetos que demonstram interesse em determinados eventos podem se inscrever como observadores de um sujeito. Este, por sua vez, notifica automaticamente todos os observadores cadastrados sempre que ocorre uma alteração, favorecendo o baixo acoplamento pela eliminação de dependências diretas. O repositório abstrai o acesso a dados ao oferecer uma interface análoga a uma coleção para a interação com objetos de domínio, ao mesmo tempo em que encobre a lógica de consultas, separando a lógica de negócios dos aspectos relacionados à persistência (7). Por outro lado, o padrão Strategy estabelece uma família de algoritmos, encapsulando cada um em uma classe distinta, permitindo sua intercambialidade por meio de uma interface comum (8).

2.5.4 Arquitetura modular e separação de responsabilidades

A arquitetura modular organiza o *software* em módulos autônomos, em que cada módulo reúne um conjunto de funcionalidades interconectadas e oferece uma interface bem delimitada, enquanto oculta os detalhes de implementação. Este princípio permite o desenvolvimento, a testagem e a manutenção autônoma de módulos, os quais se conectam por meio de interfaces consistentes que definem

acordos entre os componentes. Essa abordagem baseia-se na distribuição de responsabilidades, garantindo que cada módulo tenha uma única responsabilidade claramente delineada, evitando a formação de módulos que busquem englobar múltiplas funções, o que os torna complexos para serem compreendidos (17).

A coesão avalia o quão interligadas e concentradas estão as responsabilidades de um módulo, de forma que uma alta coesão sugere que seus elementos colaboram para um único propósito e resulta em módulos que são facilmente compreensíveis, modificáveis e reutilizáveis. Por outro lado, o acoplamento examina o nível de interdependência entre os módulos, um baixo acoplamento indica que os módulos possuem dependências mínimas entre si, por meio de interfaces bem definidas, o que permite modificações sem que sejam necessárias alterações em cascata em todo o sistema (15). O acoplamento é administrado por meio do encapsulamento, que oculta detalhes de implementação atrás de interfaces públicas, pela injeção de dependências, que fornece as dependências a módulos, em vez de permitir que estes criem as dependências de forma direta e pela utilização de abstrações, que possibilitam a substituição de implementações concretas sem impactar o código que delas depende (17).

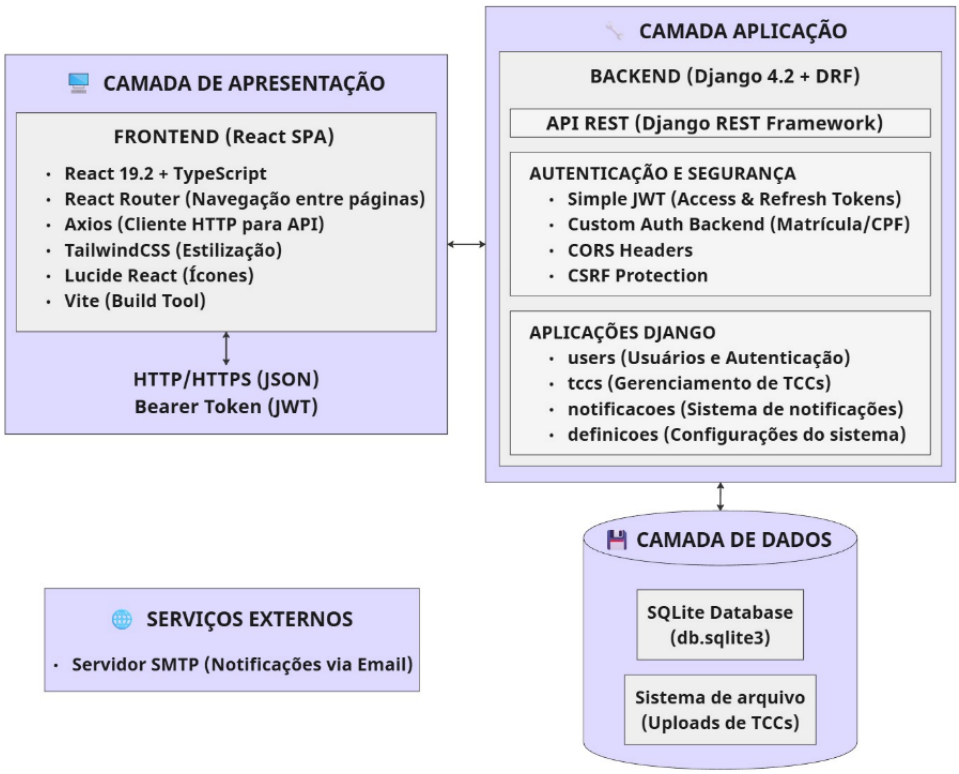
3 DESENVOLVIMENTO DO TRABALHO

3.1 Visão geral

3.1.1 Estrutura Cliente-Servidor

O sistema foi implementado seguindo padrão arquitetural cliente-servidor de três camadas, de forma que cada um dos componentes possui distintas responsabilidades e comunica-se através de interfaces normalizadas no sistema, como demonstrado na Figura 14. Dessa forma, é possível estabelecer uma organização adequada, baseada na divisão de funções bem definidas, de modo que a camada apresentação (executada no cliente) tem o foco na interação com o usuário, a camada lógica de negócio (centralizada no servidor) é responsável pela implementação das regras e a camada persistência gerencia o armazenamento das informações.

Figura 14 – Arquitetura Cliente-Servidor da Plataforma.



Fonte: autor, 2025.

Camada de apresentação

A aplicação foi criada como uma SPA (*Single Page Application*) usando a biblioteca React 19 juntamente com a linguagem TypeScript. Seu funcionamento é fundamentado na execução completa no navegador, de modo a necessitar encarregar-se apenas pela renderização da interface, registro das interações e exibição visual dos dados. Não há acesso direto ao banco de dados e toda comunicação com o servidor é feita por meio de requisições HTTP assíncronas à interface REST. Essa metodologia permite que a navegação seja realizada sem a necessidade de recarregamento completo. Nesse processo, o gerenciamento de estado local é feito pelo React, o que possibilita atualizações parciais apenas nos componentes que sofreram alterações.

Camada de lógica de negócio

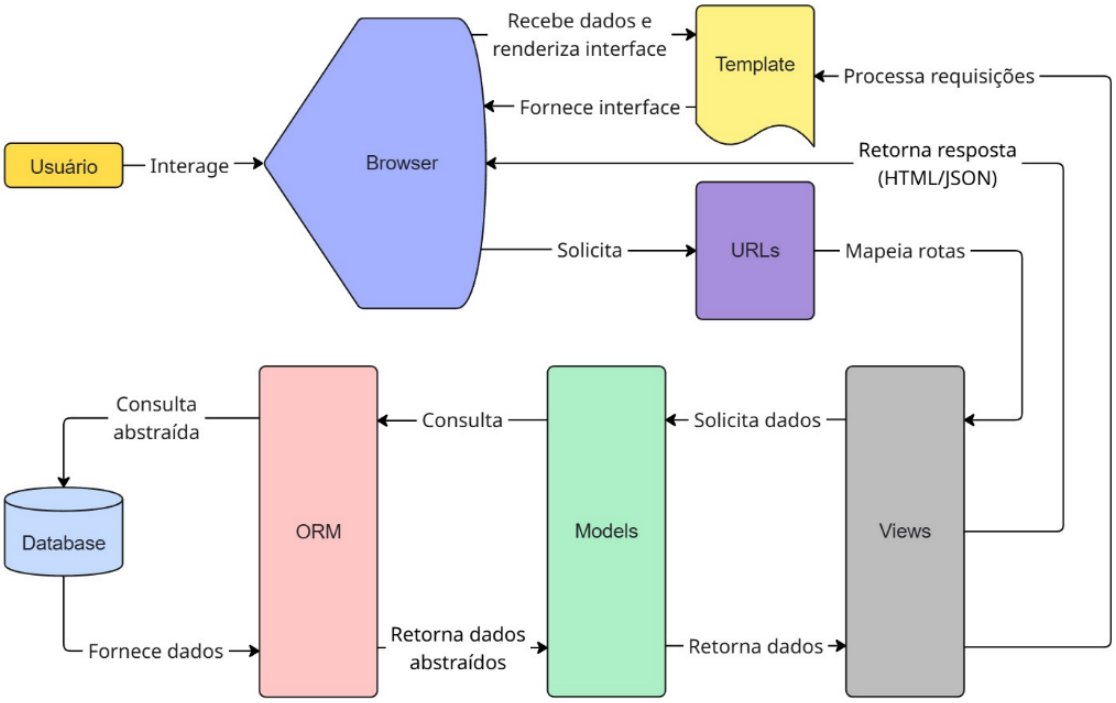
Foi desenvolvida com Django 4.2.16 e Django REST Framework 3.14, de modo a concentrar a implementação de regras de negócio complexas, como por exemplo validações de prazos fundamentadas no calendário acadêmico, controle detalhado de permissões, cálculos automáticos de notas, gerenciamento de máquina de estados com transições automatizadas e orquestração de fluxos de trabalho de avaliação. O *backend* disponibiliza funcionalidades através de uma interface REST, de forma em que cada *endpoint* corresponde a uma operação sobre um recurso do domínio. Essa abordagem *stateless* possibilita a escalabilidade horizontal sem a necessidade de sincronização de sessão. A separação arquitetural permite que equipes especializadas desenvolvam paralelamente e facilita a adição de clientes alternativos, como aplicativos móveis e simplifica os testes por meio do isolamento total das camadas.

Camada de persistência

A camada de persistência utiliza o SQLite e tem suporte configurado para migração para PostgreSQL. O ORM (Object Relational Mapper) do Django oculta as variações entre os sistemas gerenciadores, de modo a possibilitar que os modelos sejam estabelecidos uma única vez e convertidos automaticamente para o dialeto SQL adequado. Essa camada é acessada apenas pelo *backend* e não contém acesso direto pelo *frontend*. Assegura-se a consistência, independentemente do fluxo de

execução, por meio de validações de integridade, restrições de unicidade e índices estabelecidos no nível do modelo, assim como exemplificado na Figura 15.

Figura 15 – Fluxo de Requisições no Padrão MTV do Django.



Fonte: autor, 2025.

3.1.2 Padrão de Comunicação

A interação entre o *frontend* e o *backend* foi realizada em conformidade com os princípios da arquitetura REST, adotando o protocolo HTTP como camada de transporte e o formato JSON para a serialização de dados. A interface foi elaborada em conformidade com restrições essenciais, englobando a separação entre cliente e servidor, a comunicação sem estado (*stateless*), a possibilidade de respostas que podem ser armazenadas em *cache*, a uniformidade da interface e a organização em camadas do sistema. Cada recurso do domínio é disponibilizado por meio de uma URL exclusiva, enquanto as operações estão associadas a métodos HTTP padronizados: *GET*, *POST*, *PUT* e *PATCH* para a atualização e *DELETE* para a eliminação. *URLs* aderem à convenção *RESTful* e empregam substantivos no plural para representar coleções e identificadores para recursos específicos, além de ações customizadas que fogem do padrão CRUD, utilizando sufixos verbais descritivos.

O JSON foi selecionado como formato de serialização em virtude de sua leveza, facilidade de leitura por humanos e ampla aceitação e compatibilidade nativa com JavaScript. Os serializadores do Django REST Framework realizam a conversão automática entre objetos Python e sua representação em JSON, mantendo os tipos de dados e possibilitando a serialização de relacionamentos, por meio de identificadores numéricos ou através de objetos aninhados completos.

A autenticação foi realizada por meio da utilização de *tokens* JWT em lugar de sessões armazenadas no servidor. Após a realização de um *login* bem-sucedido, o *backend* envia um *token* de acesso válido por 60 minutos, além de um *token* de atualização que possui validade de 7 dias. O *token* de acesso é inserido no cabeçalho de autorização das requisições (utilizando o esquema portador) e o *backend* realiza a validação através da verificação da assinatura criptográfica, da expiração e das declarações, de modo a dispensar necessidade de consultar o banco de dados a cada requisição. Quando o *token* de acesso expira, o *frontend* utiliza automaticamente o *token* de atualização para obter um novo par, de modo a dispensar que seja necessária uma nova autenticação. Os *tokens* de atualização são renovados a cada uso, o que resulta na criação de um novo *token* e na inclusão do *token* anterior em uma lista de revogação, com o intuito de impedir a reutilização de *tokens* que possam estar comprometidos, restringindo assim a janela de vulnerabilidade.

A interface faz uso de códigos de estado HTTP semânticos, como demonstrado na Tabela 3. As respostas de erro são estruturadas de maneira padronizada, retornando mensagens descritivas ou objetos de validação que relacionam campos a erros específicos, assim é possível um tratamento centralizado via interceptadores.

Tabela 3 – Códigos de *status* HTTP.

Código	*Status*	Descrição
200	*Ok*	Sucesso
201	*Created*	Recurso criado
204	*No Content*	Sucesso sem corpo
400	*Bad Request*	Erro de validação
401	*Unauthorized*	*Token* ausente ou expirado
403	*Forbidden*	Permissão negada
404	*Not Found*	Recurso não encontrado
500	*Internal Server Error*	Falha no servidor

Fonte: autor, 2025.

3.1.3 Tecnologias Empregadas

A escolha das tecnologias fundamentou-se em critérios como maturidade, qualidade da documentação, engajamento da comunidade e conformidade técnica em relação aos requisitos do projeto.

O React 19 foi utilizado para o desenvolvimento da interface em função de seu funcionamento se basear em uma abordagem fundamentada em componentes, possibilitando a criação de interfaces complexas utilizando da composição hierárquica de componentes reutilizáveis. O sistema de *hooks* simplifica a gestão de estado e de efeitos colaterais, resultando em um código mais conciso e passível de testes, enquanto a reconciliação virtual aprimora as atualizações ao calcular as mínimas diferenças. TypeScript incorpora um sistema de tipos estático, o que possibilita autocompletar de forma inteligente, identificação antecipada de erros, refatoração segura e aprimoramento da documentação por meio de interfaces. O React *Router* DOM 7 administra a navegação ao associar *URLs* a componentes, possibilitando rotas com parâmetros dinâmicos, navegação programática e rotas protegidas que realizam a verificação de autenticação. Axios atua como um cliente HTTP e proporciona uma interface prática, realizando a transformação automática de JSON. Desse modo é possível a utilização de interceptadores para a manipulação de requisições e respostas, além de possibilitar o cancelamento de solicitações. O TailwindCSS disponibiliza classes utilitárias que permite a criação ágil de interfaces, de modo a resultar em um arquivo CSS final reduzido, ao eliminar estilos que não são utilizados. Vite foi selecionado como empacotador devido à sua capacidade de proporcionar inicialização imediata, substituição de módulo extremamente ágil e compilações otimizadas.

O Django 4.2 foi escolhido como o *framework web* devido à sua capacidade de fornecer um mapeamento objeto-relacional eficaz, um sistema automático de migrações, um painel administrativo integrado, um sistema de autenticação sólido e um ecossistema abrangente.

O Django REST Framework estende as funcionalidades do Django, de modo a disponibilizar conjuntos de visualizações que implementam operações CRUD com um código reduzido e serializadores que realizam a conversão entre os modelos e o JSON. Inclui também um sistema de permissões granulares, paginação automática e

filtros. O djangorestframework-simplejwt fornece autenticação baseada em *tokens web* JSON, suporte a listas de revogação, personalização de declarações e rotação automática. O SQLite é utilizado como sistema de gerenciamento durante o desenvolvimento devido à sua leveza e à ausência da necessidade de um servidor separado, enquanto o PostgreSQL pode ser utilizado para o ambiente de produção, de modo a dar suporte integral ao ACID (propriedades fundamentais de transações em bancos de dados) e tipos de dados avançados e excelente desempenho.

3.1.4 Circulação de Dados e Integração

O fluxo de dados adota um padrão unidirecional consistente, no qual toda a comunicação entre as camadas transita pela interface REST, criando um ponto de integração singular com regras centralizadas. Ao interagir com a interface, o usuário realiza a validação dos dados no *frontend* através de algumas verificações de segurança e transmite uma requisição HTTP com o token JWT no cabeçalho. O *backend* recebe uma requisição, valida o *token* através de *middleware* que extrai o identificador do usuário e verifica as permissões através de classes personalizadas que analisam o tipo de usuário e seus relacionamentos. Em seguida, a lógica de negócio é executada consultando ou alterando o banco de dados através do mapeamento objeto-relacional e retornando uma resposta em formato JSON com o código de estado apropriado. O *frontend* processa a resposta, modifica o estado por meio de *hooks* e da *Context* API e *re-renderiza* automaticamente os componentes impactados, apresentando mensagens ao usuário por meio de notificações temporárias ou alertas embutidos.

Interceptadores do Axios acompanham as respostas ao examinar os códigos de *status*. Quando o *backend* emite um código 401, sinalizando que o *token* expirou, o interceptador interrompe o processamento e executa o procedimento de renovação, enviando uma requisição ao *endpoint* designado para atualização, utilizando um *token* de renovação válido. Uma nova resposta, contendo um par de *tokens*, é processada por meio da atualização do armazenamento local e do contexto de autenticação. Após uma renovação bem-sucedida, a solicitação original é reexecutada de maneira automática com um novo *token* e a resposta é encaminhada ao código que a iniciou,

como se nenhum erro tivesse acontecido, assegurando uma experiência sem interrupções, sem exigir um novo *login*.

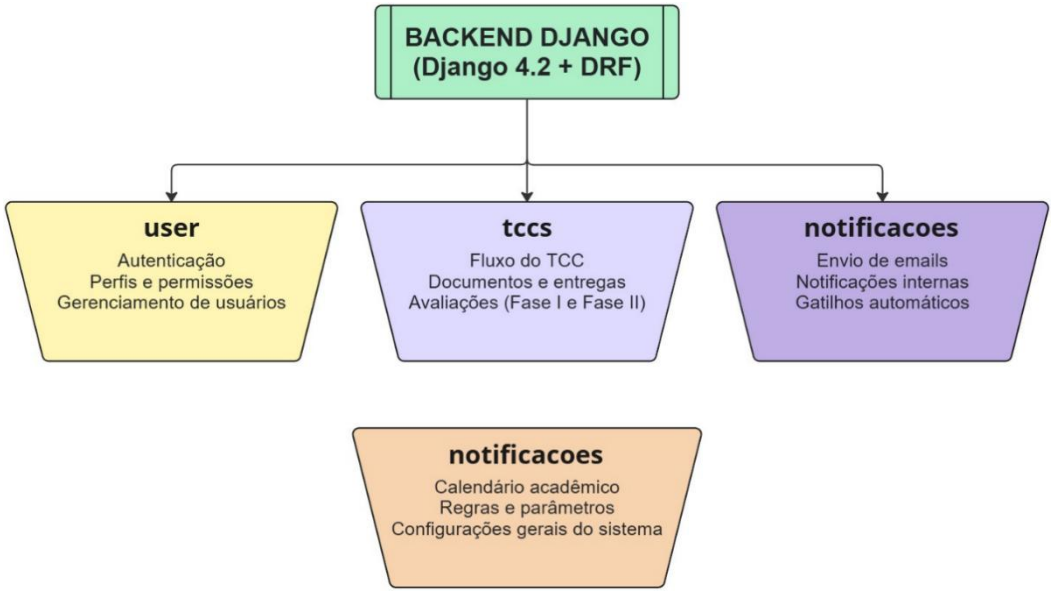
O sistema implementa uma camada dupla de notificações: notificações internas, que são armazenadas no banco de dados e estão relacionadas ao destinatário e um TCC opcional, que são carregadas periodicamente pelo *frontend* por meio de consultas realizadas a cada 30 segundos, atualizando contadores e listagens, por sua vez, as notificações via correio eletrônico são enviadas pelo *backend* mediante um sistema configurado com servidor SMTP, respeitando três níveis de controle, incluindo uma *flag* global que indica a ativação do envio, preferências detalhadas por tipo que são configuradas individualmente e a validação do endereço cadastrado. Os sinais do Django, integrados aos modelos principais, automatizam a geração de notificações sempre que eventos significativos ocorrem, como alterações de estado, aprovações e conclusões, assegurando uma comunicação uniforme, independentemente do fluxo de execução.

3.2 *Backend*

3.2.1 Estrutura modular do Django

O *backend* foi elaborado de acordo com a arquitetura modular do Django, de modo que a aplicação é segmentada em quatro módulos autônomos, assim como apresentado na Figura 16, sendo cada um responsável por um domínio particular do sistema. Essa divisão fundamenta-se no modelo de alocação de responsabilidades, o que simplifica a manutenção e possibilita a realização de testes unitários independentes para cada componente.

Figura 16 – Módulos do *backend* da plataforma.



Fonte: autor, 2025.

O módulo inicial, *users*, desenvolve um modelo de usuário personalizado que amplia as classes fundamentais de autenticação do Django, trocando o identificador convencional por um endereço de e-mail. Adicionando ao modelo principal, foram desenvolvidos modelos auxiliares que abordam preferências visuais e de notificação, estabelecendo vínculos um-para-um com o usuário, os quais asseguram informações específicas para cada perfil.

O módulo tccs engloba os modelos de domínio primário, de modo a abranger o TCC e suas entidades associadas, tais como solicitações de orientação, documentos, comissões, avaliações, agendamentos e histórico de eventos. Para gerenciar o acesso a esses recursos, o módulo desenvolve classes de permissão personalizadas, as quais ampliam o sistema de permissões do DRF. Ademais, emprega o sistema de sinais do Django para automatizar processos que devem ser ativados em decorrência de alterações nos dados.

O módulo de notificações opera em colaboração com o módulo TCCs, de modo a implementar estruturas para o armazenamento de avisos internos e das preferências de correio eletrônico dos usuários. A geração de notificações é realizada de forma automatizada através de sinais vinculados aos eventos do módulo TCCS e assegura que os usuários sejam avisados sobre alterações significativas. A transmissão de e-

mails é efetuada por meio do sistema de correio do Django, o qual possui um *backend* SMTP que pode ser configurado.

Por fim, o módulo de definições concentra as configurações do sistema que podem ser modificadas pelo coordenador, mediante modelos que armazenam o calendário acadêmico, códigos de cadastro, documentos de referência e parâmetros de e-mail. Este módulo também faz uso de sinais para a implementação dinâmica de configurações e dispensa a reinicialização do servidor, esse dinâmica possibilita que as modificações sejam imediatamente refletidas no sistema.

3.2.2 Interface de Programação de Aplicação REST

A API REST foi desenvolvida empregando o DRF, com a implementação de grupos de *views* fundamentados em recursos que realizam o mapeamento automático de operações CRUD para *endpoints* HTTP. Essa metodologia possibilita que cada elemento do domínio (TCCs, usuários, notificações, entre outros) seja disponibilizado por meio de *URLs* normatizadas, todas organizadas sob o prefixo /*api*/ através do sistema de roteamento do *framework*.

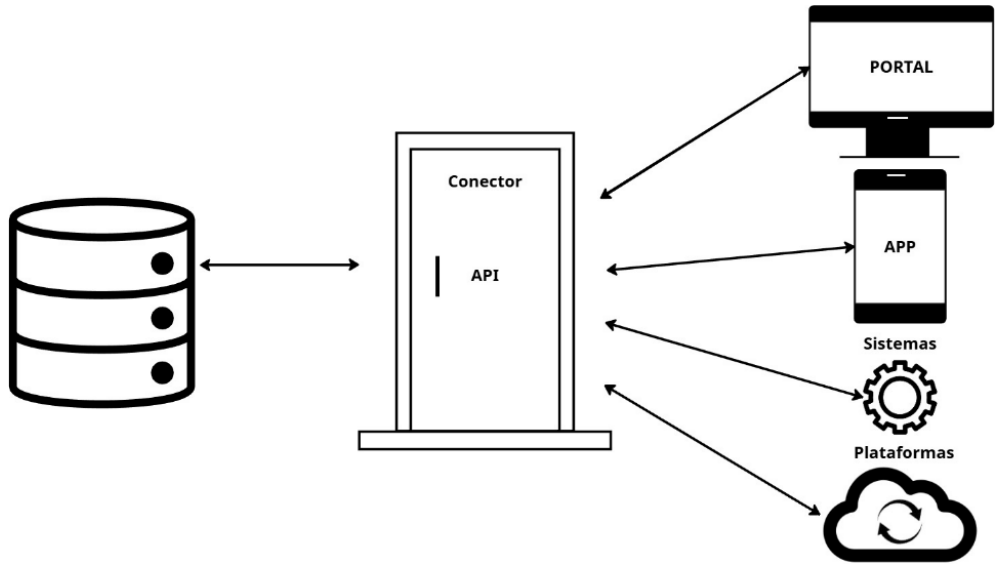
A camada de serialização funciona como um elo entre as representações internas em Python e o formato JSON empregado na API. Os serializadores de modelo realizam a conversão automática de objetos provenientes do banco de dados para o formato JSON (ou no sentido inverso) e implementam métodos de validação personalizados, de modo que aplicam regras de negócio particulares durante esse procedimento. Os serializadores incorporam campos calculados que oferecem dados aninhados e ajustam a estrutura da resposta conforme o contexto de uso. Para operações de listagem, são fornecidas versões resumidas visando a eficiência, enquanto as visualizações individuais contêm informações mais detalhadas. As validações intricadas consultam modelos correlatos e o calendário acadêmico a fim de verificar prazos e imposições antes de autorizar as operações.

Além das validações de dados, o sistema de permissões regula o acesso de cada usuário a diferentes recursos. Foram implementadas classes personalizadas que ampliam o sistema de permissões do *framework*, realizando a verificação da autenticação do usuário e de seu relacionamento particular com os recursos

requisitados. Essas análises avaliam a classificação do usuário e exploram as interações por meio de chaves estrangeiras para identificar, por exemplo, se um docente atua como orientador de um determinado TCC. As permissões são implementadas nos grupos de *views* e podem ser modificadas para operações específicas através de decoradores. Assim, é possível obter um controle detalhado sobre cada *endpoint*.

A estruturação dos *endpoints* emprega roteadores que criam automaticamente *URLs* fundamentadas nos recursos estabelecidos, em conformidade com as convenções *RESTful*, com a dinâmica ilustrada na Figura 17. Além das operações padrão de CRUD, ações específicas do domínio são disponibilizadas por meio de ações personalizadas, sinalizadas por decoradores, como a formação de bancas, o envio de avaliações e a aprovação de documentos, cada uma possuindo suas próprias normas de permissão e validação.

Figura 17 – Arquitetura de integração via API REST.



Fonte: autor, 2025.

3.2.3 Autenticação e autorização

A autenticação foi realizada por meio da implementação da biblioteca *djangorestframework-simplejwt*, a qual amplia as funcionalidades do DRF,

proporcionando suporte a JWT. Essa estratégia sem estado se ajusta à arquitetura com *frontend* desacoplado, removendo a exigência de conservação de sessões no servidor. Os *tokens* de acesso têm validade de 60 minutos e são usados para validar requisições. Já os *tokens* de atualização duram 7 dias e permitem renovar o acesso sem necessidade de nova autenticação.

A criação de *tokens* emprega uma visualização personalizada que substitui o procedimento convencional da biblioteca, a fim de incorporar dados suplementares no *payload*, como a categoria de usuário e as preferências estéticas. Assim, o *frontend* obtém os *tokens* e também informações fundamentais para a configuração inicial da *interface*. Os *tokens* são autenticados pelo algoritmo HS256 e incorporam reivindicações padrão de JWT (identificador, marcas de tempo de emissão e expiração), além de campos personalizados.

Com o intuito de assegurar a segurança, mesmo após o *logout* ou a possível violação de *tokens*, foi estabelecido um sistema de invalidação que utiliza um modelo de lista negra, o qual registra identificadores únicos de *tokens* revogados. No processo de *logout*, o *token* de atualização é inserido na lista por meio de um método específico, tornando-se permanentemente inválido. Adicionalmente, a rotação automática foi ajustada para criar novos *tokens* de atualização a cada renovação, colocando o *token* anterior na lista negra de forma automática, o que diminui consideravelmente as janelas de vulnerabilidade, caso algum *token* seja capturado.

O sistema de autorização opera em parceria com a autenticação e emprega um campo que indica o tipo de usuário no modelo principal, associado ao sistema de permissões do *framework*. As *views* realizam a filtragem personalizada de objetos com base no tipo de usuário autenticado, consultando relacionamentos de chave estrangeira para verificar a propriedade e a participação em recursos. Os alunos têm acesso apenas aos seus respectivos trabalhos de conclusão de curso, os professores podem visualizar os TCCs dos quais são orientadores, enquanto os coordenadores dispõem de uma visão abrangente, tudo gerenciado por meio dessa filtragem dinâmica.

3.2.4 Máquina de estados

A máquina de estados foi desenvolvida para gerenciar o ciclo de vida integral do TCC através de um campo textual no modelo que registra o estado corrente. Os valores admissíveis estão estabelecidos em um módulo de constantes, o qual abrange 5 etapas, representadas na Figura 18, divididas em 14 estados diferentes, entre os quais se encontram três estados terminais que simbolizam conclusões definitivas do processo. A centralização em um módulo de constantes facilita a manutenção e assegura a consistência em todo o sistema.

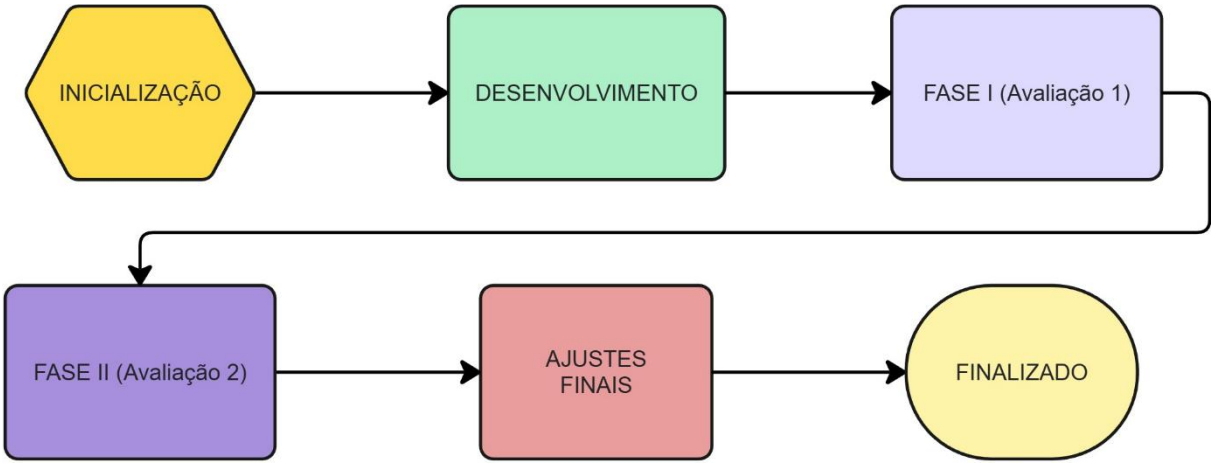
As transições entre estados são executadas por meio de métodos nas visualizações que avaliam de forma criteriosa o estado presente antes de autorizar alterações. Essas validações empregam lógica condicional, a qual analisa não apenas o valor do estado, mas também campos de controle associados que atuam como pré-requisitos para determinadas transições. A autorização para a avaliação exige que determinadas *flags* estejam ativadas, assegurando que as etapas intermediárias foram devidamente realizadas. Certas transições são automatizadas por meio do sistema de sinais do Django: ao serem vinculados aos modelos de avaliação, os sinais verificam se todas as avaliações foram finalizadas e atualizam o estado automaticamente, dispensando a necessidade de intervenção manual.

Foi desenvolvido um comando de gerenciamento no Django que é executado periodicamente através de um agendador de tarefas, para realizar transições que dependem do tempo. Esse comando verifica os agendamentos de defesa, aplicando filtros por data e situação atual e realiza automaticamente a transição para a fase de apresentação assim que o momento agendado é alcançado.

Além do campo referente ao estado principal, o modelo TCC apresenta diversos campos booleanos que atuam como controles complementares do fluxo que englobam indicadores para a aprovação de continuidade, autorização para avaliação e restrição de edições durante o processo de análise. Esses campos possibilitam um controle detalhado, desvinculado do estado presente. Adicionalmente, foram introduzidos campos de autorização manual que possibilitam ao coordenador outorgar exceções aos prazos do cronograma para TCCs específicos, sem comprometer o cronograma geral. Todas essas alterações de estado e indicadores geram

automaticamente registros de eventos por meio de sinalizações, estabelecendo um rastreamento abrangente de auditoria que registra toda a evolução do TCC.

Figura 18 – Fluxo de estados do Trabalho de Conclusão de Curso.



Fonte: autor, 2025.

3.2.5 Sistema de arquivos

A gestão de documentos emprega campos de arquivos do Django, acompanhados por funções personalizadas de *upload*, as quais definem a localização e a maneira de armazenamento dos arquivos. Essas funções criam caminhos organizados de forma hierárquica, estabelecendo uma estrutura na qual cada TCC possui uma pasta exclusiva, dividida por gênero documental. Para assegurar a singularidade dos nomes e prevenir conflitos, identificadores exclusivos são gerados de forma automática e adicionados ao nome de cada arquivo, enquanto o nome original fornecido pelo usuário é mantido em um campo distinto para apresentação.

A validação dos uploads foi realizada em três níveis que operam de maneira integrada para assegurar a integridade. No que tange ao modelo, os validadores conferem as extensões de arquivo aceitas e o tamanho máximo, rejeitando de imediato arquivos que sejam inadequados. No âmbito do serializador, validações particulares asseguram que a extensão é adequada para o tipo de documento que está sendo enviado. Por exemplo, monografias devem obrigatoriamente estar em formato PDF, enquanto outros tipos de documentos aceitam formatos adicionais. Por fim, no âmbito das visualizações, as verificações de permissão analisam o calendário

acadêmico e os campos de liberação manual para estabelecer se o envio ocorre dentro do prazo permitido para o respectivo tipo de documento e usuário.

O controle de versões foi estabelecido por meio de um campo numérico no modelo de documento, o qual é calculado automaticamente no momento da sua criação. O sistema registra documentos anteriores do mesmo modelo para o TCC em questão e fornece o próximo número sequencial, elaborando um histórico abrangente de revisões. Restrições de unicidade em diversos campos asseguram que não existam versões replicadas para a mesma combinação de TCC e categoria documental.

O gerenciamento do ciclo de vida dos documentos é realizado por meio de um campo de *status* que apresenta três valores distintos: pendente, aprovado e rejeitado, além de um campo textual destinado a fornecer um *feedback* detalhado. Alterações no *status* geram imediatamente sinais que produzem automaticamente registros de eventos na linha do tempo do TCC e notificações para o estudante, instaurando um modelo de observador, no qual diversos componentes respondem a essas modificações sem um acoplamento direto entre eles.

3.2.6 Método de avaliação

A avaliação foi instaurada por meio de dois modelos fundamentais que se relacionam com as etapas do processo. Cada modelo dispõe de cinco campos numéricos com precisão decimal, destinados aos critérios de avaliação correspondentes à fase, junto a um campo calculado que resulta na soma das notas dos critérios. A distinção dos modelos possibilita que as etapas apresentem critérios variados, ao mesmo tempo em que conservam uma estrutura semelhante. Os limites máximos autorizados para cada critério são armazenados no modelo de calendário acadêmico em campos decimais, possibilitando à instituição uma flexibilidade na estipulação da importância relativa de cada elemento.

A verificação das notas acontece em diversos momentos para assegurar a consistência. Nos serializadores, os métodos de validação analisam o calendário em vigor e confirmam se cada nota individual não ultrapassa o peso estabelecido para tal critério na fase pertinente e o modelo de calendário conta com uma validação que

assegura que a soma dos pesos atinja exatamente 10,0, gerando um erro em situações de configuração inconsistente.

A determinação das notas é realizada por meio de propriedades computadas nos modelos, que promovem os cálculos exigidos no momento de seu acesso. A pontuação final de cada avaliador é calculada por meio da soma dos cinco critérios estabelecidos. As notas finais de cada etapa (NF1 e NF2) são determinadas por meio de características no modelo de TCC que realizam agregações médias nas avaliações pertinentes, assegurando que o cálculo sempre represente o estado atual. A média final é determinada por um processo análogo e é calculada de forma automática com base nas duas notas finais.

Para viabilizar a avaliação em duplo-cega, foi estabelecido um campo de arquivo opcional no modelo da banca que possibilita o *upload* da versão anonimizada da monografia. O serializador implementa uma lógica condicional que verifica se o arquivo existe e retorna apenas ele em vez do documento original na representação de dados para os avaliadores, de modo a ocultar a identificação do aluno e do orientador.

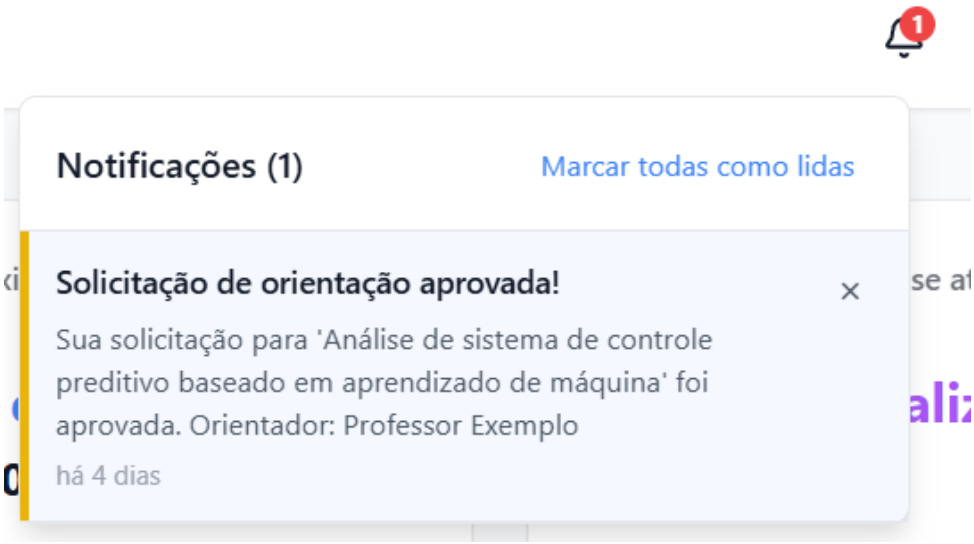
O desfecho do processo é estabelecido automaticamente por meio de um sinal interligado ao modelo de avaliação da segunda etapa. Ao registrar a última avaliação, o sistema verifica se a média final atinge o patamar mínimo exigido para aprovação e atualiza o campo referente ao resultado final no TCC correspondente, essa dinâmica dispensa a necessidade de cálculos ou determinações manuais.

3.2.7 Mecanismos de notificações

O sistema de notificações foi concebido em duas camadas que se complementam e atuam em conjunto, a fim de assegurar que os usuários recebam informações sobre eventos pertinentes. A camada inicial é composta por notificações internas organizadas em um modelo de banco de dados que inclui campos para tipo, título, mensagem, prioridade, URL de ação, estado de leitura e uma referência opcional ao TCC correspondente. Essa configuração possibilita a apresentação de notificações na interface *web*, como apresentado na Figura 19, além de estar diretamente associada aos recursos citados. Com o objetivo de aumentar a eficiência

do desempenho, foram elaborados índices de banco de dados que combinam campos frequentemente acessados, como por exemplo nos casos do destinatário e o estado de leitura. Essa dinâmica possibilita agilizar as pesquisas por notificações não lidas.

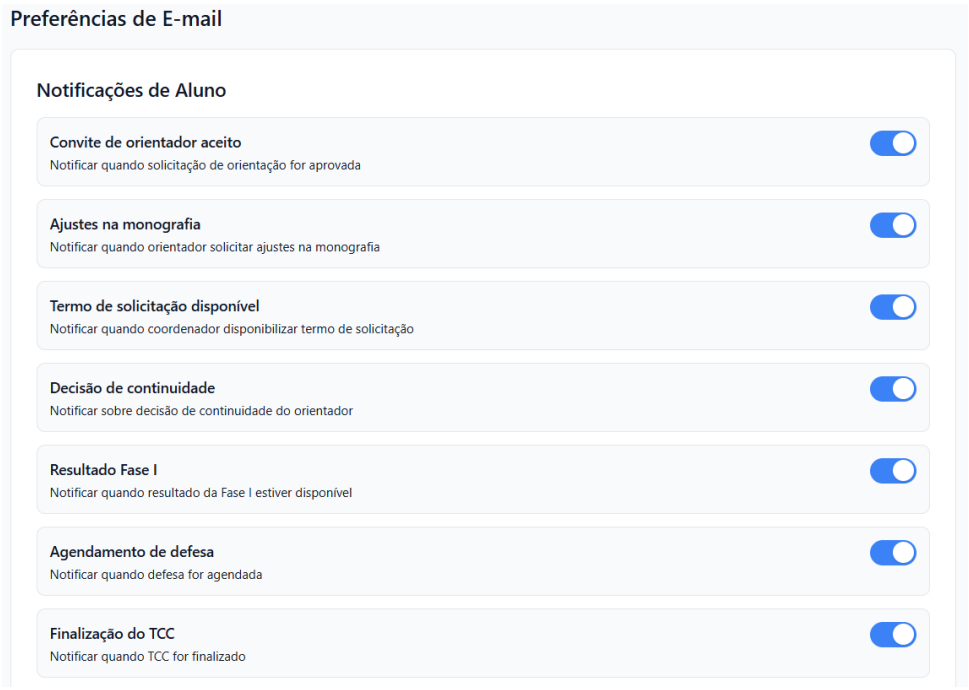
Figura 19 – Exemplo de notificação interna.



Fonte: autor, 2025.

A segunda camada realiza o envio de e-mails por meio da plataforma de correio do Django, empregando configurações SMTP guardadas em um modelo de banco de dados em vez de utilizar arquivos de configuração fixos. Essa metodologia possibilita ao coordenador realizar ajustes nos parâmetros de conexão, como servidor, porta e credenciais, por meio da interface administrativa, dispensando a necessidade de acesso ao código-fonte ou de reestruturação do sistema. Os parâmetros são recuperados de forma dinâmica do banco de dados sempre que há a necessidade de enviar um e-mail.

Figura 20 – Preferências de E-mail do perfil do discente.



Fonte: autor, 2025.

Com o intuito de respeitar as preferências pessoais, foi instituído um modelo de preferências de e-mail, exemplificados na Figura 20, que contém campos booleanos distintos para cada categoria de notificação, categorizados conforme o tipo de usuário. Assim, cada perfil possui alternativas pertinentes às suas necessidades, sem ser inundado com preferências inadequadas. O mecanismo de envio analisa três condições por meio de consultas antes de enviar cada e-mail: a ativação global do envio nas configurações do sistema, a autorização do usuário específico para aquele tipo de notificação em suas preferências e a existência de um endereço de e-mail válido registrado. O e-mail é enviado de forma efetiva somente quando todas as condições são atendidas.

A total automação do sistema é realizada por meio de sinais do Django que estão interligados aos modelos centrais do sistema. Esses sinais acompanham ocorrências como alterações de estado em TCCs, validação de documentos e envio de avaliações. Quando um evento significativo acontece, os receptores de sinal analisam as alterações ao confrontar o estado atual com o anterior e acionam uma camada de serviços encarregada de gerar notificações. A arquitetura fundamentada em eventos garante a consistência de funcionamento devido ao fato de que as notificações são

geradas de forma independente da localização no código em que a ação foi realizada. Esse dinamismo permite eliminar a chance de que notificações sejam omitidas em um determinado fluxo específico.

3.2.8 Gestão de prazos e calendário

O calendário acadêmico foi estabelecido através de um modelo que centraliza a gestão temporal do sistema para possibilitar ao coordenador estabelecer um único calendário por semestre, assim como demonstrado na Figura 21. Os campos de data registram prazos referentes a cada fase do processo, de forma que todos os campos de data são opcionais, de modo a possibilitar uma configuração flexível que se adapte às exigências de cada período letivo.

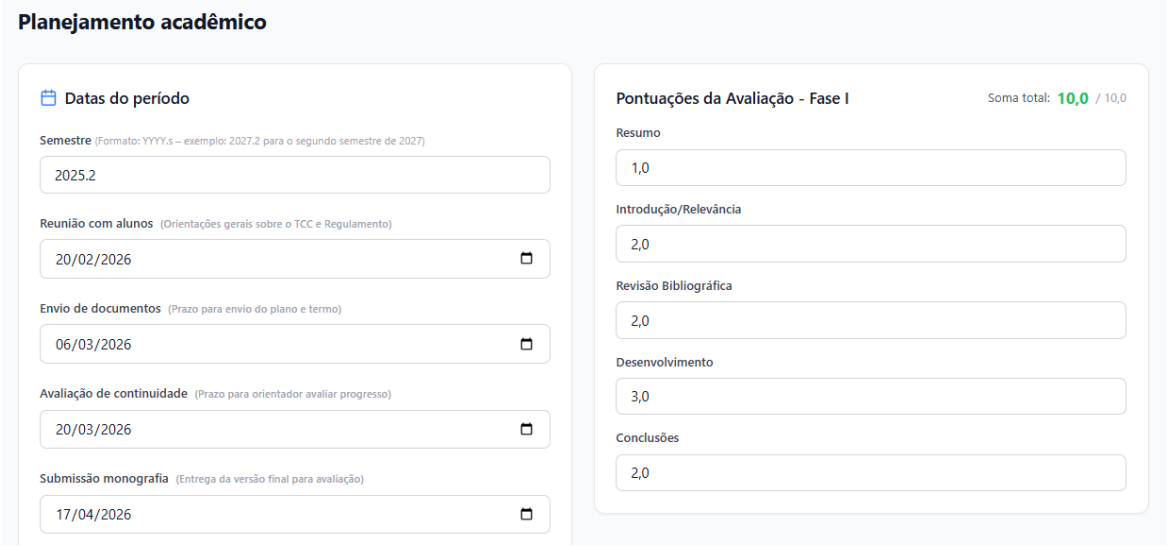
Figura 21 – Cartão de calendário acadêmico interno.

Fonte: autor, 2025.

A fim de assegurar a observância desses prazos, foram introduzidas validações temporais nos serializadores por meio de métodos que verificam o calendário vigente assim como demonstrado na Figura 22 e na Figura 23. Essas validações realizam a comparação entre a data presente e o campo relacionado à operação em curso, bloqueando ações que estejam fora do intervalo autorizado. Por exemplo, ao realizar a validação do envio de uma monografia, o sistema procura o prazo determinado para essa categoria de documento e confere se a tentativa se realiza dentro do intervalo aceitável. Uma verificação suplementar analisa campos de liberação manual no modelo TCC através de uma lógica alternativa, possibilitando que as operações avancem, mesmo fora do prazo estipulado, desde que o coordenador tenha concedido uma exceção individual.

A consistência dos pesos utilizados na avaliação é assegurada através da validação no modelo de calendário. O método de limpeza compila todos os valores de peso de cada etapa e verifica se a soma resultante satisfaz o resultado da soma total ser 10, caso esse requisito não seja cumprido, uma notificação de erro é gerada ao tentar salvar a configuração. Essa verificação é realizada de forma independente para cada etapa e assegura que elas apresentem uma alocação correta de pontos. Os pesos aprovados são empregados nas validações de notas e nos cálculos das notas finais, essa dinâmica permite que o sistema de avaliação cumpra a configuração adequada.

Figura 22 – Configuração de datas e pontuações parte 1.



Fonte: autor, 2025.

Figura 23 – Configuração de datas e pontuações parte 2.

Fonte: autor, 2025.

O sistema de liberação individual acrescenta ao calendário global diversos campos booleanos no modelo TCC, com cada campo representando um tipo particular de prazo. Quando acionados pelo coordenador, esses campos possibilitam que o referido TCC realize operações mesmo além dos prazos estabelecidos no calendário.

As validações nos serializadores estabelecem uma lógica alternativa que analisa tanto o prazo do calendário quanto a situação desses campos de liberação, autorizando a operação caso alguma das condições seja verdadeira. Essa estrutura dual oferece a flexibilidade necessária para situações excepcionais, sem prejudicar o controle temporal abrangente do sistema.

3.3 *Frontend*

3.3.1 Arquitetura de componentes

A estrutura do *frontend* foi elaborada com base nos princípios de componentização do React, de modo a organizar a interface em uma hierarquia de quatro níveis, cada um com responsabilidades gradativamente mais específicas. O primeiro nível abarca componentes essenciais que englobam elementos primordiais, como botões, campos de texto e etiquetas, de forma a assegurar uma consistência visual via variáveis CSS compartilhadas. O segundo nível introduz elementos

pertencentes a domínios específicos, como uma linha do tempo horizontal para a visualização de fases, uma linha do tempo vertical para eventos cronológicos, além de painéis de alertas de prazos que contêm uma lógica de negócio integrada, a qual consulta o calendário acadêmico e realiza o cálculo de estados visuais. O terceiro nível é composto por fluxos complexos que envolvem modais de *upload* com validação, formulários de avaliação que realizam validação dinâmica das notas de acordo com os pesos estabelecidos e o cálculo em tempo real dos totais. O quarto nível compreende *layouts* integrais organizados por perfil, os quais incluem cabeçalho personalizado, menu lateral contextual, painel de notificações e área de conteúdo, tudo conforme o padrão de composição.

A biblioteca é composta por 28 componentes reutilizáveis organizados em um diretório específico, além de contar com 11 componentes de estrutura. Uma linha do tempo horizontal apresenta 14 estados organizados em 7 grupos visuais, com indicadores distintos por meio de colorações, ícones e uma animação pulsante que sinaliza a etapa atual, utilizando um cálculo dinâmico para o posicionamento. A versão vertical apresenta eventos em ordem cronológica e metadados oriundos do campo JSON, de modo a utilizar virtualização para aprimorar a renderização de históricos extensos. Todos os elementos respeitam um padrão de construção através de características de seus descendentes e a aplicação de variáveis CSS personalizadas via atributos de dados no elemento raiz. Essa dinâmica assegura a adaptação automática ao tema escolhido e elimina a necessidade de propagação manual das propriedades.

3.3.2 Roteamento e navegação

A implementação do sistema de roteamento foi realizada por meio do React *Router* DOM na versão 7, englobando cerca de 35 rotas organizadas hierarquicamente em quatro perfis, conforme o contexto funcional. Todas as rotas funcionais são resguardadas por um componente que implementa três camadas de validação sequenciais: a verificação de autenticação, que redireciona para a página de *login* na ausência de *tokens*, a validação do tipo de usuário em relação a uma lista de tipos permitidos, que realiza um redirecionamento contextual para a rota padrão do perfil quando o acesso não é autorizado e a gestão de estados de carregamento, que

apresenta um indicador visual durante a validação do *token* JWT e o carregamento do perfil.

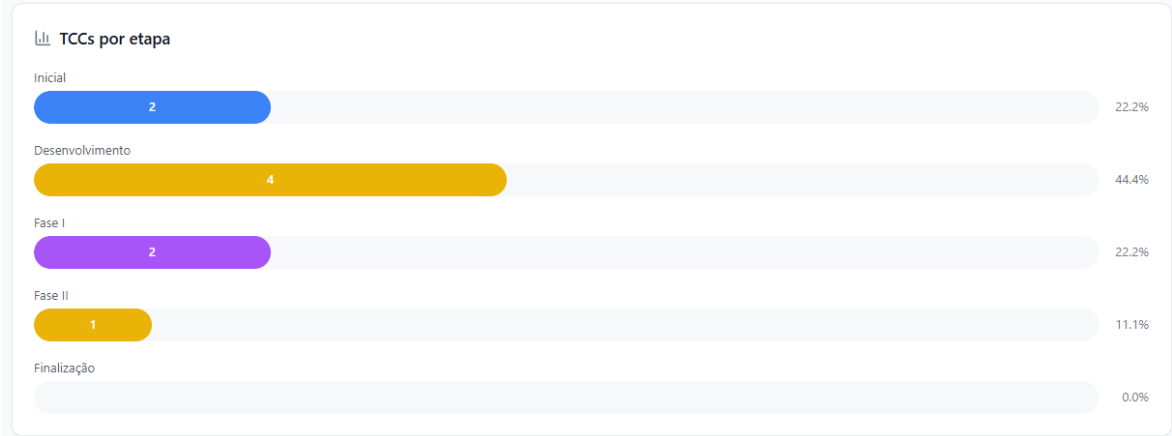
As rotas foram estruturadas com prefixos distintos de acordo com cada perfil: para os alunos, são disponibilizados recursos como *dashboard*, visualização de trabalho e gerenciamento de documentos, para os professores, são oferecidas rotas parametrizadas destinadas a orientandos específicos, coorientações e bancas, os coordenadores empregam caminhos administrativos que abrangem gestão de trabalhos com visualização parametrizada, gerenciamento de solicitações, administração de usuários, os avaliadores restringem-se a um cronograma de avaliações e à submissão de pareceres. Um sistema de contagem numérica de *badges* foi incorporado aos itens do menu, mostrando contadores de ações não concluídas que são atualizados de forma dinâmica por meio de *hooks* personalizados, os quais consultam o *status* de trabalhos, documentos e avaliações. A detecção da rota ativa é realizada com a aplicação de destaque visual por meio de classes CSS condicionais.

3.3.3 Gerenciamento de estado

A gestão de estado foi realizada através da *Context* API *do* React através de contextos específicos para autenticação, notificações e tema. O contexto de autenticação concentra *tokens* JWT, informações integrais do usuário e estados de carregamento, de modo a implementar um fluxo de *login* que realiza uma requisição ao *endpoint* de autenticação, armazena os *tokens* no armazenamento local, obtém o perfil completo (incluindo preferências, sincroniza com o contexto de tema) e redireciona de forma automática. O *logout* realiza a invalidação formal do *token* através de uma solicitação ao *backend*, elimina o armazenamento local, restabelece o estado e efetua o redirecionamento. A persistência entre sessões é estabelecida por meio de um efeito que verifica a existência de *tokens*, realizando a busca automática do perfil durante a montagem. Isso ocorre com uma sincronização bidirecional entre o armazenamento local, o cliente HTTP e o estado do React, por meio de um *callback* registrado que notifica o contexto quando os *tokens* são renovados de forma automática.

O contexto de notificações administra um conjunto de objetos, computa o total de mensagens não lidas por meio de filtragem e implementa o carregamento inicial por meio de requisições simultâneas, além de possibilitar a marcação, de forma individual ou em grupo, como lidas e a exclusão mediante confirmação. A atualização periódica, efetuada através de *polling* a cada 30 segundos, realiza a busca automática por novas notificações, de modo a adotar uma estratégia de atualizações que altera o estado local imediatamente, antes da validação pelo *backend*. O contexto temático se mantém vigente entre quatro alternativas, com tamanhos de fonte organizados em três categorias e três opções de família tipográfica. É organizado implementando a persistência dupla no armazenamento local do navegador e no *backend* através de um *endpoint* específico. Uma aplicação prática emprega atributos de dados HTML no elemento principal e possibilitando a seleção de grupos de variáveis CSS através de seletores de atributo que possui sincronização automática utilizando uma funcionalidade definida para observar alterações e acionar requisições ao *backend*, assim como apresentado no caso da divisão de estados do fluxo de TCCs no painel do coordenador, apresentado na Figura 24.

Figura 24 – Exemplo de divisão de estados do fluxo de TCCs.



Fonte: autor, 2025.

3.3.4 Comunicação com *backend*

A comunicação foi realizada utilizando o Axios, que conta com uma instância personalizada dotada de interceptadores. Isto permite a injeção automática de

autenticação, a renovação automática de tokens expirados e um tratamento padronizado de erros. A instância foi estabelecida com uma URL base através de uma variável de ambiente, possibilitando a troca entre diferentes ambientes sem a necessidade de alteração no código.

O interceptador de requisição insere automaticamente o *token* JWT no cabeçalho de autorização, utilizando o formato *bearer*, antes de cada requisição. O interceptador de resposta estabelece um tratamento avançado para o erro 401, reconhecendo o *token* expirado e dando início ao processo de renovação automática. Esse processo utiliza um sistema de filas, coordenado por meio de uma variável de controle booleana e um *array* de *callbacks*, com o objetivo de evitar que várias requisições simultâneas tentem renovar o *token*.

A função de renovação realiza uma solicitação ao *endpoint* de atualização e envia o *token* de renovação. Ela extrai o novo *token* de acesso da resposta e o armazena localmente. Posteriormente, a função realiza notificação através de um *callback* e reexecuta todas as requisições que estejam enfileiradas. Caso a renovação não seja bem-sucedida devido ao *token* de atualização também ter expirado, realiza-se uma limpeza para eliminar os *tokens* e redirecionar o usuário para a página de *login*. Foi instaurado um tratamento específico para os códigos 403, 404 e 500. O método foi executado contando com uma função auxiliar que padroniza a extração das mensagens de erro. A estruturação do código foi realizada através de módulos de serviço especializados, de modo que cada módulo encapsula requisições pertinentes a um domínio específico e exporta funções que devolvem dados tipados.

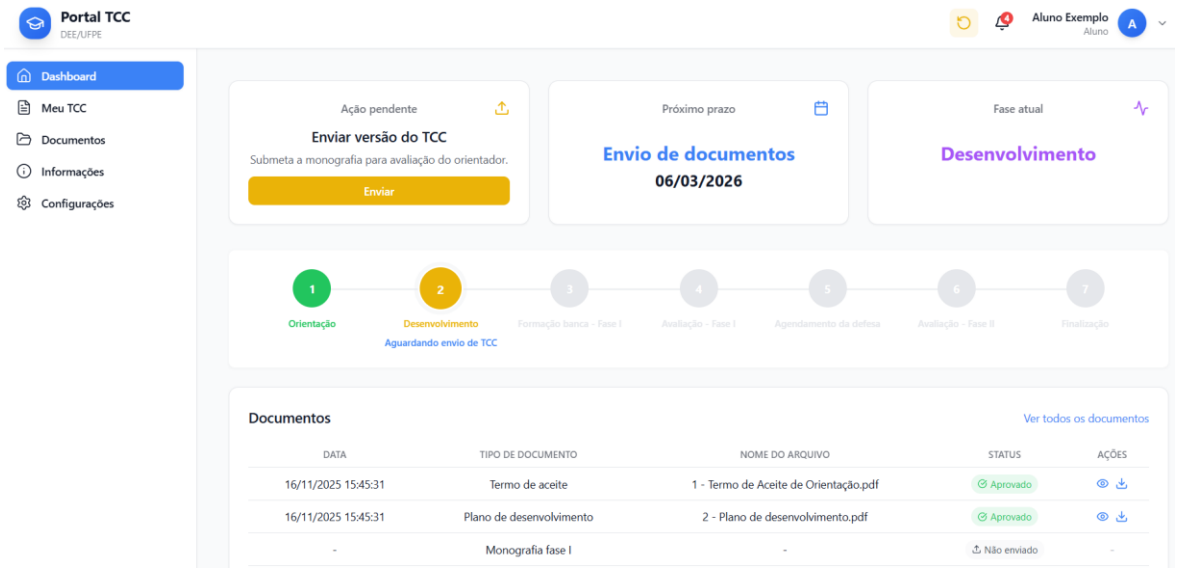
3.3.5 Funcionalidades por perfil

A implementação foi rigorosamente distribuída por perfil, assegurando que cada interface exiba unicamente funcionalidades consideradas estritamente pertinentes.

A interface do aluno incorpora um painel composto por cartões informativos que apresentam o *status* atual, o orientador, datas relevantes e notas, quando disponíveis, assim como representado na Figura 25. Também inclui uma linha do tempo horizontal, com a etapa atual salientada por meio de uma animação pulsante, além de um componente de alerta de prazo que calcula os dias restantes e aplica níveis de

urgência, diferenciados por meio de cores. A seção dedicada a ações rápidas exibe botões contextuais que surgem de forma dinâmica de acordo com o estado atual, enquanto a página de documentos incorpora uma listagem que permite a aplicação de filtros por tipo, indicadores visuais de *status* por meio de *badges* coloridos, visualização *inline* do *feedback* do orientador e um sistema para *download*.

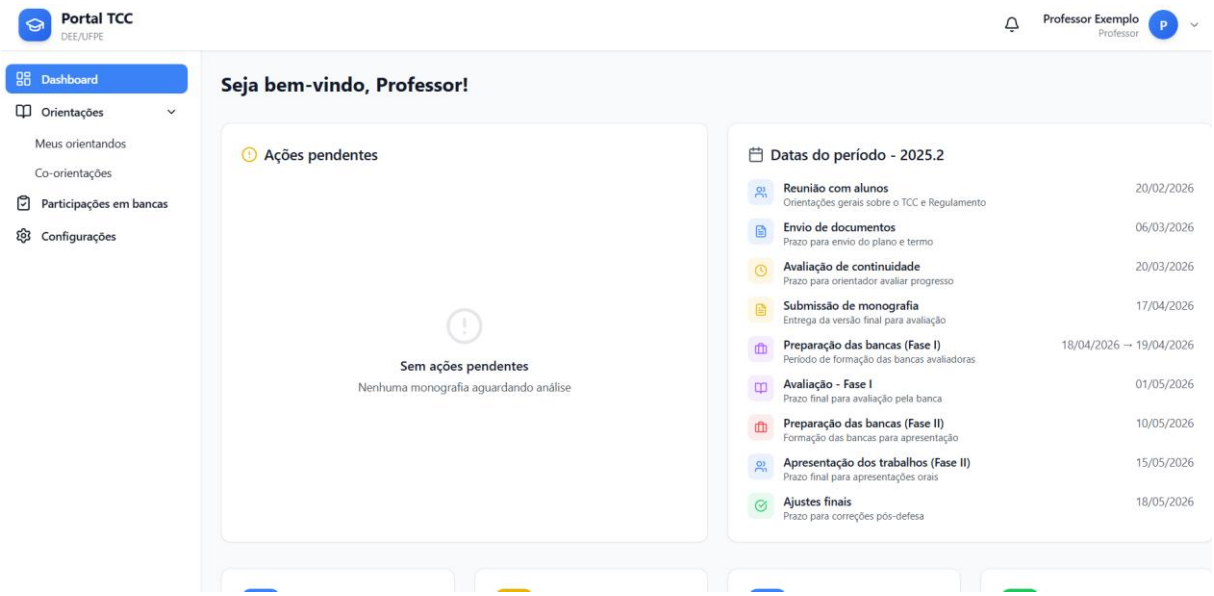
Figura 25 – Visão de tela principal de funcionalidades do perfil do aluno.



Fonte: autor, 2025.

A interface destinada aos professores agrega orientações, co-orientações e bancas por meio de um painel, demonstrado na Figura 26, que apresenta solicitações pendentes em cartões expansíveis, os quais incluem ações diretas para aceitar ou recusar, sendo obrigatória a apresentação de uma justificativa. A relação de orientandos apresenta cartões expansíveis que mostram informações resumidas, acompanhados de indicadores numéricos de ações pendentes, enquanto a página de detalhes estrutura o conteúdo em abas que são carregadas sob demanda, contendo visão geral, documentos, linha do tempo e avaliações. O sistema de aprovação para continuidade exige a confirmação através de um *modal* e valida a presença de uma monografia aprovada como pré-requisito, ao mesmo tempo que a liberação para avaliação verifica o sinalizador de continuidade e a monografia aprovada.

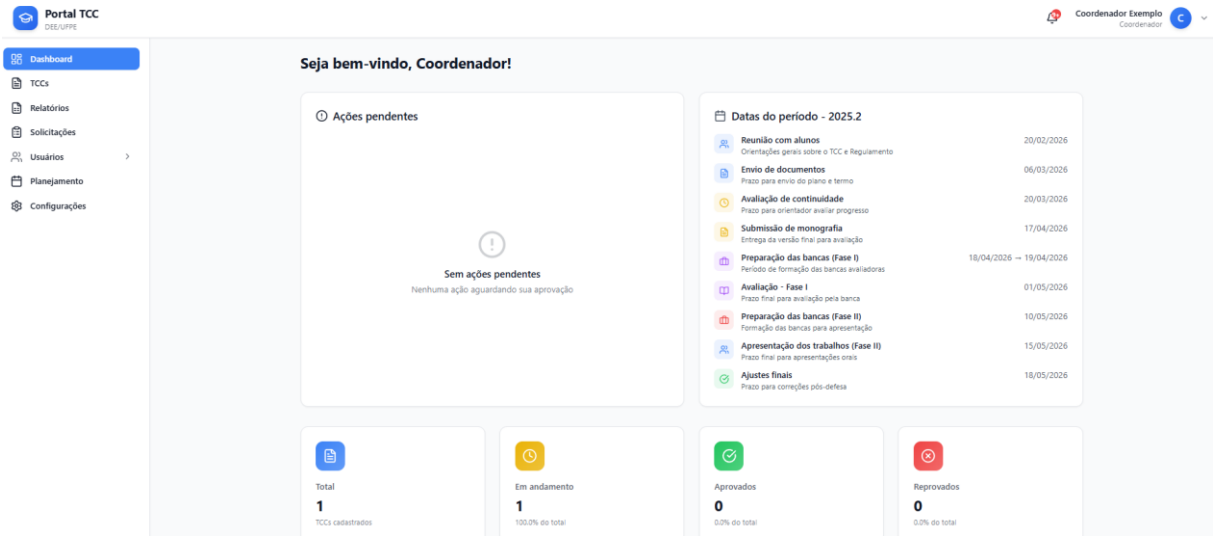
Figura 26 – Visão de tela principal de funcionalidades do perfil do professor.



Fonte: autor, 2025.

A interface do coordenador disponibiliza um painel administrativo abrangente, que apresenta estatísticas calculadas em tempo real que englobam trabalhos por etapa, distribuição de orientações e taxa de aprovação, assim como ilustrado na Figura 27. Além disso, inclui também visualizações gráficas através de componentes personalizados fundamentados em elementos HTML estilizados. A relação de trabalhos estabelece um sistema de filtros segmentados por etapas, orientador e semestre. A página de detalhes exibe seções estruturadas que contêm informações básicas que podem ser editadas, uma grade de caixas de seleção para liberações manuais acompanhada de um *modal* de confirmação que requer a inserção de senha administrativa, além de um módulo destinado à formação de banca, que apresenta um combobox para busca, validação da composição e um campo opcional para o *upload* de versão anonimizada, com funcionalidade de arrastar e soltar. O planejamento introduz um formulário destinado ao calendário acadêmico, que conta com validações de natureza cronológica e exige a soma exata de 10,0 pontos para os pesos dos critérios.

Figura 27 – Visão de tela principal de funcionalidades do perfil do coordenador.



Fonte: autor, 2025.

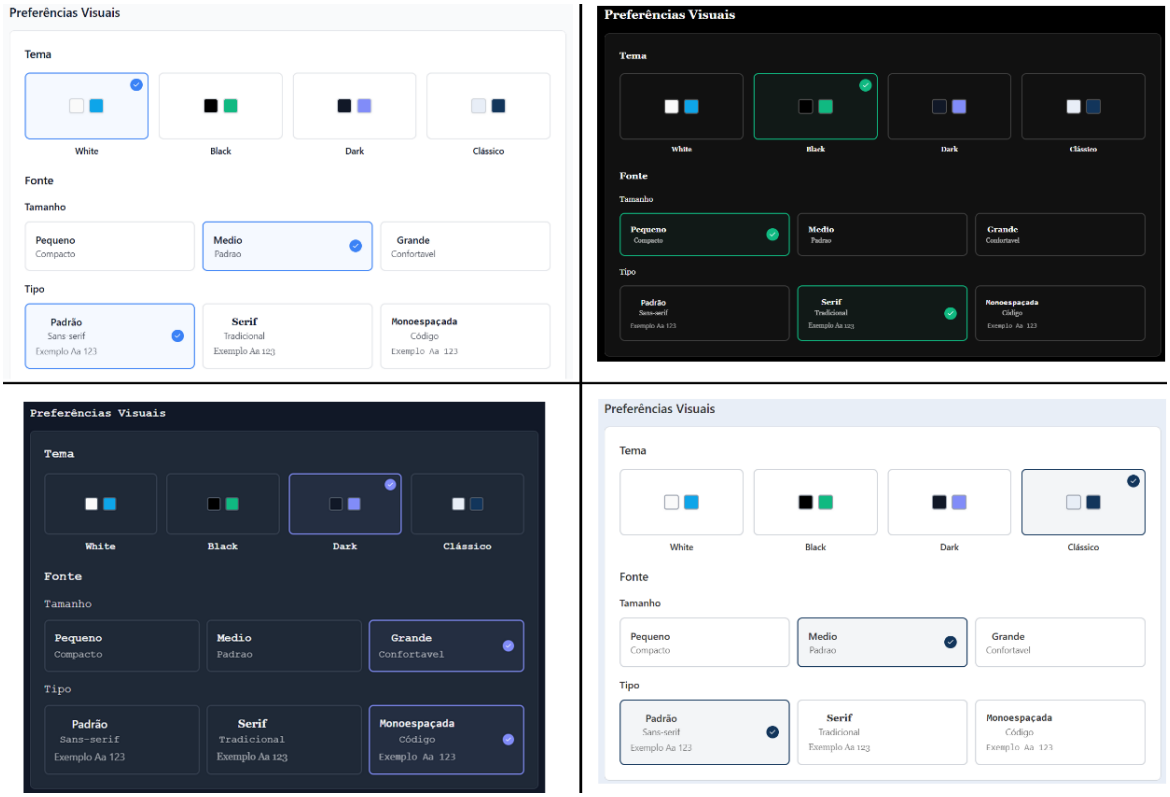
A interface destinada ao avaliador externo foi elaborada com uma abordagem minimalista, de modo a apresentar as bancas através de cartões que exibem o título, o nome do aluno ou uma indicação de anonimato, a fase, o prazo e o *status*. O formulário de avaliação contém cinco campos numéricos, cada um com pesos máximos extraídos do calendário e dispõe de uma validação dinâmica em tempo real que restringe entradas que ultrapassem os limites estabelecidos. Além disso, inclui um campo destinado a pareceres textuais e realiza o cálculo automático da nota total, que é apresentado de forma destacada. O sistema de rascunho realiza o salvamento automático em períodos regulares e assegura a preservação do progresso parcial. Ao clicar no botão de envio final, um *modal* de confirmação é exibido e informa sobre a impossibilidade de realizar edições posteriores, além de realizar a desabilitação o formulário após a submissão.

3.3.6 Sistema de temas visuais

O mecanismo de temas foi desenvolvido utilizando variáveis CSS personalizadas que contém atributos de dados HTML no elemento raiz, de forma a possibilitar a troca dinâmica e integral dos esquemas de cores sem a necessidade de recarregamento. Foram implementados os temas demonstrados na Figura 28: claro (com fundo branco puro e tonalidades suaves), escuro (com fundo preto puro e

elevado contraste), *dark* (com fundo cinza escuro e contraste médio) e um clássico (inspirado em interfaces acadêmicas tradicionais). Cada tema estabelece valores para 23 variáveis CSS dispostas em categorias semânticas, de modo a incluir fundos, textos com hierarquia, bordas, estados e gradientes de marca.

Figura 28 – Visão de tela principal de funcionalidades do perfil do coordenador.



Fonte: autor, 2025.

O *framework* de CSS utilitário foi ajustado para vincular variáveis personalizadas a classes utilitárias, possibilitando a utilização direta em atributos de classe que se ajustam automaticamente ao tema em uso. O sistema permite personalização adicional do tamanho da fonte, disponibilizando três níveis que ajustam o tamanho base e a escala tipográfica, além da família de fontes, com três opções que permitem a modificação da fonte em toda a interface. O contexto estabelece uma persistência tripla, assegurando a sincronização: armazenamento local para aplicação imediata, carregamento automático das preferências armazenadas no *backend* após o *login* e sincronização ativa de alterações por meio de requisições sempre que o usuário

modifica uma opção, implementando uma estratégia otimista com atualização imediata da interface e requisições em segundo plano.

3.3.7 Responsividade

A interface foi elaborada com uma abordagem *mobile-first*, utilizando um sistema de *breakpoints* que assegura uma experiência apropriada em dispositivos que variam de 320 pixels a 2560 pixels ou mais. Estilos base adotam um *viewport* reduzido como referência inicial, aplicando ajustes progressivos por meio de prefixos de *breakpoint* que ativam estilos apenas acima de certos limites, assegurando uma interface utilizável em dispositivos móveis por padrão, à medida que aprimoramentos são adicionados gradualmente.

Os componentes de estrutura realizam adaptações sincronizadas: o menu lateral se transforma em um ícone de hambúrguer em telas estreitas, sendo possível abrir um painel sobreposto, o cabeçalho é reduzido, de forma a preservar unicamente as informações essenciais e a área de conteúdo ajustado o espaçamento interno. Componentes de conteúdo desenvolvem responsividade através de diversas técnicas como tabelas com rolagem horizontal em dispositivos de tela reduzida e alteram para uma exibição completa em telas maiores, cartões que utilizam grade responsiva e adaptam automaticamente o número de colunas e formulários que ajustam seu *layout* de uma única coluna vertical em dispositivos móveis para múltiplas colunas em desktops. As otimizações voltadas para dispositivos com touchscreen englobam uma área mínima de toque de 44 por 44 pixels, estados ativos que complementam o efeito de *hover* convencional e menus *dropdown* ajustados para exibição em tela cheia em dispositivos móveis.

3.3.8 Validações e *feedback* ao usuário

Foi instituído um sistema abrangente de validações, estruturado em diversas camadas. As validações realizadas no lado do cliente em formulários abarcam a presença de campos obrigatórios, que são sinalizados por um asterisco e impedem a submissão quando não preenchidos, campos numéricos que validam tanto o formato

quanto o intervalo, campos de e-mail que asseguram a conformidade através de expressões regulares, além de formulários de avaliação que implementam uma validação dinâmica, monitorando se os valores não ultrapassam os limites máximos preestabelecidos em tempo real, conforme o usuário digita.

O sistema de notificações temporárias coordena uma sequência de *toasts* que surgem no canto superior direito, com uma animação de entrada e permanecendo visíveis por três segundos, de forma a se extinguir através de uma animação de saída. O sistema apresenta quatro variantes visuais, cada uma com cores e ícones específicos:

- Sucesso: fundo verde
- Erro: fundo vermelho
- Alerta: fundo amarelo
- Informação: fundo azul

Ações prejudiciais demandam validação por meio de modais ajustáveis, os quais modificam ícones e paletas de cores, entrando em um estado de carregamento durante a execução que inibe botões e apresenta um indicador de progresso. Componentes de esqueleto foram desenvolvidos para representar estados de carregamento, exibindo *placeholders* animados com gradiente pulsante. Os indicadores de progresso incluem uma linha do tempo com uma barra visual que conecta etapas finalizadas e distintivos numéricos que utilizam cores para diferenciar os níveis de urgência.

3.4 Banco de dados

O banco de dados foi elaborado por meio do sistema de mapeamento objeto-relacional do Django, o qual converte automaticamente classes em Python para tabelas de banco de dados, assim como transforma os relacionamentos em chaves estrangeiras. A configuração obtida consiste em 16 modelos fundamentais dispostos nos quatro módulos da aplicação, sendo que cada um representa uma tabela no banco de dados. A implementação e uso do SQLite durante a fase de desenvolvimento e produção justifica-se em razão da simplicidade do banco de dados, porém o sistema

oferece suporte a migração ao PostgreSQL através de uma configuração que possibilita a troca sem necessidade de modificação no código.

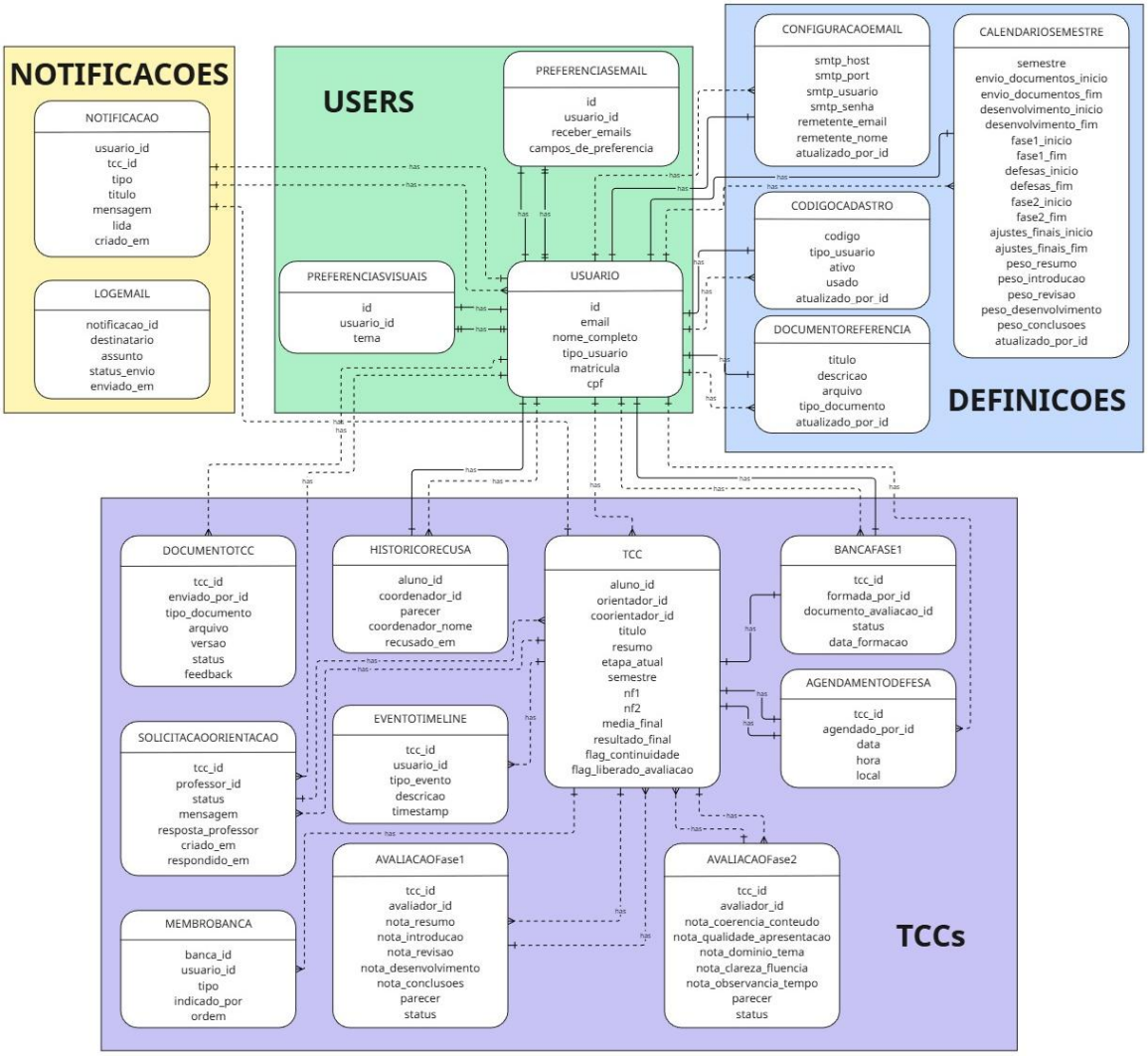
3.4.1 Modelo Entidade-Relacionamento

A organização da estrutura foi realizada através do agrupamento dos 16 modelos nos módulos funcionais, esquematizados na Figura 29, de modo a realizar a divisão de responsabilidades também na camada de dados.

A classe Usuário amplia as classes fundamentais de autenticação do *framework*, estabelecendo o e-mail como o campo de identificação exclusivo, em substituição ao tradicional nome de usuário. Essa escolha exprime o cenário acadêmico em que os e-mails institucionais funcionam como identificadores inerentes. O campo denominado "tipo de usuário" retém um dos quatro tipos possíveis através de um campo de texto com opções limitadas, sendo essencial para o funcionamento do sistema de permissões. A implementação adota a herança em uma tabela única para realizar a centralização de todos os tipos em um único banco de dados. Essa abordagem facilita consultas que englobam diversos tipos, além de prevenir junções desnecessárias.

Para complementar o modelo principal de usuário, foram desenvolvidos modelos de preferências que estabelecem relacionamentos um-para-um. A normalização mencionada dissocia informações de configuração da tabela principal, tornando mais simples a inclusão de preferências no futuro, sem a necessidade de alterar a estrutura central. A geração desses modelos auxiliares é realizada de forma automatizada através de sinais vinculados ao evento de criação de usuário, essa dinâmica assegura que cada usuário tenha suas preferências configuradas com valores padrão adequados. O modelo de TCC serve como o núcleo do sistema e estabelece vínculos de chave estrangeira para três tipos de usuários com funções distintas: aluno, orientador e coorientador. O campo que representa o estado atual armazena texto proveniente do módulo de constantes, enquanto diversos campos de controle, configurados como booleanos, administram as *flags* do fluxo. Os resultados das avaliações são guardados em campos decimais, os quais aceitam valores nulos até que sejam efetivamente calculados.

Figura 29 – Diagrama ER da aplicação.



Fonte: autor, 2025.

Entidades complementares estabelecem relacionamentos de chave estrangeira com o TCC, configurando composições que representam diferentes aspectos do processo. As solicitações de orientação guardam convites de alunos dirigidos aos professores, os documentos armazenam todos os arquivos junto a metadados de versionamento, os eventos registram, de forma cronológica, todas as ações relevantes, o histórico conserva pareceres referentes a trabalhos rejeitados e os agendamentos guardam dados acerca das apresentações. Todos esses modelos estão interconectados ao TCC por meio de uma chave estrangeira que possibilita a navegação em ambas as direções.

O modelo de banca estabelece um relacionamento um-para-um com o TCC, de modo a assegurar que cada trabalho tenha apenas uma banca, além de um campo opcional para o arquivo do documento anonimizado utilizado na avaliação em duplo-cego. Os membros da banca são geridos através de um modelo específico que determina as interações entre a banca e o usuário, apresentando uma limitação de unicidade na combinação para evitar a inserção repetida do mesmo docente. Os modelos de avaliação para cada fase criam conexões entre o TCC e o avaliador, além de impor uma limitação de exclusividade na combinação, garantindo que cada avaliador execute apenas uma avaliação por etapa. Os campos de nota são configurados como decimais e possuem validadores que proíbem a inserção de valores negativos, a medida que o campo de *status* gerencia o ciclo de vida da avaliação, com estados como pendente, enviado e bloqueado.

O modelo de calendário concentra toda a administração do tempo por meio de campos de data facultativos, permitindo flexibilidade, além de incluir campos decimais para avaliações que totalizam obrigatoriamente 10,0. Diversos modelos de configuração retêm códigos de cadastro organizados conforme o tipo de usuário, documentos de referência disponíveis para *download* e parâmetros SMTP destinados ao envio de e-mail. O modelo de notificação estabelece vínculos para o destinatário e um TCC opcional, incluindo índices estratégicos que visam à otimização de consultas recorrentes, como a busca por notificações não lidas.

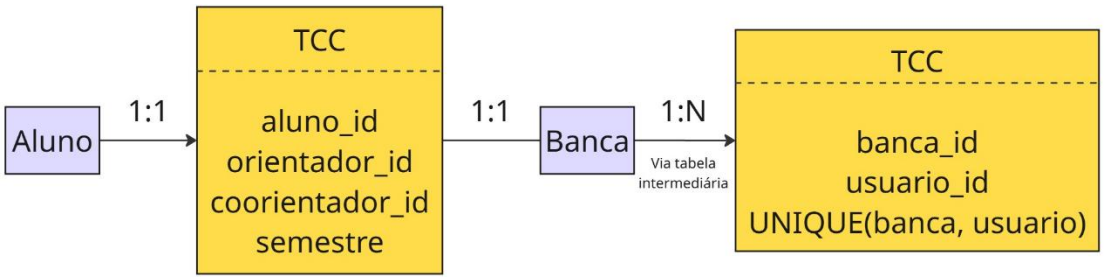
3.4.2 Relacionamentos

As interações entre os modelos foram estabelecidas por meio de chaves estrangeiras e relacionamentos um-para-um do ORM, cada um ajustado com uma estratégia de deleção adequada, que equilibra a integridade referencial à preservação histórica. O modelo de TCC ilustra essa abordagem dual de deleção: o vínculo com o aluno adota a deleção em cascata, assegurando a eliminação total dos trabalhos quando o discente é retirado do sistema. Enquanto isso, os relacionamentos com o orientador e coorientador adotam a configuração de valor nulo, o que permite a preservação do registro do TCC mesmo após a remoção do docente, mantendo, dessa forma, a integridade histórica dos registros acadêmicos. Adicionalmente a esses relacionamentos, limitações impõem restrições às escolhas disponíveis com

base no tipo de usuário, assegurando, por exemplo, que apenas docentes possam ser selecionados na função de orientadores. Uma imposição de exclusividade entre aluno e semestre proíbe que um estudante tenha mais de um TCC no mesmo período acadêmico.

O sistema de bancas ilustra composições mais sofisticadas por meio de diversos níveis de inter-relações. O modelo de banca estabelece um relacionamento um-para-um com o TCC, assim como ilustrado na Figura 30, empregando a exclusão em cascata para uma composição rigorosa, assegurando que os ciclos de vida da banca e do TCC permaneçam interligados. Os integrantes da banca são administrados por meio de um modelo intermediário que estabelece vínculos com deleção em cascata em ambas as direções – tanto para a banca quanto para o usuário – configurando uma estrutura de relacionamento muitos-para-muitos que é gerida manualmente. A limitação de unicidade estabelecida entre a banca e os membros impede que um mesmo docente seja alocado repetidamente na mesma banca.

Figura 30 – Diagrama relacional de TCC e bancas avaliadoras.



Fonte: autor, 2025.

Os modelos de avaliação adotam uma estratégia mista de exclusão: a exclusão em cascata é utilizada para o relacionamento com o TCC, vinculando o ciclo de vida da avaliação ao trabalho, enquanto a configuração de valor nulo no relacionamento com o avaliador possibilita a preservação das avaliações históricas, mesmo após a remoção do docente do sistema. A limitação de unicidade associada entre o TCC e o avaliador assegura que cada docente ofereça, no máximo, uma avaliação por trabalho em cada etapa.

Com o intuito de aprimorar o desempenho, foram elaborados índices de maneira estratégica em campos e combinações que são comumente empregados nas

consultas. Índices simples sinalizam campos isolados, como destinatário, em notificações, ao passo que índices compostos aprimoram buscas em vários campos, como destinatário e estado de leitura, de forma simultânea. Esses índices aceleram de maneira significativa operações rotineiras, como a localização de notificações não lidas de um usuário ou a busca por eventos relacionados a um TCC específico.

Foram estabelecidos campos de auditoria em modelos de configuração para monitoramento de modificações. Os vínculos para o usuário que efetuou a modificação utilizam uma configuração com valor nulo, possibilitando ao sistema registrar a identidade daqueles que efetuaram alterações, sem a necessidade de que esses usuários permaneçam no sistema de forma indefinida. Os campos de data com atualização automática registram o momento em que ocorreu cada alteração. Todos os vínculos foram estabelecidos com denominações descritivas em ordem inversa, o que facilita a navegação em ambas as direções, possibilitando o acesso tanto aos objetos correlacionados quanto ao objeto de origem por meio de consultas diretas.

3.4.3 Modelo de usuários customizado

O modelo de usuário foi desenvolvido através da extensão das classes fundamentais de autenticação do Django, de modo a possibilitar a personalização total dos campos e das funcionalidades relacionadas à autenticação. No sistema, o e-mail é tratado como o campo de identificação padrão ao invés do campo nome de usuário convencional do *framework*. Essa opção se alinha ao contexto acadêmico, no qual e-mails institucionais funcionam como identificadores únicos naturais. O campo de e-mail apresenta uma restrição de unicidade, a qual assegura que os valores sejam únicos no nível do banco de dados por meio de uma *constraint*, evitando assim a criação de cadastros duplicados.

A implementação abrange um gerenciador personalizado que amplia o gerenciador padrão de usuários do Django, sobrescrevendo os métodos de criação com o intuito de assegurar que as senhas sejam sempre armazenadas de maneira segura. O procedimento para a criação de usuários regulares aplica um *hash* de forma automática à senha fornecida antes do armazenamento, enquanto o processo para a criação de superusuários inclui a configuração automática de permissões

administrativas e determina o tipo como coordenador, estabelecendo que os administradores do sistema são invariavelmente coordenadores.

O campo denominado tipo de usuário institui um armazenamento em formato de texto, com opções limitadas estabelecidas no próprio modelo, sendo essencial para a totalidade do sistema de permissões e filtragem de dados. Com o objetivo de assegurar a consistência das informações, o método de limpeza aplica validações condicionais que se baseiam no tipo de usuário: caso o tipo seja aluno, é imprescindível que o campo referente ao curso não esteja desprovido de informações. Se o tipo corresponder a professor ou coordenador, a inserção do campo departamento se torna obrigatória. Se o tipo for avaliador externo, a afiliação institucional deve ser disponibilizada. Essas validações são realizadas antes da gravação no banco de dados, assegurando a integridade desde o momento da criação.

Com o objetivo de otimizar a administração e as consultas, houve a implementação de modelos *proxy* específicos para cada categoria de usuário. Esses modelos não geram tabelas extras no banco de dados, de modo a atuarem como visualizações restritas da tabela de usuários. Cada *proxy* implementa um gerenciador personalizado, com dinâmica de sobreposição do método de consulta para realizar automaticamente a filtragem por tipo, isto possibilita que os administradores operem com interfaces distintas para cada tipo de usuário, sem a necessidade de sempre indicar o filtro manualmente.

Os modelos de preferências estabelecem vínculos unidimensionais com o modelo de usuário, sendo gerados de forma automática por meio de indícios fornecidos pelo Django. Os receptores de sinal vinculados ao evento de criação de usuário identificam o cadastro de novos usuários e geram automaticamente os registros de preferências correspondentes. Essa automação assegura que cada usuário tenha suas preferências sempre ajustadas e dispensa a necessidade de verificações de presença em todo o código que emprega tais configurações.

3.5 Segurança

A segurança do sistema foi estabelecida por meio de diversas camadas de proteção, em conformidade com os princípios da defesa em profundidade e as diretrizes sugeridas pela OWASP. A arquitetura inclui um sistema de autenticação sólida fundamentada em *tokens*, autorização detalhada através do controle de acesso baseado em papéis, validação de dados em diversas camadas, proteção contra fragilidades recorrentes e configurações de comunicação entre cliente e servidor.

3.5.1 Segurança de autenticação

A implementação do sistema de autenticação foi realizada através de JWT e assegura uma autenticação sem estado, adequada para uma arquitetura com *frontend* dissociado. A configuração determina políticas restritivas para expiração e renovação, de modo a reduzir a janela de vulnerabilidade em situações de comprometimento.

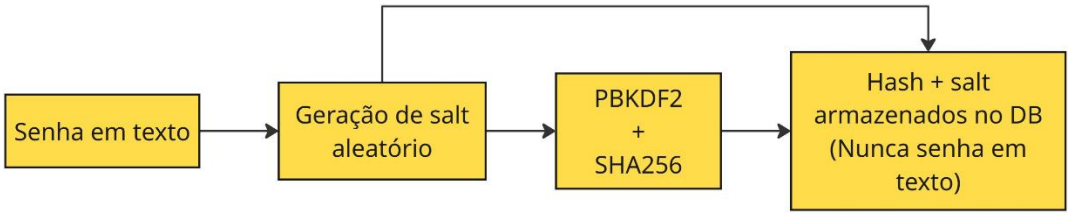
A arquitetura adota dois tipos de *tokens* que possuem ciclos de vida diferentes: *tokens* de acesso, com duração de 60 minutos, os quais são inseridos no cabeçalho de autorização de cada requisição autenticada e *tokens* de atualização, que têm validade de 7 dias e são utilizados unicamente para a geração de novos *tokens* de acesso. O sistema realiza a rotação automática por meio de uma configuração específica, na qual cada utilização para obter um novo *token* resulta na emissão de um novo *token* de atualização e na invalidação automática do anterior, o que diminui a janela de exploração caso o *token* seja interceptado. A configuração do algoritmo emprega HS256 para a assinatura criptográfica, assegurando a integridade e a autenticidade por meio de uma chave secreta compartilhada.

Foi implantada uma lista de revogação, armazenada em um banco de dados, utilizando um modelo específico. Ao realizar o *logout* por meio de um *endpoint* específico, o token de atualização é inserido em uma lista que invalida tanto o token de atualização quanto os tokens de acesso associados, inibindo a reutilização após um *logout* explícito. Verificações são realizadas de forma automática através de *middleware* em cada tentativa de uso, de modo a realizar consulta a uma lista e rejeitar tokens que tenham sido explicitamente invalidados. A rotação automática auxilia esse

mecanismo ao incluir o token anterior à lista sempre que um novo é gerado, estabelecendo uma camada adicional de segurança.

As senhas dos usuários são armazenadas por meio da função de derivação de chave baseada em senha PBKDF2, utilizando o algoritmo de *hash* SHA256, que é implementada como padrão pelo *framework* Django. Esse método é exemplificado na Figura 31 e aplica um *hash* iterativo com um *salt* gerado aleatoriamente para cada senha, o que previne *Rainbow Table Attack* (ataque criptográfico usado para descobrir senhas a partir de hashes). A configuração abrange quatro validadores que são aplicados tanto na criação quanto na alteração das senhas: o validador de similaridade com atributos impede a utilização de senhas que derivem de nome ou e-mail, o validador de comprimento assegura uma complexidade básica, o validador de senhas comuns refuta aquelas que constam em listas públicas de credenciais comprometidas e o validador numérico impede senhas que sejam compostas unicamente por dígitos. As credenciais de serviço SMTP são codificadas em *base64* antes de serem armazenadas por meio de métodos personalizados no modelo de configuração, o que impede a exposição direta em *dumps* de banco de dados ou interfaces administrativas. Adicionalmente, a chave secreta é armazenada em uma variável de ambiente.

Figura 31 – Processo de hashing de senhas com PBKDF2 e salt.



Fonte: autor, 2025.

3.5.2 Segurança de autorização

O sistema de autorização estabelece um controle de acesso fundamentado em papéis, por meio de permissões personalizadas que ampliam a estrutura básica do

DRF, assegurando que os usuários tenham acesso apenas aos recursos para os quais possuem autorização explícita.

Nove classes de permissão customizadas foram implementadas, cada uma responsável por validar aspectos específicos por meio de métodos que conferem autenticação e relacionamentos. As permissões estabelecem uma lógica hierárquica por meio de verificações condicionais: coordenadores detêm acesso administrativo total, alunos têm acesso restrito a seus próprios recursos mediante a comparação de identificadores, professores podem acessar recursos nos quais atuam como orientadores ou coorientadores por meio de consultas aos relacionamentos e avaliadores têm acesso apenas às avaliações que lhes foram atribuídas, por meio da verificação dos membros da banca. O sistema de controle de visibilidade estabelece três camadas para eventos na linha do tempo, por meio de um campo enumerado: acessíveis a todos os participantes, acessíveis apenas ao orientador e ao coordenador, ou acessíveis exclusivamente ao coordenador. Esta abordagem possibilita a ocultação de informações sensíveis, como validações internas, de usuários que não possuem autorização.

Todas as operações analisam a propriedade antes de permitir a execução, por meio de métodos de permissão em nível de objeto, assegurando que apenas os proprietários, os usuários associados ou o coordenador possam acessar recursos determinados. O sistema distingue as permissões de leitura e escrita por meio da verificação do método HTTP. Ademais, as consultas ao banco de dados são filtradas através de métodos sobrescritos nas coleções de visualizações, com a dinâmica da aplicação de filtros fundamentados no tipo de usuário antes de devolver os resultados, o que impede a enumeração decorrente de tentativas de acesso através de identificadores.

O sistema estabelece permissões distintas para cada operação por meio de um decorador de ação personalizado aplicado a métodos específicos. As operações críticas, como a aprovação de continuidade, a liberação para avaliação e a formação de bancas, incluem validações que restringem o acesso exclusivamente a perfis autorizados. A validação da autorização ocorre em três níveis complementares: a configuração global assegura que os *endpoints* rejeitem requisições não autenticadas, as permissões específicas validam o tipo de usuário junto com o relacionamento e a

lógica de negócio realiza validações adicionais que verificam o estado do TCC, a conformidade com prazos e os valores das *flags* de controle.

3.5.3 Segurança de dados

A segurança da informação adota validações em diversas camadas e medidas de prevenção contra tipos frequentes de ataques, assegurando a integridade dos dados que são armazenados e processados.

As informações são validadas em três camadas distintas e independentes. No *frontend*, os componentes realizam a validação do lado do cliente. São analisados os critérios de formato, tamanho e regras fundamentais. No *backend* do sistema, a camada de serialização realiza uma segunda validação, que analisa tipos, formatos, comprimentos e regras complexas, incluindo a verificação se o aluno possui outro TCC em andamento por meio de consulta ao banco de dados, se as notas não ultrapassam os pesos estabelecidos e se as extensões estão de acordo com o tipo de documento. Os serializadores implementam campos específicos em vez de permitir a aceitação de todos os campos do modelo, o que ajuda a prevenir ataques de atribuição em massa, nos quais um invasor poderia alterar campos que não deveriam ser modificados. A terceira camada é implementada nos modelos por meio de métodos de limpeza e validadores de campo, assegurando que os requisitos sejam cumpridos, mesmo que as camadas anteriores sejam contornadas. Isso inclui a verificação da unicidade por meio de restrições em nível de banco e a consistência de dados correlacionados.

O mapeamento objeto-relacional produz, de forma automática, consultas SQL parametrizadas por meio de uma API de consultas, desvinculando completamente o código SQL dos dados fornecidos pelos usuários. Todas as interações empregam técnicas de ORM ou consultas brutas parametrizadas, evitando, em qualquer circunstância, a concatenação direta, assegurando que a entrada seja sempre considerada como dado e não como código executável, mesmo na presença de caracteres especiais.

A transferência de arquivos realiza uma validação rigorosa em diversas camadas. A validação de extensões assegura que somente arquivos com extensões

autorizadas sejam aceitos por meio de um validador específico, permitindo a inclusão de PDFs para monografias e de arquivos DOC/DOCX para outras categorias, evitando o *upload* de arquivos executáveis. A validação de tamanho estabelece um teto de 10 megabytes através de uma função personalizada, a fim de impedir ataques de negação de serviço. A validação do tipo MIME complementa a verificação da extensão ao realizar uma análise do cabeçalho binário, em vez de confiar apenas no nome apresentado, evitando o contorno dessa segurança por meio de renomeação. O sistema cria nomes de arquivos a partir de identificadores únicos universais, utilizando a biblioteca UUID, em vez de empregar nomes fornecidos, o que impede *Path Traversal* (vulnerabilidade de segurança que permite acessar arquivos e diretórios fora do caminho permitido pelo sistema). Uma função personalizada elabora uma estrutura hierárquica utilizando exclusivamente o identificador de TCC e o tipo de documento regulados pela aplicação, os quais são concatenados com um UUID gerado, juntamente com metadados que incluem o nome original, armazenado em campos distintos.

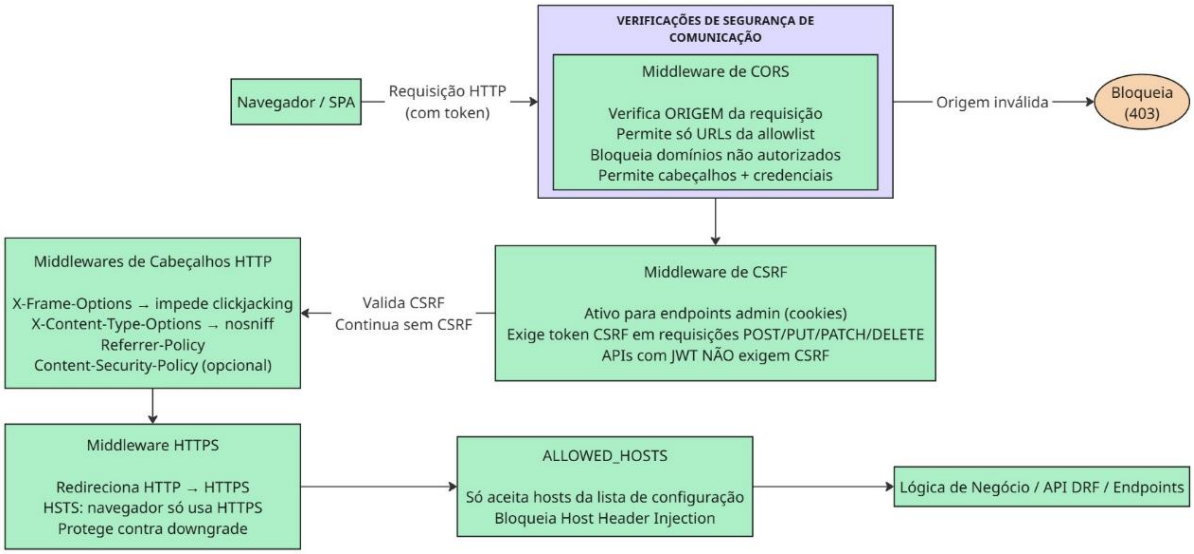
3.5.4 Segurança de comunicação

A comunicação estabelece medidas de proteção contra ataques de origem cruzada e falsificação de requisições por meio da configuração adequada de políticas e da utilização de middlewares de segurança, como ilustrado na Figura 32.

O compartilhamento de recursos entre diferentes origens foi estabelecido via uma biblioteca específica, que contém uma lista explícita de origens autorizadas prevista nas configurações. A mencionada lista branca limita a execução de requisições via JavaScript apenas para o *frontend* implantado nas *URLs* designadas, evitando que sites mal-intencionados hospedados em outros domínios realizem requisições em nome de usuários autenticados. A configuração possibilita o envio de cookies e cabeçalhos por meio da *flag* de credenciais, a qual é essencial para o funcionamento de *tokens* armazenados localmente, que são incluídos em cabeçalhos personalizados. O *middleware* foi colocado próximo ao vértice da pilha, assegurando que as verificações sejam realizadas antes do tratamento da lógica de negócio, permitindo a rejeição antecipada de requisições provenientes de fontes não autorizadas.

A defesa contra a falsificação de requisições entre sites é realizada por meio de um *middleware* específico integrado na pilha. Em interfaces REST que utilizam autenticação fundamentada em *tokens*, as solicitações não estão sujeitas à validação de *token* de proteção, uma vez que os *tokens* armazenados localmente devem ser inclusos de forma explícita por meio de JavaScript, o qual é restringido pela política de mesma origem quando a requisição é originada de um domínio distinto. A proteção continua em vigor para *endpoints* administrativos que utilizam a autenticação de sessão convencional, resguardando o painel administrativo mediante requisições modificadoras que exigem um *token* válido inserido no cabeçalho ou no campo do formulário.

Figura 32 – Fluxo de segurança de comunicação em requisições HTTP na API.



Fonte: autor, 2025.

O sistema incorpora cabeçalhos de segurança HTTP nas respostas através de um *middleware*. O cabeçalho de opções de frame impede o *clickjacking* ao orientar os navegadores a não permitirem a incorporação em frames de páginas externas. Algumas configurações complementares englobam o redirecionamento obrigatório para HTTPS, a implementação de segurança de transporte HTTP estrita e a prevenção da detecção de tipo MIME, que determina o respeito ao tipo declarado. A configuração de *hosts* autorizados limita os nomes aos quais a aplicação pode responder, com base em uma lista estabelecida nas configurações obtidas a partir de

variáveis de ambiente. O objetivo é a prevenção contra ataques de envenenamento de *cache* e injeção de cabeçalho do *host*.

3.5.5 Prevenção contra vulnerabilidades comuns

O sistema estabelece medidas de proteção direcionadas às vulnerabilidades elencadas no OWASP Top 10, que corresponde ao conjunto de riscos mais significativos para aplicações *web*.

O *framework frontend* realiza o escape automático de variáveis renderizadas por meio do mecanismo padrão do React, o que previne a injeção de *scripts*. Quando elementos apresentam informações oriundas do *backend* ou da inserção de usuário, o *framework* realiza, de maneira automática, o processo de escape de caracteres especiais em HTML, convertendo-os em entidades, o que impede sua interpretação como marcação ou código executável. No *backend*, as respostas da interface fornecem apenas dados no formato JSON através de serializadores, que não são reconhecidos como HTML pelo navegador, dessa forma conferindo uma camada adicional de segurança.

Todas as visualizações realizam a verificação de permissões antes de efetuar operações por meio de classes aplicadas no nível das visualizações. Não há *endpoints* que dependam unicamente de parâmetros de URL sem realizar a verificação da relação entre o usuário autenticado e o recurso requisitado. A validação de propriedades é realizada em nível de objeto, através da verificação de relacionamentos mediante a comparação de chaves estrangeiras. Dessa forma é possível assegurar que o acesso seja negado caso não se possua a autorização adequada, mesmo que o invasor conheça o identificador. Aspectos críticos, como notas finais, médias calculadas e resultado final, são definidos como somente leitura nos serializadores, de modo que qualquer alteração direta através de requisições é impedida.

O sistema elabora rotas de arquivos utilizando unicamente identificadores de banco e UUIDs gerados internamente, por meio de bibliotecas com segurança criptográfica, sem incluir entradas diretas de usuários. A função personalizada emprega unicamente o identificador do TCC e o tipo de documento da instância para

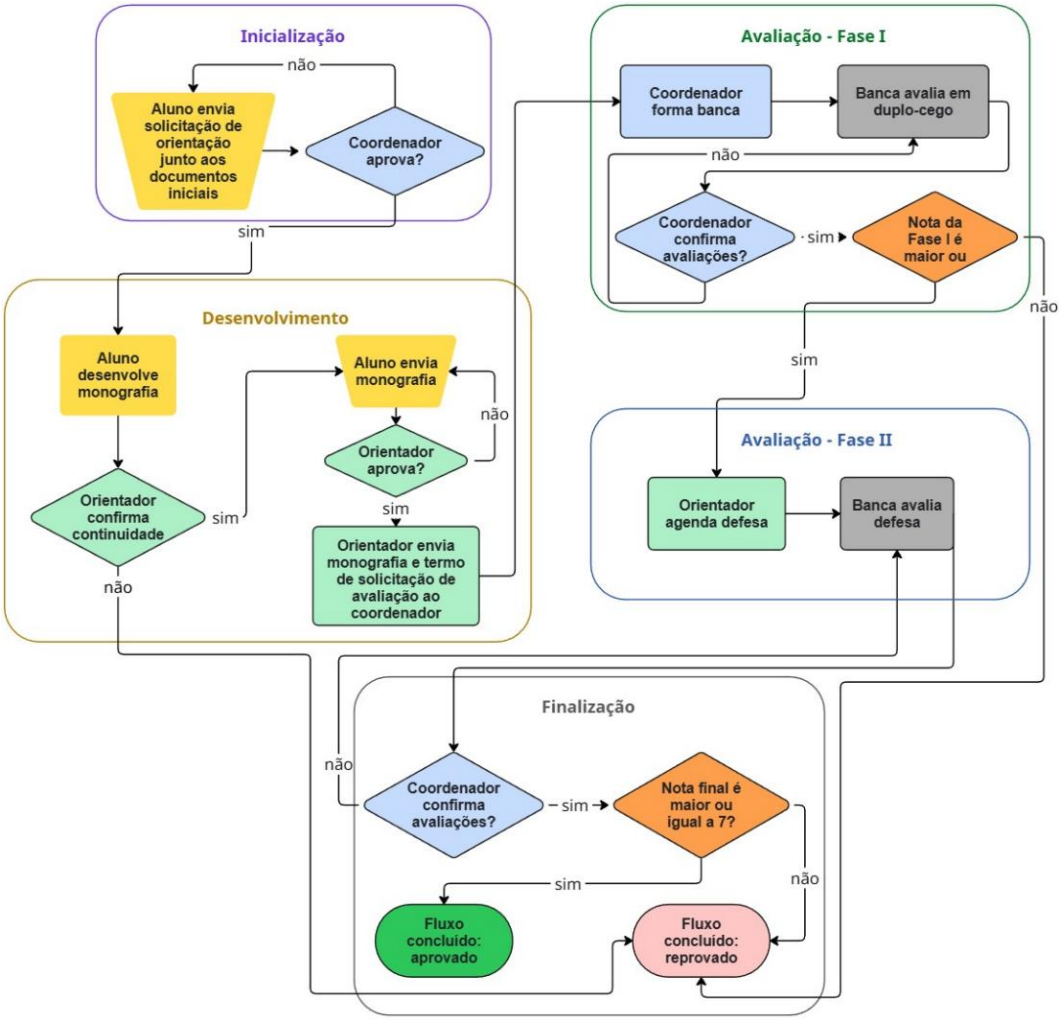
organizar diretórios, criando um nome exclusivo por meio de UUID, independentemente do nome original.

Um sistema de linha do tempo estabelece a auditoria automática de ações críticas por meio da criação de eventos documentados em um modelo específico. Qualquer alteração em TCC, aprovação de documento, validação de avaliação e transição entre etapas provoca um evento por meio de sinais interligados aos modelos, registrando o usuário responsável, o *timestamp* da execução, o tipo de evento e informações específicas. Os registros são gerados com autorizações que impedem alterações após a sua criação e implementam um *log* imutável, essa dinâmica permite o rastreamento para fins de auditoria. Configurações delicadas, como chave secreta, credenciais bancárias e SMTP, são administradas através de variáveis de ambiente carregadas com a utilização de uma biblioteca específica, de modo a assegurar que as credenciais não sejam inseridas no código-fonte e nem commitadas no repositório. A variável de ambiente configura a *flag* de depuração, a qual deve ser desativada em ambientes de produção, a fim de evitar a exposição de rastreamentos de pilha, detalhes de consultas SQL e diretórios de arquivos em mensagens de erro.

4 RESULTADOS

Os resultados obtidos nesta seção avaliam a implementação da plataforma, conduzidos com base no processamento das etapas do fluxo de trabalho de conclusão de curso. A sequência lógica dos processos pode ser apresentada no formato de fluxograma, como apresentado na figura 33.

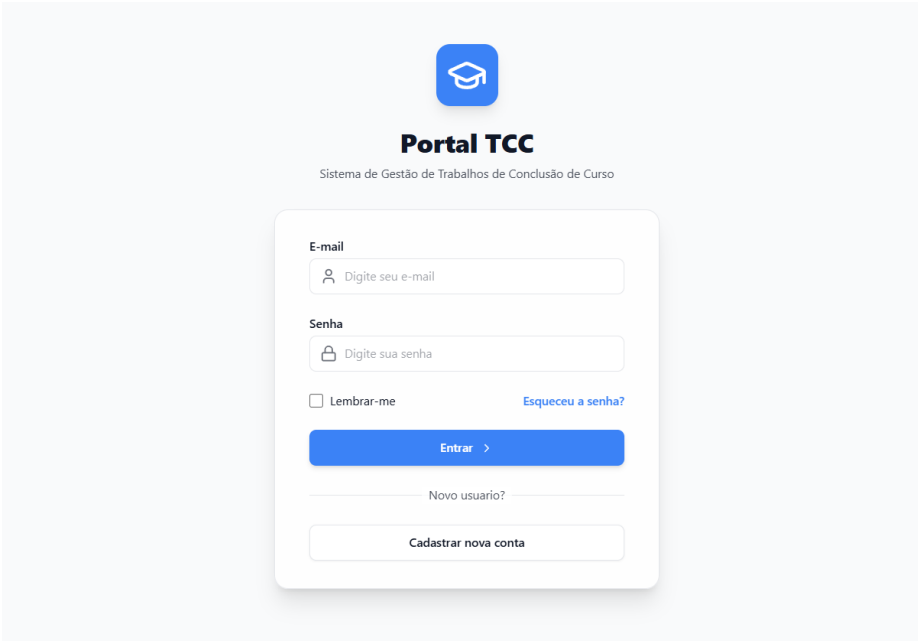
Figura 33 – Fluxograma de fluxo da aplicação.



Fonte: autor, 2025.

Para realizar a execução de um fluxo de TCC é necessária a realização do cadastro de perfil pessoal do usuário na tela de *login*, apresentada na Figura 34.

Figura 34 – Tela de *login*.



Fonte: autor, 2025.

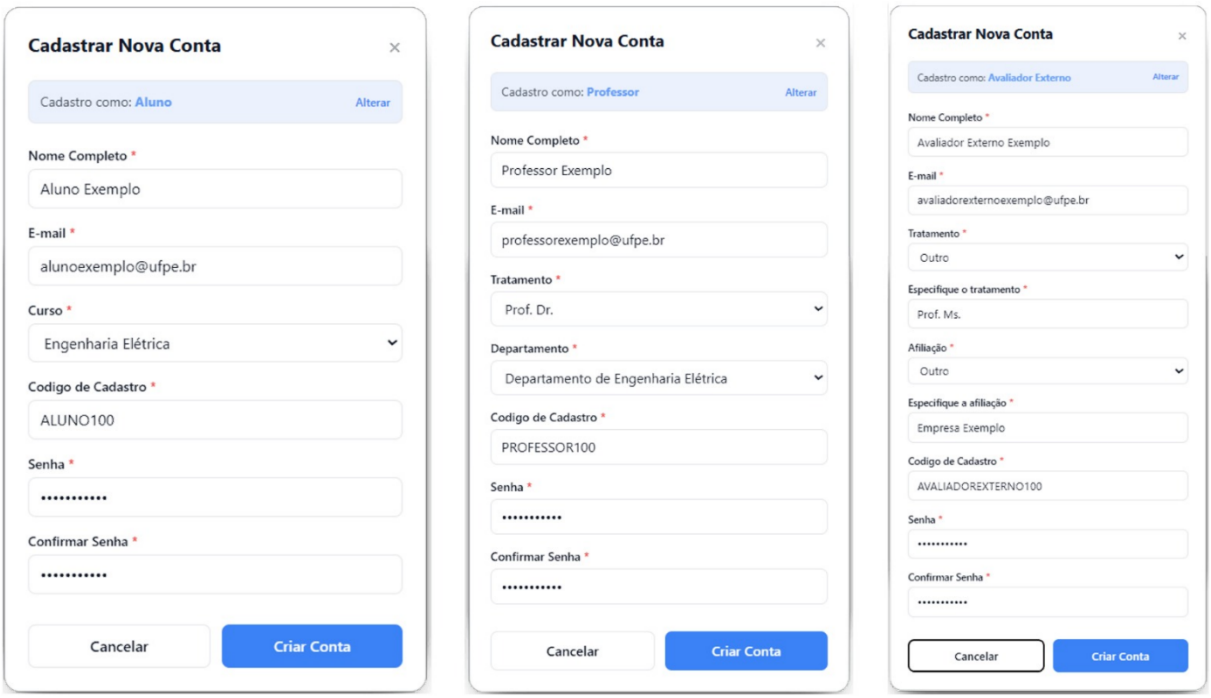
Os três perfis de cadastro disponíveis na tela de cadastro são Aluno, Professor e Avaliador Externo, como demonstrado na Figura 35. O cadastro de perfil do administrador é realizado via acesso direto ao painel do *backend* da aplicação.

Figura 35 – Opções de perfis para cadastro.

Fonte: autor, 2025.

Os requerimentos de cadastro possuem campos distintos de acordo com cada tipo de perfil, como apresentado na Figura 36, de modo que os campos Nome completo, E-mail, código de cadastro e senha são comuns a todos perfis. Os códigos de cadastro solicitados são determinados na seção Planejamento do perfil do coordenador.

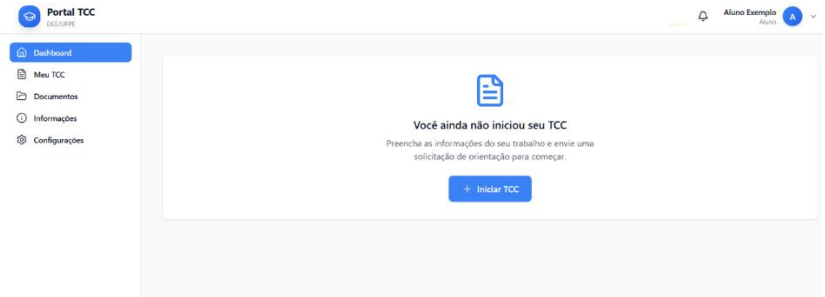
Figura 36 – Telas de cadastro.



Fonte: autor, 2025.

Após o cadastro, o usuário tem acesso ao perfil para inicialização das atividades, exemplificado na Figura 37.

Figura 37 – Tela inicial do aluno.

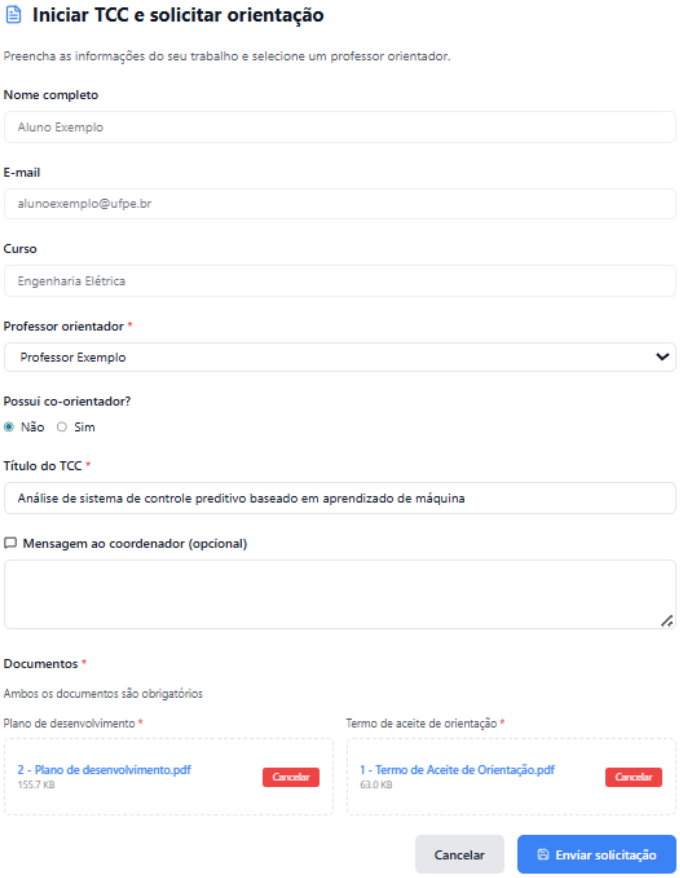


Fonte: autor, 2025.

Etapa de inicialização

A solicitação de orientação é realizada pelo aluno através do formulário inicial, onde será solicitada a seleção do orientador (professor interno), indicação do coorientador (opcional), título do TCC e *upload* do plano de desenvolvimento e termo de aceite de orientação, como apresentado na Figura 38.

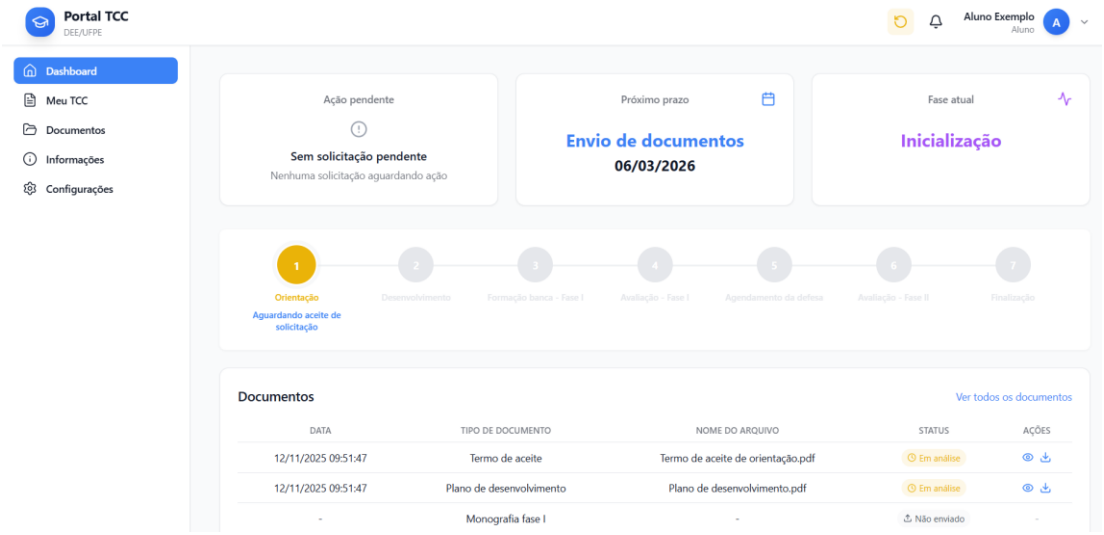
Figura 38 – Tela de solicitação de orientação.



Fonte: autor, 2025.

Após o envio da solicitação de orientação, a Figura 39 demonstra que a tela do *dashboard* do aluno é liberada para visualização de ações, prazo, fase atual, *timeline* de eventos e histórico de documentos.

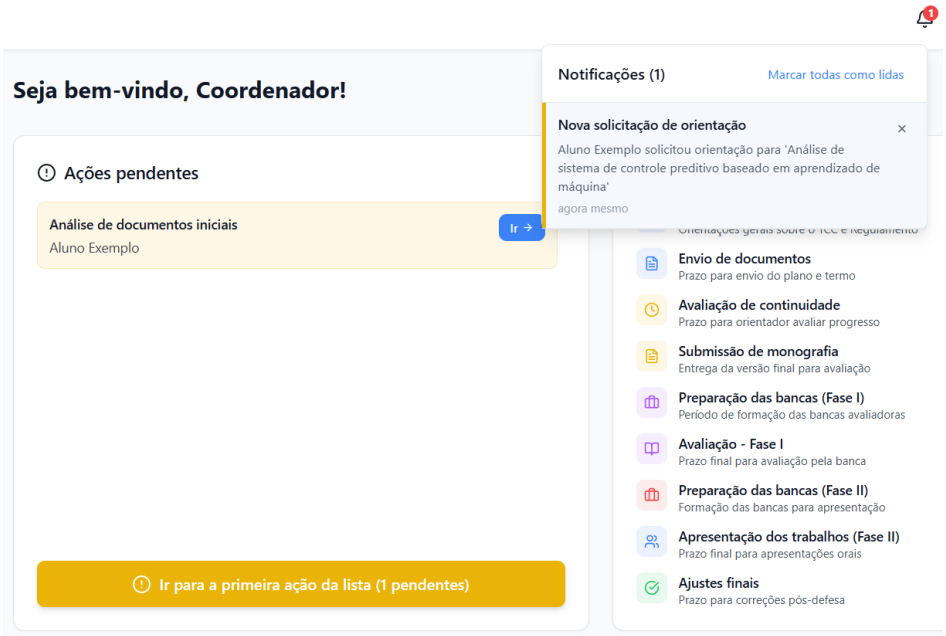
Figura 39 – Tela do aluno, após envio de solicitação.



Fonte: autor, 2025.

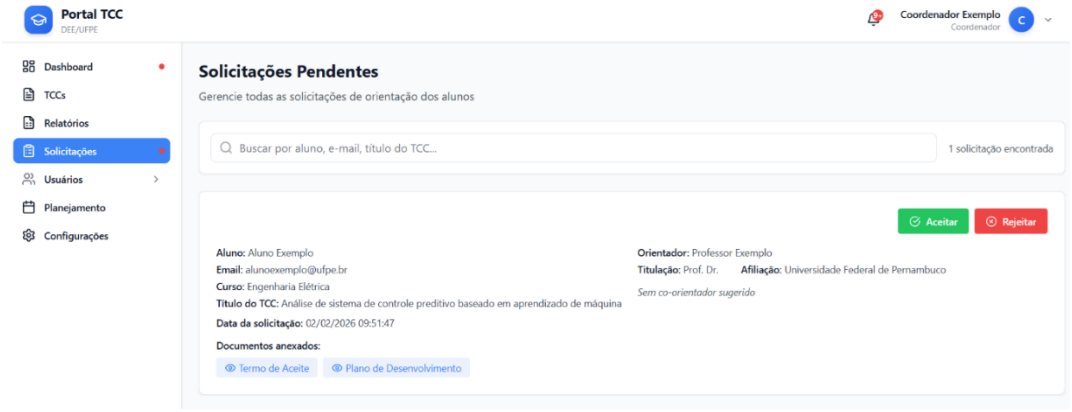
O coordenador recebe a notificação, como apresentado na Figura 40, para realizar aprovação ou solicitar ajustes de informações, como exibido na Figura 41.

Figura 40 – Notificação de solicitação de orientação no perfil do coordenador.



Fonte: autor, 2025.

Figura 41 – Tela de avaliação de solicitação de orientação.

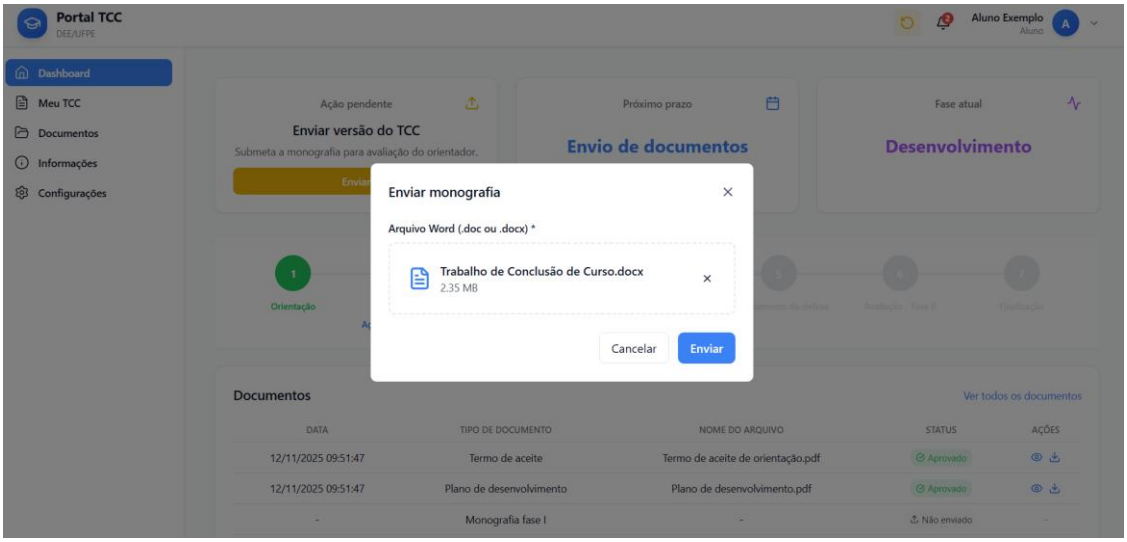


Fonte: autor, 2025.

Etapa de desenvolvimento

Com a aprovação do coordenador, o aluno recebe a notificação de aprovação e o campo de envio de TCC ao orientador é liberado, como observado na Figura 42.

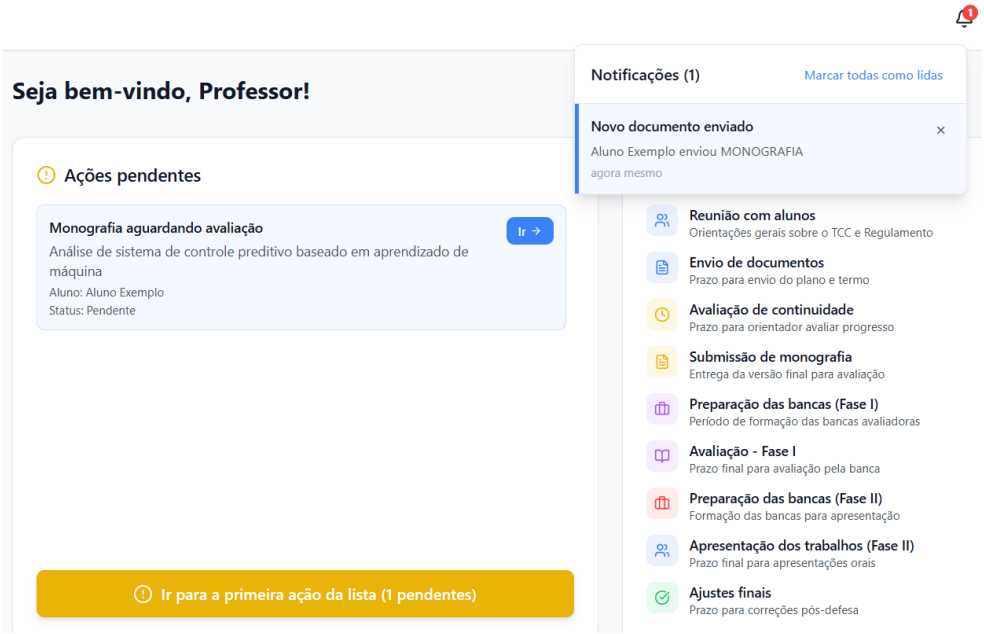
Figura 42 – Tela de envio de TCC.



Fonte: autor, 2025.

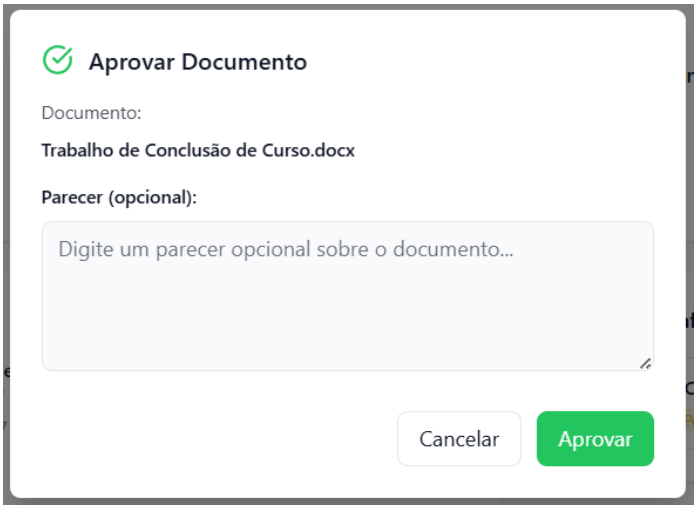
O envio do TCC do aluno gera uma notificação ao orientador, que por sua vez, pode aprovar ou solicitar ajustes, como demonstrado na Figura 43 e na Figura 44.

Figura 43 – Notificação de envio de TCC no perfil do orientador.



Fonte: autor, 2025.

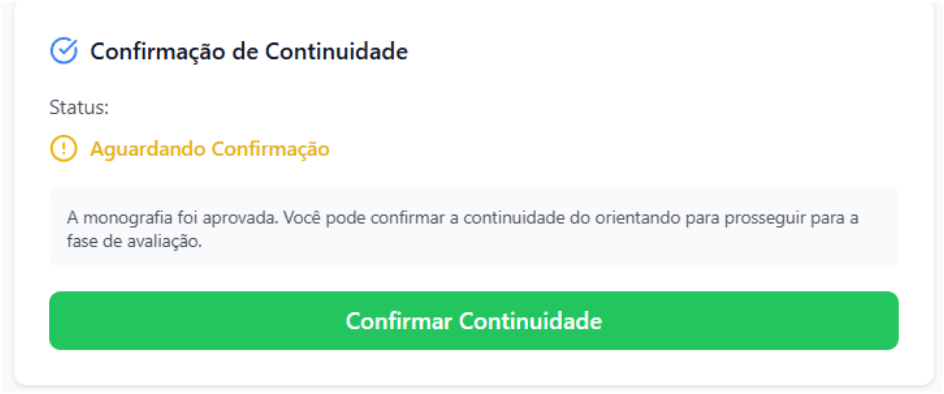
Figura 44 – Tela de aprovação de TCC.



Fonte: autor, 2025.

Na data determinada pelo coordenador, na tela de Planejamento, a confirmação de continuidade deve ser confirmada ou recusada, como exemplificado na Figura 45. Em caso de recusa, o fluxo do aluno será encerrado.

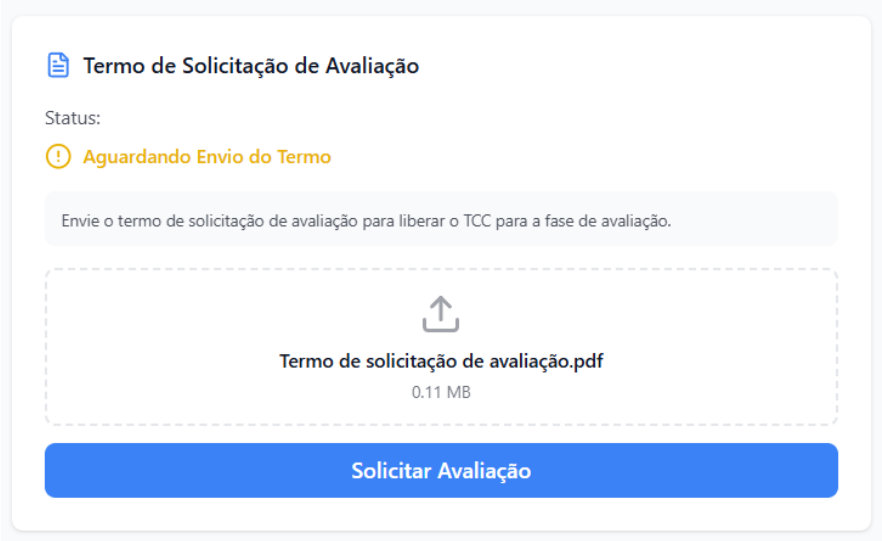
Figura 45 – Tela de confirmação de continuidade.



Fonte: autor, 2025.

Após a confirmação de continuidade e aprovação do TCC pelo orientador, o campo de envio do termo de solicitação de avaliação será liberado para que o orientador realize o *upload* e envio do documento, solicitando ao coordenador a formação da banca de avaliação, assim como apresentado na Figura 46.

Figura 46 – Tela de termo de solicitação de avaliação.

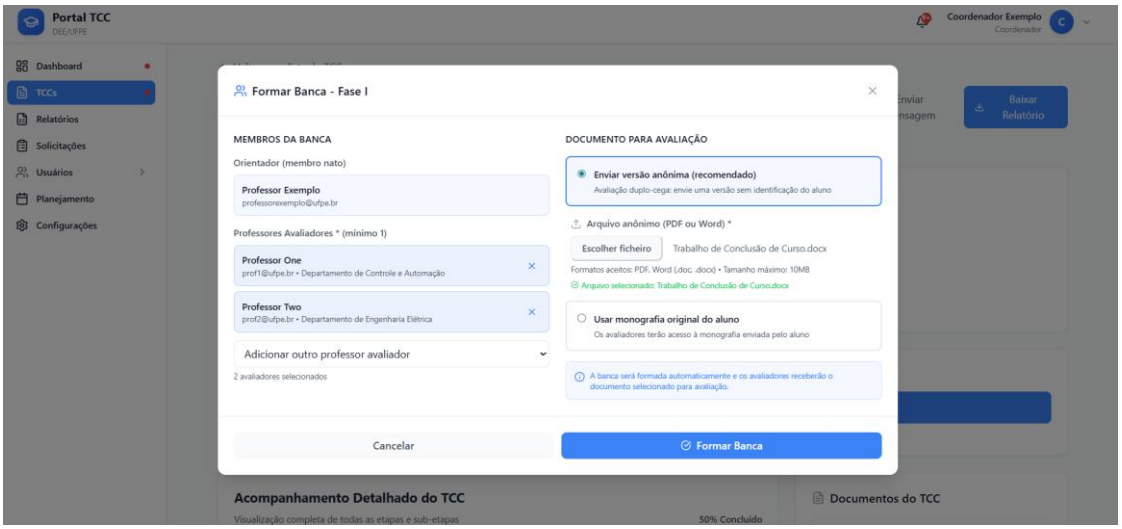


Fonte: autor, 2025.

Etapa Fase I

Na sequência, o coordenador recebe a notificação e realiza a formação da banca, como demonstrado na Figura 47.

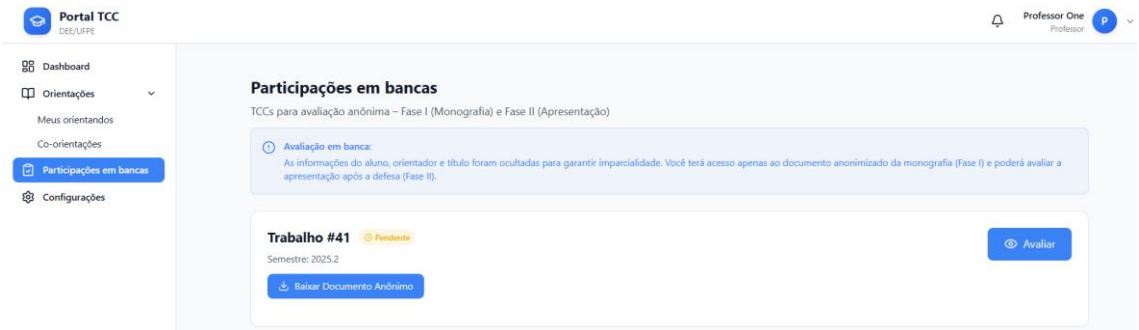
Figura 47 – Tela de formação de banca.



Fonte: autor, 2025.

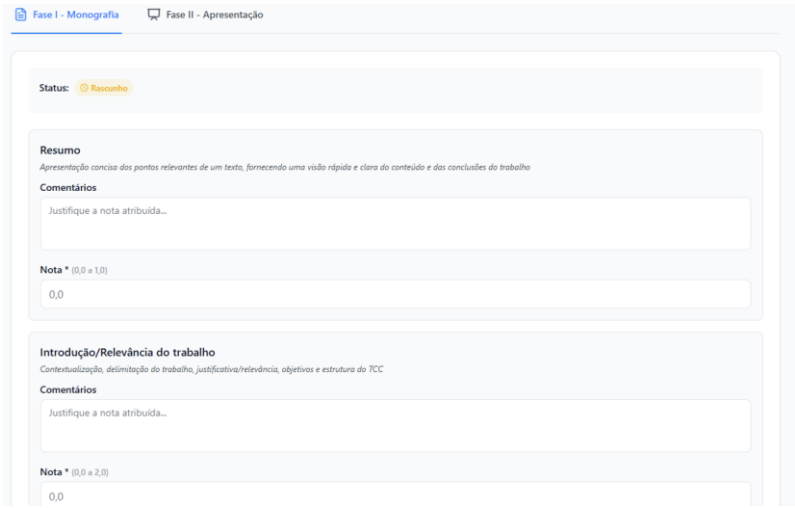
A banca convidada recebe a notificação e pode acessar a página de avaliação, contendo o TCC e o formulário de avaliação da Fase I, como é possível observar na Figura 48 e Figura 49.

Figura 48 – Tela de participação de banca do professor.



Fonte: autor, 2025.

Figura 49 – Tela de avaliação da Fase I.



Fonte: autor, 2025.

Após o envio das avaliações de toda banca, o coordenador pode realizar a validação das avaliações ou solicitar ajustes.

Com a confirmação das notas da banca e a validação do coordenador, caso o aluno seja aprovado, o orientador é notificado e pode realizar o agendamento da defesa, como demonstrado na Figura 50. Caso o aluno não atinja a nota mínima necessária para a aprovação, o fluxo do TCC é encerrado.

Etapa Fase II

Figura 50 – Tela de agendamento de defesa.

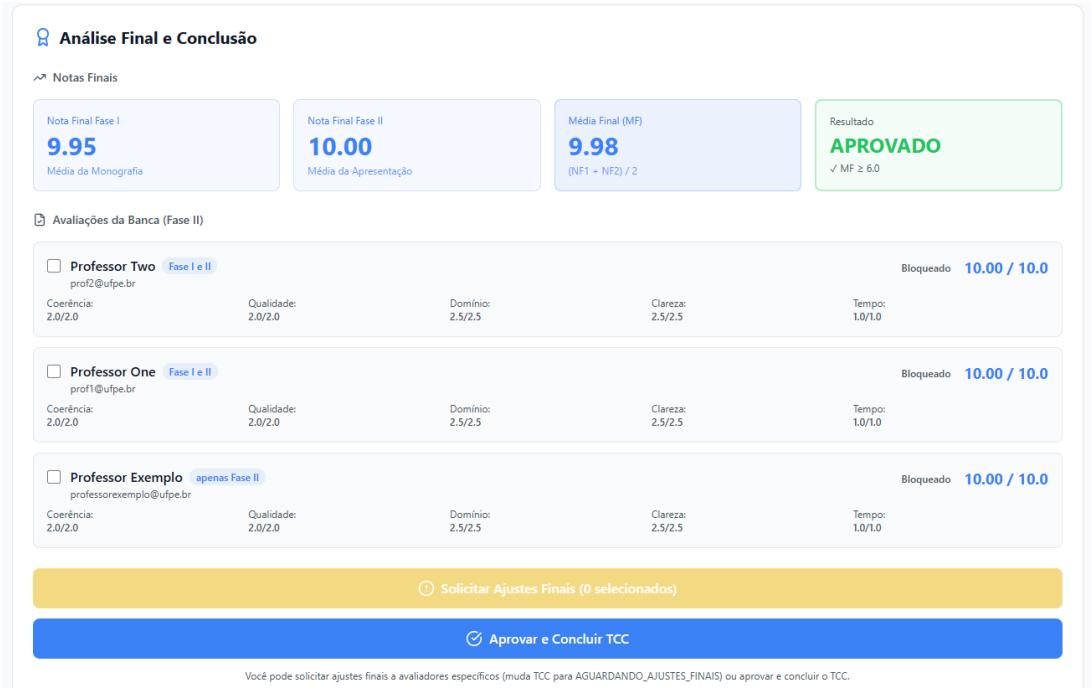
Fonte: autor, 2025.

No dia e hora marcados para a defesa, os formulários de avaliação da segunda fase serão liberados para a banca realizar as avaliações.

Após o envio de todas as notas da banca, o coordenador realiza a validação das notas da Fase II, através do painel apresentado na Figura 51. Com a confirmação das notas pelo coordenador, o fluxo é concluído e a aprovação será efetiva caso o discente alcance a nota mínima exigida para aprovação.

Etapa de finalização

Figura 51 – Tela de finalização do fluxo de TCC.



Fonte: autor, 2025.

5 CONCLUSÕES E PROPOSTAS DE CONTINUIDADE

A plataforma desenvolvida atende ao objetivo de centralizar, padronizar e acompanhar o ciclo de vida do TCC no DEE/UFPE, de forma a reduzir a dependência de procedimentos manuais e ampliar a rastreabilidade das etapas para alunos, orientadores, avaliadores e coordenação. A implementação de arquitetura cliente-servidor com *backend* em Django/DRF e *frontend* em React/TypeScript, integrados por API REST com JWT, mostrou-se adequada para manutenção evolutiva, integração e controle de acesso baseado em papéis. Além disso, a estrutura em módulos do *backend*, a máquina de estados para governar fases e transições, o sistema de notificações e os controles de prazos contribuíram para a conformidade processual e a transparência operacional dos fluxos acadêmicos descritos nos capítulos 3 e 4.

Os resultados indicam a viabilidade prática da solução, resultando no aumento da eficiência administrativa, maior clareza procedimental e na centralização de informações institucionais.

Com base nos resultados obtidos, propõem-se as seguintes direções para trabalhos futuros:

- Desenvolver uma versão móvel (aplicativo), de modo a permitir aos usuários o acompanhamento e a execução de procedimentos via dispositivos móveis;
- Adaptar a plataforma para outros procedimentos acadêmicos, como por exemplo o gerenciamento de estágios supervisionados.

REFERÊNCIAS

1. **BANKS, Alex; PORCELLO, Eve. Learning React: Modern Patterns for Developing React Apps**. 2nd ed. Sebastopol: O'Reilly Media, 2020.
2. **CHRISTIE, Tom. Django REST Framework**. 2011. Disponível em: *https*://www.Django-*rest-framework*.org/. Acesso em: 8 jan. 2025.
3. **DAVENPORT, Thomas H. Process Innovation: Reengineering Work through Information Technology**. Boston: Harvard Business School Press, 1993.
4. **FACEBOOK. React: A JavaScript library for building user interfaces**. 2013. Disponível em: *https*://React.dev/. Acesso em: 8 jan. 2025.
5. **FIELDING, Roy T. Architectural Styles and the Design of Network-based *Software* Architectures**. 2000. Tese (Doutorado em Ciência da Computação) – University of California, Irvine, 2000.
6. **FORCIER, Jeff; BISSEX, Paul; CHUN, Wesley J. Python *Web* Development with Django**. Upper Saddle River: Addison-Wesley Professional, 2008.
7. **FOWLER, Martin. Patterns of Enterprise *Application* Architecture**. Boston: Addison-Wesley Professional, 2002.
8. **GAMMA, Erich et al. Design Patterns: Elements of Reusable Object-Oriented *Software***. Reading: Addison-Wesley Professional, 1994.
9. **GREENFELD, Daniel Roy; GREENFELD, Audrey Roy**. Two Scoops of Django 4.x: Best Practices for the Django *Web Framework*. 6th ed. San Diego: Two Scoops Press, 2022.
10. **JONES, M. et al. JSON *Web Token* (JWT)**. RFC 7519, IETF, maio 2015. Disponível em: *https*://datatracker.ietf.org/doc/HTML/rfc7519. Acesso em: 8 jan. 2025.
11. **LAKATOS, Eva Maria; MARCONI, Marina de Andrade**. Fundamentos de Metodologia Científica. 8. ed. São Paulo: Atlas, 2017.
12. **LAUDON, Kenneth C.; LAUDON, Jane P. Sistemas de Informação Gerenciais**. 14. ed. São Paulo: Pearson, 2020.
13. **MICROSOFT. TypeScript: JavaScript with syntax for types**. 2012. Disponível em: *https*://www.typescriptlang.org/. Acesso em: 8 jan. 2025.
14. **OWASP. OWASP Top Ten 2021**. 2021. Disponível em: *https*://owasp.org/www-*project*-top-ten/. Acesso em: 8 jan. 2025.
15. **PRESSMAN, Roger S.; MAXIM, Bruce R. *Software* Engineering: A Practitioner's Approach**. 8th ed. New York: McGraw-Hill Education, 2014.
16. **SEVERINO, Antônio Joaquim. Metodologia do Trabalho Científico.** 23. ed. São Paulo: Cortez, 2016.
17. **SOMMERVILLE, Ian. *Software* Engineering**. 10th ed. Harlow: Pearson, 2015.