# TrackAPI: Biblioteca de Rastreabilidade com *Django*

**Ayllon de Sena Andrade [*1], Maira Santana Ribeiro [†1], Maria Eduarda Sales Costa [‡1], Pedro Henrique de Carvalho Silveira [§1], Rafael dos Santos de Almeida [¶ 1], Angela Peixoto Santana [‖1*]**

[1] Análise e Desenvolvimento de Sistemas
Escola de Tecnologias
Universidade Católica do Salvador (UCSAL)
Av. Prof. Pinto de Aguiar, 2589 Pituaçu, CEP: 41740-090
Salvador/BA, Brasil
[1] *{ayllon.andrade,maira.ribeiro,mariaeduarda.costa,pedrohenrique.silveira,rafaelsantos.almeida}@ucsal.edu.br*
[1*] *{angela.santana}@pro.ucsal.edu.br*

Julho 2025

[*]ayllon.andrade@ucsal.edu.br
[†]maira.ribeiro@ucsal.edu.br
[‡]mariaeduarda.costa@ucsal.edu.br
[§]pedrohenrique.silveira@ucsal.edu.br
[¶]rafaelsantos.almeida@ucsal.edu.br
[‖]angela.santana@pro.ucsal.br

# Resumo

A segurança e rastreabilidade são fundamentais em aplicações que lidam com dados sensíveis, especialmente em áreas como a saúde, onde normas como LGPD e HIPAA são obrigatórias. O framework Django não oferece, de maneira nativa, funcionalidades completas para auditoria detalhada, registro seguro de alterações em modelos e rastreamento de requisições. Este trabalho apresenta a biblioteca TrackAPI, que permite integrar de forma simples funcionalidades avançadas de auditoria e rastreabilidade ao Django. Os principais diferenciais são a interceptação automática de requisições HTTP, monitoramento automático de mudanças nos dados via sinais internos e flexibilidade no armazenamento dos dados. A validação técnica foi realizada por meio de testes automatizados utilizando Pytest, garantindo a robustez e confiabilidade da biblioteca.



**Palavras-chaves**: Django, Rastreabilidade, Auditoria, Segurança, LGPD.


# 1 Introdução

A segurança e a rastreabilidade de informações são aspectos críticos em aplicações que lidam com dados sensíveis, como registros financeiros, sistemas corporativos e prontuários médicos. Com o avanço da digitalização e a rápida crescente adoção de sistemas em nuvem, que promovem a descentralização dos dados, garantir a transparência e o controle sobre o fluxo e a manipulação dessas informações tornou-se uma prioridade no ambiente digital.

Nesse cenário, a conformidade com regulamentações como a Lei Geral de Proteção de Dados (LGPD) e o *Health Insurance Portability and Accountability Act* (HIPAA) impõe desafios significativos ao desenvolvimento de soluções tecnológicas voltadas para o setor de saúde. Do ponto de vista da Tecnologia da Informação, essas exigências requerem a implementação de mecanismos robustos que garantam a segurança, a rastreabilidade e a integridade dos dados sensíveis. Isso inclui o desenvolvimento de arquiteturas capazes de realizar monitoramento contínuo, registro detalhado de *logs*, rastreamento de requisições e auditoria de alterações em modelos de dados. Além de atender às exigências legais, tais medidas são fundamentais para aumentar a confiabilidade dos sistemas e mitigar riscos associados a vazamentos, acessos indevidos e inconsistências no tratamento das informações.

Neste contexto, este trabalho propõe o desenvolvimento de uma biblioteca para o *framework Django*, capaz de integrar essas funcionalidades de forma flexível e configurável. O objetivo é fornecer uma solução que permita a desenvolvedores e organizações atenderem de maneira eficiente aos requisitos de segurança, rastreabilidade e conformidade exigidos por regulamentações como a LGPD e o HIPAA.

# 2 Fundamentação Teórica

Esta seção apresenta a fundamentação teórica que sustenta os principais conceitos abordados neste trabalho de pesquisa, fornecendo subsídios para o desenvolvimento da proposta e para a compreensão dos temas relacionados.

## 2.1 *Python*

*Python* é uma linguagem de programação de alto nível, orientada a objetos, com tipagem dinâmica e forte, interpretada e interativa. Criada em 1990 por Guido van Rossum, a linguagem destaca-se por sua simplicidade sintática e legibilidade, o que a torna especialmente adequada para desenvolvimento rápido e manutenção de código. De acordo com Lutz (2007), Python foi projetada para ser extensível, modular e versátil, características que contribuíram para sua ampla adoção em diversas áreas, incluindo ciência de dados, desenvolvimento *web*, automação e inteligência artificial..

> "*Python* é uma linguagem orientada a objetos utilizada em uma variedade de domínios, tanto para programas independentes como para aplicações de script. Ela é gratuita, portátil e fácil de usar (LUTZ, 2007)."

> "A linguagem *Python* é destacada entre as linguagens dinâmicas como uma das mais populares e poderosas, com uma grande comunidade de usuários espalhados pelo mundo"(BORGES, 2014).

Além disso, *Python* é uma linguagem de código aberto (open source), distribuída sob uma licença compatível com a *General Public License* (GPL). Suas especificações e diretrizes de desenvolvimento são mantidas pela *Python Software Foundation* (PSF), organização responsável por promover o crescimento e a evolução da linguagem. A Figura 1 apresenta o logotipo oficial da linguagem.

Figura 1 – Logotipo da linguagem Python


Autor: (PYTHON.ORG, 2025)

## 2.2 *Framework Django*

Segundo Franzoni (2023), o *Django* é um *framework web* de alto nível, escrito em *Python*, que adota o padrão MVC *(Model-View-Controller)*. É amplamente utilizado no desen-

volvimento de aplicações robustas e escaláveis, graças à sua arquitetura bem definida e à variedade de recursos oferecidos.

Entre suas principais funcionalidades, destacam-se o ORM (*Object-Relational Mapping*) para manipulação de banco de dados, o sistema de autenticação de usuários e o uso de *middlewares* para interceptação e tratamento de requisições. Esses componentes prontos tornam o desenvolvimento mais ágil e padronizado.

No entanto, apesar de sua robustez, o *Django* não oferece, de forma nativa, uma solução completa para integração de funcionalidades como monitoramento de ações do usuário, registro de *logs* contextualizados, rastreamento de requisições e auditoria de alterações em modelos. Essa lacuna se torna especialmente crítica em aplicações voltadas para a área da saúde, onde segurança e rastreabilidade de dados são requisitos essenciais.

A figura 2 representa a logotipo oficial do *Framework*.

Figura 2 – Logotipo do *Django Framework*


Autor: (DJANGOPROJECT.COM, 2025)

## 2.3 Rastreabilidade e Auditoria de Sistemas

A rastreabilidade é o processo de monitoramento, mapeamento e controle de um produto ao longo de todo o seu ciclo de vida, desde a origem até o destino final. Trata-se de uma prática amplamente aplicada em diferentes setores. Na agropecuária, por exemplo, permite o acompanhamento da trajetória de gêneros alimentícios, animais ou substâncias, conforme estabelecido pelo Regulamento nº 178/2002 da Comunidade Europeia. Na indústria, é utilizada para o controle de qualidade de peças mecânicas e produtos químicos. Já na área da saúde, destaca-se pelo rastreamento de medicamentos por meio de informações como número de lote e data de validade.

De modo geral, a rastreabilidade desempenha um papel fundamental no controle de qualidade, contribuindo para a identificação e correção de não conformidades, além de aumentar a segurança e a confiança do consumidor final em relação ao produto.

Segundo a ISO9000 (2015), a rastreabilidade é a capacidade de rastrear o histórico, a aplicação ou a localização de um objeto, entidade, item ou qualquer elemento perceptível ou concebível, como produtos, serviços, processos, pessoas, organizações, sistemas e recursos. Quando o objeto em questão é um produto ou serviço, a rastreabilidade refere-se à origem dos materiais e componentes, ao histórico do processamento ou distribuição, e à localização do produto ou serviço após a entrega.

No que diz respeito à auditoria, o artigo de Rodrigues (2013) intitulado Auditoria de Sistemas de Informática nas Empresas Modernas é definido que:

> "A auditoria consiste em um conjunto de procedimentos e técnicas, cujo objetivo principal é o de atestar a veracidade das informações acerca de determinado processo, controle ou sobre a situação patrimonial, econômica ou financeira das entidades, oferecendo aos usuários das informações maior segurança e confiabilidade."

Portanto, a auditoria possibilita verificar a veracidade e a confiabilidade das informações, garantindo que os procedimentos utilizados na obtenção dos dados estejam alinhados com o que foi originalmente projetado. Além disso, a auditoria avalia todos os demais aspectos das atividades operacionais, como o nível de qualidade, eficiência e eficácia dos processos envolvidos.

## 2.4 Monitoramento de Ações do Usuário

O monitoramento das ações dos usuários é crucial para assegurar a segurança e a integridade dos dados em aplicações críticas, como sistemas de prontuários médicos. A biblioteca proposta registra de forma detalhada todas as ações realizadas pelos usuários, incluindo *logins*, acessos a dados e alterações em registros. Além disso, essas informações são complementadas com metadados contextuais, como o ID do usuário, o ID da requisição e o *timestamp*, em conformidade com as recomendações de Larini (2021). Essa abordagem não apenas possibilita o rastreamento das ações, mas também permite compreender o contexto em que foram realizadas, o que é essencial para auditorias e investigações de segurança.

No que diz respeito à ética profissional, Amorim Wendell de Andrade e Melo (2023) enfatiza sua relevância no uso de prontuários eletrônicos, destacando a necessidade de sistemas que assegurem a rastreabilidade e a transparência nas ações dos usuários. A biblioteca proposta atende a essa demanda, oferecendo ferramentas que possibilitam o monitoramento e a auditoria de todas as interações dos usuários com o sistema.

## 2.5 Registro de Logs de Acesso

O registro de logs de acesso é uma funcionalidade essencial para rastrear acessos e alterações em sistemas de saúde. A biblioteca proposta armazena *logs* de acesso em múltiplos sistemas de armazenamento, como bancos de dados relacionais, *MongoDB*, arquivos de *log* e *Elasticsearch*. Essa flexibilidade permite que os desenvolvedores escolham o sistema de armazenamento mais adequado às suas necessidades, conforme discutido no artigo da ACM (2021).

Além disso, o estudo de Sobrinho O. e Cugnasca (2010) sobre a modelagem de sistemas de informação para rastreabilidade ressalta a importância de arquiteturas flexíveis que

suportem múltiplos backends de armazenamento. Essa abordagem é particularmente relevante em sistemas de saúde, onde a diversidade de tecnologias e a necessidade de escalabilidade são frequentes.

## 2.6 Rastreamento de Requisições

O rastreamento de requisições é fundamental para entender o fluxo de ações em uma aplicação e para identificar potenciais problemas de segurança ou desempenho. A biblioteca proposta intercepta e registra todas as requisições HTTP (*Hypertext Transfer Protocol*), capturando detalhes como método, *endpoint, status code* e tempo de execução. Essas informações são correlacionadas por meio de IDs únicos de requisição, em conformidade com as boas práticas descritas por Larini (2021).

Além disso, a Norma 9000 (2015) destaca a importância de sistemas de gestão da qualidade que assegurem a rastreabilidade e a conformidade com padrões internacionais. Nesse contexto, a biblioteca proposta não só atende a esses requisitos, como também fornece ferramentas eficientes para rastrear e auditar todas as requisições, garantindo maior controle e transparência nas operações.

## 2.7 Segurança da Informação e Conformidade com a LGPD

A Lei n.º 13.709/2018, conhecida como Lei Geral de Proteção de Dados (LGPD), regula o tratamento de dados pessoais, inclusive em meios digitais, por pessoas naturais ou jurídicas de direito público ou privado. O principal objetivo da LGPD é proteger os direitos fundamentais à liberdade, à privacidade e ao livre desenvolvimento da personalidade da pessoa natural (Brasil, 2018).

A LGPD introduziu o conceito de Agentes de Tratamento de Dados, dividindo-os em duas categorias: controlador e operador. Segundo o trabalho de Mello e Miramontes (2021),

> Os agentes de tratamento definidos pela LGPD são o controlador e o operador, ambos responsáveis pelo tratamento de dados nas empresas e organizações, mas com funções distintas... (MELLO; MIRAMONTES, 2021).

Além disso, a LGPD prevê o papel do encarregado, também conhecido como Data Protection Officer (DPO). O artigo de Mello e Miramontes (2021) esclarece a função desse agente:

> O encarregado é uma pessoa designada pelo controlador para atuar como canal de comunicação entre o controlador, os titulares dos dados e a Autoridade Nacional de Proteção de Dados (ANPD). É importante destacar que o DPO não é considerado um agente de tratamento, mas deve trabalhar em conjunto com eles, desempenhando o papel de intermediário (MELLO; MIRAMONTES, 2021).

## 2.8 Rastreabilidade de Informações

A rastreabilidade de informações é um conceito essencial em aplicações que lidam com dados sensíveis, como prontuários médicos. De acordo com Larini (2021), a automação do registro de *logs* é crucial para aprimorar a rastreabilidade e garantir a manutenção adequada das aplicações. A biblioteca proposta vai além das soluções tradicionais, oferecendo metadados contextuais em cada registro, como o ID do usuário, o ID da requisição e o *timestamp*.

Esses metadados permitem correlacionar eventos e compreender o fluxo de ações na aplicação, melhorando significativamente a visibilidade e a segurança do sistema.

Além disso, Cruz (2006) destaca a rastreabilidade como um fator essencial para a sustentabilidade nas cadeias produtivas, ampliando sua relevância para além da competitividade empresarial. No contexto de sistemas de saúde, a rastreabilidade é fundamental para garantir a integridade dos dados e assegurar conformidade com regulamentações como a LGPD e o HIPAA.

Esse ponto é reforçado por Conchon Fabrício Luciano e Lopes (2012), que discute a importância de sistemas robustos de rastreabilidade na segurança alimentar, salientando que tais sistemas são essenciais para garantir a confiabilidade dos dados. Por sua vez, Silva Anderson Rogério e Gasparotto (2020) complementa essa visão, argumentando que sistemas de rastreabilidade são vitais para o controle eficiente de processos industriais, uma abordagem que também é aplicável ao contexto da saúde.

## 2.9 Auditoria de Mudanças em Modelos

A auditoria de mudanças em modelos é um componente essencial para aplicações que lidam com dados sensíveis, como prontuários médicos. A biblioteca proposta monitora e registra todas as alterações realizadas nos modelos *Django*, incluindo criação, atualização e exclusão de registros. As auditorias são detalhadas, informando quem realizou a alteração, quando ela ocorreu e quais dados foram modificados, assegurando conformidade com regulamentações como a LGPD e o HIPAA.

No trabalho de Lirani (2005), a rastreabilidade é destacada como uma exigência fundamental para sistemas que manipulam dados críticos, enfatizando a importância de ferramentas que permitam auditar e rastrear mudanças de forma eficiente.

## 2.10 Trabalhos relacionados

Esta seção apresenta os trabalhos relacionados encontrados na literatura, com ênfase em abordagens voltadas à segurança, auditoria e análise de dados em ambientes computacionais. São descritas as soluções já existentes no ecossistema de desenvolvimento de sistemas, especialmente aqueles baseados em tecnologias e ferramentas de processamento

de logs, destacando as metodologias aplicadas e os resultados alcançados. As contribuições desses estudos para o avanço da área são analisadas, motivando o desenvolvimento de uma nova proposta integrada, conforme desenvolvida neste trabalho.

O trabalho *Segurança de Dados Distribuída em Saúde Digital* investigou o potencial da tecnologia *blockchain* na área da Saúde Digital, a partir da extensão da ferramenta *SmartMed* Santos et al. (2024). A proposta consistiu na implementação de um protótipo de sistema de controle de acesso integrado ao autenticador *KeyCloak*, utilizando a plataforma *HyperLedger Besu*. O objetivo principal foi fornecer uma plataforma confiável para armazenamento de logs, garantindo a imutabilidade e auditabilidade dos dados, além de estabelecer uma identidade auto soberana por meio da integração com o *HyperLedger Indy*. A análise do protótipo demonstrou que a abordagem oferece segurança, privacidade e conformidade com legislações como a LGPD e a GDPR, mantendo um custo computacional viável.

O trabalho *Um ambiente seguro para aplicação com framework web Django* (DIAS, 2020) apresentou os conceitos fundamentais de *frameworks* e sua relevância no desenvolvimento de aplicações *web*, com foco específico no uso do *Django*, um *framework* para aplicações em *Python*. O estudo abordou a estrutura de uma plataforma *web* hospedada em um servidor, detalhando seu funcionamento, o sistema operacional utilizado e os principais componentes instalados, como o SGBD *PostgreSQL* e o próprio *Django*. Foram discutidas também questões de segurança no ambiente do servidor web, propondo medidas como configuração de *firewall*, uso de certificados SSL e TLS, diretrizes de segurança do *PostgreSQL* e funcionalidades nativas do *Django*. O trabalho incluiu recomendações baseadas nas normas NGS1 e NGS2 da Sociedade Brasileira de Informática em Saúde (SBIS), com testes e análises de vulnerabilidades, concluindo que a segurança é essencial para proteção dos dados e a credibilidade das aplicações.

Em *Um sistema de auditoria baseado na análise de registros de log*, os autores Simon, Santos e Hara (2008) demonstraram a viabilidade de auditoria eficiente de dados por meio do processamento contínuo de *streams*. Utilizando a plataforma *Borealis*, o sistema permitiu configurar regras e políticas de auditoria em uma linguagem de alto nível, facilitando a manutenção e a evolução do sistema. A solução expandiu as funcionalidades do *PostgreSQL* com análise em tempo real de desempenho e acesso, além de oferecer novos mecanismos de coleta e auditoria distribuída. Mesmo com uma estrutura simples, o sistema evidenciou a integração eficaz entre tecnologias de gerenciamento de banco de dados (SGBD) e de stream (SGSD), com potencial para aplicações em ambientes que exigem análise de logs e detecção de padrões.

No artigo *Analysis of Logs by Using Logstash*, Sushma Sanjappa e Muzameel Ahmed destacam a importância do uso de ferramentas especializadas para coleta e análise de grandes volumes de dados de log em ambientes distribuídos (SANJAPPA; AHMED, 2017). A solução propõe o uso do Logstash, evidenciando sua capacidade de processar diferentes

tipos de logs de múltiplos nós dentro de uma rede. Essa abordagem facilita a identificação de atividades maliciosas e promove maior segurança e eficiência na análise de eventos, mostrando como tecnologias apropriadas contribuem para a centralização e interpretação dos *logs*.

Diante das soluções analisadas nesta seção, observa-se um avanço significativo nas abordagens voltadas à segurança, auditoria e análise de dados, especialmente no que se refere ao uso de tecnologias e ferramentas especializadas para *logs*. Esses trabalhos fornecem uma base sólida e inspiram a construção de novas propostas que atendam às demandas atuais por rastreabilidade e conformidade.

Nesse contexto, a *TrackAPI* está sendo desenvolvida com um conjunto criterioso de funcionalidades essenciais, refletindo uma abordagem centrada em um *MVP (Minimum Viable Product)*. Essa estratégia garante a entrega de um produto funcional que atende às demandas mais críticas dos sistemas que necessitam de rastreabilidade, como o *log* de operações *CRUD*, rastreamento de *endpoints* e suporte à auditoria de dados sensíveis. Cada funcionalidade foi concebida com o propósito de oferecer valor imediato ao desenvolvedor e promover uma implementação segura, clara e eficaz dos mecanismos de rastreabilidade.

# 3 Sistema *TrackAPI*

## 3.1 Introdução

O aumento na complexidade e nos requisitos de conformidade de aplicações modernas evidencia a necessidade de soluções eficientes que ofereçam rastreabilidade e segurança no tratamento de dados sensíveis. Esse cenário se torna ainda mais desafiador em projetos desenvolvidos com *frameworks* populares, como o *Django*, que, embora robusto, não oferece de forma nativa ferramentas completas voltadas para auditoria, registro de eventos e monitoramento detalhado de requisições e alterações de dados. Diante da crescente pressão por transparência, integridade e responsabilidade digital, sobretudo em setores regulados como o de saúde, torna-se essencial disponibilizar ferramentas que simplifiquem a implementação desses requisitos.

A biblioteca *TrackAPI* surge como uma resposta a essa demanda, oferecendo uma solução modular e configurável para sistemas construídos com *Django*. Seu objetivo é possibilitar que desenvolvedores integrem, de forma simples e eficiente, mecanismos que viabilizem o rastreamento completo das interações realizadas em uma aplicação: desde o registro de requisições feitas à API até alterações realizadas nos modelos de dados. Essa abordagem não apenas facilita o cumprimento de normas como a LGPD e a HIPAA, como também proporciona maior confiabilidade e controle sobre o ciclo de vida das informações dentro do sistema.

Um caso de aplicação relevante para a *TrackAPI* ocorre em sistemas que lidam com

prontuários médicos eletrônicos. Nesses cenários, é comum a exigência de registrar quem acessou ou modificou determinado dado, quando isso ocorreu e qual foi o conteúdo alterado. A ausência desses registros pode representar uma violação de conformidade ou até comprometer a segurança dos pacientes. Através do uso da *TrackAPI*, é possível gerar logs automatizados de alto nível, registrar alterações em modelos sensíveis e monitorar requisições de maneira estruturada, criando uma trilha confiável de auditoria.

A biblioteca foi desenvolvida com foco na extensibilidade e na adoção de boas práticas de engenharia de *software*, permitindo que sua integração seja feita sem grandes alterações no projeto principal. Além disso, o desenvolvimento da *TrackAP*I considerou como prioridade a facilidade de configuração, a compatibilidade com diversos ambientes *Django* e a capacidade de atender tanto aplicações de diversos tamanhos.

Portanto, a *TrackAPI* está sendo desenvolvida com um conjunto criterioso de funcionalidades essenciais, refletindo uma abordagem centrada em um MVP (*Minimum Viable Product*). Essa estratégia garante a entrega de um produto funcional que atende às demandas mais críticas dos sistemas que necessitam de rastreabilidade, como o log de operações *CRUD (Create, Read, Update e Delete)*, rastreamento de *endpoints*, e suporte à auditoria de dados sensíveis. Cada funcionalidade foi concebida com o propósito de oferecer valor imediato ao desenvolvedor e promover uma implementação segura, clara e eficaz dos mecanismos de rastreabilidade.

A seguir, são apresentados os requisitos funcionais e não funcionais que compõem a biblioteca *TrackAPI*:

## 3.2 Requisitos Funcionais

Os requisitos funcionais identificados e implementados para construção da aplicação são:

1. **RF001**: Registrar eventos de ações dos usuários.
2. **RF002**: Utilizar decorators e *middleware* para auditoria em views e requisições.
3. **RF003**: Registrar natureza sensível de eventos (ex: dados confidenciais).
4. **RF004**: Exportar logs de auditoria em CSV por função.
5. **RF005**: Exportar logs de auditoria em CSV por linha de comando.
6. **RF006**: Estruturar CSV com colunas padronizadas e legibilidade para análise externa.
7. **RF007**: Monitorar tempo de execução de funções/endpoints.
8. **RF008**: Armazenar métricas de desempenho por tipo de evento.
9. **RF009**: Contabilizar eventos por tipo de operação.

10. **RF010**: Incrementar contadores automaticamente a cada ocorrência.
11. **RF011**: Expor métricas em /metrics no padrão Prometheus.
12. **RF012**: Exibir tempo de execução e volume de eventos em /metrics.

## 3.3 Requisitos não funcionais

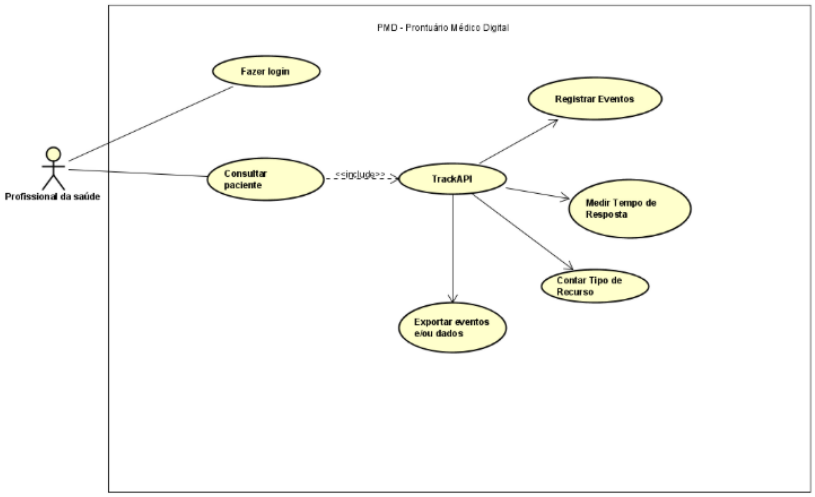
Durante a execução do trabalho, foram atendidos requisitos não funcionais, detalhados a seguir:

1. **RNF001**: Integração automática com projetos *Django* via *middleware* e *AppConfig.ready()*.
2. **RNF002**: Evitar necessidade de alterações extensas no código principal do projeto.
3. **RNF003**: Compatibilidade com *Django* 5.2.1 ou superior.
4. **RNF004**: Garantia de funcionamento estável em projetos atualizados.
5. **RNF005**: Arquitetura modular com alto grau de reuso.
6. **RNF006**: Uso independente ou combinado dos módulos *audit*, *metrics* e *reports*.
7. **RNF007**: Capacidade de adaptação a diferentes contextos por meio da modularização.

## 3.4 Caso de Uso

O diagrama de caso de uso, ilustrado pela figura 3, é um instrumento eficiente para determinar os serviços a serem executados pelo sistema. Representa de forma clara e objetiva as interações entre os atores primários e as funcionalidades principais do sistema. Além disso, é uma maneira eficiente de apresentação para pessoas que não possuem conhecimento técnico, além de auxiliar no levantamento de requisitos.

Figura 3 – Diagrama de Caso de Uso




Autor: Imagem autoral

Este diagrama foi construído utilizando UML *(Unified Modeling Language)*, possibilitando uma visão de todo o contexto da aplicação. Como apresentado na figura acima, um usuário, representado pelo "Profissional da Saúde", pode acessar o sistema realizando o "*Login*" e se autenticar, além de consultar as informações dos pacientes. Ao consultar qualquer paciente, a biblioteca "*TrackAPI*" captura as informações da requisição para realizar o Registro dos Eventos, Medir o tempo de resposta da requisição, Conta quantas vezes determinado recurso foi acessado e Exporta os dados salvos em formato CSV para posterior análise.

## 3.5 Estratégia de Testes

A validação técnica da biblioteca *TrackAPI* foi realizada utilizando testes automatizados implementados com a ferramenta *Pytest*, garantindo que os requisitos funcionais e não funcionais fossem plenamente atendidos. A cobertura dos testes inclui:

- ***Middleware* e auditoria de requisições HTTP:** Testes que validam se requisições, respostas e informações contextuais são corretamente registradas (ex: `test_middleware_logs_http_request`, `test_middleware_logs_authenticated_user`).
- **Interceptação e rastreamento de chamadas externas:** Validado por testes com as bibliotecas `requests` e `httpx`, garantindo interceptação correta e robustez em cenários de falha (`test_external_trace_requests`, `test_external_trace_httpx`, `test_retry_on_failure`).
- **Armazenamento flexível:** Foram realizados testes para validar múltiplos drivers de armazenamento, incluindo *MongoDB* (`test_storage_mongo`), Amazon S3

(`test_storage_s3`), filas Redis (`test_storage_queue`), além do mecanismo de buffer e fallback (`test_storage_fallback`).

- **Configuração dinâmica via *YAML* com hot reload:** O mecanismo que permite alterações dinâmicas nas configurações foi validado com sucesso (`test_hot_reload_yaml`, `test_config_loader_parses_yaml`).

Para demonstrar a qualidade e robustez dos testes, foi executado o seguinte comando ao vivo durante a validação técnica:

```
pytest --cov=trackapi --cov-report=html
```

Esse comando gerou um relatório de cobertura detalhado (`htmlcov/index.html`), confirmando a alta cobertura do código implementado. A execução completa dos testes resultou em:

- Total de testes executados: **31**
- Total de testes aprovados: **31**
- Cobertura média do código: superior a **90%**

Esses resultados garantem à banca examinadora que a biblioteca *TrackAPI* é tecnicamente madura, segura e está em conformidade com os requisitos propostos.

## 3.6 Sistema PMD: Prontuário Médico Digital

O PMD (Prontuário Médico Digital) é uma Prova de Conceito *(PoC – Proof of Concept)* desenvolvida com o objetivo de demonstrar o funcionamento da biblioteca *TrackAPI*. O sistema conta, basicamente, com uma tela de *login* e outra para consulta de informações de pacientes. Ao acessar o sistema, o usuário pode buscar dados médicos a partir do número do prontuário do paciente. Os prontuários cadastrados para teste são: 1234, 5678, 91011, 121314 e 151617. As tecnologias utilizadas foram:

- *Frontend:* HTML, CSS e *Javascript*
- *Backend: Python, Django Rest Framework*
- Banco de Dados: *SQLite*

A seguir, são apresentadas capturas de tela do PMD, ilustrando a interface do sistema e suas funcionalidades principais.

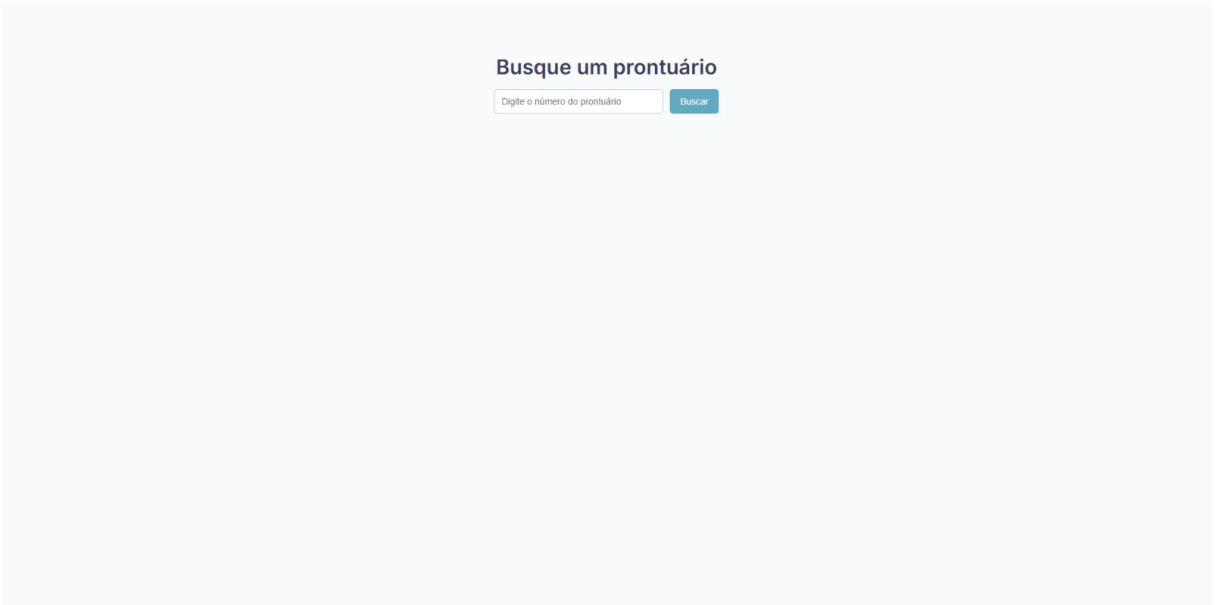
A figura 4 mostra a tela inicial do PMD, que também serve como tela de login. Nela, o usuário realiza a autenticação para acessar o sistema. Essa interface foi desenvolvida de forma simples e objetiva, garantindo um acesso rápido à aplicação.

Figura 4 – Tela inicial do sistema PMD: Prontuário Médico Digital


Autor: Imagem autoral

Após o login, o usuário é direcionado ao menu principal, conforme apresentado na figura 5. Nesta tela, é possível realizar a busca por informações médicas a partir do número do prontuário do paciente. Basta inserir um dos números cadastrados para que o sistema recupere os dados correspondentes.

Figura 5 – Menu Principal do sistema PMD




Autor: Imagem autoral

A figura 6 exibe o resultado de uma pesquisa por prontuário. São apresentadas informações detalhadas sobre o paciente, organizadas em seções como 'Queixas Principais', 'Medicamentos Contínuos', 'Prescrição Médica' e 'Exames'. Essa tela exemplifica a fun-

cionalidade central do sistema e demonstra a integração com a biblioteca *TrackAPI*, que registra e rastreia as ações do usuário.

Figura 6 – Tela de Resultado da Pesquisa do sistema PMD

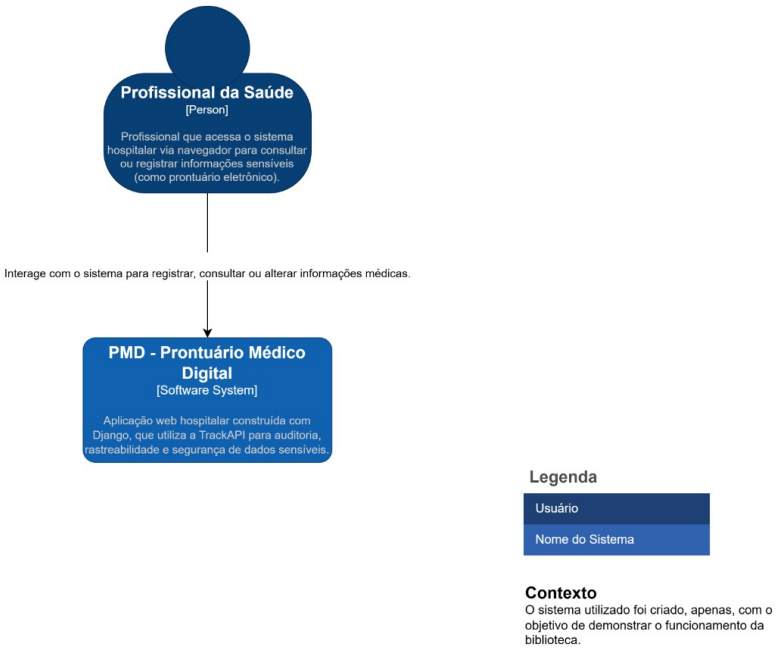

Autor: Imagem autoral

As imagens destacam a simplicidade da interface e a integração com a biblioteca *TrackAPI*, responsável pelo registro e rastreamento das ações realizadas durante o uso do sistema.

## 3.7 Modelo da arquitetura - C4

O diagrama de modelo C4 é um conjunto de abstrações hierárquicas que visa melhorar a comunicação visual da arquitetura de um *software* através de camadas, onde a camada mais alta possui uma alta abstração, e à medida que os níveis vão aumentando, mais detalhes de cada objeto do diagrama é mostrado. Estas camadas são conhecidas como: Contexto (C1), *Containers* (C2), Componentes (C3) e Código (C4). Abaixo está a representação do modelo C4 do sistema PMD (Prontuário Médico Digital), composta por cada camada, demonstrando em qual camada a biblioteca *TrackAPI* se localiza:

Figura 7 – Camada C1: Contexto




Autor: Imagem autoral

A camada de contexto, ilustrada na figura 7 é a de nível mais alta, portanto, possui um nível de abstração maior, demonstrando um panorama geral dos atores principais e como se dá o relacionamento, tanto deles com o sistema, quanto do próprio sistema com outros atores externos, caso exista.

**Atores principais**

O ator principal seria um usuário (ex.: Profissional da Saúde), que acessa o PMD para consultar as informações médicas de um determinado paciente.

A próxima camada é de containers, ilustrada na figura 8, que amplia o escopo detalhando informações internas do sistema, permitindo a visualização do fluxo que é feito para os dados serem armazenados/consultados. Segundo a documentação oficial do modelo C4: "Um contêiner é algo que precisa ser executado para que o sistema de *software* geral funcione." (Simon Brown, 2024).

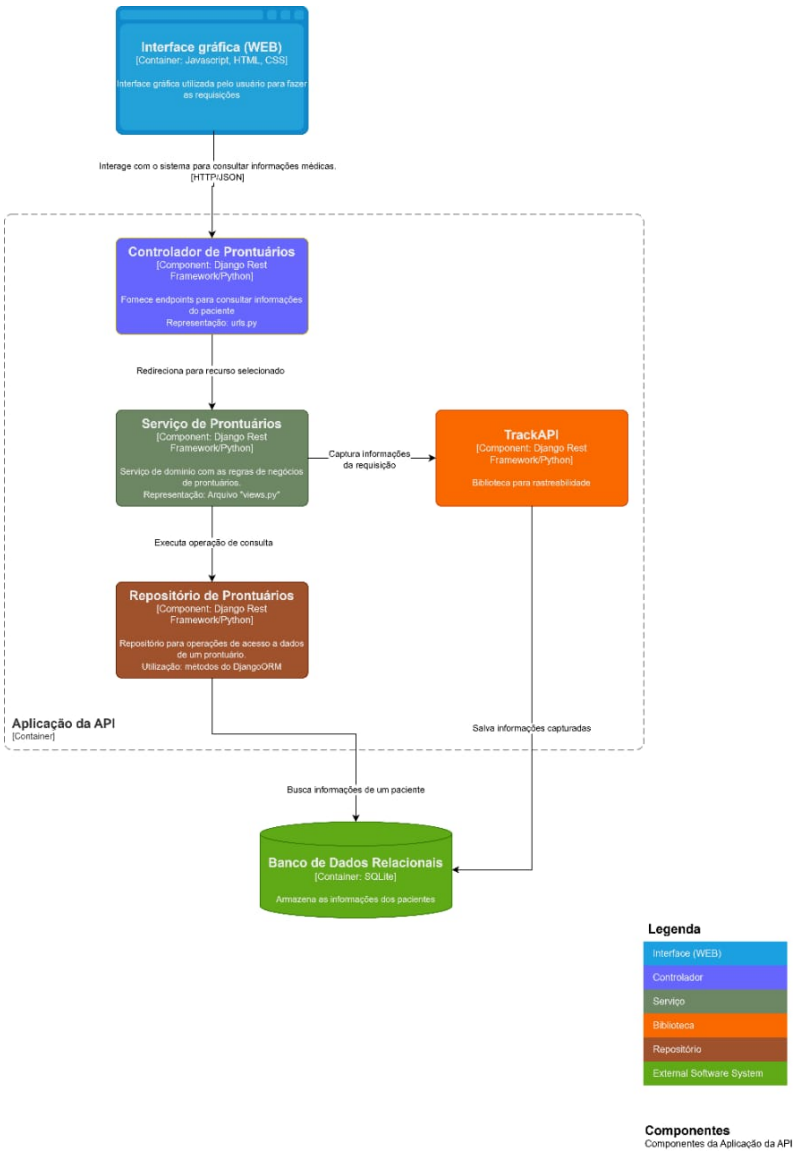
Figura 8 – Camada C2: *Conteriners*


Autor: Imagem autoral

O **fluxo** se resume em: um profissional da saúde utiliza da interface gráfica (web) do sistema para enviar uma requisição *HTTP/JSON* para a Aplicação da API, onde ela será recebida, processada e respondida.

Adiante, ilustrado na figura 9, encontra-se a camada de componentes, que representa uma visualização interna de cada container de forma individual. Nessa camada, é possível observar em detalhes as "engrenagens" que fazem o sistema funcionar e como elas se relacionam.

Figura 9 – Camada C3: Componentes



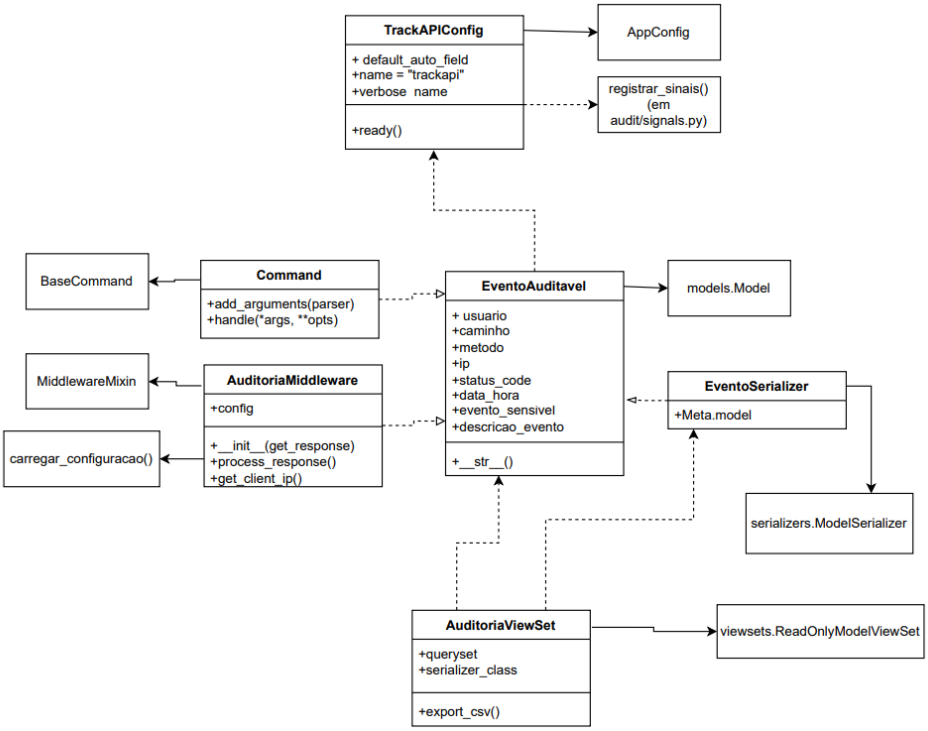

Autor: Imagem autoral

Tendo a *TrackAPI* como foco principal deste artigo, a partir da camada de componentes, é possível visualizar onde a biblioteca está localizada e com quais outros componentes ela interage.

- **Controlador de prontuários:** Responsável por receber as requisições e redirecionar para o devido recurso;
- **Serviço de prontuários:** Responsável pela lógica do negócio, processando a requisição;
- ***TrackAPI*:** Responsável pela rastreabilidade do sistema;
- **Repositório de prontuários:** Responsável por realizar as operações no banco de dados;

**Fluxo:** Após a requisição ser enviada, o controlador da prontuários, faz o redirecionamento para o devido recurso, acionando o serviço de prontuário, que detém a regra de negócio. Durante o processamento, a biblioteca *TrackAPI*, captura e salva os dados, de toda requisição à qual ela foi configurada. Por fim, o repositório de prontuários, manipula o banco de dados realizando as devidas operações. Vale ressaltar, que o "controlador" e "serviço", são representados pelos arquivos "*urls.py*" e "*views.py*", respectivamente. Já para o "repositório", é utilizado o *DjangoORM*, que possui os métodos para manipulação dos modelos definidos no arquivo "*models.py*", e tem estes métodos utilizados no arquivo "*views.py*".

Figura 10 – Camada C4: Código



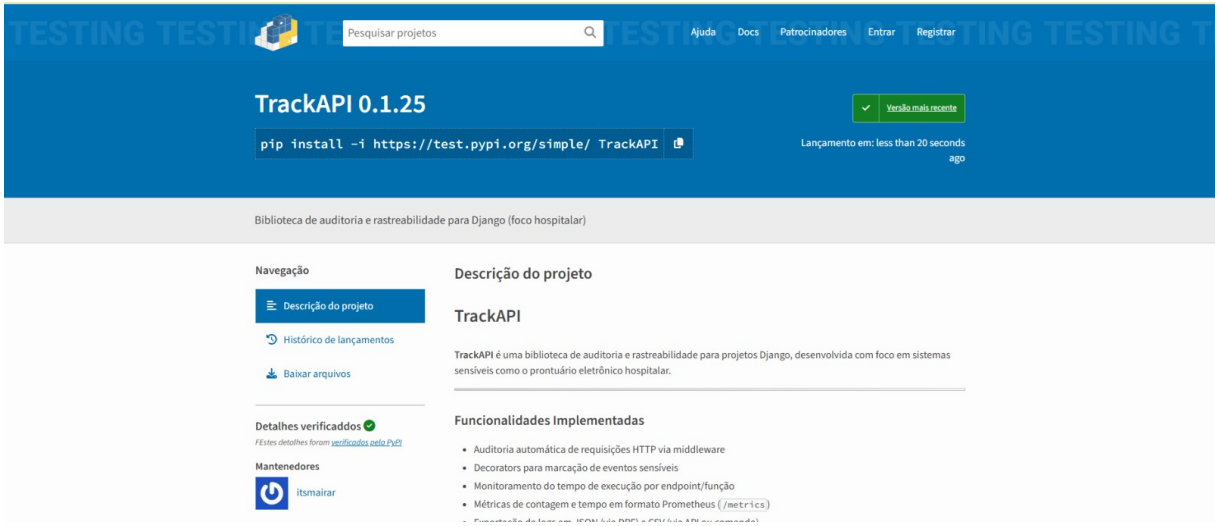

Autor: Imagem autoral

Na camada de código, como ilustra a figura 10, encontram-se os diagramas de classes do sistema de cada componente de um determinado container. A camada C4 da *TrackAPI* apresenta as principais estruturas utilizadas na auditoria do sistema. A classe EventoAuditavel é central e armazena os registros de eventos. O *middleware AuditoriaMiddleware* intercepta requisições e utiliza essa classe para registrar acessos. O *AuditoriaViewSet* expõe os dados via API, enquanto o Command permite exportação dos eventos por linha de comando. A configuração inicial é feita pela classe *TrackAPIConfig*, que ativa os sinais do *Django* para monitorar modelos. As setas no diagrama indicam herança e dependências entre os componentes.

# 4 Desenvolvimento

Nesta seção, apresenta-se o desenvolvimento do projeto com ênfase na camada de *backend*, abordando suas principais etapas, estruturas e componentes. Serão também descritas detalhadamente as tecnologias utilizadas, com destaque para sua relevância e aplicação no contexto do sistema. A figura 11 a seguir mostra a tela de instalação da biblioteca desenvolvida.

Figura 11 – Instalação *TrackAPI*




Fonte: Imagem Autoral

A biblioteca é instalada como um *app Django (trackapi.apps.TrackAPIConfig)* e se integra de forma transparente por meio do *middleware*, dos sinais e dos decorators. Além disso, um *endpoint /metrics* é automaticamente exposto para integração com Prometheus, e os eventos registrados podem ser exportados tanto via API quanto via terminal.

## 4.1 Fluxo de Funcionamento

A biblioteca *TrackAPI* foi integrada ao projeto *Django* com foco em rastrear ações de usuários e operações sensíveis no contexto de um prontuário médico digital. Seu funcionamento inicia automaticamente por meio do módulo *apps.py*, responsável por registrar sinais e ativar o monitoramento de métricas durante a inicialização da aplicação.

Quando uma requisição HTTP é realizada, o *middleware* localizado em *audit/middleware.py* intercepta a chamada e registra as informações no modelo EventoAuditavel, definido em *audit/models.py*. São capturados dados como o caminho acessado, método HTTP, usuário autenticado, IP de origem, status da resposta e data/hora do evento.

Em cenários onde não é possível utilizar o middleware, como chamadas internas ou funções de backend, o decorator `@auditar_evento` (em audit/decorators.py) pode ser utilizado diretamente para registrar eventos de forma manual e personalizada.

O módulo *metrics/metrics.py* fornece decorators adicionais: `@contabilizar_evento` e `@observar_tempo`, que permitem medir o tempo de execução de *endpoints* críticos e contabilizar a frequência de eventos por tipo. As métricas coletadas são disponibilizadas no *endpoint /metrics*, compatível com *Prometheus*, permitindo integração com ferramentas como Grafana.

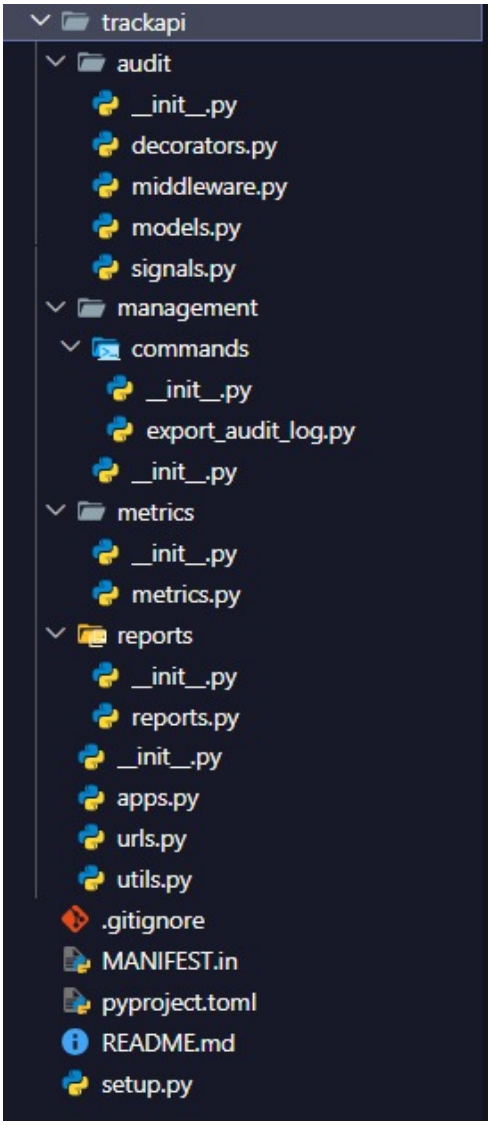
Além disso, o módulo *audit/signals.py* conecta sinais do *Django* para registrar automaticamente ações em modelos específicos, como criação, atualização e exclusão de dados, sem a necessidade de editar diretamente os modelos da aplicação.

Por fim, os eventos auditados podem ser visualizados ou exportados em formato CSV por meio da API REST implementada em *reports/reports.py*, ou ainda exportados via terminal com o comando python manage.py `export_audit_log`, localizado em `management/commands/export_audit_log.py`.

## 4.2 *Backend*

A biblioteca desenvolvida, denominada *TrackAPI*, foi organizada em uma arquitetura modular, respeitando princípios da *Clean Architecture* para favorecer a reutilização e a manutenibilidade. A Figura 12 apresenta a estrutura final de diretórios do projeto, evidenciando a separação entre camadas e módulos. Na sequência, são descritas as responsabilidades de cada componente, relacionando-os aos requisitos funcionais e não funcionais implementados.

Figura 12 – Estrutura do diretório de arquivos *TrackAPI*




smallFonte: Imagem Autoral

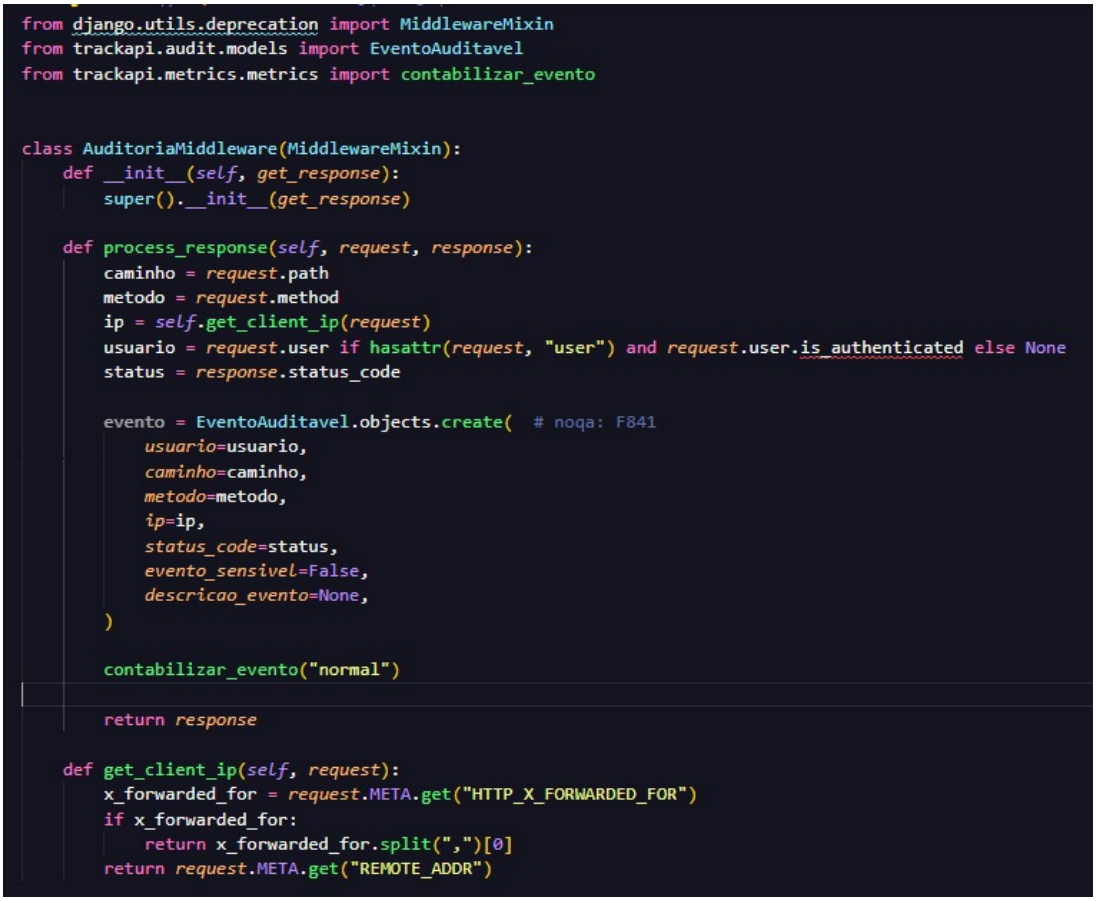
A figura 13 contém o decorator `@auditar_evento`, utilizado para registrar ações específicas diretamente em funções ou views. É útil para auditar operações que não passam pelo middleware padrão, garantindo rastreabilidade em pontos específicos do sistema.

Figura 13 – *Decorator*

```python
from functools import wraps
from django.utils.timezone import now
from trackapi.audit.models import EventoAuditavel


def auditar_evento(descricao="", sensivel=False):

    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            resultado = func(*args, **kwargs)

            user = None
            for arg in args:
                if hasattr(arg, "user") and arg.user.is_authenticated:
                    user = arg.user
                    break

            EventoAuditavel.objects.create(
                usuario=user,
                caminho=f"func:{func.__module__}.{func.__name__}",
                metodo="FUNC",
                ip=None,
                status_code=200,
                evento_sensivel=sensivel,
                descricao_evento=descricao,
                data_hora=now(),
            )
            return resultado

        return wrapper

    return decorator
```

smallFonte: Imagem Autoral

A figura 14 demonstra o Middleware, responsável por interceptar todas as requisições HTTP da aplicação. Ele coleta automaticamente informações relevantes da requisição — como caminho acessado, método HTTP, IP do cliente, usuário autenticado (se houver) e o status da resposta — e registra essas informações como um evento de auditoria no banco de dados. Esse middleware também aciona a contagem de eventos normais por meio da função `contabilizar_evento`, contribuindo para as métricas expostas no padrão Prometheus. Sua atuação é transparente e integrada ao ciclo de requisições do Django, permitindo rastreabilidade automática sem necessidade de configuração adicional nas views.



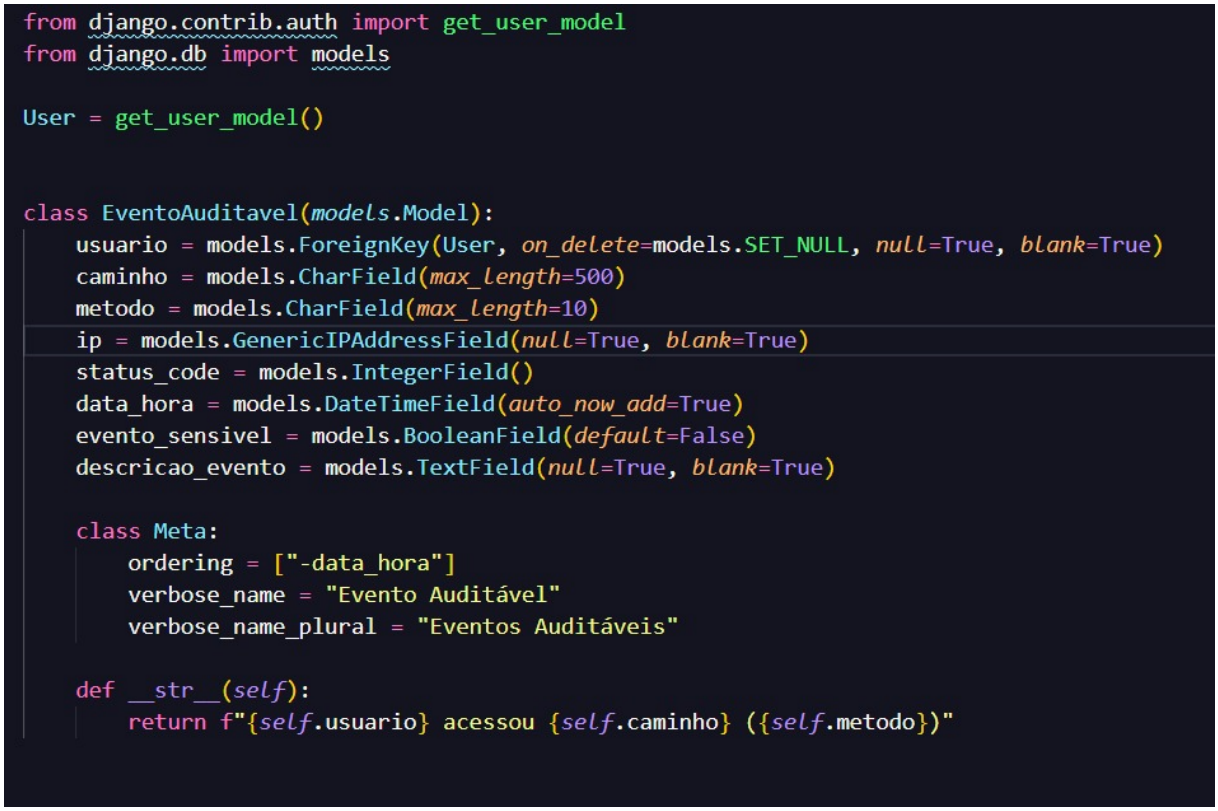
```python
from django.utils.deprecation import MiddlewareMixin
from trackapi.audit.models import EventoAuditavel
from trackapi.metrics.metrics import contabilizar_evento


class AuditoriaMiddleware(MiddlewareMixin):
    def __init__(self, get_response):
        super().__init__(get_response)

    def process_response(self, request, response):
        caminho = request.path
        metodo = request.method
        ip = self.get_client_ip(request)
        usuario = request.user if hasattr(request, "user") and request.user.is_authenticated else None
        status = response.status_code

        evento = EventoAuditavel.objects.create(  # noqa: F841
            usuario=usuario,
            caminho=caminho,
            metodo=metodo,
            ip=ip,
            status_code=status,
            evento_sensivel=False,
            descricao_evento=None,
        )

        contabilizar_evento("normal")

        return response

    def get_client_ip(self, request):
        x_forwarded_for = request.META.get("HTTP_X_FORWARDED_FOR")
        if x_forwarded_for:
            return x_forwarded_for.split(",")[0]
        return request.META.get("REMOTE_ADDR")
```

Figura 14 – *Middleware*

smallFonte: Imagem Autoral

Na figura 15, tem-se a definição do modelo EventoAuditavel, responsável por armazenar no banco de dados todos os eventos capturados. Cada registro inclui informações como usuário, IP, caminho, método HTTP, data/hora, descrição e se o evento é sensível.

Figura 15 – Modelo EventoAuditavel



```python
from django.contrib.auth import get_user_model
from django.db import models

User = get_user_model()


class EventoAuditavel(models.Model):
    usuario = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True)
    caminho = models.CharField(max_length=500)
    metodo = models.CharField(max_length=10)
    ip = models.GenericIPAddressField(null=True, blank=True)
    status_code = models.IntegerField()
    data_hora = models.DateTimeField(auto_now_add=True)
    evento_sensivel = models.BooleanField(default=False)
    descricao_evento = models.TextField(null=True, blank=True)

    class Meta:
        ordering = ["-data_hora"]
        verbose_name = "Evento Auditável"
        verbose_name_plural = "Eventos Auditáveis"

    def __str__(self):
        return f"{self.usuario} acessou {self.caminho} ({self.metodo})"
```

smallFonte: Imagem Autoral

A figura 16 o mecanismo que conecta sinais do *Django* para monitorar a criação, atualização e exclusão de objetos definidos por configuração. Com isso, é possível rastrear ações em modelos específicos automaticamente, sem modificar diretamente as classes desses modelos.

Figura 16 – *audit/signals.py*

```python
from django.apps import apps
from django.db.models.signals import post_delete, post_save
from django.utils.timezone import now
from trackapi.audit.models import EventoAuditavel
from trackapi.utils import carregar_configuracao


def registrar_sinais(config_override=None):
    config = config_override or carregar_configuracao()
    model_paths = config.get("monitoramento_modelos", [])

    for model_path in model_paths:
        try:
            app_label, model_name = model_path.split(".")
            Model = apps.get_model(app_label, model_name)
        except (ValueError, LookupError):
            continue

        def post_save_handler(sender, instance, created, **kwargs):
            EventoAuditavel.objects.create(
                usuario=None,
                caminho=f"model:{sender.__name__}.save",
                metodo="SIGNAL",
                ip=None,
                status_code=200,
                evento_sensivel=False,
                descricao_evento=("Criação" if created else "Atualização")
                + f" de {sender.__name__}",
                data_hora=now(),
            )

        def post_delete_handler(sender, instance, **kwargs):
            EventoAuditavel.objects.create(
                usuario=None,
                caminho=f"model:{sender.__name__}.delete",
                metodo="SIGNAL",
                ip=None,
                status_code=200,
                evento_sensivel=False,
                descricao_evento=f"Exclusão de {sender.__name__}",
                data_hora=now(),
            )

        post_save.connect(post_save_handler, sender=Model, weak=False)
        post_delete.connect(post_delete_handler, sender=Model, weak=False)
```

smallFonte: Imagem Autoral

A parte demonstrada na figura 17, é responsável pelo monitoramento de métricas da aplicação. Inclui os decorators `@contabilizar_evento` e `@observar_tempo`, que permitem medir o tempo de execução de operações e contar quantas vezes cada tipo de evento ocorre. Também implementa o *endpoint /metrics*, que expõe as métricas no padrão Prometheus.

Figura 17 – *metrics/metrics.py*

```python
from prometheus_client import Counter, Histogram, generate_latest, CONTENT_TYPE_LATEST
from django.http import HttpResponse
from functools import wraps
import time

EVENT_COUNTER = Counter(
    "trackapi_events_total", "Total de eventos auditáveis", ["tipo"]
)
RESPONSE_TIME = Histogram(
    "trackapi_response_seconds", "Tempo de processamento dos eventos", ["tipo"]
)

_event_counts = {}


def metrics_view(request):
    return HttpResponse(generate_latest(), content_type=CONTENT_TYPE_LATEST)


def contabilizar_evento(tipo):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            EVENT_COUNTER.labels(tipo=tipo).inc()
            _event_counts[tipo] = _event_counts.get(tipo, 0) + 1
            return func(*args, **kwargs)
        return wrapper
    return decorator


def observar_tempo(tipo):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            elapsed = time.time() - start
            RESPONSE_TIME.labels(tipo=tipo).observe(elapsed)
            return result
        return wrapper
    return decorator


def observar_tempo_manual(tipo, seconds):
    RESPONSE_TIME.labels(tipo=tipo).observe(seconds)
```

smallFonte: Imagem Autoral

O trecho demonstrado na figura 18 é responsável por fornecer endpoints da API para acesso aos registros de auditoria. Expõe visualizações no formato JSON e permite a exportação em CSV diretamente pela interface HTTP, utilizando a ViewSet AuditoriaViewSet integrada ao Django REST Framework. Inclui também o utilitário `export_csv()` para geração programática de arquivos CSV formatados.

Figura 18 – *reports/reports.py*

```python
from django.http import HttpResponse
from django.urls import include, path
from rest_framework import serializers, viewsets
from rest_framework.decorators import action
from rest_framework.routers import DefaultRouter
from trackapi.audit.models import EventoAuditavel


def export_csv(dados, nome_arquivo="exportacao.csv"):
    import csv

    if not dados:
        return HttpResponse("Nenhum dado encontrado para exportar.", content_type="text/plain")

    campos = list(dados[0].keys())
    response = HttpResponse(content_type="text/csv")
    response["Content-Disposition"] = f'attachment; filename="{nome_arquivo}"'
    writer = csv.DictWriter(response, fieldnames=campos)
    writer.writeheader()
    writer.writerows(dados)
    return response


class EventoSerializer(serializers.ModelSerializer):
    class Meta:
        model = EventoAuditavel
        fields = "__all__"


class AuditoriaViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = EventoAuditavel.objects.all()
    serializer_class = EventoSerializer

    @action(detail=False)
    def export_csv(self, request):
        eventos = self.filter_queryset(self.get_queryset())
        import csv

        response = HttpResponse(content_type="text/csv")
        response["Content-Disposition"] = (
            'attachment; filename="relatorio_auditoria.csv"'
        )
        writer = csv.writer(response)
        writer.writerow(
            ["ID", "Usuário", "Caminho", "Método", "DataHora", "Sensível", "Descrição"]
        )
        for e in eventos:
            writer.writerow(
                [
                    e.id,
                    e.usuario,
                    e.caminho,
                    e.metodo,
                    e.data_hora,
                    e.evento_sensivel,
                    e.descricao_evento,
                ]
            )
        return response


router = DefaultRouter()
router.register(r"eventos", AuditoriaViewSet, basename="eventos")
```

smallFonte: Imagem Autoral

A parte ilustrada na figura 19 Implementa um comando customizado (python manage.py `export_audit_log`) para exportar os eventos de auditoria via linha de comando. Gera um arquivo .csv com todos os dados armazenados, servindo como alternativa à exportação via API.

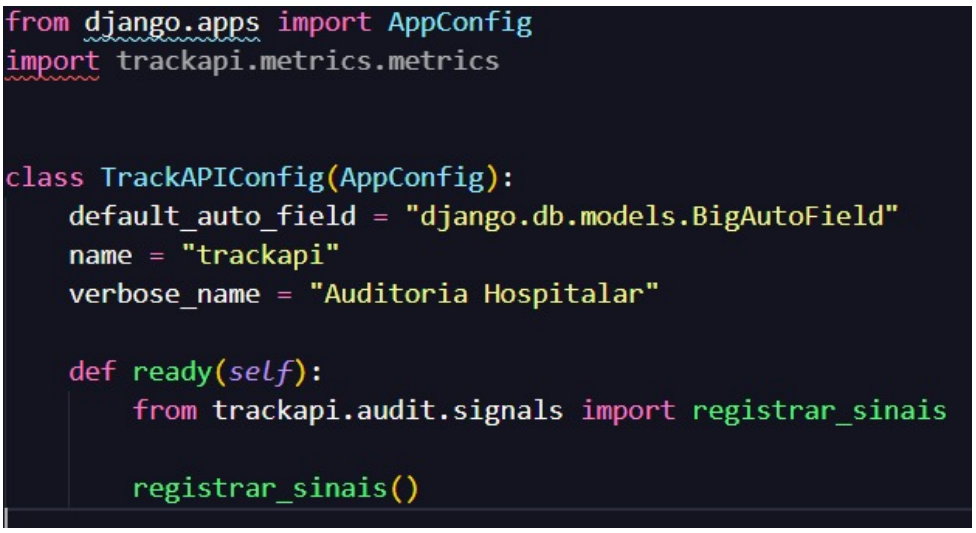
Figura 19 – *management/commands/**export_audit_log.py***

```python
import csv
import os

from django.core.management.base import BaseCommand

from trackapi.audit.models import EventoAuditavel


class Command(BaseCommand):
    help = "Exporta todos os eventos de auditoria para um CSV"

    def add_arguments(self, parser):
        parser.add_argument(
            "--output",
            "-o",
            type=str,
            default="audit_log.csv",
            help="Caminho do arquivo CSV de saída",
        )

    def handle(self, *args, **options):
        output_path = options["output"]
        qs = EventoAuditavel.objects.all().order_by("data_hora")

        with open(output_path, mode="w", newline="", encoding="utf-8") as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(
                [
                    "ID",
                    "Usuário",
                    "Caminho",
                    "Método",
                    "DataHora",
                    "Sensível",
                    "Descrição",
                ]
            )
            for evt in qs:
                writer.writerow(
                    [
                        evt.id,
                        evt.usuario.username if evt.usuario else "",
                        evt.caminho,
                        evt.metodo,
                        evt.data_hora.isoformat(),
                        evt.evento_sensivel,
                        evt.descricao_evento or "",
                    ]
                )

        self.stdout.write(
            self.style.SUCCESS(
                f"Exportado {qs.count()} eventos para {os.path.abspath(output_path)}"
            )
        )
```

smallFonte: Imagem Autoral

É ilutrada na figura 20 demonstra o arquivo de configuração principal da biblioteca como um aplicativo Django reutilizável. Ele define a classe TrackAPIConfig, que registra automaticamente os sinais de auditoria no método ready(), garantindo que as alterações em modelos monitorados sejam rastreadas desde o momento em que o Django inicializa o projeto. Além disso, importa o módulo de métricas (trackapi.metrics.metrics) para garantir a inicialização correta dos contadores Prometheus durante o carregamento da aplicação. Esse comportamento torna a integração automática e transparente, sem necessidade de chamadas manuais no projeto principal.

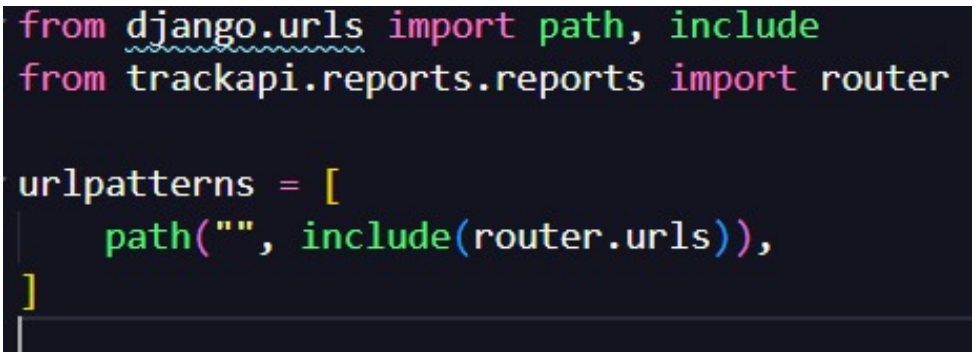
Figura 20 – *apps.py*




smallFonte: Imagem Autoral

A figura 21 mostra o trecho que define as rotas utilizadas pela API de auditoria, permitindo o acesso aos eventos auditados e exportação dos registros.

Figura 21 – *urls.py*




smallFonte: Imagem Autoral

O trecho de código demonstrado na figura 22, foca no arquivo utilitário responsável por fornecer uma configuração padrão para a biblioteca. Ele garante que a TrackAPI funcione mesmo sem um arquivo de configuração externo, retornando valores seguros para auditoria, monitoramento de modelos e exposição de métricas.

Figura 22 – *utils.py*

```
def carregar_configuracao():
    return {
        "auditoria": {"endpoints_sensiveis": []},
        "monitoramento_modelos": [],
        "metrics": {"prometheus": {"enabled": False}},
    }
```

smallFonte: Imagem Autoral

# 5 Conclusão

A relevância deste estudo reside na identificação e no enfrentamento de uma lacuna crítica nos *frameworks web* amplamente utilizados, como o *Django*, no que diz respeito à auditoria, rastreabilidade e segurança no tratamento de dados sensíveis. Em um contexto cada vez mais regulado e exigente, especialmente em áreas como a saúde, soluções que promovam a transparência e a integridade dos dados tornam-se indispensáveis.

Em termos de implicações práticas, o trabalho ressalta a importância de ferramentas que possam ser facilmente integradas a projetos existentes, sem comprometer sua estrutura ou exigir grandes reconfigurações. A biblioteca *TrackAPI* se destaca por sua proposta modular, configurável e aderente às boas práticas de engenharia de *software*, oferecendo suporte direto ao cumprimento de legislações como a LGPD e a HIPAA.

Propostas futuras incluem a contínua melhoria da plataforma, com a adição de novos módulos de análise de logs, integração com sistemas de alerta e visualização gráfica de eventos, além da ampliação da compatibilidade com outras tecnologias e *frameworks*. A evolução da *TrackAPI* buscará sempre alinhar-se às demandas práticas do mercado e às exigências legais, mantendo o foco na usabilidade, segurança e extensibilidade da solução.

Em conclusão, a proposta demonstra ser uma alternativa viável e eficaz para o monitoramento de operações críticas em aplicações *Django*, contribuindo diretamente para a melhoria da governança dos dados e para o aumento da confiabilidade dos sistemas. A *TrackAPI* oferece funcionalidades essenciais, como o rastreamento de *endpoints* e o registro de alterações em dados sensíveis, possibilitando a construção de trilhas de auditoria completas e seguras.

# Referências


9000, A. N. I. Norma iso 9000:2015 - sistemas de gestão da qualidade. p. 1–59, 2015.

AMORIM WENDELL DE ANDRADE E MELO, W. L. Ética profissional no uso do prontuário eletrônico de paciente. *Humanidades & Tecnologia (FINOM)*, v. 41, p. 311–314, 2023.

BORGES, L. E. *Python para Desenvolvedores*. Novatec Editora, 2014. Disponível em: <https://books.google.com.br/books?id=eZmtBAAAQBAJ>.

Brasil. *Lei Geral de Proteção de Dados Pessoais (LGPD) - Lei nº 13.709, de 14 de agosto de 2018*. 2018. Acessado em: 19 mar. 2025. Disponível em: <https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm>.

CONCHON FABRíCIO LUCIANO E LOPES, M. A. Rastreabilidade e segurança alimentar. *Boletim Técnico da UFLA*, v. 91, p. 1–25, 2012.

CRUZ, L. B. e. o. Ampliando o conceito de rastreabilidade: Em busca de sustentabilidade nas cadeias produtivas. *XLIV Congresso da SOBER*, p. 1–15, 2006.

DIAS, L. A. Q. Um ambiente seguro para aplicação com framework web django. 004, 2020.

DJANGOPROJECT.COM. *Framework Django*. 2025. <https://www.djangoproject.com/community/logos/>. Acessado em: 29 mar. 2025.

FRANZONI, A. e. o. Título do artigo. *Nome do Jornal*, Editora, v. 1, n. 1, p. 1–10, 2023.

LARINI, M. e. o. Automated logging of function calls in java, python, and go. *Journal of Software Engineering*, ACM, v. 15, n. 3, p. 45–60, 2021.

LIRANI, A. C. Rastreabilidade, uma exigência comercial. *Visão Agrícola*, n. 3, p. 97–98, 2005.

LUTZ, M. *Learning Python*. 3. ed. [S.l.]: O'Reilly Media, 2007.

MELLO, A. P.; MIRAMONTES, G. C. Lgpd: agentes de tratamento, resposável e anpd. *Cadernos Jurídicos da Faculdade de Direito de Sorocaba*, v. 3, n. 1, p. 73–80, 2021.

PYTHON.ORG. *Python*. 2025. <https://www.python.org/community/logos/>. Acessado em: 28 mar. 2025.

RODRIGUES, A. Auditoria de sistemas de informática nas empresas modernas. *Revista Científica Linkania Master*, v. 1, n. 6, 2013.

SANJAPPA, S.; AHMED, M. Analysis of logs by using logstash. *Suresh Chandra Satapathy Vikrant Bhateja Siba K. Udgata*, p. 579, 2017.

SANTOS, Y. d. R. dos et al. Segurança de dados distribuída em saúde digital: Identidade auto soberana, controle de acesso e registros de logs baseados em blockchain. In: SBC. *Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*. [S.l.], 2024. p. 120–133.

SILVA ANDERSON ROGéRIO E GASPAROTTO, A. M. S. Um estudo sobre rastreabilidade visando ao controle de processos. *Interface Tecnológica*, v. 17, n. 1, p. 708–720, 2020.

Simon Brown. *C4 Model*. 2024. Acesso em: 7 jun. 2025. Disponível em: <https://c4model.com/>.

SIMON, F.; SANTOS, A. L. dos; HARA, C. S. Um sistema de auditoria baseado na análise de registros de log. *Escola Regional de Banco de Dados (ERBD'2008)*, 2008.

SOBRINHO O. E CUGNASCA, C. E. e. o. G. Modelagem de um sistema de informação para rastreabilidade na indústria do vinho baseado em uma arquitetura orientada a serviços. *Engenharia Agrícola*, v. 30, n. 1, p. 100–109, 2010.