Anna Clara Rodrigues Peres

ThreatLens: Desenvolvimento de API para o *back-end*

Uberlândia - MG

Agosto / 2026

Anna Clara Rodrigues Peres

ThreatLens: Desenvolvimento de API para o *back-end*

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Universidade Federal de Uberlândia

Faculdade de Computação

Bacharelado em Sistemas de Informação

Orientador: Paulo Henrique Ribeiro Gabriel

Uberlândia - MG

Agosto / 2026

Agradecimentos

Concluir este trabalho marca o fim de uma etapa que começou muito antes da primeira linha de código do ThreatLens ser escrita, e agradecer é reconhecer que nenhuma dessas etapas foi trilhada sozinha.

Começo agradecendo à minha família – meus pais – Raquel e Walter, minhas irmãs Rhanna, Ana Paula e Andresa, meu irmão Bruno e a Caymmi – por serem a base sobre a qual toda a minha trajetória acadêmica foi construída. O apoio incondicional, a paciência nos momentos de ausência e o incentivo constante, mesmo à distância dos detalhes técnicos deste trabalho, foram o que me permitiu chegar até aqui.

Ao meu orientador, Prof. Dr. Paulo Henrique Ribeiro Gabriel, agradeço não apenas pela orientação técnica cuidadosa e pelas contribuições fundamentais ao longo de todo o desenvolvimento deste trabalho, mas também pela paciência e pela disponibilidade em cada revisão e ajuste de rota que este trabalho exigiu.

Agradeço especialmente a Pedro Miguel, parceiro no desenvolvimento do ThreatLens e responsável pelo *front-end* da aplicação – há um valor particular em construir algo junto com alguém com quem aprendi e cresci ao longo desse processo, e com quem também poderei comemorar essa conquista.

Ao PET-SI (Programa de Educação Tutorial – Sistemas de Informação), agradeço pelas oportunidades de aprendizado, pesquisa e extensão que moldaram boa parte da minha formação, muito além do que uma grade curricular tradicional ofereceria – e aos tutores e colegas que tornaram essa experiência tão marcante, minha sincera gratidão.

Por fim, agradeço aos professores da Faculdade de Computação (FACOM/UFU) que contribuíram para minha formação, e aos colegas de curso que tornaram tão difícil me despedir dessa jornada – obrigada por dividirem comigo as noites de estudo, os prazos apertados e as pequenas vitórias ao longo da graduação.

Resumo

Este trabalho apresenta o desenvolvimento do ThreatLens, uma aplicação *back-end* (API REST) responsável por centralizar, autenticar e disponibilizar, de forma segura, filtrada e agregada, mensagens de inteligência de ameaças cibernéticas já coletadas e classificadas por trabalhos anteriores desenvolvidos na Faculdade de Computação da Universidade Federal de Uberlândia (FACOM/UFU). Para isso, foi implementado um módulo de autenticação e gestão de usuários baseado em *JSON Web Token* (JWT), com *refresh tokens* persistidos e revogáveis, e um módulo de consulta de mensagens (*posts*) coletadas de canais públicos do Telegram, com suporte a filtros por fonte, relevância, categoria, período e busca por texto livre no conteúdo das mensagens, paginação, ordenação, estatísticas agregadas e nuvem de palavras. A camada de acesso a múltiplas fontes de dados foi projetada por meio do Padrão de Projeto *Strategy*, permitindo a incorporação de novas fontes sem alteração da lógica de negócio já implementada. A avaliação funcional, realizada por meio de requisições HTTP submetidas aos *endpoints* implementados, confirmou o atendimento aos requisitos funcionais e não funcionais especificados. Como resultado, a arquitetura proposta avança na direção do preenchimento da lacuna de integração identificada entre os trabalhos relacionados, por meio de uma camada de API única e extensível – ainda que apenas a fonte Telegram esteja integrada no escopo atual, ficando a incorporação de novas fontes e a integração com o módulo de alertas como direções para trabalhos futuros.

Palavras-chave: *Cyber Threat Intelligence*. API REST. Spring Boot. Segurança Cibernética. Telegram. *JSON Web Token*.

Abstract

This work presents the development of ThreatLens, a *back-end* application (REST API) responsible for centralizing, authenticating, and securely serving filtered and aggregated cyber threat intelligence messages already collected and classified by previous works developed at the School of Computer Science of the Federal University of Uberlândia (FACOM/UFU). To this end, a user authentication and management module was implemented based on *JSON Web Token* (JWT), with persisted, revocable refresh tokens, along with a module for querying messages (*posts*) collected from public Telegram channels, supporting filtering by source, relevance, category, and period, pagination, sorting, aggregated statistics, and word clouds. The multi-source data access layer was designed using the *Strategy* design pattern, allowing new sources to be incorporated without changes to the already implemented business logic. The functional evaluation, carried out through HTTP requests submitted to the implemented *endpoints*, confirmed compliance with the specified functional and non-functional requirements. As a result, the proposed architecture advances toward bridging the integration gap identified among the related works through a single, extensible API layer – although only the Telegram source is currently integrated, leaving the incorporation of new sources and integration with the alert module as directions for future work.

Keywords: *Cyber Threat Intelligence*. REST API. Spring Boot. Cybersecurity. Telegram. *JSON Web Token*.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface* (Interface de Programação de Aplicações)
CTI	*Cyber Threat Intelligence* (Inteligência de Ameaças Cibernéticas)
DBSCAN	*Density-Based Spatial Clustering of Applications with Noise*
DTO	*Data Transfer Object* (Objeto de Transferência de Dados)
FACOM	Faculdade de Computação
IP	*Internet Protocol* (Protocolo de Internet)
IoC	*Indicator of Compromise* (Indicador de Comprometimento)
JDBC	*Java Database Connectivity*
JPA	*Jakarta Persistence API*
JVM	*Java Virtual Machine* (Máquina Virtual Java)
JWT	*JSON Web Token*
KNN	*K-Nearest Neighbors* (K Vizinhos Mais Próximos)
LDA	*Latent Dirichlet Allocation* (Alocação Latente de Dirichlet)
LTS	*Long-Term Support* (Suporte de Longo Prazo)
ORM	*Object-Relational Mapping* (Mapeamento Objeto-Relacional)
OTP	*One-Time Password* (Senha de Uso Único)
PLN	Processamento de Linguagem Natural
REST	*Representational State Transfer*
SMTP	*Simple Mail Transfer Protocol* (Protocolo Simples de Transferência de Correio)
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
STIX	*Structured Threat Information eXpression*
SVM	*Support Vector Machine* (Máquina de Vetores de Suporte)

TAXII	*Trusted Automated eXchange of Indicator Information*
TCC	Trabalho de Conclusão de Curso
TF-IDF	*Term Frequency–Inverse Document Frequency*

Sumário

1 Introdução

O crescimento da conectividade global ampliou, na mesma proporção, a superfície de exposição de indivíduos e organizações a ataques cibernéticos. No Brasil, esse cenário se manifesta de forma concreta: em 2021, um ataque cibernético comprometeu sistemas do Ministério da Saúde, como o e-SUS e o ConecteSUS, expondo dados de 223 milhões de pessoas, incluindo CPFs, nomes, datas de nascimento e informações de veículos (G1, 2021). Mais recentemente, o país se consolidou como o principal alvo de ataques cibernéticos da América Latina, tendo sido alvo de 356 bilhões de tentativas de ataque ao longo de 2024, segundo dados do laboratório de inteligência de ameaças da Fortinet (Security Leaders, 2025). Diante desse cenário, a Inteligência de Ameaças Cibernéticas (do inglês *Cyber Threat Intelligence* – CTI) consolidou-se como uma abordagem fundamental para que equipes de segurança antecipem, identifiquem e respondam a potenciais ataques antes que estes se concretizem.

Uma das fontes mais relevantes para a obtenção de CTI são os dados não estruturados publicados em redes sociais e em fóruns da *Dark Web*. Extrair valor desses dados, no entanto, é um desafio: o volume de mensagens é elevado, o conteúdo é majoritariamente irrelevante para fins de segurança e a identificação manual de postagens relevantes é inviável em escala. Segundo o relatório *X-Force Threat Intelligence Index*, da IBM, os ciberataques ocorrem majoritariamente por meio da coleta de credenciais, phishing e exploração de vulnerabilidades conhecidas, o que reforça a necessidade de monitoramento contínuo dessas fontes (IBM, 2023). Por isso, técnicas de mineração de texto, Processamento de Linguagem Natural (PLN) e aprendizado de máquina têm sido amplamente empregadas para automatizar a coleta, classificação e análise dessas informações (RAHMAN; HEZAVEH; WILLIAMS, 2023).

No contexto da Faculdade de Computação da Universidade Federal de Uberlândia (FACOM/UFU), essa problemática foi endereçada por uma sequência de trabalhos acadêmicos que, em conjunto, implementam um pipeline de inteligência de ameaças: Vieira (2025) desenvolveu um bot para a captura automatizada de mensagens de canais públicos do Telegram; Jesus Filho (2023) propôs um modelo de aprendizado de máquina supervisionado para classificar essas mensagens quanto à sua relevância para a cibersegurança; Reis (2024) aplicou aprendizado não supervisionado para identificar as temáticas predominantes entre as mensagens já classificadas; e Borges (2025) implementou um sistema de geração de alertas que notifica usuários sobre novas postagens relevantes. Cada uma dessas etapas foi desenvolvida de forma independente, resultando em módulos funcionais, porém ainda não integrados em um sistema único que permita a analistas de segurança consultar, filtrar e explorar esses dados de forma centralizada e segura.

É diante dessa lacuna que se justifica o presente trabalho. O objetivo geral deste trabalho é o desenvolvimento de uma aplicação *back-end* (API REST), denominada ThreatLens, responsável por centralizar, autenticar e disponibilizar de forma segura, filtrada e agregada as mensagens já coletadas e classificadas pelos trabalhos apresentados no parágrafo anterior – atuando como a camada de API que conecta os dados produzidos pelas etapas de coleta e classificação a um eventual consumidor final (*front-end* ou outro sistema cliente) e preenchendo a lacuna de integração identificada entre esses trabalhos.

Para alcançar esse objetivo, foram definidos os seguintes objetivos específicos:

- Implementar um módulo de autenticação e gestão de usuários com registro, verificação de e-mail, login *stateless* baseado em JWT com *refresh token* persistido e revogável, recuperação e troca de senha, e controle de acesso por papéis;
- Implementar um módulo de consulta de mensagens (*posts*) com suporte a filtros por fonte, relevância, categoria, período e busca por texto livre no conteúdo das mensagens, paginação e ordenação;
- Disponibilizar um *endpoint* de estatísticas agregadas, retornando totais, percentuais de relevância e distribuição por fonte e por nível de relevância;
- Disponibilizar um *endpoint* de nuvem de palavras, calculando a frequência dos termos mais utilizados no conteúdo dos *posts* filtrados;
- Projetar a camada de acesso a dados de forma extensível, por meio do Padrão Strategy, permitindo a incorporação de novas fontes sem alteração da lógica de negócio já implementada.

O restante deste documento está organizado da seguinte forma: o Capítulo 2 apresenta os trabalhos relacionados, situando a aplicação proposta em relação a eles e a plataformas de CTI já estabelecidas na literatura e no mercado. O Capítulo 4 descreve as principais tecnologias empregadas no desenvolvimento da aplicação. O Capítulo 5 apresenta os requisitos levantados, a modelagem de dados e da arquitetura, e a implementação dos módulos que compõem o sistema. O Capítulo 6 apresenta a avaliação funcional da aplicação, demonstrando o atendimento aos requisitos especificados por meio de requisições submetidas aos *endpoints* implementados. Por fim, o Capítulo 7 retoma os objetivos definidos nesta introdução, discute as contribuições e limitações do trabalho e aponta direções para trabalhos futuros.

2 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados a este TCC. A seção 2.1 apresenta os quatro trabalhos do mesmo projeto de pesquisa em que este TCC está inserido, na ordem em que ocorrem no pipeline de inteligência de ameaças: captura, classificação, análise e notificação. A seção 2.2 situa a aplicação proposta em relação a plataformas de CTI já estabelecidas na literatura e no mercado, e a seção 2.3 discute como ela se posiciona frente a todos esses trabalhos.

2.1 Trabalhos do Grupo de Pesquisa

Vieira (2025) desenvolveu um sistema de *back-end* para a captura e armazenamento de mensagens publicadas em canais públicos do Telegram, construído com NestJS, TypeScript, Prisma e PostgreSQL. Nos experimentos conduzidos, foram coletadas 760 mensagens, das quais aproximadamente 16% foram classificadas como relevantes para o contexto de cibersegurança. Esse trabalho alimenta, junto com outras fontes previstas, a base de dados que a aplicação proposta neste trabalho consome.

Jesus Filho (2023), em dissertação de mestrado, propôs um modelo para identificar se uma postagem de fóruns da *Dark Web* é relevante para a cibersegurança, combinando Indicadores de Comprometimento (IoCs), palavras-chave contextuais e análise manual na rotulagem dos dados. Entre os algoritmos testados (SVM, Regressão Logística, LightGBM e XGBoost), o LightGBM com TF-IDF Unigram apresentou os melhores resultados. Esse trabalho produz a probabilidade/relevância que qualifica cada postagem armazenada na base de dados consumida pela aplicação proposta neste trabalho.

Reis (2024) identificou as temáticas predominantes entre as postagens já classificadas por Jesus Filho (2023), aplicando aprendizado não supervisionado (K-means, DBSCAN, KNN) e modelagem de tópicos (LDA). O algoritmo K-means obteve o melhor desempenho, organizando os dados em três clusters: segurança de dados e contas, dados pessoais e identidade, e comunidade de hacking. Enquanto Jesus Filho (2023) resolve *quais* postagens são relevantes, Reis (2024) aprofunda *do que tratam* essas postagens.

Borges (2025) desenvolveu, com Django REST Framework, uma API para o gerenciamento de perfis parametrizáveis de alerta (palavras-chave, fontes e limiar de relevância), que notifica usuários por e-mail sobre novas postagens correspondentes. É o trabalho mais próximo, em escopo funcional, da aplicação proposta neste trabalho, já que ambos consomem a mesma base de dados classificada e expõem uma API ao usuário final – porém, como mencionado na introdução, essa integração ainda não foi realizada.

2.2 Plataformas de Consulta e Compartilhamento de CTI

Além dos trabalhos apresentados na seção anterior, é importante situar a aplicação proposta frente a plataformas já consolidadas com propósito semelhante. O MISP (*Malware Information Sharing Platform*) é uma plataforma de código aberto para coleta, correlação e compartilhamento de IoCs entre organizações, com suporte a STIX/OpenIOC e API REST (WAGNER et al., 2016). O OpenCTI organiza o conhecimento sobre ameaças como um grafo estruturado (STIX 2.1) e expõe uma API GraphQL para integração com fontes como MISP, VirusTotal e MITRE ATT&CK (Filigran, 2026); diferentemente do MISP, não possui artigo acadêmico revisado por pares – a referência disponível é apenas a documentação oficial do fabricante, Filigran.

Essas plataformas, no entanto, foram concebidas para consolidar indicadores estruturados de múltiplas organizações segundo protocolos padronizados (STIX/TAXII), o que pressupõe dados de entrada já estruturados e demanda conhecimento especializado de configuração. O ThreatLens, por sua vez, é uma API de propósito específico, construída sob medida para consultar, por meio de filtros estruturados (fonte, relevância, categoria e período), o conteúdo já classificado a partir de mensagens originalmente não estruturadas coletadas pelos trabalhos apresentados na seção anterior, sem a sobrecarga arquitetural de uma plataforma como o MISP ou o OpenCTI.

2.3 Discussão

A Tabela 1 resume como os trabalhos apresentados se relacionam: os quatro trabalhos internos resolvem etapas isoladas do pipeline (captura, classificação, análise, notificação), enquanto MISP e OpenCTI resolvem o problema de consulta/compartilhamento em um nível mais genérico, orientado a indicadores estruturados.

O foco deste TCC não está na coleta, na classificação ou na geração de novos dados, tampouco em ser uma plataforma genérica de compartilhamento entre organizações; a aplicação proposta complementa os trabalhos relacionados, funcionando como o ponto de convergência entre eles e um eventual sistema de consumo final – ressalvado que, no escopo atual, a integração com o módulo de alertas de Borges (2025) ainda não foi realizada.

Tabela 1 – Comparação entre a aplicação proposta e os trabalhos relacionados

Trabalho	**Coleta dados**	**Expõe API REST**	**Autentica usuários**	**Consome dados classificados**	**Público-alvo**
Vieira (2025)	Sim (Telegram)	Não	Não	Não	Sistema interno
Jesus Filho (2023)	Não	Não	Não	Sim	Sistema interno
Reis (2024)	Não	Não	Não	Sim	Sistema interno
Borges (2025)	Não	Sim	Não	Sim	Analista (e-mail)
MISP / OpenCTI	Não	Sim	Sim	Sim	Organizações
Este trabalho	**Não**	**Sim**	**Sim**	**Sim**	**Analista (*front-end*)**

3 O Sistema ThreatLens

O ThreatLens é uma plataforma web voltada à consulta e análise de dados de inteligência de ameaças cibernéticas. Seu propósito é disponibilizar, de forma segura e estruturada, mensagens coletadas de fontes abertas da internet e classificadas quanto à sua relevância para a cibersegurança, permitindo que analistas as consultem, filtrem e visualizem sem necessidade de acesso direto às bases de dados subjacentes.

A plataforma surgiu da necessidade de integrar os resultados de um pipeline de coleta e classificação de dados – desenvolvido ao longo de uma sequência de trabalhos acadêmicos na FACOM/UFU e descrito no Capítulo 2 – com uma interface acessível para consumo por analistas de segurança. Antes do ThreatLens, os dados produzidos por esse pipeline permaneciam acessíveis apenas diretamente pelo banco de dados, sem controle de acesso por usuário, sem filtros estruturados e sem nenhuma forma de visualização agregada.

3.1 Visão Geral da Plataforma

A plataforma é composta por dois componentes principais. O *back-end*, objeto deste TCC, é uma API REST responsável por autenticar usuários, controlar o acesso e expor os dados de inteligência de ameaças de forma filtrada, paginada e agregada. O *front-end* é um painel de visualização (*dashboard*) destinado a analistas de segurança, desenvolvido separadamente e fora do escopo deste trabalho, que consome os *endpoints* da API e apresenta as informações de forma interativa.

A plataforma oferece ao analista três perspectivas complementares sobre os dados: a consulta de publicações individuais com filtros por fonte, relevância, categoria, período e busca por texto livre no conteúdo das mensagens; estatísticas agregadas que fornecem um panorama consolidado do conjunto de dados; e uma nuvem de palavras calculada sobre as publicações filtradas, que permite identificar rapidamente os temas predominantes. Essa combinação permite ao analista partir de uma visão geral e, quando necessário, detalhar publicações específicas de interesse.

3.2 Posição no Pipeline

O ThreatLens não realiza coleta nem classificação de dados – ele integra a etapa de acesso e consulta de um pipeline mais amplo, cujas etapas anteriores são descritas no Capítulo 2. As etapas de **coleta**, **classificação** e **análise de temas** precedem o

ThreatLens e são responsáveis por obter as mensagens de canais públicos do Telegram, atribuir a cada uma delas um nível de relevância para a cibersegurança e identificar as categorias temáticas predominantes. Os resultados dessas etapas são persistidos em um banco de dados PostgreSQL externo, denominado `asgard`.

O ThreatLens lê esses dados já processados – exclusivamente em modo de leitura, sem interferir no esquema externo – e os expõe por meio da API REST. Uma quinta etapa, de **notificação** por e-mail, opera de forma paralela ao ThreatLens; sua integração à plataforma está prevista como direção para trabalho futuro.

Essa posição determina uma característica fundamental da plataforma: o ThreatLens depende das etapas anteriores para que haja dados a consultar, mas não tem controle sobre o ritmo, o volume ou o esquema dos dados que recebe. Isso orientou decisões de projeto como o acesso somente leitura ao banco externo e a manutenção de um banco de dados próprio para armazenar os dados de usuários, tokens de sessão e códigos de verificação de e-mail, mantendo os dados de inteligência isolados das informações de gerenciamento da plataforma. Os detalhes técnicos dessas decisões são apresentados no Capítulo 5.

3.3 Público-Alvo

A plataforma é destinada a profissionais e pesquisadores da área de segurança da informação e CTI que necessitam acompanhar e analisar o cenário de ameaças digitais em fontes abertas. Ela permite que esses profissionais consultem os dados produzidos pelo pipeline sem dependência de acesso direto ao banco de dados ou conhecimento do esquema interno, reduzindo a barreira técnica de acesso à informação e centralizando a análise em uma interface única.

4 Tecnologias Empregadas

Neste capítulo são descritas as tecnologias empregadas para o desenvolvimento deste trabalho, desde as ferramentas de *build* – termo que designa o processo automatizado de transformar o código-fonte escrito pelo desenvolvedor em um programa pronto para ser executado – e a linguagem de programação (seções 4.1 e 4.2) até o *framework* web, os mecanismos de autenticação e persistência de dados e as bibliotecas de teste utilizadas (seções 4.3 a 4.14).

4.1 Maven

O Apache Maven (Apache Software Foundation, 2026) é uma ferramenta de gerenciamento e compreensão de software que utiliza o conceito de *Project Object Model* (POM) para gerenciar o *build*, relatórios e a documentação de um projeto a partir de um arquivo central. No projeto ThreatLens, o Maven foi essencial para automatizar o ciclo de vida de compilação e gerenciar a complexa árvore de dependências da API em Spring Boot, declaradas no arquivo `pom.xml` e recuperadas do Repositório Central.

A implementação adotou o uso do Maven Wrapper (`mvnw`), o que permitiu que o projeto fosse compilado e executado de forma consistente em diferentes ambientes de desenvolvimento sem a necessidade de instalação prévia da ferramenta no sistema local. Adicionalmente, utilizou-se o `spring-boot-maven-plugin` para o empacotamento da aplicação em um artefato JAR executável, otimizando o resultado final ao excluir bibliotecas que atuam apenas em tempo de compilação.

4.2 Java 21

Java (Oracle Corporation, 2026a) é uma linguagem de programação orientada a objetos desenvolvida originalmente pela Sun Microsystems em 1995 e atualmente mantida pela Oracle. Uma de suas principais características é a portabilidade: o código-fonte é compilado para *bytecode*, que é executado pela Máquina Virtual Java (JVM – *Java Virtual Machine*), tornando a aplicação independente do sistema operacional subjacente.

A versão 21 é uma versão LTS (*Long-Term Support*), com suporte garantido até 2029, o que a torna a escolha natural para projetos que exigem estabilidade a longo prazo. Entre as melhorias introduzidas nessa versão estão atualizações de desempenho da JVM e avanços na API de concorrência com *Virtual Threads*. No ThreatLens, o Java 21 foi adotado por ser a versão LTS mais recente à época do desenvolvimento e por ser

amplamente suportado pelo ecossistema Spring Boot.

4.3 Spring Boot

Spring Boot (VMware, Inc., 2026a) é um *framework* de código aberto baseado no ecossistema Spring. Seu objetivo é simplificar a criação de aplicações Java ao eliminar a necessidade de configurações manuais extensas, adotando o princípio de *convention over configuration* – ou seja, as configurações padrão são predefinidas pelo *framework* para o caso de uso mais comum, cabendo ao desenvolvedor sobrescrever apenas o que for necessário para o seu contexto específico.

Em versões anteriores do Spring, era comum a necessidade de arquivos XML extensos para configurar o contexto da aplicação, definir *beans* e registrar *servlets*. O Spring Boot elimina essa complexidade por meio de anotações e autoconfiguração, permitindo que uma aplicação web esteja pronta para execução com poucas linhas de código. Além disso, ele embute um servidor HTTP diretamente no JAR gerado, dispensando a necessidade de um servidor de aplicação externo.

No ThreatLens, foi utilizada a versão 4.0.5 do Spring Boot, que traz mudanças de nomenclatura em relação às versões anteriores – o *starter* de aplicações web, por exemplo, passou a se chamar `spring-boot-starter-webmvc`. A aplicação faz uso de diversos módulos do ecossistema Spring Boot, como segurança, persistência, envio de e-mails e validação, todos descritos nas seções seguintes.

4.4 JSON Web Token

JSON Web Token (JWT) (JONES; BRADLEY; SAKIMURA, 2015) é um padrão aberto definido pela RFC 7519 que especifica um formato compacto e autocontido para transmissão segura de informações entre partes na forma de um objeto JSON assinado digitalmente. Um token JWT é composto por três partes separadas por pontos: o *header*, que indica o algoritmo de assinatura utilizado; o *payload*, que contém as informações (*claims*) sobre o usuário; e a *signature*, que garante a integridade do token.

Por ser autocontido, o JWT permite que o servidor valide a identidade do usuário sem precisar consultar um banco de dados a cada requisição – basta verificar a assinatura do token. No ThreatLens, o JWT foi adotado para implementar autenticação *stateless*, com tokens de acesso de curta duração complementados por *refresh tokens* persistidos em banco de dados, estratégia detalhada no Capítulo 5.

A biblioteca utilizada para geração e validação dos tokens foi a JJWT (Okta, Inc., 2026), versão 0.11.5, adotada por oferecer suporte completo à especificação RFC 7519 e por integrar-se nativamente ao ecossistema Spring Boot.

4.5 Spring Security

Spring Security (VMware, Inc., 2026d) é um *framework* do ecossistema Spring dedicado à autenticação e autorização em aplicações Java. Ele atua por meio de uma cadeia de filtros HTTP que interceptam cada requisição antes que ela chegue aos controladores da aplicação, permitindo verificar a identidade do usuário e validar suas permissões de acesso.

No ThreatLens, o Spring Security foi utilizado para proteger os *endpoints* da API, garantindo que apenas usuários autenticados possam acessar recursos sensíveis. A configuração define rotas públicas – como registro, *login* e verificação de e-mail – e rotas protegidas, que exigem um token válido. A integração com o mecanismo de autenticação baseado em *JSON Web Token* (JWT), descrito na seção 4.4, é feita por meio de um filtro customizado, detalhado no Capítulo 5.

4.6 BCrypt

BCrypt (PROVOS; MAZIÈRES, 1999) é um algoritmo de *hash* adaptativo desenvolvido por Niels Provos e David Mazières, projetado especificamente para o armazenamento seguro de senhas. Ao contrário de funções de *hash* genéricas como SHA-256, o BCrypt incorpora automaticamente um *salt* – valor aleatório adicionado à senha antes do *hash* – e um fator de custo configurável, que determina o tempo de processamento necessário para gerar o *hash*. Esse fator pode ser aumentado conforme o poder computacional cresce, mantendo o algoritmo resistente a ataques de força bruta ao longo do tempo.

No ThreatLens, o BCrypt foi utilizado por meio da classe `BCryptPasswordEncoder`, fornecida pelo próprio Spring Security, para proteger as senhas dos usuários antes de armazená-las no banco de dados. Nenhuma senha é persistida em texto puro – apenas seu *hash* BCrypt.

4.7 Spring Data JPA e Hibernate

Spring Data JPA (VMware, Inc., 2026b) é um módulo do ecossistema Spring que simplifica o acesso a bancos de dados relacionais por meio da Jakarta Persistence API (JPA). Ele elimina a necessidade de implementar manualmente operações repetitivas de consulta e persistência, oferecendo repositórios prontos que, a partir de convenções de nomenclatura de métodos, geram automaticamente as *queries* SQL correspondentes.

O Hibernate (Red Hat, Inc., 2026) é o *framework* ORM (*Object-Relational Mapping*) utilizado como implementação subjacente da JPA. Seu papel é realizar o mapeamento entre as classes Java anotadas com `@Entity` e as tabelas do banco de dados

relacional, abstraindo a escrita manual de SQL para as operações mais comuns.

No ThreatLens, o Spring Data JPA com Hibernate foi utilizado para gerenciar o domínio de autenticação – entidades como usuários, *refresh tokens* e códigos de verificação. Por se tratar de um *schema* controlado pelo próprio projeto, o uso de um ORM é adequado e produtivo. Para o acesso ao banco externo de *posts*, foi adotada uma abordagem diferente, descrita na seção seguinte.

4.8 JDBC

JDBC (*Java Database Connectivity*) (Oracle Corporation, 2026b) é uma API padrão da linguagem Java para acesso a bancos de dados relacionais. Ela define um conjunto de interfaces que permitem executar consultas SQL, recuperar resultados e gerenciar conexões de forma independente do banco de dados utilizado – cada banco fornece seu próprio *driver* que implementa essas interfaces.

O Spring Framework oferece a classe `NamedParameterJdbcTemplate`, que adiciona uma camada de conveniência sobre o JDBC puro, permitindo o uso de parâmetros nomeados nas *queries* SQL em vez de índices posicionais, tornando o código mais legível e menos suscetível a erros.

No ThreatLens, o JDBC foi adotado para acessar o banco externo de *posts*, cujo *schema* é mantido por um *pipeline* de coleta externo ao projeto. Como esse *schema* não é controlado pela aplicação, utilizar um ORM como o Hibernate seria inadequado – o JDBC puro oferece acesso direto e sem acoplamento ao *schema* de terceiros.

4.9 PostgreSQL

PostgreSQL (PostgreSQL Global Development Group, 2026) é um sistema gerenciador de banco de dados relacional de código aberto, desenvolvido e mantido pelo PostgreSQL Global Development Group. É reconhecido pela robustez, conformidade com o padrão SQL e suporte a recursos avançados como transações ACID, índices parciais e tipos de dados customizados.

No ThreatLens, a aplicação utiliza dois bancos PostgreSQL distintos. O primeiro é o banco primário, cujo *schema* é controlado pelo próprio projeto e armazena os dados de autenticação – usuários, *refresh tokens* e códigos de verificação. O segundo é um banco externo, mantido por um *pipeline* de coleta independente, que armazena as mensagens do Telegram já classificadas e que são consumidas pela API de *posts*. Essa separação reflete a divisão de responsabilidades entre os dois domínios de dados da aplicação.

4.10 Flyway

Flyway (Redgate Software, 2026) é uma ferramenta de migração de banco de dados que versiona e aplica alterações de *schema* de forma incremental e rastreável. O funcionamento é baseado em *scripts* SQL numerados sequencialmente – chamados de *migrations* – que são executados automaticamente na inicialização da aplicação. O Flyway mantém um registro interno das *migrations* já aplicadas, garantindo que cada *script* seja executado apenas uma vez e na ordem correta, independentemente do ambiente em que a aplicação estiver rodando.

No ThreatLens, o Flyway foi utilizado para gerenciar o *schema* do banco primário, assegurando que a estrutura de tabelas seja reproduzível entre os ambientes de desenvolvimento, teste e produção. O *schema* externo de *posts*, por ser mantido por um *pipeline* independente, não é gerenciado pelo Flyway.

4.11 Banco de Dados H2

H2 (H2 Database Engine, 2026) é um banco de dados relacional escrito em Java, projetado para ser leve e de fácil incorporação em aplicações. Uma de suas principais características é o modo *in-memory*, no qual o banco é criado inteiramente na memória durante a execução e descartado ao término, sem necessidade de instalação ou configuração de um servidor externo.

No ThreatLens, o H2 foi utilizado exclusivamente na execução dos testes automatizados, operando em modo de compatibilidade com o PostgreSQL. Isso permite que a suíte de testes rode de forma isolada e autossuficiente, sem depender de uma instância real do PostgreSQL disponível no ambiente de testes.

4.12 Spring Mail e SMTP

SMTP (*Simple Mail Transfer Protocol*) (POSTEL, 1982) é o protocolo padrão para envio de mensagens de correio eletrônico, definido originalmente pela RFC 821. Ele especifica como clientes e servidores de e-mail se comunicam para transferir mensagens entre si, operando tipicamente nas portas 25, 465 (SSL) ou 587 (TLS).

Spring Mail (VMware, Inc., 2026c) é um módulo do ecossistema Spring que abstrai o envio de e-mails via SMTP, oferecendo uma API simples para composição e envio de mensagens sem que o desenvolvedor precise lidar diretamente com as configurações do protocolo.

No ThreatLens, o Spring Mail foi utilizado para o envio de códigos OTP (*One-Time Password*) aos usuários – tanto no fluxo de verificação de e-mail no cadastro quanto

no fluxo de recuperação de senha. Em ambiente de desenvolvimento, o serviço de envio foi configurado com o servidor SMTP do Gmail.

4.13 Lombok

Lombok (Project Lombok, 2026) é uma biblioteca Java que reduz a escrita de código repetitivo – conhecido como *boilerplate* – por meio de anotações processadas em tempo de compilação. Com ela, construtores, *getters*, *setters*, métodos `equals`, `hashCode` e `toString` são gerados automaticamente pelo compilador, sem que o desenvolvedor precise escrevê-los manualmente. Por atuar apenas em tempo de compilação, o Lombok não adiciona nenhuma dependência ao artefato final gerado.

No ThreatLens, o Lombok foi utilizado nas entidades e classes de serviço por meio de anotações como `@RequiredArgsConstructor`, para geração de construtores com as dependências necessárias, `@Data`, para geração automática de *getters* e *setters*, e `@Builder`, para criação de objetos por meio do padrão de projeto *Builder*. O `spring-boot-maven-plugin` foi configurado para excluir o Lombok do JAR de produção.

4.14 JUnit 5, Mockito e AssertJ

JUnit 5 (JUnit Team, 2026) é o *framework* de testes unitários padrão do ecossistema Java. Sua versão 5, também chamada de JUnit Jupiter, introduziu uma arquitetura modular e novos recursos de anotações, tornando a escrita e organização de testes mais flexível em relação às versões anteriores.

Mockito (Mockito Contributors, 2026) é a biblioteca de *mocks* mais utilizada no ecossistema Java. Ela permite criar objetos simulados (*mocks*) de dependências externas, isolando a unidade sendo testada e controlando o comportamento dessas dependências durante o teste, sem a necessidade de instanciar a aplicação completa.

AssertJ (AssertJ Contributors, 2026) é uma biblioteca que oferece uma API fluente para escrita de asserções em testes, permitindo encadear verificações em uma única expressão e obter mensagens de erro detalhadas em caso de falha, sem dependência de métodos estáticos como os utilizados nas asserções nativas do JUnit.

No ThreatLens, as três bibliotecas foram utilizadas em conjunto na suíte de testes automatizados. O MockMvc, fornecido pelo próprio Spring, complementou esse conjunto ao permitir simular requisições HTTP aos controladores sem a necessidade de subir um servidor real.

edefinição ao e-mail cadastrado. Deve também permitir ao usuário autenticado alterar sua senha, mediante confirmação da senha atual.

5 Desenvolvimento

Este capítulo descreve o processo de desenvolvimento do ThreatLens, a camada de API REST que compõe o *back-end* do sistema. São apresentados os requisitos levantados como ponto de partida, seguidos da modelagem de dados e da arquitetura adotada, da implementação dos módulos principais e das decisões técnicas que orientaram as escolhas realizadas ao longo do projeto. O ThreatLens foi desenvolvido em Java 21 com Spring Boot 4.0.5 e tem como responsabilidade expor, de forma segura e filtrada, mensagens de inteligência de ameaças coletadas por um *pipeline* externo – vindas de canais do Telegram – para consumo por analistas de segurança via *front-end* dedicado. O código-fonte do projeto encontra-se disponível no repositório <https://github.com/annaclararodrigues/threatlens-backend>.

5.1 Requisitos do Sistema

Antes de iniciar o desenvolvimento, realizou-se o levantamento dos requisitos do sistema. Em Engenharia de Software, essa etapa consiste em identificar e documentar as funcionalidades esperadas e as restrições de qualidade que a solução deve atender (PRESSMAN; MAXIM, 2016). Os requisitos foram definidos com base no problema central abordado pelo ThreatLens: a necessidade de fornecer, de maneira segura e estruturada, acesso a dados de *threat intelligence* para analistas de segurança, com controle de autenticação, filtragem de conteúdo e apresentação de estatísticas agregadas.

5.1.1 Requisitos Funcionais

Por definição, um requisito funcional descreve a função de uma aplicação ou de um componente. Nesta seção, estão listados os principais requisitos funcionais pertencentes à aplicação desenvolvida neste trabalho. Foram estabelecidos nove requisitos funcionais, descritos nas tabelas 2, 3, 4, 5, 6, 7, 8, 9 e 10.

Tabela 2 – RF001: Cadastro de Usuários

RF001	**Cadastro de Usuários**
O sistema deve permitir o registro de novos usuários mediante informação de nome de usuário, endereço de e-mail e senha. O cadastro deve ser seguido de verificação obrigatória do endereço de e-mail por meio de código temporal (OTP) de quatro dígitos, enviado automaticamente ao e-mail informado. O acesso à API somente é liberado após a confirmação do e-mail.	

Tabela 3 – RF002: Autenticação de Usuários

RF002	**Autenticação de Usuários**
O sistema deve autenticar os usuários por meio de e-mail e senha. Após autenticação bem-sucedida, deve ser emitido um par de tokens: um *access token* JWT de curta duração, utilizado para autorizar requisições às rotas protegidas, e um *refresh token* de longa duração, armazenado em *cookie* HttpOnly no cliente e persistido no banco de dados para controle de revogação.	

Tabela 4 – RF003: Renovação e Encerramento de Sessão

RF003	**Renovação e Encerramento de Sessão**
O sistema deve disponibilizar *endpoint* para renovação transparente dos tokens de acesso, a partir do *refresh token* vigente, sem exigir novo *login* do usuário. Deve também oferecer *endpoint* de *logout* que revoga o *refresh token* no banco de dados e limpa os *cookies* no cliente, encerrando a sessão de forma segura.	

Tabela 5 – RF004: Recuperação e Troca de Senha

RF004	**Recuperação e Troca de Senha**
O sistema deve oferecer ao usuário a possibilidade de recuperar o acesso à conta em caso de esquecimento de senha, por meio do envio de um código de redefinição ao e-mail cadastrado. Deve também permitir ao usuário autenticado alterar sua senha, mediante confirmação da senha atual.	

Tabela 6 – RF005: Gerenciamento Administrativo de Usuários

RF005	**Gerenciamento Administrativo de Usuários**
O sistema deve disponibilizar, para usuários com perfil administrador, endpoints para listagem paginada de contas de usuário com suporte a filtro por *role* e busca textual, remoção de contas com proteção contra exclusão de administradores e contra *lockout* do sistema pela remoção do último administrador restante. O acesso a essas operações deve ser restrito exclusivamente ao perfil `ADMIN`, sendo negado a usuários comuns.	

Tabela 7 – RF006: Consulta de Posts com Filtros

RF006	**Consulta de Posts com Filtros**
O sistema deve expor *endpoint* para listagem paginada de mensagens (*posts*) coletadas das fontes de inteligência de ameaças. A listagem deve suportar filtragem por fonte (ex.: Telegram), nível de relevância (baixo, médio ou alto), categoria e período de tempo – podendo este ser definido por intervalo explícito de datas ou por períodos predefinidos (dia, semana, mês, ano ou todos). A ordenação deve ser configurável por data, por *score* de relevância, por identificador, por fonte ou por conteúdo.	

Tabela 8 – RF007: Estatísticas Agregadas de Posts

RF007	**Estatísticas Agregadas de Posts**
O sistema deve oferecer *endpoint* dedicado ao retorno de estatísticas sobre os *posts* disponíveis, incluindo total de mensagens, percentual de mensagens classificadas como relevantes, distribuição por nível de relevância (baixo, médio e alto) e contagem por fonte. Os mesmos filtros de período e fonte aplicáveis à listagem de posts devem ser suportados nas estatísticas.	

Tabela 9 – RF008: Nuvem de Palavras dos Posts

RF008	**Nuvem de Palavras dos Posts**
O sistema deve oferecer *endpoint* dedicado ao cálculo da frequência das palavras mais utilizadas no conteúdo dos *posts*, de modo a alimentar uma visualização de nuvem de palavras no *frontend*. Devem ser suportados os mesmos filtros de fonte, relevância, categoria e período aplicáveis à listagem e às estatísticas de *posts*, além de um parâmetro que limite a quantidade máxima de palavras retornadas. Palavras irrelevantes (*stopwords*) em português e inglês, bem como termos muito curtos, devem ser desconsiderados do cálculo.	

Tabela 10 – RF009: Suporte a Múltiplas Fontes de Dados

RF009	**Suporte a Múltiplas Fontes de Dados**
A arquitetura do sistema deve ser extensível para incorporar novas fontes de inteligência de ameaças além do Telegram, sem necessidade de alteração na camada de serviço ou nos contratos de API. A adição de uma nova fonte deve requerer apenas a implementação de um novo componente de acesso a dados, seguindo a interface definida pelo sistema.	

5.1.2 Requisitos Não Funcionais

Requisitos não funcionais são os requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenção e tecnologias envolvidas (CYSNEIROS; LEITE, 1997). Não é necessário que sejam especificados pelo cliente, uma vez que são características mínimas de um software de qualidade, ficando a cargo do desenvolvedor optar por atender esses requisitos ou não. Neste trabalho, foram especificados quatro requisitos não funcionais, descritos nas tabelas 11, 12, 13 e 14.

Tabela 11 – RNF001: Segurança de Credenciais

RNF001	**Segurança de Credenciais**
As senhas dos usuários nunca devem ser armazenadas em texto plano. O sistema deve aplicar o algoritmo BCrypt para *hashing* de senhas antes da persistência, garantindo resistência a ataques de força bruta e *rainbow table*. Os tokens de acesso devem ser armazenados em *cookies* com os atributos HttpOnly e SameSite=Strict, mitigando ataques do tipo *Cross-Site Scripting* (XSS) e *Cross-Site Request Forgery* (CSRF).	

Tabela 12 – RNF002: Autenticação *Stateless*

RNF002	**Autenticação *Stateless***
O mecanismo de autenticação deve ser *stateless*, ou seja, o servidor não deve manter estado de sessão em memória. A validação de identidade deve ser realizada integralmente a partir do JWT contido em cada requisição, sem consulta ao banco de dados a cada chamada.	

Os requisitos apresentados nesta seção orientaram as decisões de arquitetura e implementação descritas nas seções seguintes. Em particular, os requisitos de segurança

Tabela 13 – RNF003: Versionamento do Esquema de Banco de Dados

RNF003	**Versionamento do Esquema de Banco de Dados**
O esquema do banco de dados primário da aplicação deve ser versionado e gerenciado por ferramenta de migração, de modo que qualquer ambiente possa ser provisionado de forma reprodutível e controlada, sem intervenção manual. Alterações no esquema devem ser rastreáveis e reversíveis.	

Tabela 14 – RNF004: Separação entre Dados Próprios e Dados Externos

RNF004	**Separação entre Dados Próprios e Dados Externos**
A aplicação deve manter separação clara entre o banco de dados de sua responsabilidade (usuários, tokens, códigos de verificação) e o banco de dados externo de *posts*, cuja evolução de esquema é controlada por sistema terceiro. Essa separação deve impedir que ferramentas de migração da aplicação interfiram no esquema externo, e que o modelo de objetos da aplicação se acople a tabelas não gerenciadas por este projeto.	

(RNF001 e RNF002) motivaram a adoção de JWT com *refresh token* persistido e *cookies* HttpOnly; e os requisitos RNF003 e RNF004 justificaram a configuração de dois *datasources* distintos e o uso do Flyway exclusivamente sobre o banco primário.

5.2 Modelagem

Uma vez especificados os requisitos do sistema, passou-se à fase de modelagem. Nesta etapa, foram definidas as estruturas de dados que sustentam a aplicação e a organização geral das suas camadas. Dois artefatos principais foram produzidos: o modelo de entidades do banco de dados primário e a definição da arquitetura em camadas da aplicação.

5.2.1 Modelagem do Banco de Dados

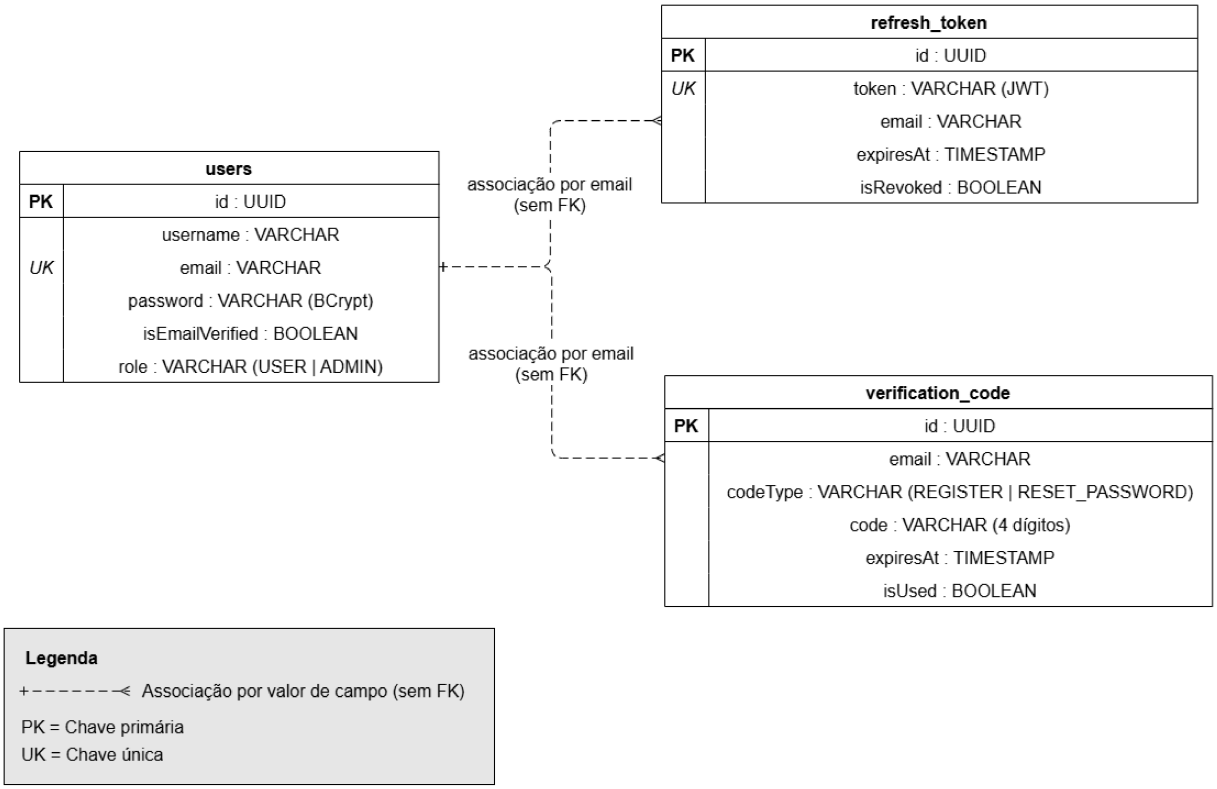
O banco de dados da aplicação é dividido em dois domínios distintos, cada um com seu próprio *datasource* e mecanismo de acesso. O primeiro domínio, denominado primário, é de responsabilidade do ThreatLens e contém os dados gerenciados pela própria aplicação: usuários, tokens de sessão e códigos de verificação. O segundo domínio, externo, pertence ao *pipeline* de coleta e classificação de mensagens do Telegram, desenvolvido separadamente, e é acessado pelo ThreatLens apenas em modo de leitura.

5.2.1.1 Banco de Dados Primário

O banco de dados primário é gerenciado via Spring Data JPA com Hibernate e tem seu esquema versionado pela ferramenta Flyway, garantindo reprodutibilidade entre ambientes. É composto por três entidades: **users**, que armazena as informações de cada usuário cadastrado; **refresh_token**, que registra os *refresh tokens* emitidos pela

aplicação, permitindo sua validação e revogação; e **verification_code**, que armazena os códigos OTP enviados por e-mail para verificação de cadastro e redefinição de senha. O diagrama entidade relacionamento dessas três entidades é apresentado na Figura 1.

Figura 1 – Diagrama Entidade Relacionamento da Aplicação



Fonte: Autoria própria

Vale observar que não há chave estrangeira explícita relacionando as três entidades entre si. A associação entre *users*, *refresh_token* e *verification_code* é feita por convenção de valor: todas as entidades compartilham o campo `email` como identificador de vínculo. Essa escolha simplifica o modelo, mas implica ausência de garantia referencial no banco de dados – um *tradeoff* entre simplicidade de implementação e integridade estrutural.

O esquema do banco primário evoluiu de forma incremental ao longo do desenvolvimento. A *migration* `V1__create_initial_schema.sql` criou as três tabelas iniciais, enquanto `V2__add_roles_to_users_table.sql` adicionou posteriormente a coluna `role` à tabela *users*, evidenciando o controle versionado de mudanças de esquema proporcionado pelo Flyway.

5.2.1.2 Banco de Dados Externo (*Posts*)

O banco de dados externo, denominado `asgard`, é mantido pelo *pipeline* de coleta e classificação de mensagens e não está sob controle deste projeto. O ThreatLens acessa esse

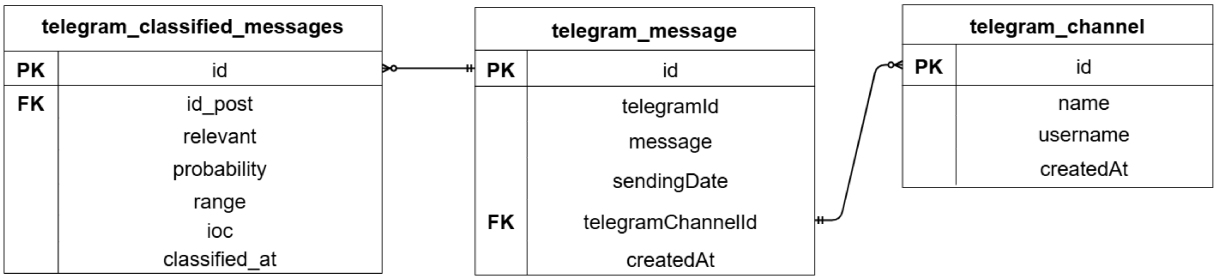
banco exclusivamente em modo de leitura, via JDBC puro, por meio de um *datasource* secundário configurado separadamente, sem uso de JPA ou Flyway.

O banco foi concebido para acomodar múltiplas fontes de dados, cada uma com seu próprio esquema e estrutura de tabelas. Para este trabalho, optou-se por implementar a integração com a fonte Telegram, por ser a de maior volume de *posts* relevantes disponível. As demais fontes, embora planejadas, não foram implementadas nesta versão e estão registradas como trabalho futuro. Essa decisão influenciou diretamente a arquitetura da camada de acesso a dados, descrita na seção 5.2.2: em vez de acoplar o código ao esquema específico do Telegram, foi adotado um padrão que permite incorporar novas fontes sem alterar a lógica existente.

Para a fonte Telegram, três tabelas do banco `asgard` são consumidas pela aplicação, conforme ilustrado na Figura 2. A tabela **telegram_message** contém as mensagens coletadas dos canais, com os campos `id`, `telegramId`, `message`, `sendingDate` e `telegramChannelId`. A tabela **telegram_channel** armazena os metadados dos canais monitorados, com os campos `id`, `name` e `username`. A tabela **telegram_classified_messages** registra a classificação de relevância atribuída a cada mensagem pelo *pipeline* de coleta, com os campos `id_post`, `relevant`, `probability`, `range`, `ioc` e `classified_at`. O campo `probability` é utilizado pela aplicação para determinar o nível de relevância da mensagem – baixo, médio ou alto –, enquanto `range` e `ioc` são retornados diretamente na resposta da API.

Vale observar que as associações entre as três tabelas são feitas por convenção de valor – `telegram_message.telegramChannelId` referencia `telegram_channel.id`, e `telegram_classified_messages.id_post` referencia `telegram_message.id` – sem restrições referenciais explícitas no banco, uma vez que seu esquema é gerenciado por um sistema externo.

Figura 2 – Diagrama das tabelas do banco externo `asgard` consumidas pelo ThreatLens



Fonte: Autoria própria

5.2.2 Arquitetura da Aplicação

A aplicação segue uma arquitetura em camadas típica de APIs REST desenvolvidas com Spring Boot, organizada no padrão Controller → Service → Repository. Não há camada de visão, uma vez que o ThreatLens é uma API pura, cujo *front-end* é desenvolvido e mantido separadamente. Os pacotes principais da aplicação estão organizados sob o pacote raiz `com.backend.threatlens` e podem ser agrupados em três categorias:

- **Camadas principais:**
  - `controller` – reúne os controladores REST responsáveis por receber e responder às requisições HTTP;
  - `service` – concentra as regras de negócio;
  - `repository` – contém os repositórios de acesso ao banco primário via Spring Data JPA e ao banco externo de *posts* via JDBC.
- **Estruturas de dados:**
  - `entity` – entidades JPA mapeadas para o banco primário;
  - `dto` – objetos de transferência de dados de entrada e saída.
- **Infraestrutura:**
  - `config` – configurações de segurança e dos dois *datasources*;
  - `filter` – filtro de autenticação JWT e limitação de requisições por IP;
  - `exception` – exceções de negócio e tratamento global de erros;
  - `utils` – utilitários para geração de tokens e manipulação de *cookies*.

O ponto arquiteturalmente mais relevante da aplicação é o uso do **Padrão Strategy** para o acesso a múltiplas fontes de *posts*. Esse padrão é materializado pela interface `PostsSourceRepository`, localizada no subpacote `repository/posts`, que define o contrato que toda fonte de dados deve implementar. A interface declara quatro métodos:

- `source()` – identifica a fonte de dados;
- `findPage()` – realiza consulta paginada com filtros;
- `count()` – retorna a contagem total de registros;
- `getStats()` – obtém estatísticas agregadas.

Cada fonte de dados corresponde a uma implementação concreta dessa interface – atualmente, apenas `TelegramRepository` está implementada. O Spring coleta automaticamente todas as implementações disponíveis e as injeta como uma `List<PostsSource Repository>` na camada de serviço, que as itera dinamicamente conforme os filtros da requisição. Com isso, a incorporação de uma nova fonte de dados requer apenas a criação de uma nova classe que implemente essa interface, sem qualquer alteração no serviço ou nos *endpoints* existentes.

5.3 Implementação

A implementação do ThreatLens está organizada em quatro módulos funcionais: autenticação e autorização, administração, consulta de *posts* e tratamento de erros. O código-fonte está disponível publicamente no repositório GitHub[1] e pode ser executado localmente com os pré-requisitos de Java 21, Maven e acesso aos dois bancos PostgreSQL descritos na seção 5.2, iniciando a aplicação com o comando `./mvnw spring-boot:run` com o perfil `dev` ativo.

5.3.1 Módulo de Autenticação e Autorização

O módulo de autenticação é responsável pelo ciclo de vida completo das contas de usuário: registro, verificação de e-mail, login, renovação de sessão, logout e recuperação de senha. Todos os *endpoints* desse módulo estão sob o prefixo `/auth` e são implementados pelo `AuthController`, que delega a lógica de negócio ao `AuthService`.

A estratégia de autenticação adotada é *stateless*, baseada em JWT. Optou-se por complementar o JWT puro com *refresh tokens* persistidos em banco de dados, o que resolve uma limitação inerente ao padrão: tokens JWT convencionais não podem ser revogados antes de sua expiração, tornando o logout não confiável. Com os *refresh tokens* armazenados na tabela `refresh_token` e marcados com o campo `isRevoked`, a aplicação garante que um logout ou uma troca de senha invalide imediatamente a sessão do usuário. O *access token* tem duração de cinco minutos, configurável via a propriedade `jwt.expiration` em `application.properties` e lida em tempo de execução por meio da anotação `@Value` no `JwtUtil`. Esse intervalo representa um equilíbrio entre segurança – reduzindo a janela de vulnerabilidade em caso de comprometimento do token – e eficiência, evitando o volume excessivo de requisições de renovação que uma duração muito curta implicaria.

Os dois tokens emitidos no login utilizam mecanismos de transporte distintos, escolhidos de acordo com o perfil de uso e os requisitos de segurança de cada um.

[1] https://github.com/annaclararodrigues/threatlens-backend

O *access token* é transportado via *header* HTTP `Authorization: Bearer` e deve ser incluído pelo cliente em toda requisição às rotas protegidas da API. Por ter duração curta – configurável pela propriedade `jwt.expiration` – sua exposição eventual representa uma janela de vulnerabilidade limitada.

O *refresh token*, por sua vez, é transportado exclusivamente via *cookie* `HttpOnly`, restrito ao caminho `/auth/refresh`. Essa separação é deliberada: por ser um token de longa duração, ele não deve ser manipulado diretamente pelo código da aplicação cliente. O atributo `HttpOnly` impede que código JavaScript acesse o cookie; o atributo `SameSite=Strict` impede que ele seja enviado em requisições originadas de outros domínios; e a restrição de caminho garante que o cookie seja enviado apenas ao *endpoint* `/auth/refresh`, sem acompanhar requisições às demais rotas da API. Essas propriedades, em conjunto, reduzem a superfície de ataque do *refresh token* em relação a ataques do tipo XSS e CSRF (**??**). O atributo `Secure` é configurável por perfil de ambiente: desabilitado em desenvolvimento para permitir execução sem HTTPS, e habilitado em produção, garantindo transmissão exclusivamente em conexões seguras.

O fluxo de registro exige verificação de e-mail antes que o usuário possa realizar o primeiro acesso. Ao se cadastrar, um código OTP de quatro dígitos com validade de dez minutos é gerado e enviado por e-mail via SMTP. Somente após a validação desse código – por meio do *endpoint* `/auth/verify` – o campo `isEmailVerified` do usuário é marcado como verdadeiro e os tokens de sessão são emitidos. O fluxo de recuperação de senha segue o mesmo mecanismo: um código do tipo `RESET_PASSWORD` é gerado e enviado ao e-mail informado, e sua validação autentica o usuário para que a nova senha seja definida.

As senhas são armazenadas exclusivamente como *hashes* BCrypt, gerados pelo `BCryptPasswordEncoder` do Spring Security. Nenhuma senha é persistida em texto puro em nenhum momento do fluxo.

A proteção dos *endpoints* é configurada no `SecurityConfig`, que define as rotas públicas – como `/auth/register`, `/auth/login` e `/auth/refresh` – e exige autenticação válida para todas as demais. A validação do *access token* a cada requisição é realizada pelo `JwtAuthFilter`, um filtro HTTP customizado registrado antes do filtro padrão do Spring Security. O filtro extrai o token do *header* `Authorization: Bearer`, valida sua assinatura HMAC-SHA256 e expiração, carrega os dados do usuário e popula o `SecurityContext`, permitindo que os controladores identifiquem o usuário autenticado sem consulta ao banco de dados a cada requisição.

Para mitigar ataques de força bruta, o módulo implementa limitação de requisições por IP por meio do `RateLimitFilter`, registrado na cadeia de filtros do Spring Security antes do `JwtAuthFilter`. Os limites aplicados são de cinco requisições a cada quinze minutos para os *endpoints* `/auth/login` e `/auth/register`, e dez requisições a cada dez minutos para `/auth/verify`. A implementação utiliza uma janela fixa em memória,

sem dependência de bibliotecas externas. Vale observar que essa abordagem tem uma limitação conhecida: em ambientes com múltiplas instâncias da aplicação, o contador não é compartilhado entre processos, o que reduziria a eficácia do mecanismo. Essa característica está fora do escopo deste trabalho, cujo ambiente de execução é de instância única, e pode ser endereçada em trabalhos futuros por meio de cache distribuído.

Eventos críticos de autenticação são registrados por um logger de auditoria dedicado, instanciado com o nome `"AUDIT"` no `AuthService`. São auditados: login bem-sucedido e falho, logout, registro de novo usuário, troca de senha e renovação de *refresh token*. Cada registro inclui o tipo de evento, o e-mail envolvido, o IP da requisição e o *timestamp*, compondo uma trilha de auditoria consultável nos logs da aplicação.

5.3.2 Módulo de Administração

O módulo de administração expõe funcionalidades restritas a usuários com a *role* `ADMIN`, implementadas pelo `AdminController` sob o prefixo `/admin`. O controle de acesso é aplicado diretamente na classe por meio da anotação `@PreAuthorize("hasRole('ADMIN')")` do Spring Security, que exige a habilitação de segurança em nível de método via `@EnableMethodSecurity`. Qualquer requisição a esses *endpoints* por um usuário sem a *role* adequada é rejeitada com HTTP 403 antes mesmo de chegar à lógica de negócio.

A criação do primeiro usuário administrador é realizada automaticamente na inicialização da aplicação pelo `AdminBootstrapRunner`, um `CommandLineRunner` que verifica se já existe algum usuário com *role* `ADMIN` no banco de dados e, caso não exista, cria um a partir das variáveis de ambiente `ADMIN_EMAIL`, `ADMIN_USERNAME` e `ADMIN_PASSWORD`. Se essas variáveis não estiverem definidas, o *runner* apenas registra um aviso nos logs e não realiza nenhuma ação. Essa abordagem evita a exposição de um *endpoint* público de criação de administrador e impede que credenciais iniciais sejam persistidas no histórico de versionamento do esquema.

Administradores subsequentes são promovidos diretamente no banco de dados por meio de um comando SQL, sem a exposição de um *endpoint* dedicado para essa operação – dado que a promoção de privilégios administrativos é uma ação pontual e sensível, expor essa funcionalidade via API aumentaria a superfície de ataque da aplicação sem trazer benefício operacional proporcional.

O *endpoint* `GET /admin/users` lista os usuários cadastrados com suporte a paginação via `Pageable`, filtro por *role* e busca textual sobre os campos `username` e `email`. A busca utiliza o operador `LIKE` do SQL com *escaping* seguro de caracteres especiais, prevenindo que entradas maliciosas alterem o comportamento da consulta – comportamento verificado pela classe `AdminServiceTest`, descrita na seção 5.3.5.

O *endpoint* `DELETE /admin/users/{id}` remove um usuário da base de dados, eliminando em cascata seus *refresh tokens* e códigos de verificação associados. A operação é bloqueada em dois cenários: para usuários com *role* `ADMIN`, prevenindo remoção de administradores via API; e quando restar apenas um administrador no sistema, evitando que o *lockout* completo ocorra por remoção acidental do último administrador restante.

5.3.3 Módulo de Consulta de Posts

O módulo de consulta de *posts* é responsável por expor as mensagens coletadas e classificadas pelo *pipeline* externo, permitindo filtragem, paginação e agregação estatística. Os *endpoints* são implementados pelo `PostsController` e delegam a lógica de negócio ao `PostsService`, que coordena o acesso às fontes de dados por meio do Padrão Strategy descrito na seção 5.2.2.

O *endpoint* `GET /posts` retorna uma listagem paginada de *posts*, cujos parâmetros de consulta são definidos pelo `PostsQueryDTO`: `sources` (uma ou mais fontes, ex.: `TELEGRAM`), `relevance` (`LOW`, `MEDIUM` ou `HIGH`), `category` (texto livre), `search` (busca por texto livre no conteúdo da mensagem), `period` (`DAY`, `WEEK`, `MONTH`, `YEAR` ou `ALL`) ou um intervalo explícito via `from`/`to` – que tem precedência sobre `period` –, campo e direção de ordenação (`sort`: `DATE`, `SCORE`, `ID`, `SOURCE` ou `CONTENT`; `order`: `ASC` ou `DESC`) e controle de página (`page`, `size`). O contrato de resposta é definido pelo `PostResponse` (Código 5.1), um record genérico que unifica todas as fontes de dados em um único formato.

```
1  public record PostResponse(
2      String id,
3      String source,
4      String title,
5      String content,
6      String author,
7      String createdAt,
8      String category,
9      ClassificationResponse classification,
10     Map<String, Object> meta) { }
```

Código 5.1 – Contrato genérico de resposta de post

Os campos `title`, `author` e `category` são comuns à maioria das fontes previstas, mas retornados como nulos para a fonte Telegram, cujo *pipeline* não produz essas informações. Manter esses campos no contrato genérico evita a necessidade de formatos de resposta distintos por fonte, simplificando o consumo da API pelo *front-end*. O campo `meta` é tipado como `Map<String, Object>` para acomodar informações específicas

de cada fonte: para o Telegram, contém o identificador, o nome e o *username* do canal de origem, preenchidos pelo `TelegramMeta`. Cada fonte futura seguirá o mesmo padrão, implementando seu próprio record com um método `toMap()`, sem alterar o contrato do `PostResponse`.

O campo `classification` encapsula a classificação atribuída pelo *pipeline* externo (Código 5.2). O campo `score` contém a probabilidade bruta do classificador (entre 0 e 1), traduzida pelo `PostsService` para o campo `level` (`LOW`, `MEDIUM` ou `HIGH`). O campo `ioc` indica se a mensagem contém um *Indicator of Compromise* – um artefato associado a uma ameaça conhecida, como um endereço IP malicioso, um *hash* de arquivo ou um domínio suspeito.

```
1 public record ClassificationResponse(
2     boolean relevant,
3     double score,
4     RelevanceLevel level,
5     String range,
6     Boolean ioc,
7     @JsonFormat(pattern = "yyyy-MM-dd'T'HH:mm:ss")
8     LocalDateTime classifiedAt) { }
```

Código 5.2 – Resposta de classificação de relevância

O parâmetro `search` permite filtrar os *posts* pelo conteúdo da mensagem por meio de busca textual sem diferenciação de maiúsculas e minúsculas (*case-insensitive*). O filtro é propagado do `PostsQueryDTO` até o `PostsFilter` – estrutura que encapsula todos os critérios de filtragem repassados ao repositório – e aplicado no `TelegramRepository` por meio do operador `ILIKE` do PostgreSQL sobre a coluna `message`, tanto no método `findPage` quanto no `count` (que determina o total de registros para a paginação). O parâmetro é exclusivo do `GET /posts` e combinável com os demais filtros desse endpoint (`sources`, `relevance`, `category`, `period`, `from`/`to`) e com os parâmetros de ordenação e paginação. Os endpoints `GET /posts/stats` e `GET /posts/wordcloud` não expõem esse parâmetro em seus DTOs de consulta e não são afetados por ele.

O *endpoint* `GET /posts/stats` retorna estatísticas agregadas, aceitando os mesmos filtros de `sources`, `relevance` e `category` do `GET /posts` – definidos pelo `StatsQueryDTO`, porém sem os parâmetros de ordenação e paginação –, além do filtro de período via `period` (`DAY`, `WEEK`, `MONTH`, `YEAR` ou `ALL`) ou por intervalo explícito (`from`/`to`), que tem precedência sobre `period`. A resposta é estruturada pelo `PostStatsDTO` (Código 5.3), composto por três objetos: `summary`, com o total de *posts*, o total de relevantes no período e seu percentual; `relevanceDistribution`, com a contagem por nível; e

`sources`, com o volume por fonte.

```
public record PostStatsDTO(
    Summary summary,
    RelevanceDistribution relevanceDistribution,
    List<SourceCount> sources) {

    public record Summary(
        long totalPosts,
        long relevantPostsInPeriod,
        double relevantPostsPct) { }

    public record RelevanceDistribution(
        long lowCount,
        long mediumCount,
        long highCount) { }

    public record SourceCount(
        String sourceName,
        long postCount) { }
}
```

Código 5.3 – DTO de estatísticas de posts

A estratégia de paginação do `PostsService` varia conforme o número de fontes ativas. Para uma única fonte, a paginação é delegada ao banco via `OFFSET`/`LIMIT`, retornando o total real de registros de forma eficiente. Para múltiplas fontes, como não é possível uma única *query* SQL que cruze bancos distintos, cada fonte é consultada com um limite fixo de 10.000 registros (`MULTI_SOURCE_FETCH_CAP`), os resultados são mesclados em memória, ordenados e fatiados para extrair a página solicitada.

Essa abordagem tem limitações conhecidas: o total reportado pode ser impreciso quando uma fonte excede o *cap*, registros além desse limite são ignorados nas últimas páginas, e o custo de cada requisição é fixo independentemente da página solicitada. Essas limitações foram identificadas durante a implementação – quando a estratégia de mesclagem em memória já estava desenvolvida – e como o sistema opera com apenas uma fonte ativa na prática, a paginação delegada ao banco é sempre utilizada e o impacto não se manifestou. A correção definitiva, por meio de tabela unificada ou paginação por cursor (*keyset pagination*), está registrada como direção para trabalho futuro na seção 7.

O *endpoint* `GET /posts/wordcloud` (RF008) calcula a frequência das palavras mais utilizadas no conteúdo dos *posts*, para alimentar um gráfico de nuvem de palavras no

front-end. Ele reaproveita integralmente os filtros de fonte, relevância, categoria e período já utilizados por `GET /posts/stats`, acrescentando apenas o parâmetro `limit` (mínimo 1, máximo 200, *default* 50), que define a quantidade de palavras retornadas. A tokenização é realizada em SQL, no mesmo repositório que já implementa `findPage`/`count`/`getStats` (`TelegramRepository`), evitando trazer o conteúdo integral dos *posts* para a camada de aplicação: o texto da mensagem é normalizado (`lower` combinado a `regexp_replace`, removendo tudo o que não é letra) e quebrado em palavras com `regexp_split_to_table`; em seguida, palavras com dois caracteres ou menos e *stopwords* – mantidas em uma lista combinada de português e inglês em `utils/StopWords.java`, já que o conteúdo dos *posts* mistura os dois idiomas – são descartadas; o restante é agrupado e ordenado por contagem decrescente, limitado a `:limit` no próprio SQL. O mesmo `WHERE` de relevância, categoria e período já usado em `getStats`/`findPage` é reaproveitado por meio do `RELEVANCE_FILTER`.

Quando mais de uma fonte é selecionada, o `PostsService` aplica a mesma estratégia de *overfetch* descrita na seção 5.2.2 para a listagem paginada: cada fonte é consultada por um número de palavras maior que o `limit` solicitado (`MULTI_SOURCE_WORD_FETCH_CAP = 1_000`, ou o próprio `limit`, se maior), as contagens de palavras repetidas entre fontes são somadas, o resultado é reordenado e só então truncado no `limit` pedido. Essa abordagem evita que uma palavra frequente na soma de duas fontes, mas fora do *top-N* de cada fonte individualmente, seja perdida no *merge* – o mesmo problema documentado para a paginação multi-fonte de `GET /posts`. Com fonte única, a busca é feita diretamente com o `limit` solicitado, sem *overfetch*, já que não há *merge* a realizar. Como limitação conhecida, a normalização via classe POSIX `[:alpha:]` depende de o *locale*/*collation* do banco externo tratar corretamente letras acentuadas (ex.: ameaça), algo fora do controle deste projeto (seção 5.2.1.2).

5.3.4 Tratamento de Erros

O tratamento de erros da aplicação é centralizado no `GlobalExceptionHandler`, uma classe anotada com `@RestControllerAdvice` que intercepta exceções lançadas em qualquer camada da aplicação e as converte em respostas HTTP com status e mensagem adequados. Essa abordagem evita que detalhes internos da aplicação vazem para o cliente e garante um formato de resposta de erro consistente em todos os *endpoints*.

As exceções de negócio são organizadas em cinco tipos customizados. A `ResourceNotFoundException` é lançada quando uma entidade requisitada não existe no banco de dados, resultando em HTTP 404. A `BusinessRuleViolationException` é lançada quando uma operação viola uma regra de negócio definida – como tentativa de cadastro com e-mail já existente ou remoção do último administrador do sistema –, resultando em HTTP 409. A `InvalidRequestException` é lançada para erros de validação de entrada que não são cobertos pelas anotações de Bean Validation, como senhas que não

coincidem, resultando em HTTP 400. A `EmailNotVerifiedException` é lançada quando um usuário tenta realizar login sem ter concluído a verificação de e-mail, resultando em HTTP 403 – semanticamente distinta de uma violação de regra de negócio, pois representa uma pré-condição de conta não satisfeita. A `InvalidTokenException` é lançada para falhas de autenticação relacionadas a tokens – como *refresh token* inválido, revogado ou ausente –, resultando em HTTP 401.

Além das exceções de negócio, o `GlobalExceptionHandler` trata outros dois casos. Erros de validação de campos disparados pelo Bean Validation (`MethodArgumentNotValidException`) são capturados e retornados como HTTP 422, com a lista dos campos inválidos e suas mensagens de erro. Exceções não previstas são capturadas genericamente e retornadas como HTTP 500, evitando que *stack traces* sejam expostos ao cliente.

5.3.5 Testes Automatizados

A suíte de testes automatizados do ThreatLens foi desenvolvida com JUnit 5, Mockito e AssertJ, combinação padrão no ecossistema Spring Boot para testes unitários e de integração leve. Os testes de controlador utilizam o MockMvc configurado via `standaloneSetup`, que instancia apenas o controlador sendo testado sem subir o contexto Spring completo, tornando a execução mais rápida e isolada. Um *smoke test* complementa a suíte verificando que o contexto Spring inicializa corretamente com o banco H2 em memória e o Flyway desabilitado.

A suíte é composta por seis classes de teste, descritas na Tabela 15.

Tabela 15 – Classes de teste da suíte automatizada

Classe	**Tipo**	**O que cobre**
`AdminControllerTest`	Unitário (MockMvc)	Binding de parâmetros, formato JSON de resposta, códigos HTTP dos endpoints de administração e rejeição de requisições sem a *role* ADMIN (HTTP 403).
`PostsControllerTest`	Unitário (MockMvc)	Binding completo dos DTOs de consulta, estatísticas e nuvem de palavras, formato de resposta, retorno de HTTP 422 para violações de regra de negócio e rejeição de valores de `limit` fora do intervalo permitido no *endpoint* de nuvem de palavras.
`AdminServiceTest`	Unitário (Mockito)	Normalização e *escaping* da busca textual, filtro por *role*, mapeamento de entidade para DTO e regras de remoção de usuário – incluindo proteção contra remoção do último administrador.
`AuthServiceTest`	Unitário (Mockito)	Fluxos de registro, login, verificação de código OTP, troca e recuperação de senha, logout e renovação de *refresh token*.
`PostsServiceTest`	Unitário (Mockito)	Agregação de estatísticas para fonte única e múltiplas fontes, cálculo de percentuais de relevância, resolução de janelas temporais, estratégia de paginação híbrida, tratamento de valores nulos e, para a nuvem de palavras, agregação/*merge* de contagens entre fontes, truncamento no `limit` solicitado e reaproveitamento dos filtros de estatísticas.
`ThreatlensApplicationTests`	Smoke test	Inicialização completa do contexto Spring com banco H2 em memória e Flyway desabilitado.

6 Avaliação

Este capítulo apresenta a avaliação funcional do ThreatLens, realizada por meio de requisições HTTP submetidas à aplicação em execução, com o objetivo de demonstrar que os requisitos funcionais especificados na seção 5.1.1 foram efetivamente implementados e se comportam conforme o esperado. Diferentemente de aplicações com interface gráfica própria, cuja avaliação é usualmente conduzida por meio de capturas de tela do fluxo de uso, o ThreatLens é uma API pura, de modo que a evidência mais adequada de seu funcionamento é o par requisição/resposta observado em cada endpoint, obtido por meio do Postman. O capítulo está organizado por módulo funcional, na mesma divisão adotada no Capítulo 5, encerrando-se com uma discussão dos resultados obtidos e das limitações observadas durante a avaliação.

6.1 Ambiente de Avaliação

A avaliação dos módulos de autenticação e administração foi realizada em ambiente local, com a aplicação executada por meio do comando `./mvnw spring-boot:run` sob o perfil `dev`, utilizando uma instância local do PostgreSQL com o esquema do banco primário provisionado automaticamente pelo Flyway – ambiente inteiramente reproduzível a partir do repositório do projeto. Já a avaliação do módulo de consulta de posts foi realizada contra a instância real do banco externo, mantida pelo pipeline de coleta e classificação desenvolvido por Vieira (2025) e Jesus Filho (2023); por se tratar de um banco de dados de terceiro, cujo acesso é restrito e cujo esquema não é versionado por este projeto – conforme discutido no RNF004 (seção 5.1.2) – não há dump ou script de reprodução disponibilizado, e as requisições apresentadas na seção 6.4 refletem o volume de dados coletado e classificado por esses trabalhos no momento da avaliação.

6.2 Módulo de Autenticação e Autorização

Esta seção avalia os endpoints do prefixo `/auth`, correspondentes aos requisitos funcionais RF001 a RF004 (seção 5.1.1). Conforme detalhado na seção 6.1, todos os testes foram executados contra a instância local da aplicação. As rotas públicas, que dispensam autenticação, são `register`, `verify`, `resend-code`, `resend-password`, `login`, `forgot-password`, `refresh`, `logout` e `reset-password`; as demais – `change-password` e todo o prefixo `/admin` – exigem um token de acesso JWT válido.

6.2.1 Cadastro e Verificação de E-mail (RF001)

O cadastro é realizado em duas etapas, conforme descrito na seção 5.3.1: envio dos dados de registro e confirmação por código temporal enviado por e-mail.

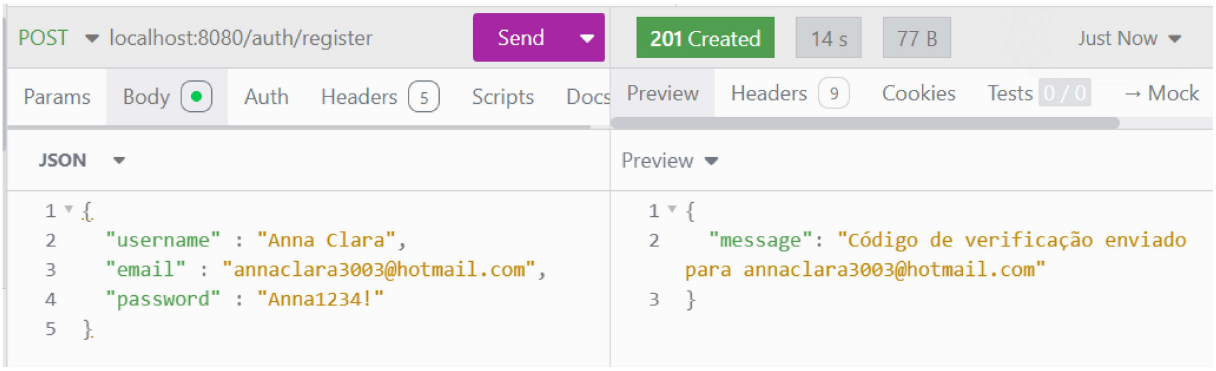
A primeira etapa é realizada pela requisição `POST /auth/register`, que recebe o seguinte corpo:

```
{
  "username": "anna",
  "email": "anna@example.com",
  "password": "Senha@123"
}
```

Em caso de sucesso, o endpoint retorna `201 Created` com o corpo:

```
{
  "message": "Código de verificação enviado para anna@example.com"
}
```

Figura 3 – Execução da requisição `POST /auth/register` no Postman



Fonte: Autoria própria

Figura 4 – Código enviado ao e-mail após a requisição `POST /auth/register`

Fonte: Autoria própria

A Figura 3 mostra a execução dessa requisição, e a Figura 4 mostra o código recebido em seguida por e-mail, confirmando o comportamento descrito na seção 5.3.1: a senha já é persistida como *hash* BCrypt e o código de verificação é gerado e enviado ao usuário. A Tabela 16 resume os cenários de erro observados nessa primeira etapa.

Tabela 16 – Cenários de erro – `POST /auth/register`

Cenário	**Mensagem**	**Status**
E-mail já cadastrado	`Email already in use`	409
Campo inválido (ex.: senha sem caractere especial)	`Dados inválidos.`	422

Com o código em mãos, a segunda etapa é realizada pela requisição `POST /auth/verify`:

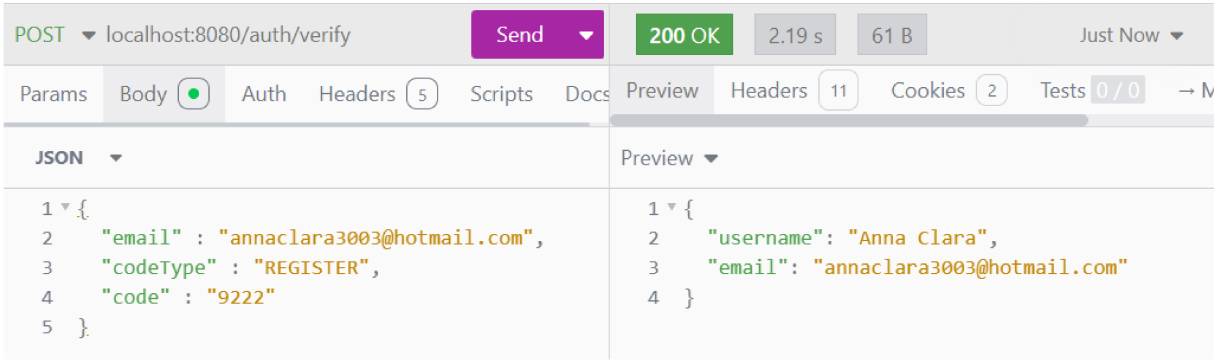
```
{
  "email": "anna@example.com",
  "codeType": "REGISTER",
  "code": "4821"
}
```

A resposta de sucesso (`200 OK`) retorna o `AuthResponseDTO`:

```
{
  "username": "anna",
```

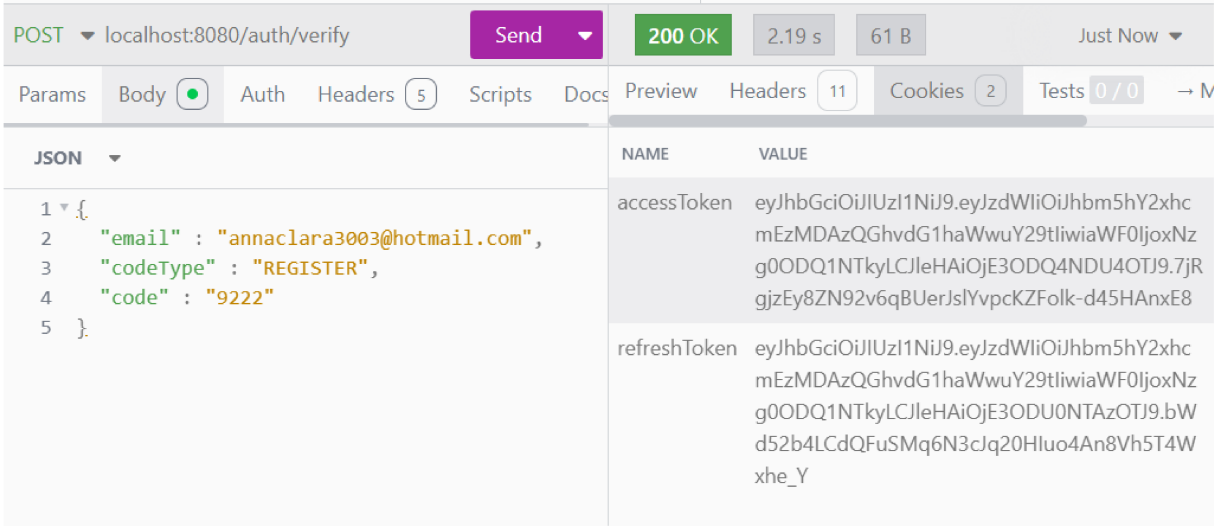
```
  "email": "anna@example.com"
}
```

Figura 5 – Execução da requisição `POST /auth/verify` no Insomnia



Fonte: Autoria própria

Figura 6 – Cookies emitidos pela execução da requisição `POST /auth/verify`



Fonte: Autoria própria

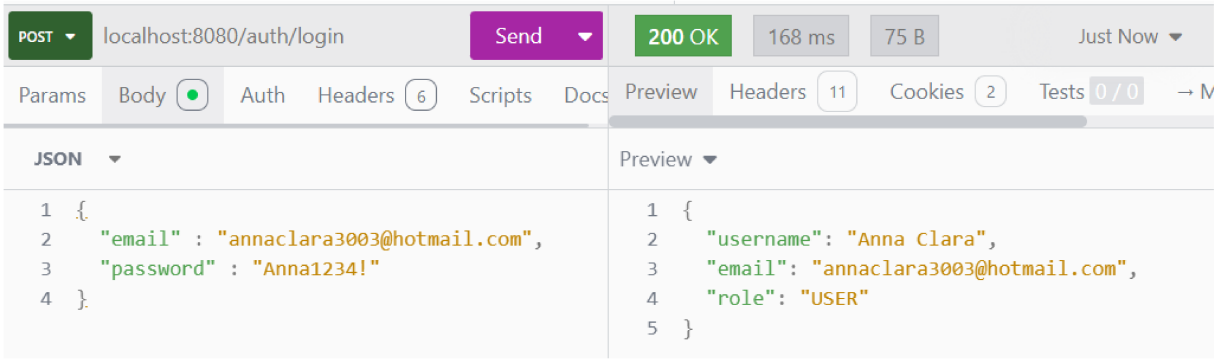
Como mostram as Figuras 5 e 6, nesse momento o campo `isEmailVerified` do usuário é marcado como verdadeiro, e os cookies `HttpOnly` `accessToken` e `refreshToken` já são emitidos na resposta – confirmando a integração entre verificação de e-mail e emissão de sessão, prevista no RF001 e no RF002. Entre os erros possíveis estão código inválido ou já utilizado (409), código expirado (409) e usuário não encontrado (404). O endpoint `POST /auth/resend-code` complementa esse fluxo, permitindo reenviar o código de registro (`200 OK`, mesmo formato de resposta) caso o usuário não o receba a tempo, sendo bloqueado quando o e-mail já foi verificado (409).

6.2.2 Autenticação de Usuários (RF002)

A autenticação é realizada pela requisição `POST /auth/login`, que recebe o seguinte corpo:

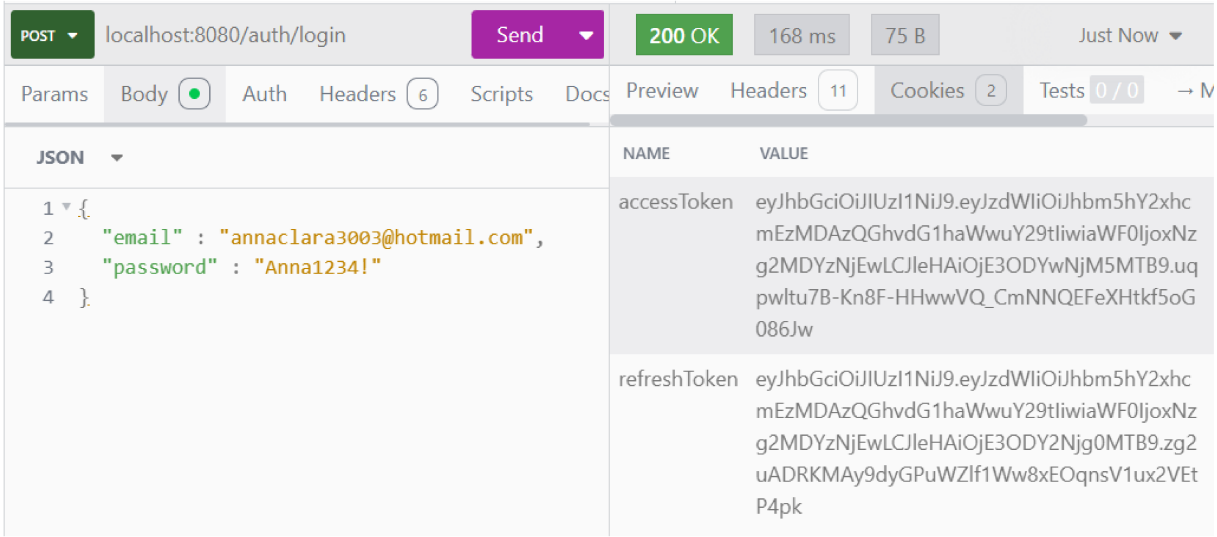
```
{
  "email": "anna@example.com",
  "password": "Senha@123"
}
```

Figura 7 – Execução da requisição `POST /auth/login` no Insomnia



Fonte: Autoria própria

Figura 8 – Cookies emitidos pela execução da requisição `POST /auth/login`



Fonte: Autoria própria

As Figuras 7 e 8 mostram, respectivamente, a execução da requisição e os cookies de sessão emitidos em resposta. A resposta de sucesso é idêntica em formato à

do `/auth/verify` (`AuthResponseDTO`, `200 OK`), com os mesmos cookies de sessão sendo emitidos. A Tabela 17 resume os erros observados.

Tabela 17 – Cenários de erro – `POST /auth/login`

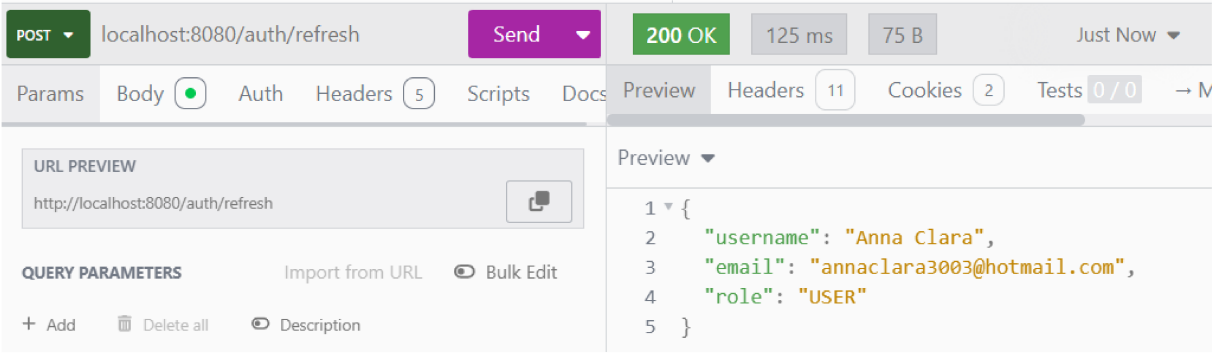
Cenário	Mensagem	Status
Credenciais inválidas	`E-mail ou senha incorretos.`	401
E-mail não verificado	`E-mail não verificado. Verifique sua caixa de entrada.`	403

O bloqueio de login para contas com e-mail não verificado confirma o comportamento especificado no RF001, evitando que uma conta incompleta acesse recursos protegidos. Cada tentativa – bem-sucedida ou não – é registrada pelo logger de auditoria `AUDIT` com o IP de origem, conforme descrito na seção 5.3.1.

6.2.3 Renovação e Encerramento de Sessão (RF003)

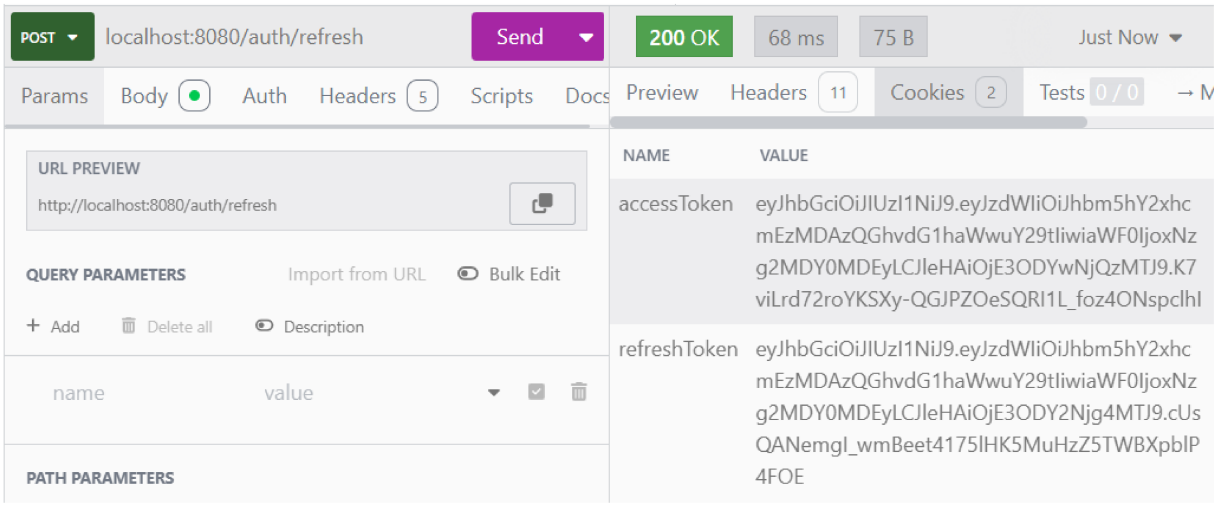
Diferentemente dos demais endpoints, a requisição `POST /auth/refresh` não possui corpo: o refresh token é lido do cookie `HttpOnly` enviado automaticamente pelo cliente. Em caso de sucesso (`200 OK`), o `AuthResponseDTO` é retornado e um novo par de cookies é emitido, revogando o refresh token anterior – implementando a rotação de tokens descrita no RNF001. A ausência do cookie, um token expirado ou um token já revogado resultam todos em `401 Unauthorized`, com mensagens distintas para cada caso (`Refresh token não encontrado.`, `Refresh token inválido ou expirado.` e `Refresh token revogado ou não encontrado.`, respectivamente).

Figura 9 – Execução da requisição `POST /auth/refresh` no Insomnia



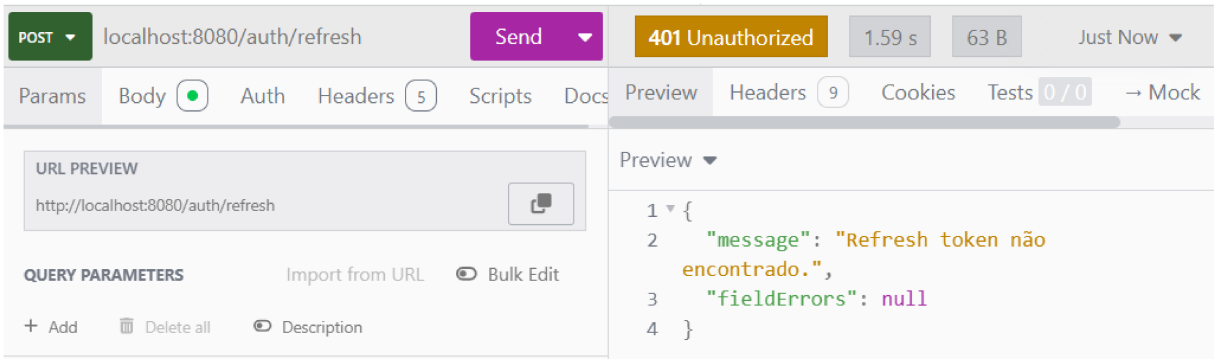
Fonte: Autoria própria

Figura 10 – Cookies emitidos pela execução da requisição `POST /auth/refresh`



Fonte: Autoria própria

Figura 11 – Cenário de erro da requisição `POST /auth/refresh` com refresh token não encontrado

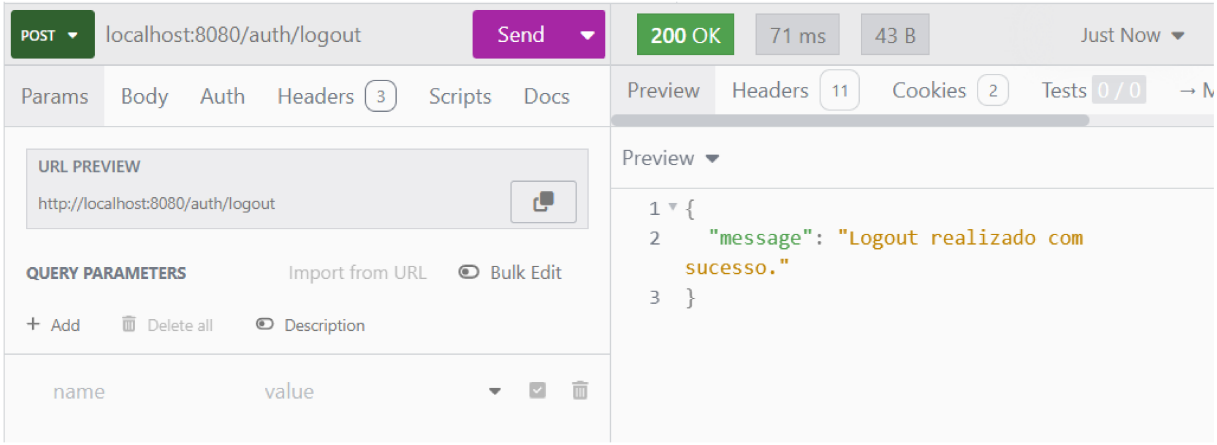


Fonte: Autoria própria

As Figuras 9 e 10 mostram a execução da requisição e o novo par de cookies emitido na resposta, enquanto a Figura 11 ilustra um dos cenários de erro descritos acima.

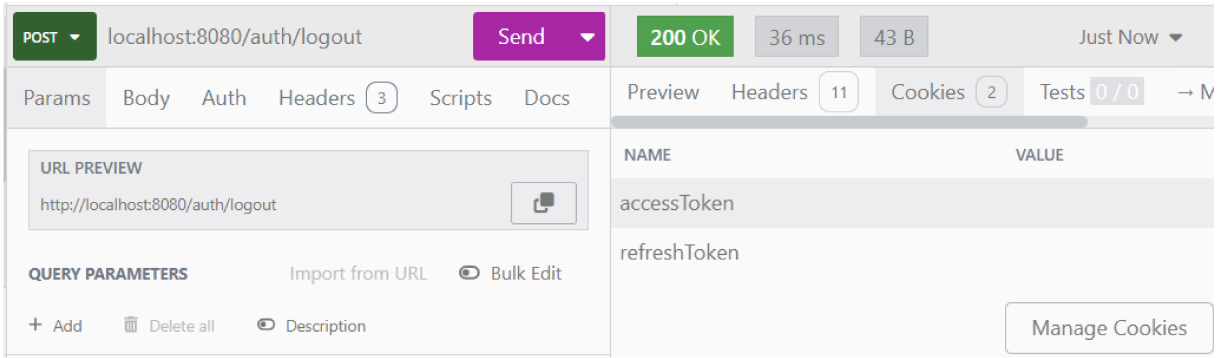
A requisição `POST /auth/logout`, também sem corpo, retorna sempre `200 OK` com a mensagem `"Logout realizado com sucesso."`, mesmo na ausência de um cookie válido – um comportamento tolerante a falhas, adequado a um endpoint de encerramento de sessão. Como efeito colateral, o refresh token é revogado no banco de dados (se existir) e ambos os cookies de sessão são limpos no cliente, completando o ciclo especificado no RF003.

Figura 12 – Execução da requisição `POST /auth/logout` no Insomnia



Fonte: Autoria própria

Figura 13 – Cookies removidos pela execução da requisição `POST /auth/logout`



Fonte: Autoria própria

As Figuras 12 e 13 confirmam, respectivamente, a resposta de sucesso e a remoção dos cookies de sessão no cliente.

6.2.4 Recuperação e Troca de Senha (RF004)

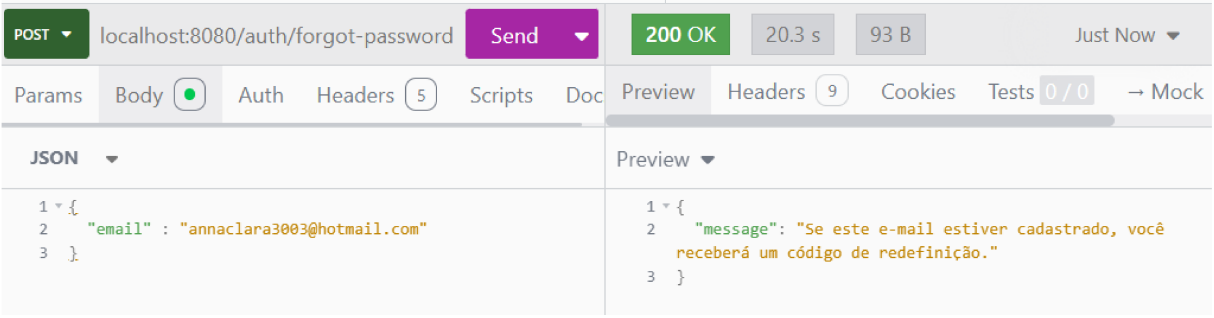
A recuperação de senha é iniciada pela requisição `POST /auth/forgot-password`, que recebe o seguinte corpo:

```
{
  "email": "anna@example.com"
}
```

Retorna `200 OK` com a mensagem `"Se este e-mail estiver cadastrado, você receberá um código de redefinição."` independentemente de o e-mail informado estar ou não cadastrado no sistema – o envio do código só ocorre quando o usuário

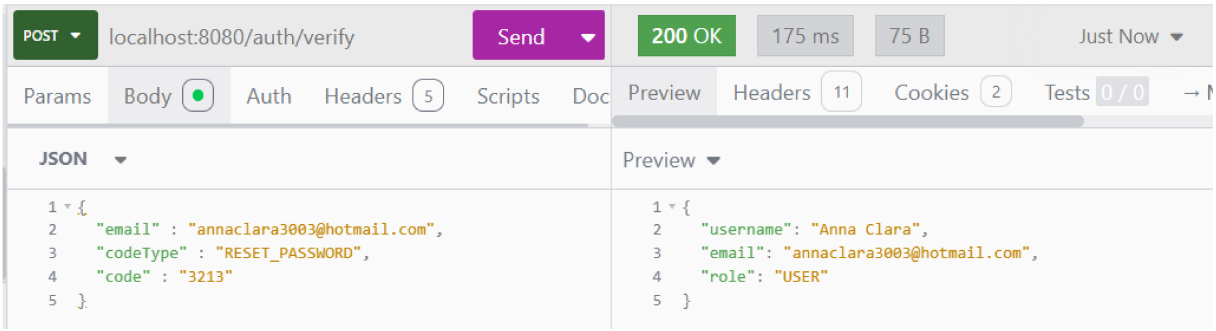
de fato existe, mas a resposta HTTP não distingue os dois casos, evitando que um solicitante infira pelo status da requisição se um endereço de e-mail está cadastrado (*user enumeration*). O endpoint `POST /auth/resend-password` segue o mesmo padrão de requisição, resposta e tratamento, reenviando o código de redefinição de senha.

Figura 14 – Execução da requisição `POST /auth/forgot-password` no Insomnia



Fonte: Autoria própria

Figura 15 – Código de redefinição de senha recebido por e-mail



Fonte: Autoria própria

A Figura 14 mostra a execução dessa requisição, e a Figura 15 mostra o código de redefinição recebido em seguida por e-mail.

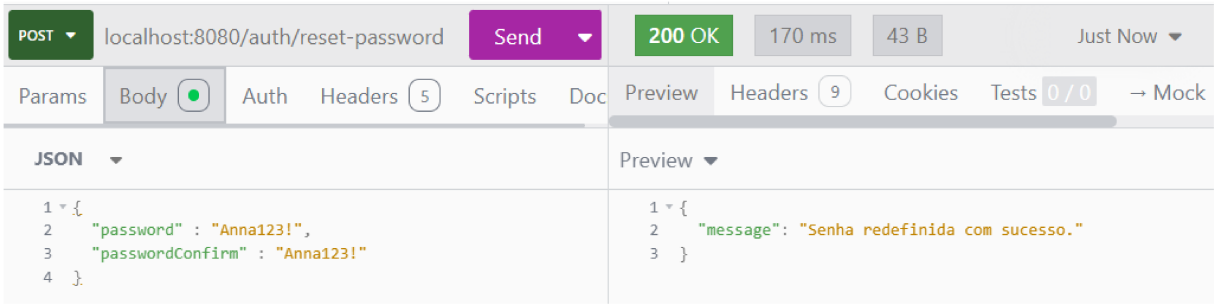
As requisições `POST /auth/reset-password` e `POST /auth/change-password` permitem a definição de uma nova senha, mas atendem a fluxos distintos. O `reset-password` completa o fluxo de recuperação de senha: a identidade do usuário é resolvida a partir do token de curta duração emitido por `/auth/verify` com `codeType=RESET_PASSWORD`, dispensando uma sessão de login completa – o que o torna utilizável por um usuário que perdeu a senha e não possui *access token* de sessão válido. Já o `change-password` exige autenticação de sessão normal e a confirmação da senha atual, sendo o endpoint de troca de senha para um usuário já logado. O corpo de `change-password` é:

```
{
```

```
  "currentPassword": "Senha@123",
  "newPassword": "NovaSenha@456",
  "newPasswordConfirm": "NovaSenha@456"
}
```

Ambos retornam `200 OK` em caso de sucesso (`"Senha redefinida com sucesso."` e `"Senha alterada com sucesso."`, respectivamente), com erros para senhas que não coincidem (400), usuário não encontrado (404) e, no caso de `change-password`, senha atual incorreta (409).

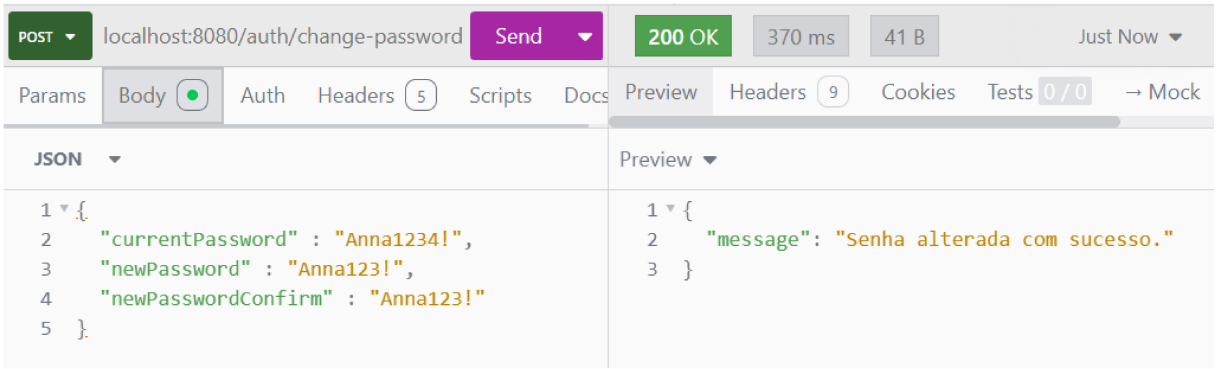
Figura 16 – Execução da requisição `POST /auth/reset-password` no Insomnia



Fonte: Autoria própria

A Figura 16 mostra a execução da requisição `POST /auth/reset-password` com o código recebido na etapa anterior.

Figura 17 – Execução da requisição `POST /auth/change-password` no Insomnia



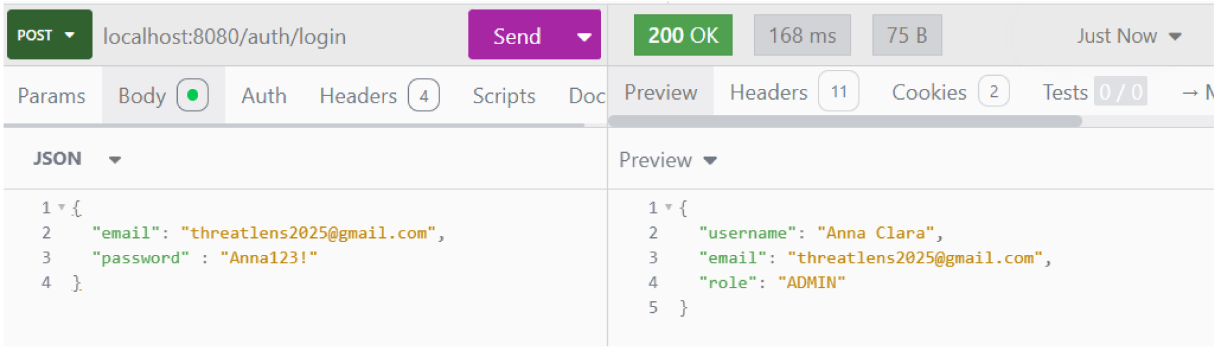
Fonte: Autoria própria

A Figura 17 mostra a execução dessa requisição por um usuário já autenticado, confirmando a troca de senha com a resposta `200 OK` descrita acima.

6.3 Módulo de Administração

O módulo de administração (RF005) é avaliado nesta seção. Todos os endpoints do prefixo `/admin` exigem, além de um token de acesso válido, que o usuário autenticado possua a *role* `ADMIN`, imposta pela anotação `@PreAuthorize("hasRole('ADMIN')")` sobre toda a classe `AdminController`. Uma requisição de um usuário autenticado sem essa *role* é rejeitada com `403 Forbidden`, tratado por um `@ExceptionHandler(AccessDeniedException.class)` dedicado no `GlobalExceptionHandler`, consistente com o formato de `ErrorResponseDTO` usado nos demais cenários de erro deste capítulo.

Figura 18 – Login como usuário administrador, utilizado para obter o *access token* com a *role* `ADMIN`



Fonte: Autoria própria

A Figura 18 mostra o login realizado com uma conta administradora, cujo *access token* é utilizado nas requisições desta seção.

A requisição `GET /admin/users` suporta paginação, filtro por *role* e busca textual por nome de usuário ou e-mail. A requisição a seguir ilustra o caso mais simples, sem nenhum filtro aplicado, retornando todos os usuários cadastrados:

```
GET /admin/users
```

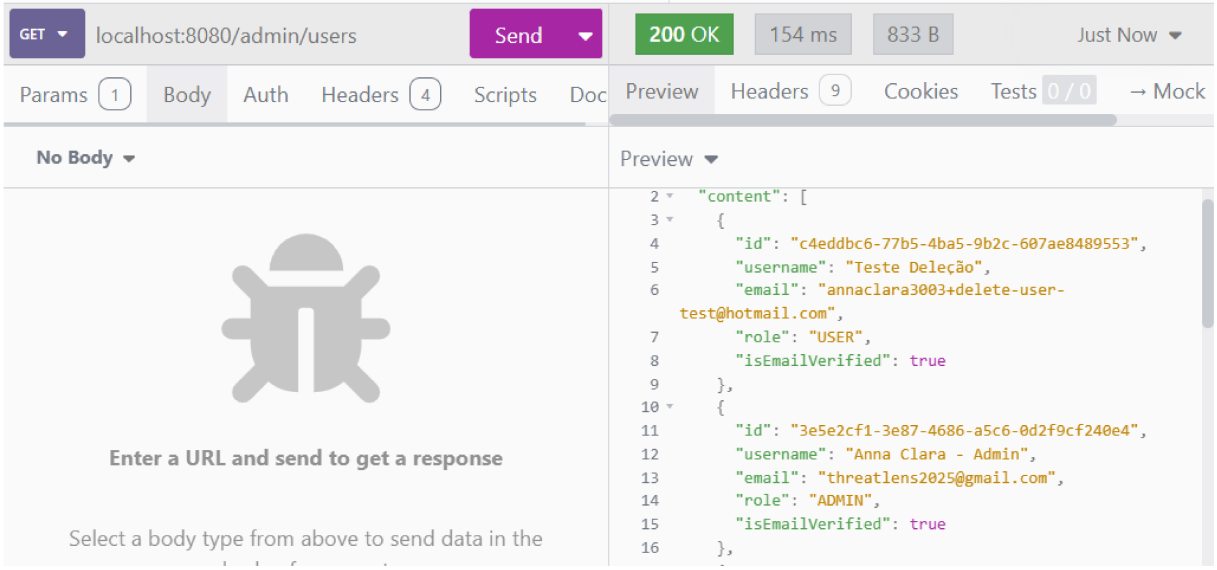
A resposta (`200 OK`) segue o formato genérico de paginação `PageResponseDTO`:

```
{
  "content": [
    {
      "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
      "username": "anna",
      "email": "anna@example.com",
      "role": "USER",
      "isEmailVerified": true
```

```
    }
    // demais usuários omitidos
  ],
  "page": 0,
  "size": 20,
  "totalElements": 5,
  "totalPages": 1,
  "last": true
}
```

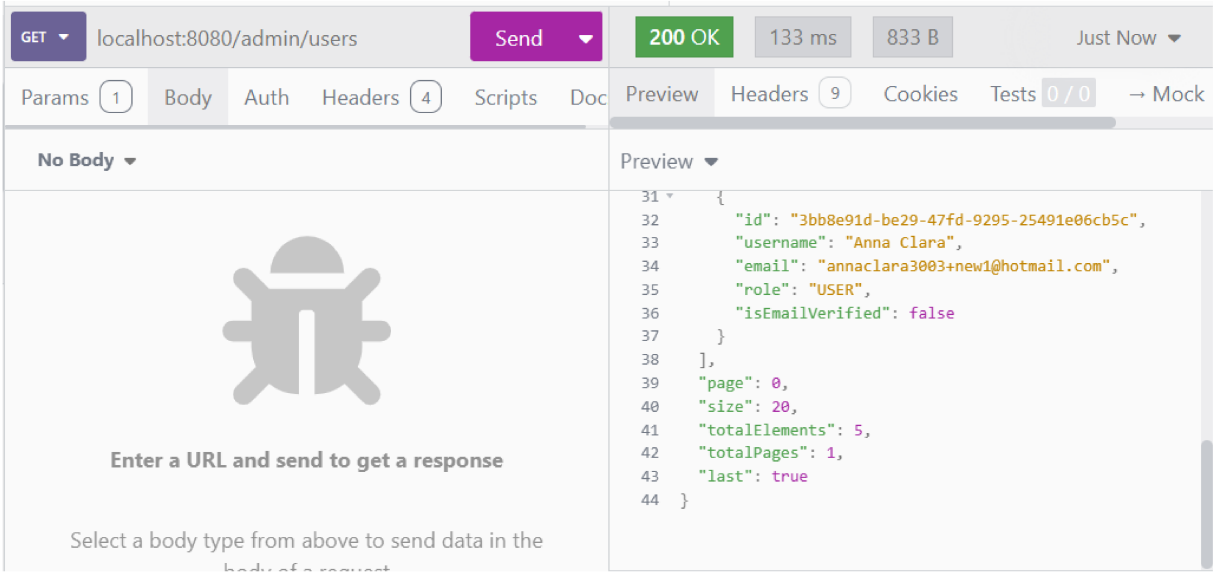

A busca textual por nome de usuário ou e-mail, quando utilizada por meio do parâmetro `search`, escapa os caracteres especiais do operador `LIKE` do SQL (`%`, `_`, `\`) antes de compor a consulta, prevenindo que uma entrada maliciosa altere seu comportamento – confirmando o tratamento descrito na seção 5.3.2.

Figura 19 – Execução da requisição `GET /admin/users` no Insomnia



Fonte: Autoria própria

Figura 20 – Metadados de paginação retornados pela requisição `GET /admin/users`



Fonte: Autoria própria

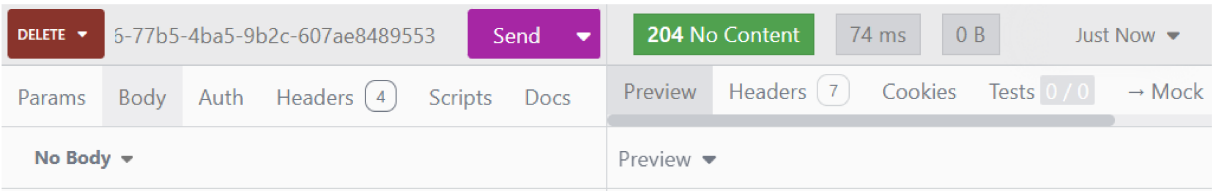
As Figuras 19 e 20 mostram, respectivamente, a execução da requisição com o token de administrador e os metadados de paginação retornados na resposta.

A requisição `DELETE /admin/users/{id}` remove um usuário e, em cascata, seus refresh tokens e códigos de verificação associados. Em caso de sucesso, retorna `204 No Content` (corpo vazio). A Tabela 18 resume os cenários de erro, que evidenciam as duas proteções de negócio especificadas no RF005.

Tabela 18 – Cenários de erro – `DELETE /admin/users/{id}`

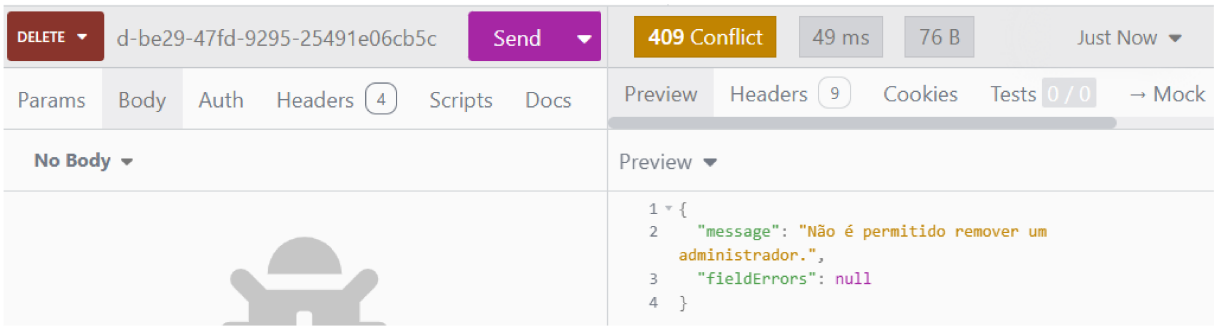
Cenário	Mensagem	Status
Usuário não encontrado	`Usuário não encontrado.`	404
Remoção do último administrador	`Não é permitido remover o último administrador.`	409

Figura 21 – Execução da requisição `DELETE /admin/users/{id}` no Insomnia



Fonte: Autoria própria

Figura 22 – Cenário de erro da requisição `DELETE /admin/users/{id}` ao tentar remover um administrador



Fonte: Autoria própria

A Figura 21 mostra a remoção bem-sucedida de um usuário comum, enquanto a Figura 22 evidencia a proteção contra remoção do último administrador restante, descrita na Tabela 18.

6.4 Módulo de Consulta de Posts

Esta seção avalia os endpoints do prefixo `/posts`, correspondentes aos requisitos funcionais RF006 a RF009 (seção 5.1.1). Conforme descrito na seção 6.1, essas requisições foram realizadas contra a instância real do banco externo `asgard`, refletindo o volume de dados coletado por Vieira (2025) e classificado por Jesus Filho (2023) no momento da avaliação. Ambos os endpoints são públicos apenas no sentido de não exigirem permissão administrativa, mas – assim como as demais rotas não listadas em `SecurityRoutes.PUBLIC` – exigem um token de acesso JWT válido.

6.4.1 Consulta de Posts com Filtros (RF006)

O endpoint `GET /posts` suporta filtragem por fonte, nível de relevância, categoria e período, além de ordenação e paginação configuráveis. A requisição a seguir ilustra o caso mais simples, sem filtros aplicados, retornando a página inicial da listagem completa de *posts*:

```
GET /posts?page=0&size=20
```

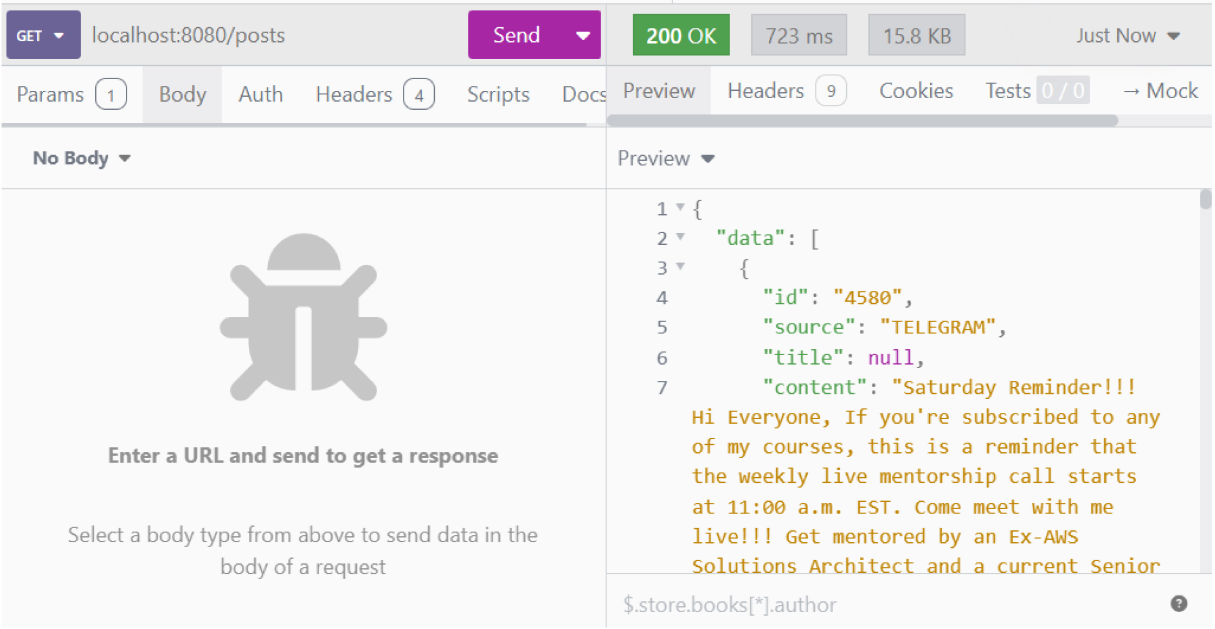
A resposta (`200 OK`) traz a lista de posts no campo `data`, seguida de metadados de paginação:

```
{
  "data": [
    {
```

```
      "id": "48213",
      "source": "TELEGRAM",
      "title": null,
      "content": "extremely critical rce phpmailer got php application
    ↪ ...",
      "author": null,
      "createdAt": "2026-03-14T09:22:10",
      "category": null,
      "classification": {
        "relevant": true,
        "score": 0.894,
        "level": "HIGH",
        "range": "7d",
        "ioc": true,
        "classifiedAt": "2026-03-14T10:05:33"
      },
      "meta": {
        "channelId": "1092",
        "channelName": "Cyber Security Experts",
        "channelUsername": "cybersecurityexperts"
      }
    }
  ],
  "pagination": {
    "page": 0,
    "size": 20,
    "total": 26757,
    "totalPages": 1338
  }
}
```

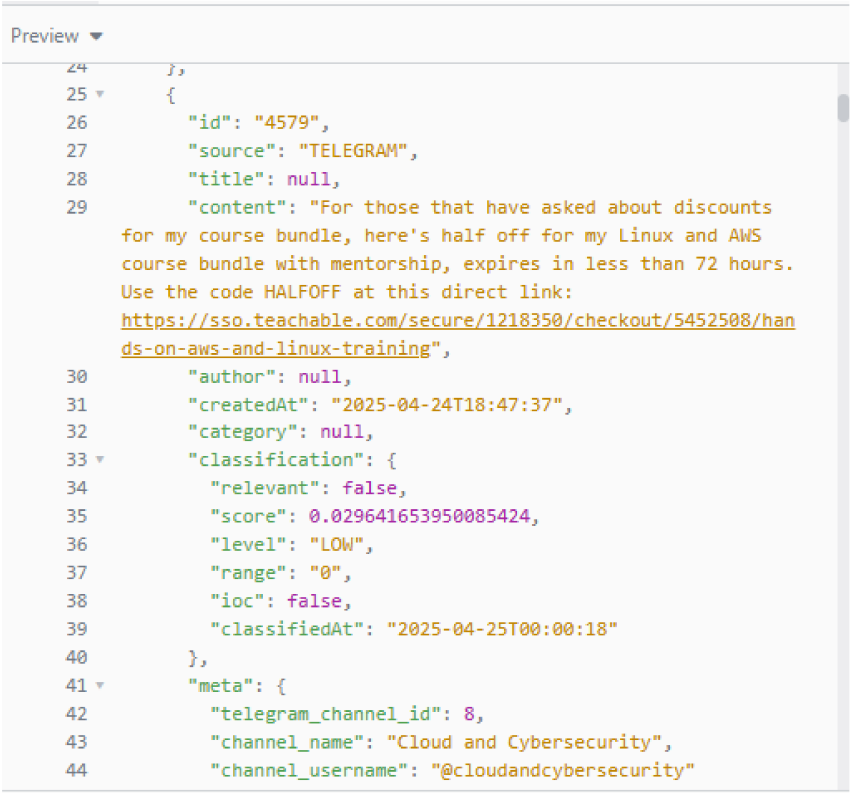

Como discutido na seção 5.3.3, os campos `title`, `author` e `category` retornam nulos para a fonte Telegram, cujo pipeline de coleta não produz essas informações – comportamento esperado do contrato genérico do `PostResponse`, projetado para acomodar múltiplas fontes sem alterar seu formato. O campo `meta` traz, neste caso, os dados do canal de origem no Telegram, preenchidos pela implementação `TelegramMeta`.

Figura 23 – Execução da requisição `GET /posts` no Insomnia



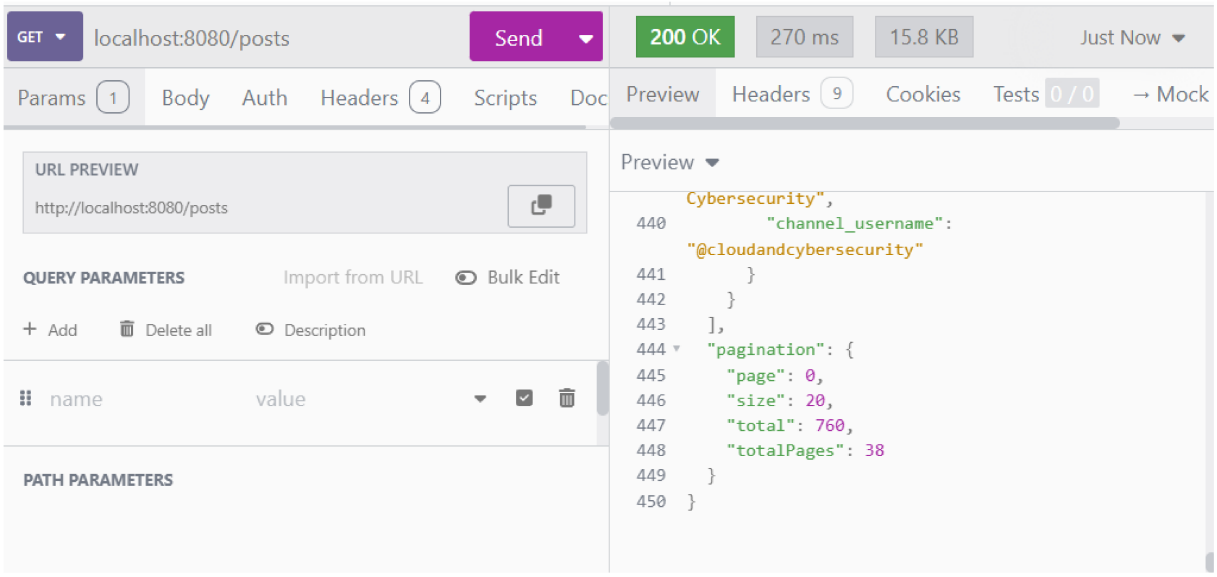
Fonte: Autoria própria

Figura 24 – Corpo da resposta da requisição `GET /posts`



Fonte: Autoria própria

Figura 25 – Metadados de paginação retornados pela requisição `GET /posts`



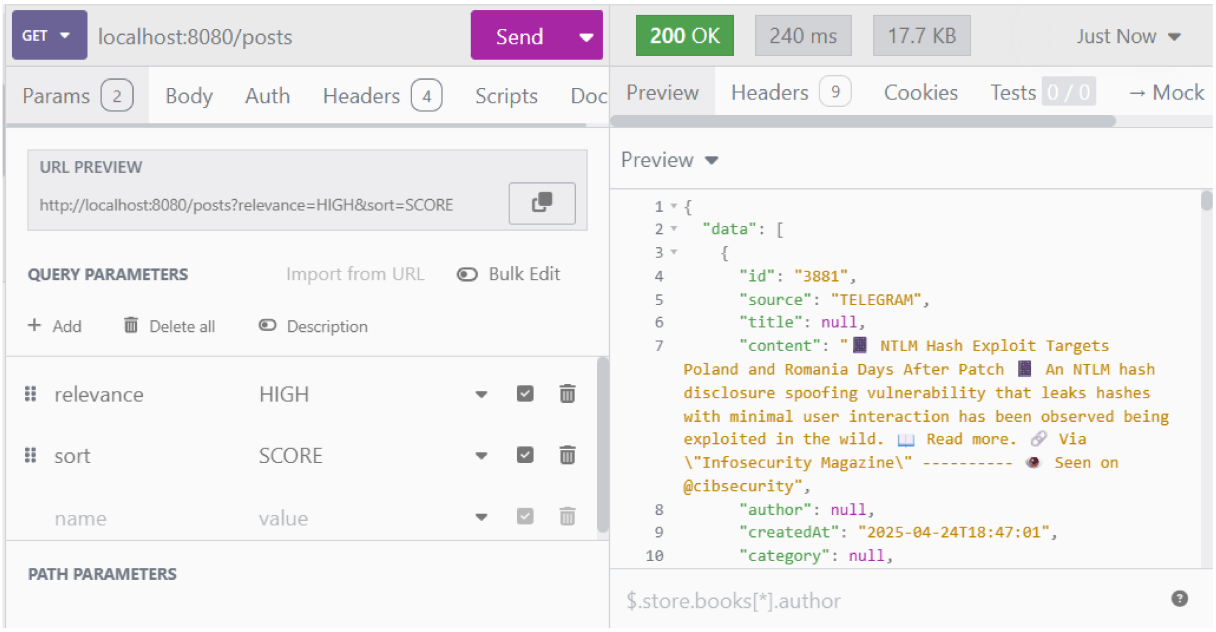
Fonte: Autoria própria

As Figuras 23 a 25 mostram, respectivamente, a execução da requisição contra o banco `asgard` real, o corpo de resposta com a lista de posts e os metadados de paginação retornados.

A combinação de filtros de `relevance` e ordenação por `sort` foi validada por meio da requisição a seguir, que retorna apenas os *posts* de relevância alta ordenados por *score* decrescente:

```
GET /posts?relevance=HIGH&sort=SCORE&order=DESC&page=0&size=20
```

Figura 26 – Execução da requisição `GET /posts` com filtro de relevância e ordenação por *score*



Fonte: Autoria própria

Figura 27 – Primeiro post retornado, evidenciando relevância `HIGH` e o maior *score* da página


```
{
  "data": [
    {
      "id": "3881",
      "source": "TELEGRAM",
      "title": null,
      "content": " NTLM Hash Exploit Targets Poland and Romania Days After Patch  An NTLM hash disclosure spoofing vulnerability that leaks hashes with minimal user interaction has been observed being exploited in the wild.  Read more.  Via \"Infosecurity Magazine\" ----------  Seen on @cibsecurity",
      "author": null,
      "createdAt": "2025-04-24T18:47:01",
      "category": null,
      "classification": {
        "relevant": false,
        "score": 0.9906197831284461,
        "level": "HIGH",
        "range": "10",
        "ioc": false,
        "classifiedAt": "2025-04-25T00:00:18"
      },
      "meta": {
        "telegram_channel_id": 1,
```


Fonte: Autoria própria

Figura 28 – Segundo post retornado, confirmando a ordenação decrescente por *score*


```
    {
      "id": "3992",
      "source": "TELEGRAM",
      "title": null,
      "content": "Ransomware Attack on Blue Yonder Disrupts
U.S. Supply Chains and Retail Operations Experts believe the
fallout will be felt across the United States, as Blue
Yonder's software supports numerous Fortune 500 companies. The
attack highlights vulnerabilities in private cloud
environments and the increasing threat ransomware poses to
supply chain infrastructure. Cyber_Security_Channel",
      "author": null,
      "createdAt": "2025-04-24T18:47:07",
      "category": null,
      "classification": {
        "relevant": false,
        "score": 0.9603421099130937,
        "level": "HIGH",
        "range": "10",
        "ioc": false,
        "classifiedAt": "2025-04-25T00:00:18"
      },
      "meta": {
        "telegram_channel_id": 2,
        "channel_name": "Cyber Security News",
        "channel_username": "@Cyber_Security_Channel"
```


Fonte: Autoria própria

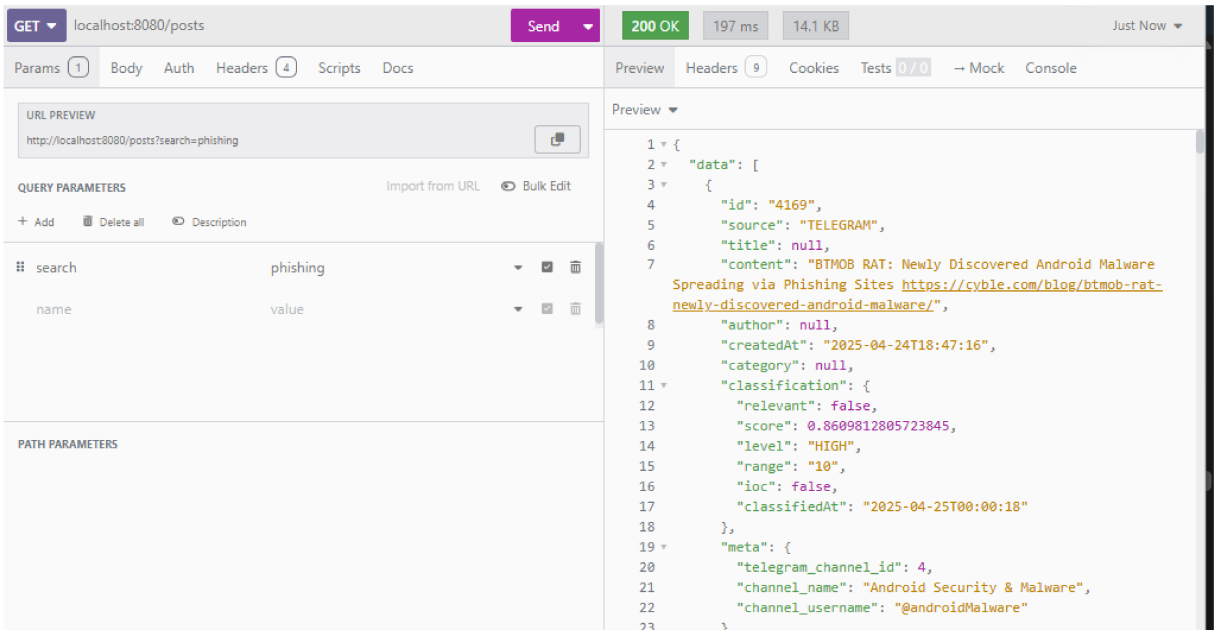
A Figura 26 mostra a execução bem-sucedida da requisição, e as Figuras 27 e 28 detalham, respectivamente, o primeiro e o segundo *post* da página retornada: ambos com `level` igual a `HIGH` e com o `score` do primeiro maior ou igual ao do segundo, confirmando que o filtro de relevância e a ordenação por *score* foram aplicados corretamente. As demais variações – `relevance=LOW`, `relevance=MEDIUM`, `sort=DATE` e os filtros de período – foram igualmente validadas de forma manual, restringindo corretamente o conjunto de resultados retornado em cada caso, sem a necessidade de demonstrar todas as combinações possíveis neste texto. As opções adicionais de ordenação por `ID`, `SOURCE` e `CONTENT`, incorporadas em uma versão posterior da API, seguem o mesmo mecanismo de ordenação já validado para `DATE` e `SCORE`.

Como apenas a fonte Telegram está integrada no escopo atual (seção 5.3.3), todas as requisições avaliadas nesta seção percorrem o caminho de paginação de fonte única, delegado diretamente ao banco via `OFFSET/LIMIT` – o campo `total` retornado reflete, portanto, a contagem real de registros que atendem ao filtro, sem as limitações de precisão que a estratégia de agregação em memória impõe ao cenário de múltiplas fontes simultâneas.

A busca por texto livre, via parâmetro `search`, foi validada por meio da requisição a seguir, que filtra os *posts* cujo conteúdo contenha o termo `phishing`:

```
GET /posts?search=phishing&page=0&size=20
```

Figura 29 – Execução da requisição `GET /posts` com filtro de busca textual (`search=phishing`)



Fonte: Autoria própria

A Figura 29 mostra a execução da requisição e o campo `total` retornado, que reflete apenas os registros cujo campo `message` satisfaz a condição `ILIKE '%phishing%'` no banco externo `asgard` – confirmando que o filtro é aplicado no banco, e não em memória. Combinações adicionais de `search` com `relevance` e com os parâmetros de ordenação são demonstradas no Anexo A, Cenário A.8.

6.4.2 Estatísticas Agregadas de Posts (RF007)

A requisição a seguir consulta as estatísticas agregadas sem nenhum filtro aplicado, cobrindo por padrão todo o histórico de *posts* (`period=ALL`, valor *default* quando o parâmetro é omitido):

```
GET /posts/stats
```

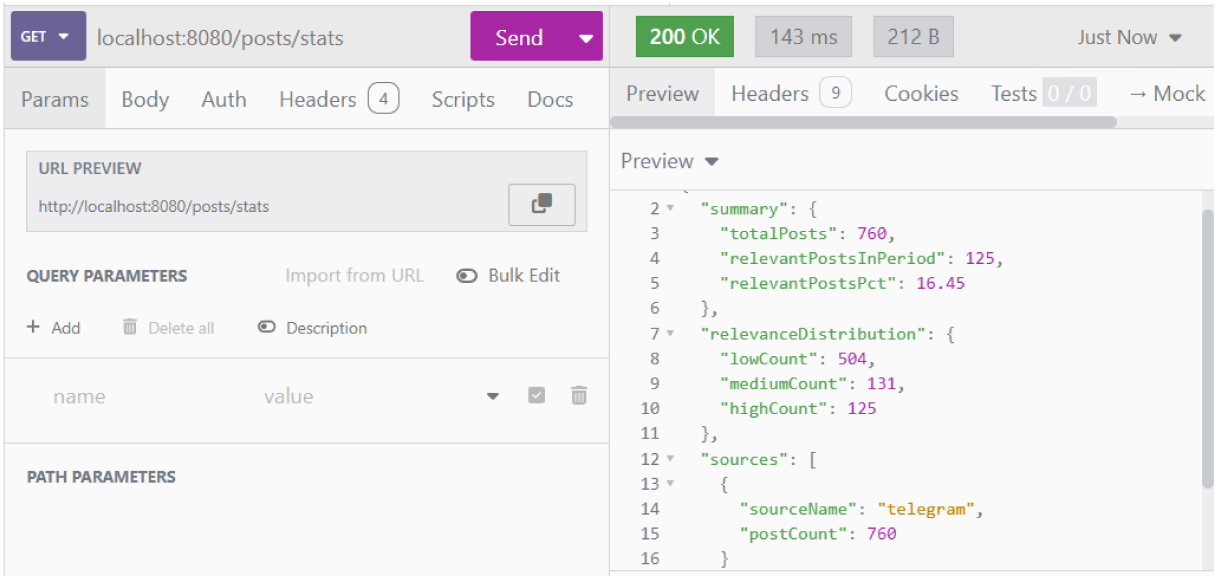
Retorna (`200 OK`) um resumo agregado de todo o histórico de *posts*:

```
{
  "summary": {
    "totalPosts": 26757,
    "relevantPostsInPeriod": 3341,
    "relevantPostsPct": 12.49
  },
```

```
  "relevanceDistribution": {
    "lowCount": 18000,
    "mediumCount": 5416,
    "highCount": 3341
  },
  "sources": [
    { "sourceName": "telegram", "postCount": 26757 }
  ]
}
```

O filtro de período (`period=DAY|WEEK|MONTH|YEAR|ALL`) foi validado manualmente, assim como sua substituição por um intervalo explícito via `from`/`to` – que, conforme especificado na arquitetura da aplicação, tem precedência sobre `period` quando ambos são informados. O percentual relatado em `relevantPostsPct` é consistente com a razão entre `relevantPostsInPeriod` e `totalPosts` arredondada a duas casas decimais, confirmando o cálculo especificado no RF007.

Figura 30 – Execução da requisição `GET /posts/stats` no Insomnia



Fonte: Autoria própria

A Figura 30 mostra a execução dessa requisição contra o banco `asgard` real, confirmando visualmente os valores usados no exemplo acima.

6.4.3 Nuvem de Palavras dos Posts (RF008)

O endpoint `GET /posts/wordcloud` reaproveita os mesmos filtros de fonte, relevância, categoria e período de `GET /posts/stats`, acrescentando apenas o parâmetro

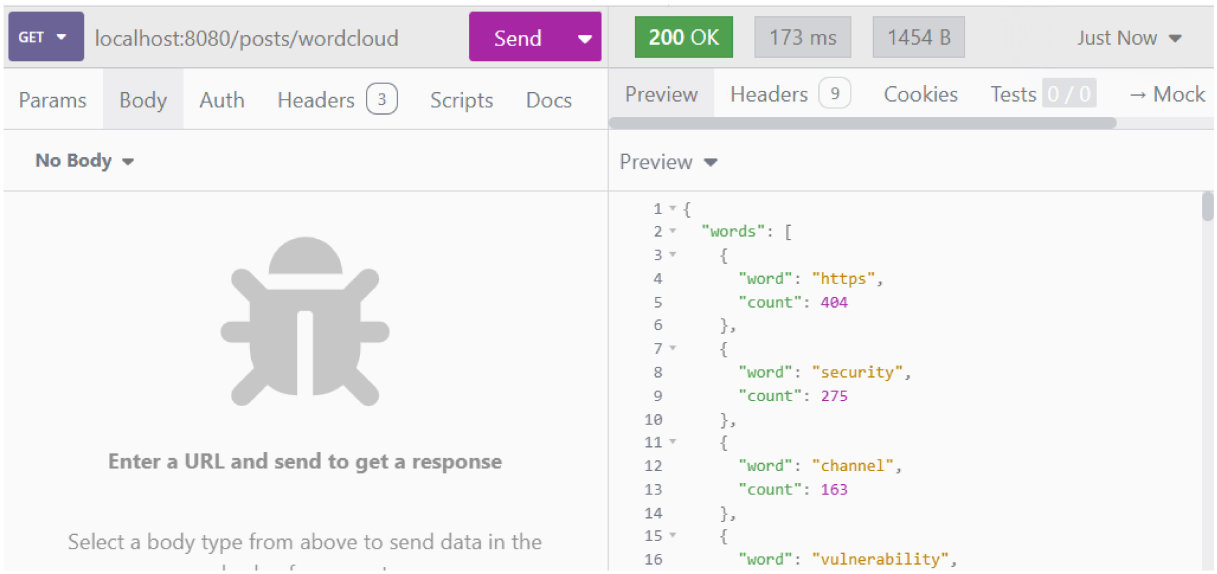
`limit`, que define a quantidade de palavras retornadas. A requisição a seguir ilustra o caso mais simples, sem nenhum filtro aplicado, usando os valores *default* de `period` (`ALL`) e de `limit` (50):

```
GET /posts/wordcloud
```

A resposta (`200 OK`) traz a lista de palavras mais frequentes no campo `words`, já ordenada por contagem decrescente e truncada em `limit` itens:

```
{
  "words": [
    { "word": "phishing", "count": 128 },
    { "word": "malware", "count": 97 }
  ]
}
```

Figura 31 – Execução da requisição `GET /posts/wordcloud` no Insomnia



Fonte: Autoria própria

A Figura 31 mostra a execução dessa requisição contra o banco `asgard` real e o corpo de resposta com a lista de palavras mais frequentes de todo o histórico de *posts*, confirmando o formato descrito acima.

A combinação de filtros foi validada manualmente com variações de `period`, `relevance` e `limit`, confirmando que a nuvem de palavras reflete corretamente o subconjunto de *posts* filtrado em cada caso e que o truncamento respeita o valor de `limit` informado, sem a necessidade de demonstrar todas as combinações possíveis neste texto.

Como apenas a fonte Telegram está integrada no escopo atual (seção 5.3.3), a requisição avaliada percorre sempre o caminho de fonte única, sem *overfetch* – o cenário de *merge* de contagens entre múltiplas fontes, descrito na seção 5.3.3, permanece sem uma execução real demonstrável enquanto apenas uma fonte estiver ativa. Da mesma forma, a dependência de *locale* do banco externo para o correto tratamento de letras acentuadas na tokenização, também discutida naquela seção, não é algo verificável por meio desta requisição isolada.

6.4.4 Suporte a Múltiplas Fontes de Dados (RF009)

O RF009 não é avaliado por meio de uma requisição direta, já que nenhuma fonte além do Telegram está integrada no escopo atual do projeto (seção 5.3.3). Sua avaliação é, portanto, arquitetural: a existência da interface `PostsSourceRepository` – implementada atualmente apenas por `TelegramRepository` e injetada como uma lista no `PostsService` – confirma que a incorporação de uma nova fonte exigiria apenas uma nova classe que implemente essa interface, sem qualquer alteração no serviço ou nos endpoints já avaliados nas seções anteriores. Este ponto é retomado na discussão de trabalhos futuros ao final deste capítulo.

6.5 Requisitos Não Funcionais

Os requisitos funcionais avaliados nas seções anteriores fornecem, por si mesmos, parte das evidências necessárias para verificar os requisitos não funcionais: os comportamentos observados nas requisições já demonstram aspectos de segurança, arquitetura e operação que os RNFs prescrevem. Esta seção retoma esses quatro requisitos explicitamente, apontando onde cada um se manifesta nos resultados já apresentados e, quando necessário, descrevendo a evidência complementar.

O **RNF001** (Segurança de Credenciais) é verificável em três momentos distintos da avaliação. Primeiro, no registro: a Figura 3 mostra que a resposta do `POST /auth/register` não contém nem devolve a senha informada – ela é imediatamente submetida ao *hash* BCrypt antes da persistência, como confirmado pelo próprio texto de resposta, que indica apenas o envio do código de verificação. Segundo, nos cookies: as Figuras 6, 8 e 10 mostram que os tokens de acesso e de renovação são sempre emitidos como cookies `HttpOnly`, atributo que impede seu acesso por *scripts* JavaScript no cliente, mitigando ataques XSS. Terceiro, na rotação de tokens: a Figura 10 evidencia que cada uso do `POST /auth/refresh` revoga o *refresh token* anterior e emite um novo par, de modo que um token capturado em trânsito tem janela de uso limitada ao intervalo entre duas renovações.

O **RNF002** (Autenticação *Stateless*) é verificável pelo próprio funcionamento dos

endpoints protegidos ao longo deste capítulo: em nenhum momento a aplicação mantém estado de sessão no servidor. A identidade do usuário é resolvida integralmente a partir da assinatura e das *claims* do JWT presente no cookie `accessToken` – sem consulta ao banco de dados a cada chamada. Isso é observável indiretamente: requisições às rotas protegidas (`/admin`, `/auth/change-password`, `/posts`) produzem `401 Unauthorized` na ausência do token (ou com token inválido), e `200 OK` na sua presença, sem qualquer mecanismo de sessão server-side envolvido. A Figura 11 reforça esse ponto: a rejeição de um *refresh token* revogado é verificada contra o banco apenas nessa rota específica – que, por sua natureza, precisa garantir que o token não foi previamente invalidado –, sendo essa a única exceção ao fluxo puramente *stateless*.

O **RNF003** (Versionamento do Esquema de Banco de Dados) é verificável pelo log de inicialização da aplicação. Ao executar `./mvnw spring-boot:run` com o perfil `dev`, o Flyway aplica automaticamente os *scripts* de migração versionados presentes em `src/main/resources/db/migration`, provisionando o banco primário sem qualquer intervenção manual – o mesmo processo que reproduz o ambiente descrito na seção 6.1. A rastreabilidade das migrações é registrada na tabela `flyway_schema_history` do banco primário, que lista cada versão aplicada, seu *checksum* e o instante de execução, permitindo identificar o estado exato do esquema em qualquer ponto do histórico de desenvolvimento.

O **RNF004** (Separação entre Dados Próprios e Dados Externos) é verificável por dois ângulos complementares. Do lado da aplicação, o Flyway está configurado para operar exclusivamente sobre o *datasource* primário (`PrimaryDataSourceConfig`), nunca sobre o `PostsDataSourceConfig` que aponta para o banco externo `asgard`: nenhuma migração toca o esquema de *posts*. Do lado da avaliação, todas as requisições ao prefixo `/posts` apresentadas na seção 6.4 leram dados do banco externo sem qualquer efeito colateral sobre o banco primário – e vice-versa: as operações de autenticação e administração avaliadas nas seções 6.2 e 6.3 não acessam o banco `asgard` em nenhum momento. A separação de responsabilidades entre os dois domínios de dados opera de forma transparente e isolada durante toda a avaliação.

7 Conclusões

Este trabalho teve como objetivo o desenvolvimento de uma API REST responsável por centralizar, autenticar e disponibilizar, de forma segura, filtrada e agregada, os dados de mensagens já coletadas e classificadas por inteligência de ameaças cibernéticas – preenchendo a lacuna de integração identificada entre os trabalhos de Vieira (2025), Jesus Filho (2023), Reis (2024) e Borges (2025), discutidos no Capítulo 2.

Os cinco objetivos específicos definidos foram alcançados:

- O módulo de autenticação e gestão de usuários foi implementado com registro, verificação de e-mail, login *stateless* baseado em JWT com *refresh token* persistido e revogável, recuperação e troca de senha, e controle de acesso por papéis;
- O módulo de consulta de mensagens foi implementado com suporte a filtros por fonte, relevância, categoria, período e busca por texto livre no conteúdo das mensagens, além de paginação e ordenação configuráveis;
- O *endpoint* de estatísticas agregadas foi disponibilizado, retornando totais, percentuais de relevância e distribuição por fonte e por nível de relevância;
- O *endpoint* de nuvem de palavras foi disponibilizado, calculando a frequência dos termos mais utilizados no conteúdo dos *posts* filtrados, com suporte aos mesmos filtros de fonte, relevância, categoria e período das estatísticas agregadas;
- A camada de acesso a dados foi projetada de forma extensível, por meio do Padrão Strategy, permitindo a incorporação de novas fontes sem alteração da lógica de negócio já implementada – ainda que, no momento da escrita deste trabalho, apenas a fonte Telegram esteja de fato integrada.

Nesse contexto, a principal contribuição deste trabalho é a camada de API que conecta, pela primeira vez, os dados produzidos separadamente pelos quatro trabalhos relacionados a um consumidor externo – um *front-end* de dashboard. Diferentemente das plataformas genéricas de mercado discutidas na seção 2.2 (MISP e OpenCTI), a aplicação desenvolvida é uma API de propósito específico, construída sob medida para o volume e o formato de dados já produzidos pela linha de pesquisa em que este trabalho está inserido, sem a sobrecarga arquitetural de uma plataforma de compartilhamento genérica entre organizações.

Algumas limitações do trabalho, já discutidas ao longo do desenvolvimento, merecem destaque. A estratégia de paginação para múltiplas fontes de dados simultâneas

impõe um limite fixo de registros buscados por fonte, o que pode comprometer a precisão do total reportado e o ordenamento de páginas mais distantes caso mais de uma fonte esteja ativa – limitação sem efeito prático no escopo atual, em que apenas o Telegram está integrado, mas relevante para a evolução futura do sistema. O mecanismo de limitação de requisições por IP opera em memória local, não sendo compartilhado entre múltiplas instâncias da aplicação em um cenário de escalabilidade horizontal. Por fim, o módulo de geração e notificação de alertas desenvolvido por Borges (2025) permanece como um sistema independente, não integrado à API desenvolvida neste trabalho.

A partir das limitações apontadas, sugerem-se as seguintes direções para trabalhos futuros:

- Integrar o módulo de geração e notificação de alertas de Borges (2025) à API desenvolvida, unificando os dois sistemas em uma única camada de consumo;
- Incorporar novas fontes de dados à camada de acesso já projetada de forma extensível, como mensagens coletadas da Deep Web;
- Substituir a estratégia atual de paginação em múltiplas fontes por uma tabela unificada ou por paginação baseada em cursor (*keyset pagination*), eliminando o limite fixo de registros por fonte;
- Substituir o mecanismo de limitação de requisições por IP, atualmente baseado em memória local, por uma solução de cache distribuído, como Redis com a biblioteca Bucket4j, viabilizando a escalabilidade horizontal da aplicação;

Apesar das limitações apontadas, o objetivo geral deste trabalho foi cumprido: a lacuna de integração identificada entre os trabalhos de Vieira (2025), Jesus Filho (2023), Reis (2024) e Borges (2025) – a ausência de uma camada única, segura e consultável sobre os dados já coletados e classificados – foi preenchida pela API desenvolvida neste trabalho, estabelecendo uma base sólida e extensível sobre a qual as direções futuras aqui apontadas podem ser construídas.

Referências

Apache Software Foundation. *Apache Maven*. 2026. <https://maven.apache.org>. Acesso em: 20 jun. 2026. Citado na página 20.

AssertJ Contributors. *AssertJ — Fluent Assertions for Java*. 2026. <https://assertj.github.io/doc>. Acesso em: 20 jun. 2026. Citado na página 25.

BORGES, M. H. G. *Desenvolvimento de um sistema de geração de alertas no contexto de ameaças cibernéticas*. Dissertação (Trabalho de Conclusão de Curso – Graduação em Sistemas de Informação) — Universidade Federal de Uberlândia, Uberlândia, Brasil, 2025. 36 p. Data de defesa: 28-abr-2025. Disponível em: <https://repositorio.ufu.br/handle/123456789/45494>. Citado 6 vezes nas páginas 13, 15, 16, 17, 66 e 67.

CYSNEIROS, L. M.; LEITE, J. C. S. d. P. Definindo requisitos não funcionais. *XI Simpósio Brasileiro de Engenharia de Software*, Fortaleza, Brasil, p. 33, 1997. Citado na página 28.

Filigran. *OpenCTI Documentation*. 2026. <https://docs.opencti.io/>. Documentação oficial do fabricante (não é artigo acadêmico revisado por pares). Citado na página 16.

G1. *Megavazamento de dados de 223 milhões de brasileiros: o que se sabe e o que falta saber*. 2021. <https://g1.globo.com/economia/tecnologia/noticia/2021/01/28/vazamento-de-dados-de-223-milhoes-de-brasileiros-o-que-se-sabe-e-o-que-falta-saber.ghtml>. Acesso em: 13 jul. 2025. Citado na página 13.

H2 Database Engine. *H2 Database Engine*. 2026. <https://h2database.com>. Acesso em: 20 jun. 2026. Citado na página 24.

IBM. *X-Force Threat Intelligence Index 2023*. 2023. <https://www.ibm.com/reports/threat-intelligence>. Acesso em: 20 jul. 2025. Citado na página 13.

JESUS FILHO, S. A. d. *Identificação de posts maliciosos na dark web utilizando Aprendizado de Máquina Supervisionado*. Dissertação (Dissertação de Mestrado) — Universidade Federal de Uberlândia, Uberlândia, Brasil, 2023. Disponível em: <https://repositorio.ufu.br/handle/123456789/41232>. Citado 7 vezes nas páginas 13, 15, 17, 42, 55, 66 e 67.

JONES, M.; BRADLEY, J.; SAKIMURA, N. *JSON Web Token (JWT)*. [S.l.], 2015. Disponível em: <https://datatracker.ietf.org/doc/html/rfc7519>. Acesso em: 20 jun. 2026. Citado na página 21.

JUnit Team. *JUnit 5*. 2026. <https://junit.org/junit5>. Acesso em: 20 jun. 2026. Citado na página 25.

Mockito Contributors. *Mockito Framework*. 2026. <https://site.mockito.org>. Acesso em: 20 jun. 2026. Citado na página 25.

Okta, Inc. *JJWT — Java JWT*. 2026. <https://github.com/jwtk/jjwt>. Acesso em: 20 jun. 2026. Citado na página 21.

Oracle Corporation. *Java 21 Documentation*. 2026. <https://docs.oracle.com/en/java/javase/21>. Acesso em: 20 jun. 2026. Citado na página 20.

Oracle Corporation. *Java Database Connectivity (JDBC)*. 2026. <https://www.oracle.com/java/technologies/javase/javase-tech-database.html>. Acesso em: 20 jun. 2026. Citado na página 23.

POSTEL, J. *Simple Mail Transfer Protocol*. [S.l.], 1982. Disponível em: <https://datatracker.ietf.org/doc/html/rfc821>. Acesso em: 20 jun. 2026. Citado na página 24.

PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. 2026. <https://www.postgresql.org>. Acesso em: 20 jun. 2026. Citado na página 23.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma Abordagem Profissional*. 8. ed. Porto Alegre: McGraw-Hill, 2016. Citado na página 26.

Project Lombok. *Project Lombok*. 2026. <https://projectlombok.org>. Acesso em: 20 jun. 2026. Citado na página 25.

PROVOS, N.; MAZIÈRES, D. A future-adaptable password scheme. In: *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1999. Disponível em: <https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf>. Acesso em: 20 jun. 2026. Citado na página 22.

RAHMAN, M. R.; HEZAVEH, R. M.; WILLIAMS, L. What are the attackers doing now? automating cyberthreat intelligence extraction from text on pace with the changing threat landscape: a survey. *ACM Computing Surveys*, Association for Computing Machinery, New York, NY, USA, 2023. ISSN 0360-0300. Citado na página 13.

Red Hat, Inc. *Hibernate ORM*. 2026. <https://hibernate.org/orm>. Acesso em: 20 jun. 2026. Citado na página 22.

Redgate Software. *Flyway — Database Migrations Made Easy*. 2026. <https://flywaydb.org>. Acesso em: 20 jun. 2026. Citado na página 24.

REIS, A. L. F. *Análise de posts maliciosos na Dark Web usando aprendizado de máquina não supervisionado*. Dissertação (Trabalho de Conclusão de Curso) — Universidade Federal de Uberlândia, Uberlândia, Brasil, 2024. Monografia (Trabalho de Conclusão de Curso). Disponível em: <https://repositorio.ufu.br/handle/123456789/44118>. Citado 5 vezes nas páginas 13, 15, 17, 66 e 67.

Security Leaders. *Brasil puxa crescimento recorde de ciberataques na América Latina, diz Threat Intel*. 2025. <https://securityleaders.com.br/brasil-puxa-crescimento-recorde-de-ciberataques-na-america-latina-aponta-threat-intel/>. Acesso em: 13 jul. 2025. Citado na página 13.

VIEIRA, C. R. *Desenvolvimento de um bot de captura de mensagens do Telegram para inteligência de ameaças cibernéticas*. Dissertação (Trabalho de Conclusão de Curso – Graduação em Ciência da Computação) — Universidade Federal de Uberlândia, Uberlândia, Brasil, 2025. 47 p. Data de defesa: 14-abr-2025. Disponível em: <https://repositorio.ufu.br/handle/123456789/45319>. Citado 7 vezes nas páginas 13, 15, 17, 42, 55, 66 e 67.

VMware, Inc. *Spring Boot*. 2026. <https://spring.io/projects/spring-boot>. Acesso em: 20 jun. 2026. Citado na página 21.

VMware, Inc. *Spring Data JPA*. 2026. <https://spring.io/projects/spring-data-jpa>. Acesso em: 20 jun. 2026. Citado na página 22.

VMware, Inc. *Spring Email*. 2026. <https://spring.io/guides/gs/sending-email>. Acesso em: 20 jun. 2026. Citado na página 24.

VMware, Inc. *Spring Security*. 2026. <https://spring.io/projects/spring-security>. Acesso em: 20 jun. 2026. Citado na página 22.

WAGNER, C. et al. MISP: the design and implementation of a collaborative threat intelligence sharing platform. In: *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security (WISCS'16)*. New York, NY, USA: ACM, 2016. p. 49–56. Disponível em: <https://doi.org/10.1145/2994539.2994542>. Citado na página 16.

Apêndices

APÊNDICE A – Declaração de uso de inteligência artificial generativa

Para a produção deste trabalho, foi utilizado o Claude (Anthropic), em diferentes versões disponibilizadas pela plataforma, principalmente entre maio e julho de 2026, mas também em etapas anteriores do desenvolvimento do projeto. A ferramenta atuou como apoio ao longo de todo o desenvolvimento do *back-end* da aplicação ThreatLens, bem como na escrita e na revisão deste documento, sempre sob supervisão, avaliação e decisão finais da autora. O uso ocorreu nas seguintes etapas: apoio na definição da estruturação e da arquitetura do projeto e dos módulos do *back-end*; apoio na configuração dos dois bancos de dados utilizados pela aplicação (o banco primário, via Spring Data JPA, e o banco externo de posts, via JDBC); apoio na criação e na revisão dos testes automatizados; elaboração de documentação técnica do projeto (arquitetura, *endpoints* e decisões de implementação), posteriormente utilizada como base para a escrita das seções de desenvolvimento deste trabalho; apoio na estruturação e reorganização de capítulos e seções deste documento; apoio na elaboração e execução de avaliações e testes dos *endpoints* da API; e apoio na revisão e formatação do documento em LaTeX – incluindo a padronização de tabelas, figuras e blocos de código, a correção de erros de compilação e a revisão de trechos do texto quanto à clareza, à coesão e à concordância, conforme apontado pelo orientador.

Cabe ressaltar os limites desse uso: em nenhum momento a ferramenta operou de forma autônoma sobre o ambiente da aplicação, executou implantações ou gerou, por conta própria, os resultados das avaliações apresentadas neste trabalho. Cada sugestão de código, texto ou estrutura recebida passou pela avaliação da autora antes de ser incorporada; a execução dos testes contra a API em funcionamento e a conferência de todos os valores, comportamentos e respostas descritos neste trabalho foram feitas manualmente pela autora, que assume a responsabilidade por sua exatidão.

Todo conteúdo gerado pela ferramenta foi revisado criticamente pela autora, que assume integral responsabilidade pelo trabalho final.

APÊNDICE B – Combinações de Filtros dos Endpoints de Posts

Este anexo complementa a Seção 6.4, registrando as combinações de parâmetros de consulta testadas manualmente para os três endpoints do módulo de posts: `GET /posts`, `GET /posts/stats` e `GET /posts/wordcloud`.

O ponto de partida do cenário é o uso real do *dashboard*: o analista abre a página de posts, aplica os mesmos filtros — fonte, relevância e janela temporal — e o *frontend* dispara, em paralelo, as três requisições para alimentar respectivamente a listagem paginada, o painel de estatísticas e o gráfico de nuvem de palavras. Cada seção deste anexo representa uma dessas sessões de uso, variando o conjunto de filtros aplicados e apresentando, para cada variação, as três requisições com seus pares de resposta esperada.

Como o banco `asgard` encerrou a coleta de dados em 2025, os filtros de período por atalho (`period=DAY|WEEK|MONTH|YEAR`) retornam conjuntos vazios e não foram incluídos. Todas as janelas temporais são definidas por intervalo explícito via `from`/`to`, que tem precedência sobre `period` conforme especificado na arquitetura da aplicação.

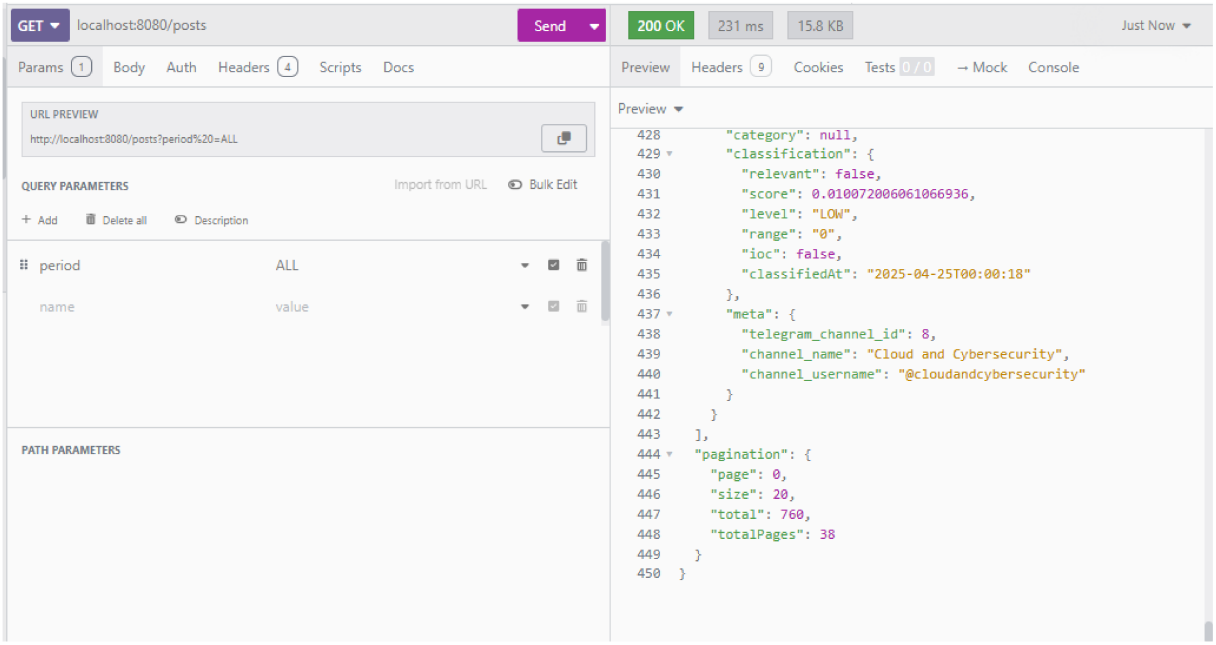
A.1 Cenário 1 — Todo o histórico, sem filtros

O analista abre o *dashboard* sem aplicar nenhum filtro. O sistema exibe o conjunto completo de dados coletados.

A.1.1 Listagem de posts (`GET /posts`)

```
GET /posts?page=0&size=20
```

Figura 32 – GET /posts sem filtros — requisição e resposta

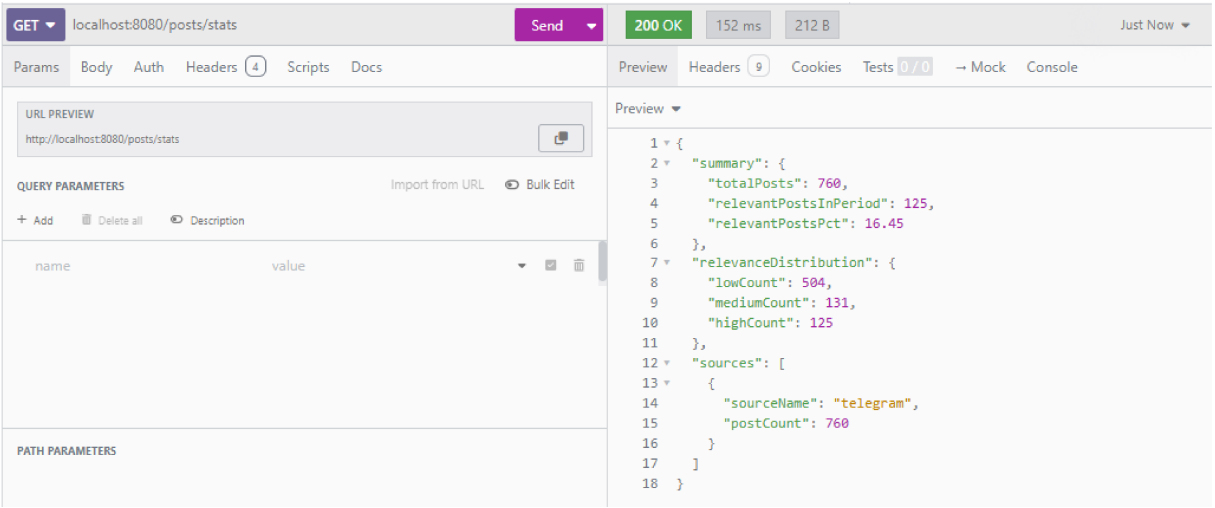


Fonte: Autoria própria

A.1.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats
```

Figura 33 – GET /posts/stats sem filtros — requisição e resposta

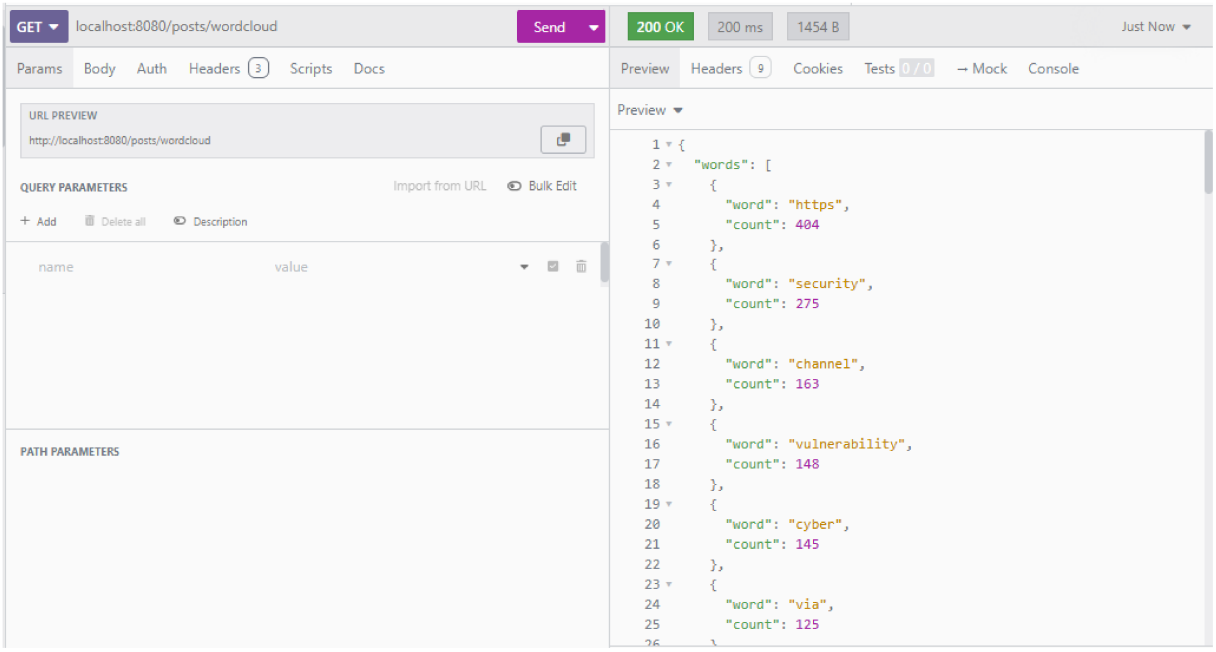


Fonte: Autoria própria

A.1.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud
```

Figura 34 – GET /posts/wordcloud sem filtros — requisição e resposta



Fonte: Autoria própria

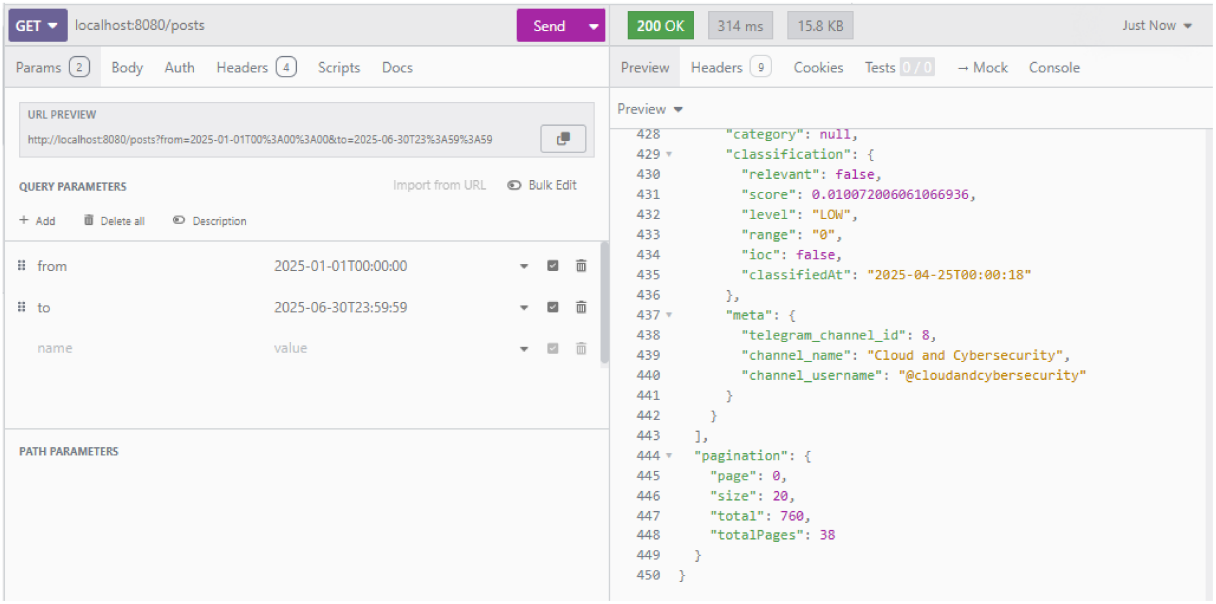
A.2 Cenário 2 — Intervalo fechado (`from`/`to`)

O analista filtra um semestre específico de coleta — janeiro a julho de 2025 — para analisar o comportamento das ameaças naquele período.

A.2.1 Listagem de posts (`GET /posts`)

```
GET /posts?from=2025-01-01&to=2025-06-31&page=0&size=20
```

Figura 35 – GET /posts com intervalo jan–jul 2025 — requisição e resposta

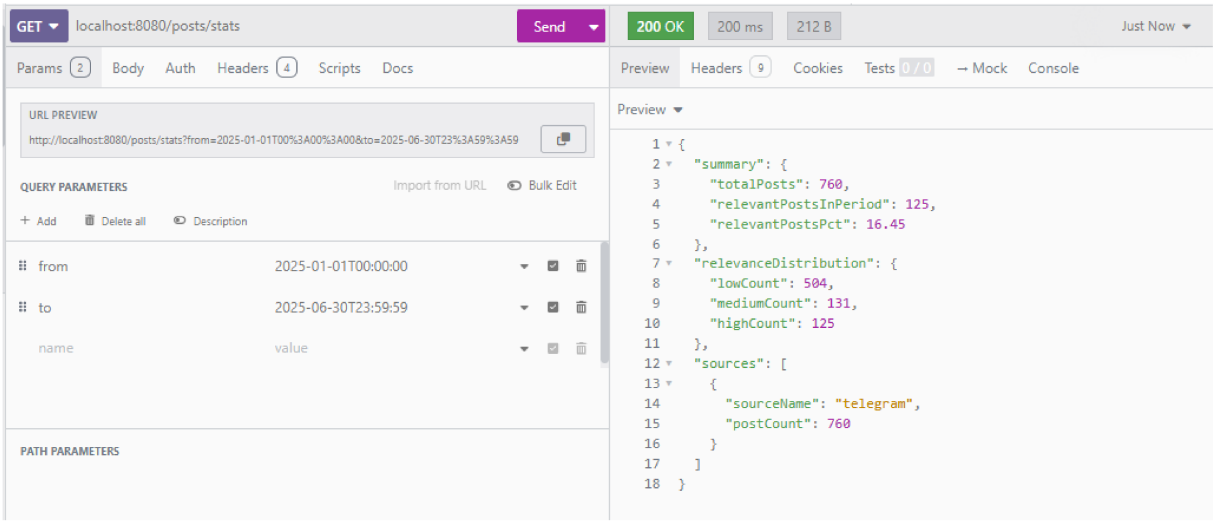


Fonte: Autoria própria

A.2.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats?from=2025-01-01&to=2025-06-31
```

Figura 36 – GET /posts/stats com intervalo jan–jul 2025 — requisição e resposta

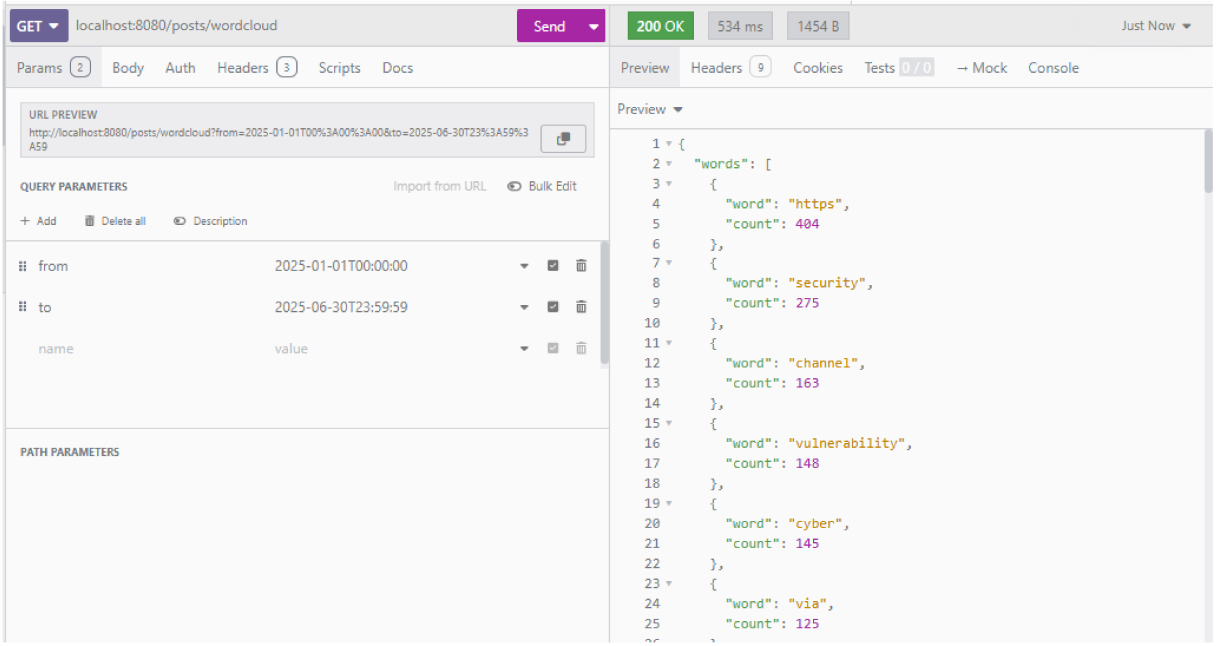


Fonte: Autoria própria

A.2.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud?from=2025-01-01&to=2025-06-31
```

Figura 37 – GET /posts/wordcloud com intervalo jan–jul 2025 — requisição e resposta



Fonte: Autoria própria

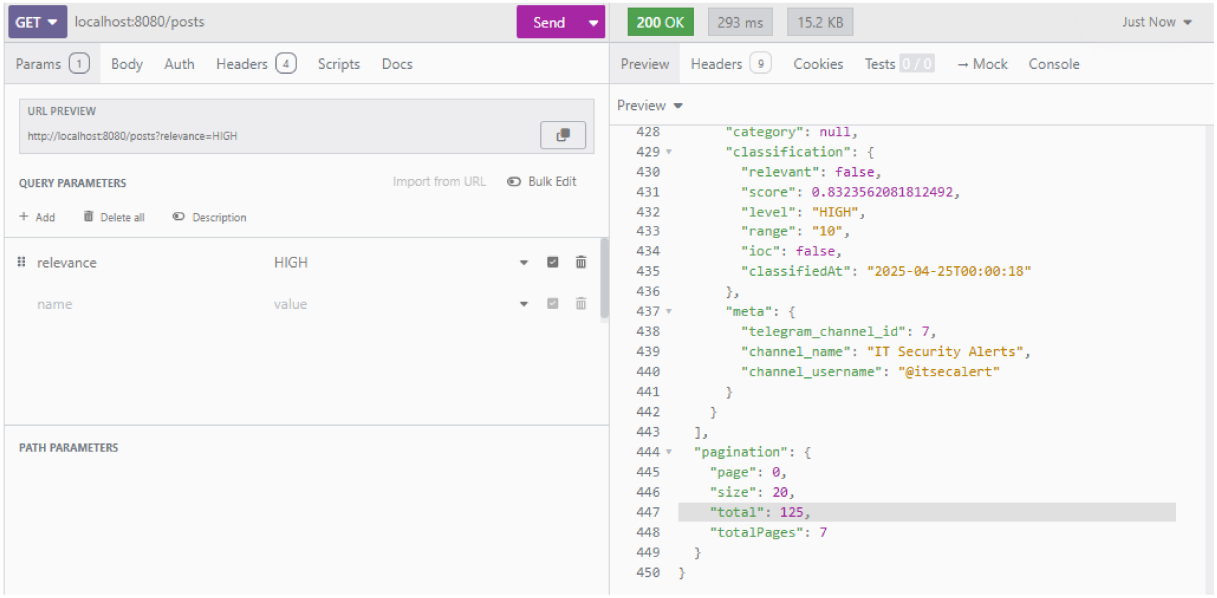
A.3 Cenário 3 — Relevância `HIGH`

O analista restringe a visualização somente aos posts de alta relevância, sem delimitar janela temporal — o painel exibe o perfil das ameaças mais críticas em todo o histórico.

A.3.1 Listagem de posts (`GET /posts`)

```
GET /posts?relevance=HIGH&page=0&size=20
```

Figura 38 – GET /posts filtrado por relevância HIGH — requisição e resposta

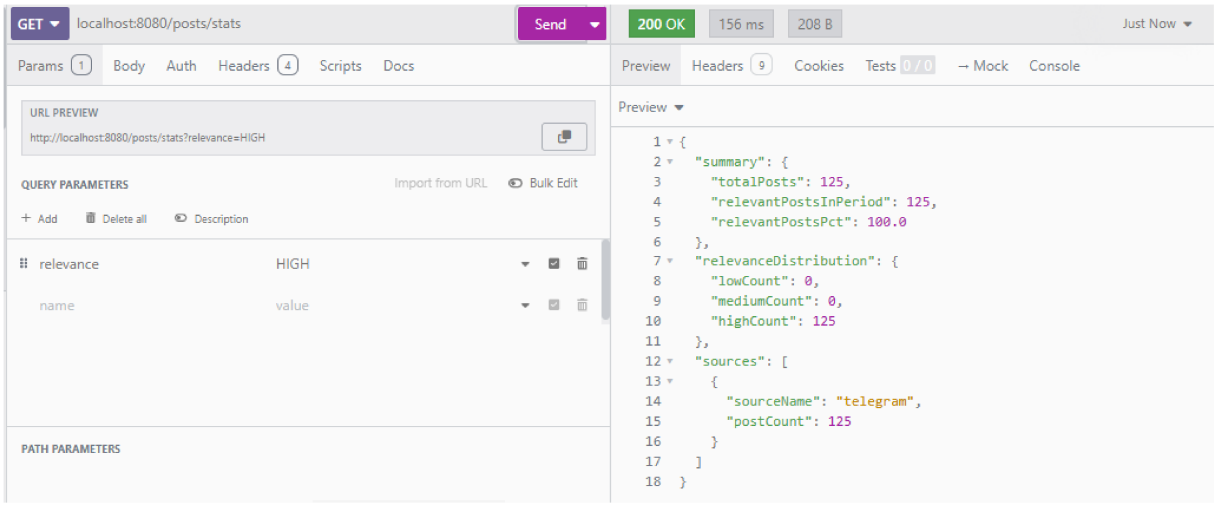


Fonte: Autoria própria

A.3.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats?relevance=HIGH
```

Figura 39 – GET /posts/stats filtrado por relevância HIGH — requisição e resposta

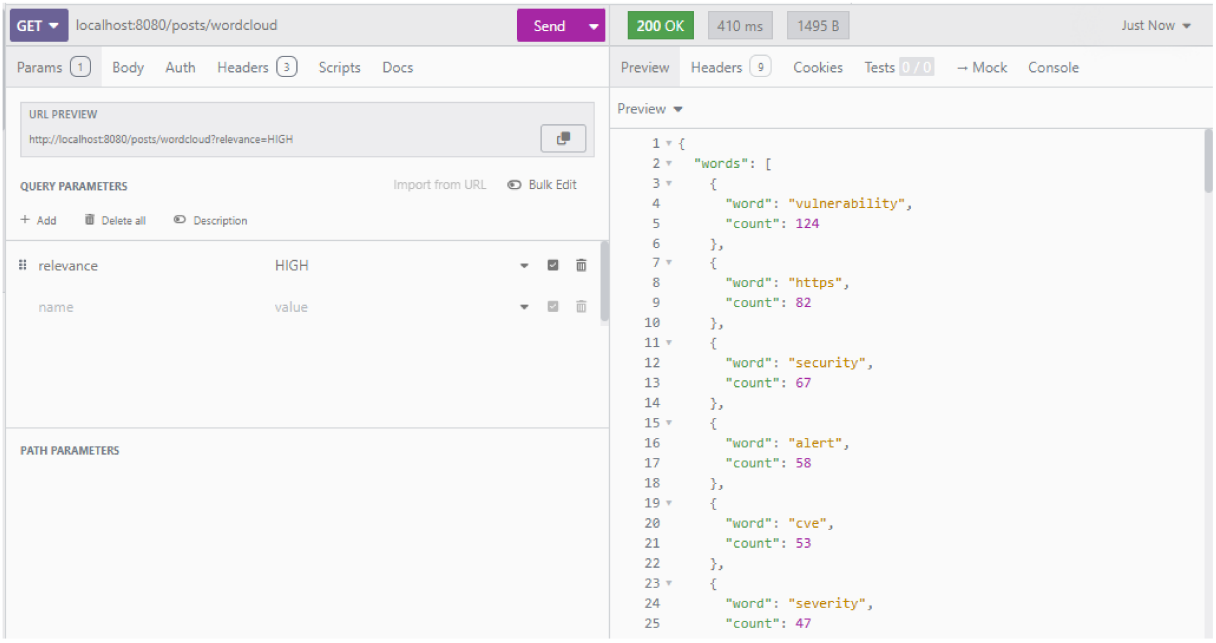


Fonte: Autoria própria

A.3.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud?relevance=HIGH
```

Figura 40 – GET /posts/wordcloud filtrado por relevância HIGH — requisição e resposta



Fonte: Autoria própria

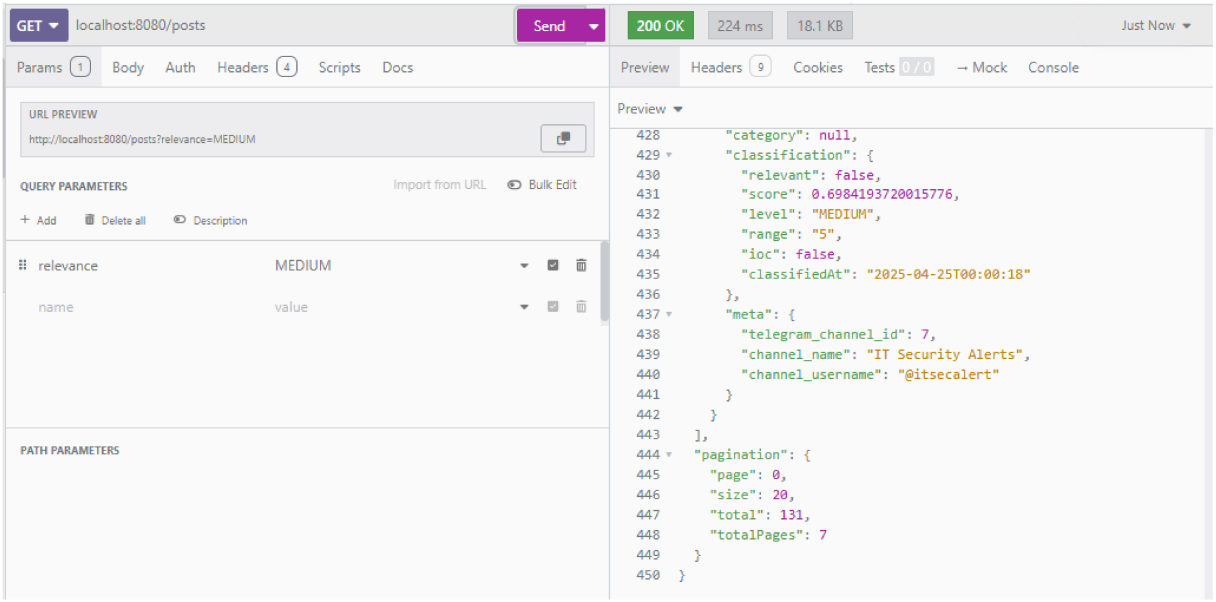
A.4 Cenário 4 — Relevância `MEDIUM`

O analista examina posts de relevância média para identificar ameaças emergentes ainda não classificadas como críticas.

A.4.1 Listagem de posts (`GET /posts`)

```
GET /posts?relevance=MEDIUM&page=0&size=20
```

Figura 41 – GET /posts filtrado por relevância MEDIUM — requisição e resposta

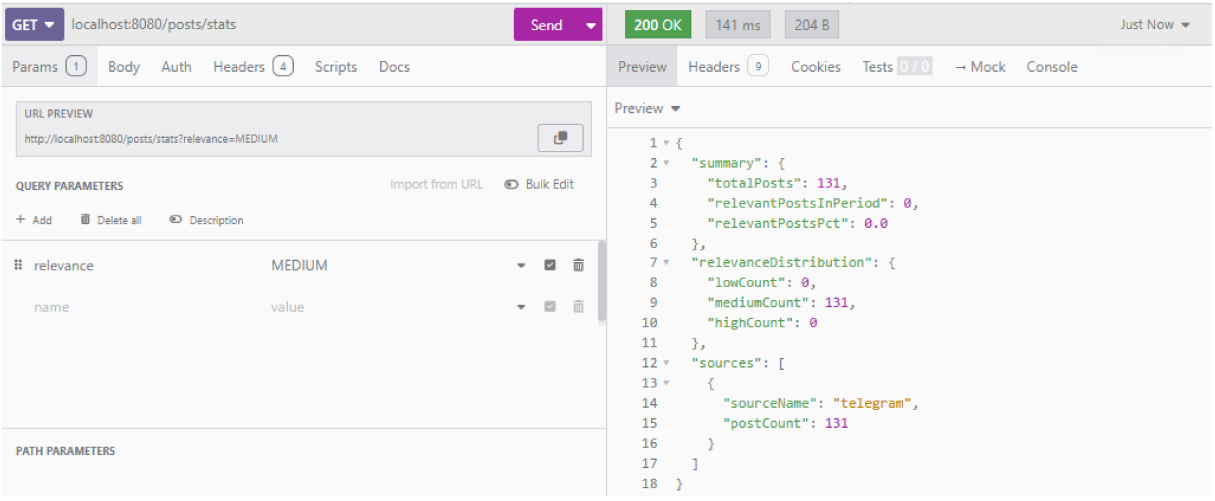


Fonte: Autoria própria

A.4.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats?relevance=MEDIUM
```

Figura 42 – GET /posts/stats filtrado por relevância MEDIUM — requisição e resposta

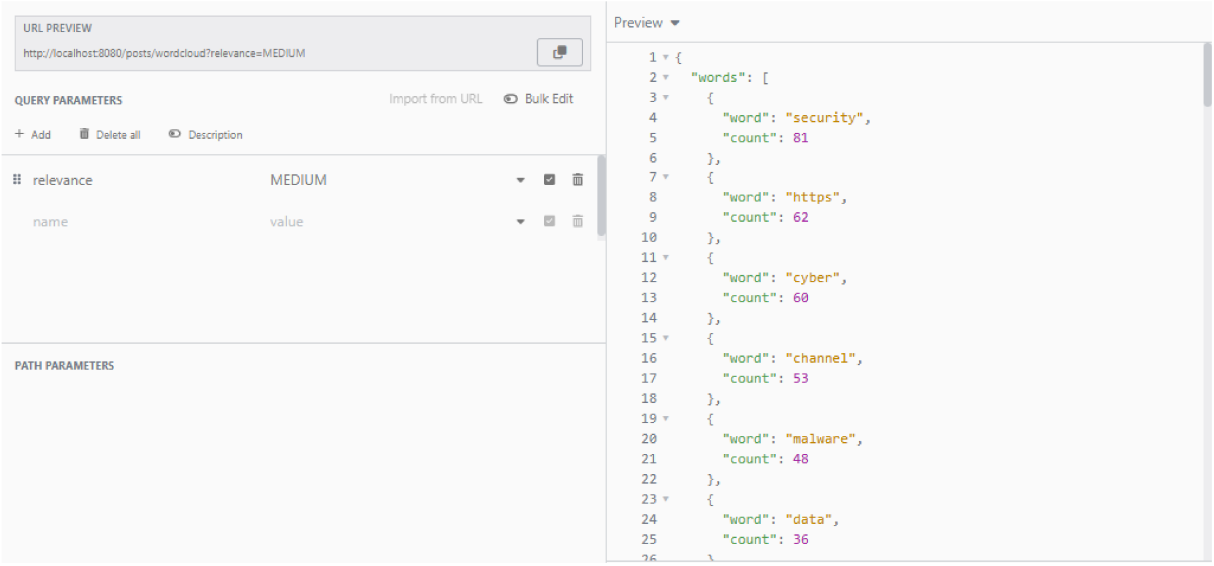


Fonte: Autoria própria

A.4.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud?relevance=MEDIUM
```

Figura 43 – GET /posts/wordcloud filtrado por relevância MEDIUM — requisição e resposta



Fonte: Autoria própria

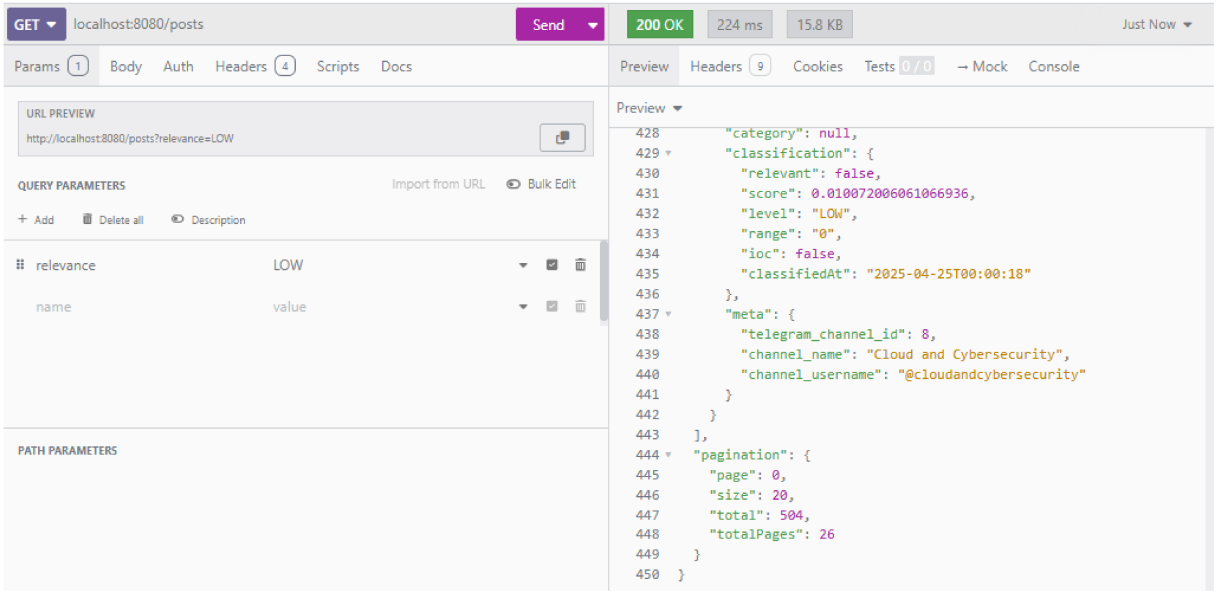
A.5 Cenário 5 — Relevância `LOW`

O analista inspeciona posts de baixa relevância para verificar o volume de ruído no canal e calibrar os limiares de classificação.

A.5.1 Listagem de posts (`GET /posts`)

```
GET /posts?relevance=LOW&page=0&size=20
```

Figura 44 – GET /posts filtrado por relevância LOW — requisição e resposta

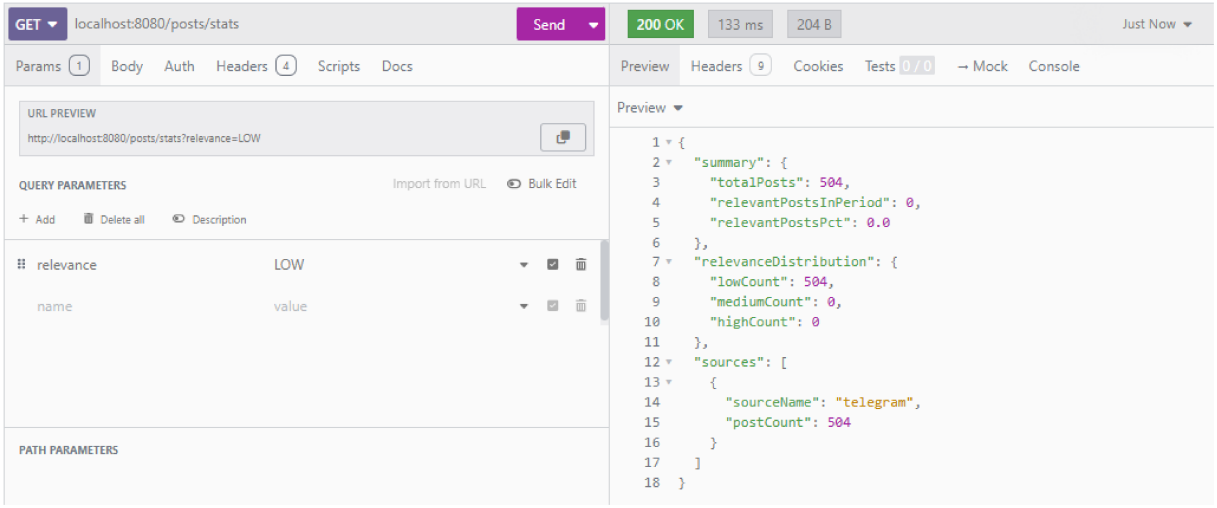


Fonte: Autoria própria

A.5.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats?relevance=LOW
```

Figura 45 – GET /posts/stats filtrado por relevância LOW — requisição e resposta

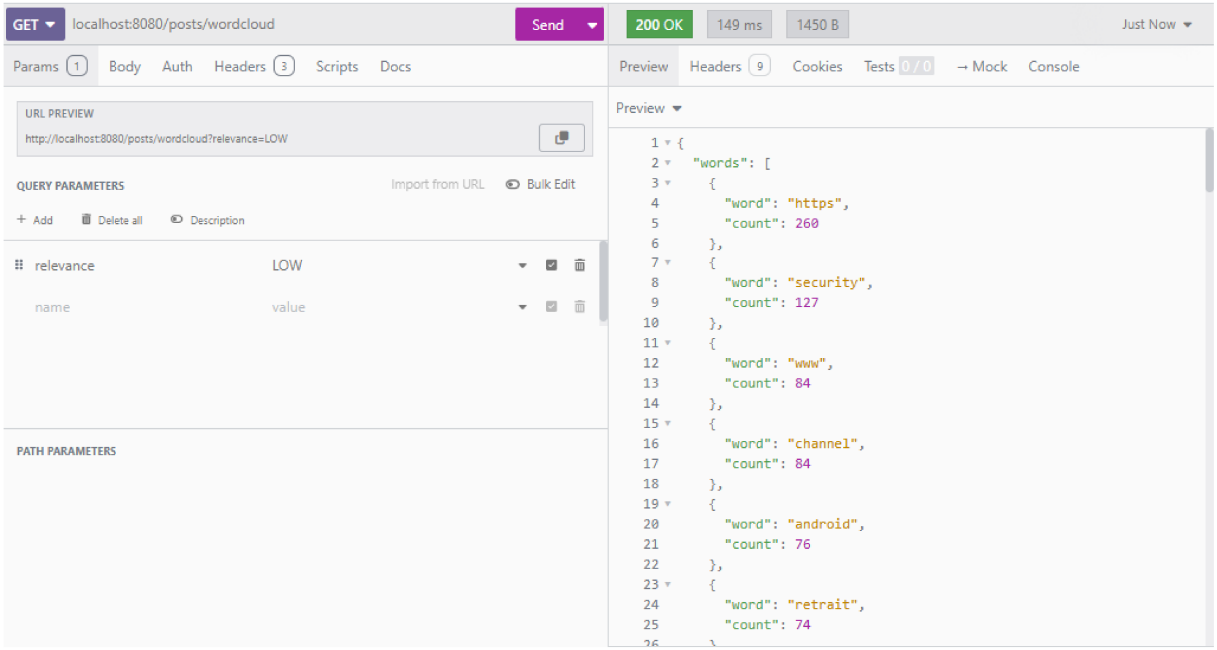


Fonte: Autoria própria

A.5.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud?relevance=LOW
```

Figura 46 – GET /posts/wordcloud filtrado por relevância LOW — requisição e resposta



Fonte: Autoria própria

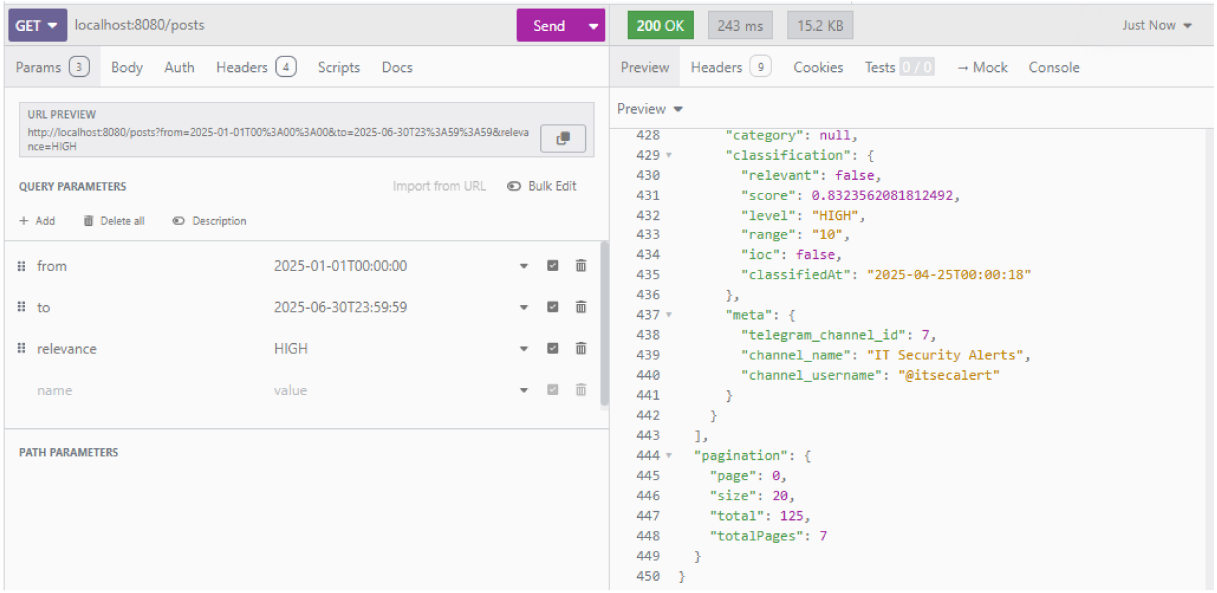
A.6 Cenário 6 — Relevância `HIGH` com intervalo `from`/`to`

O analista combina o filtro de alta relevância com uma janela temporal definida — o primeiro semestre de 2025 — para investigar picos de ameaças críticas naquele período.

A.6.1 Listagem de posts (`GET /posts`)

```
GET /posts?relevance=HIGH&from=2025-01-01&to=2025-06-30&page=0&size=20
```

Figura 47 – GET /posts com relevância HIGH e intervalo 1º sem. 2025 — requisição e resposta

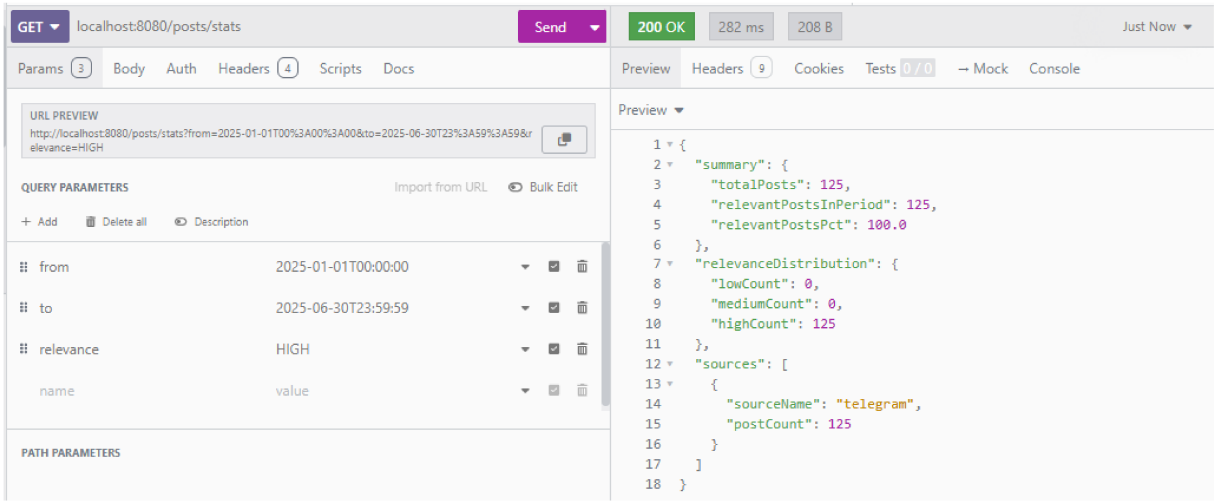


Fonte: Autoria própria

A.6.2 Estatísticas agregadas (`GET /posts/stats`)

```
GET /posts/stats?relevance=HIGH&from=2025-01-01&to=2025-06-30
```

Figura 48 – GET /posts/stats com relevância HIGH e intervalo 1º sem. 2025 — requisição e resposta

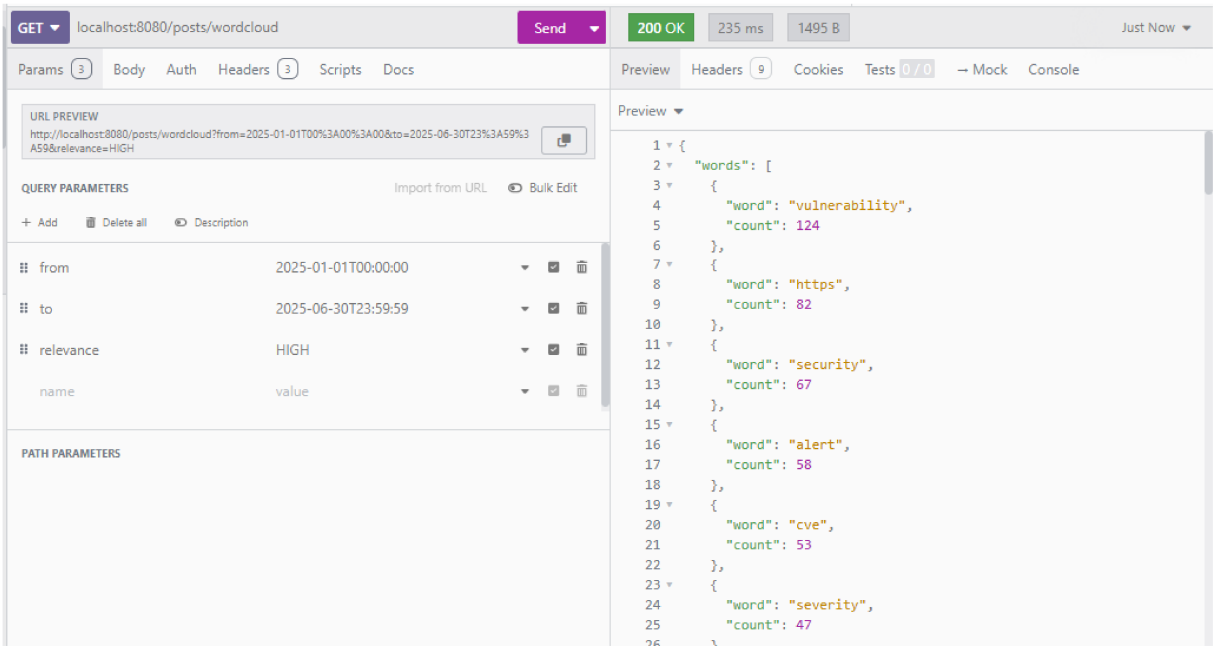


Fonte: Autoria própria

A.6.3 Nuvem de palavras (`GET /posts/wordcloud`)

```
GET /posts/wordcloud?relevance=HIGH&from=2025-01-01&to=2025-06-30
```

Figura 49 – GET /posts/wordcloud com relevância HIGH e intervalo 1º sem. 2025 — requisição e resposta



Fonte: Autoria própria

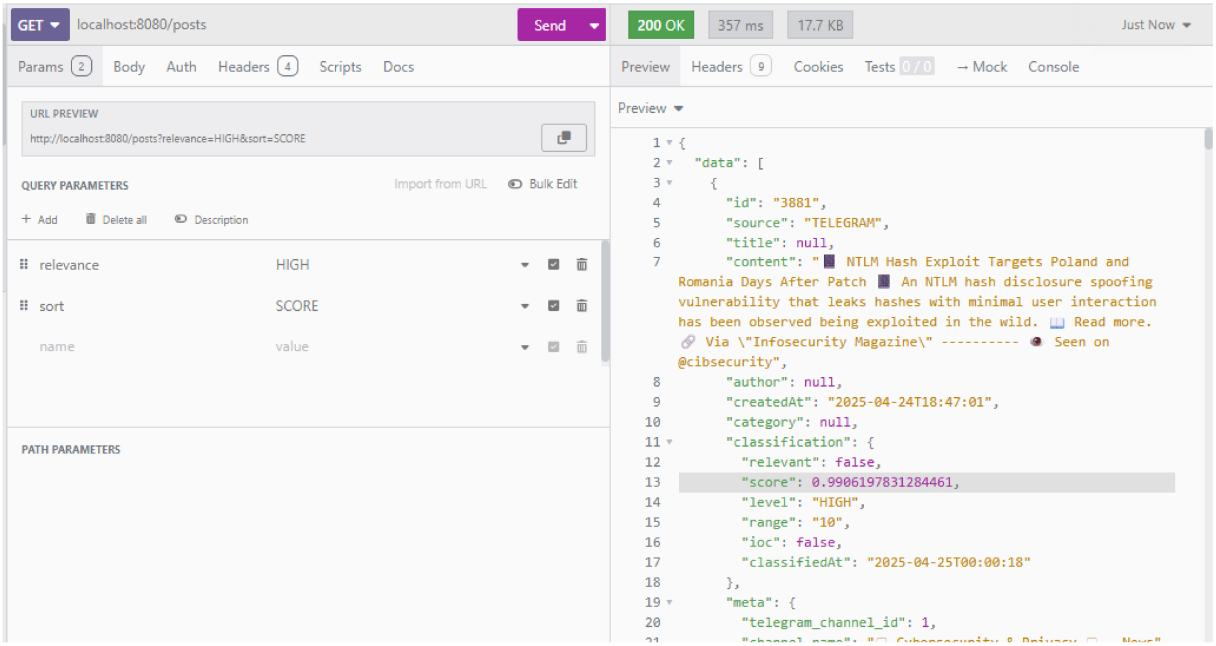
A.7 Cenário 7 — Ordenação por *score* decrescente

O analista reordena a listagem pelo *score* de classificação — do mais crítico para o menos crítico — mantendo o filtro de alta relevância para focar nas ameaças prioritárias. Os endpoints de estatísticas e nuvem de palavras não são afetados pela ordenação, portanto são exibidos com os mesmos filtros para manter a coerência do cenário no *dashboard*.

A.7.1 Listagem de posts (`GET /posts`)

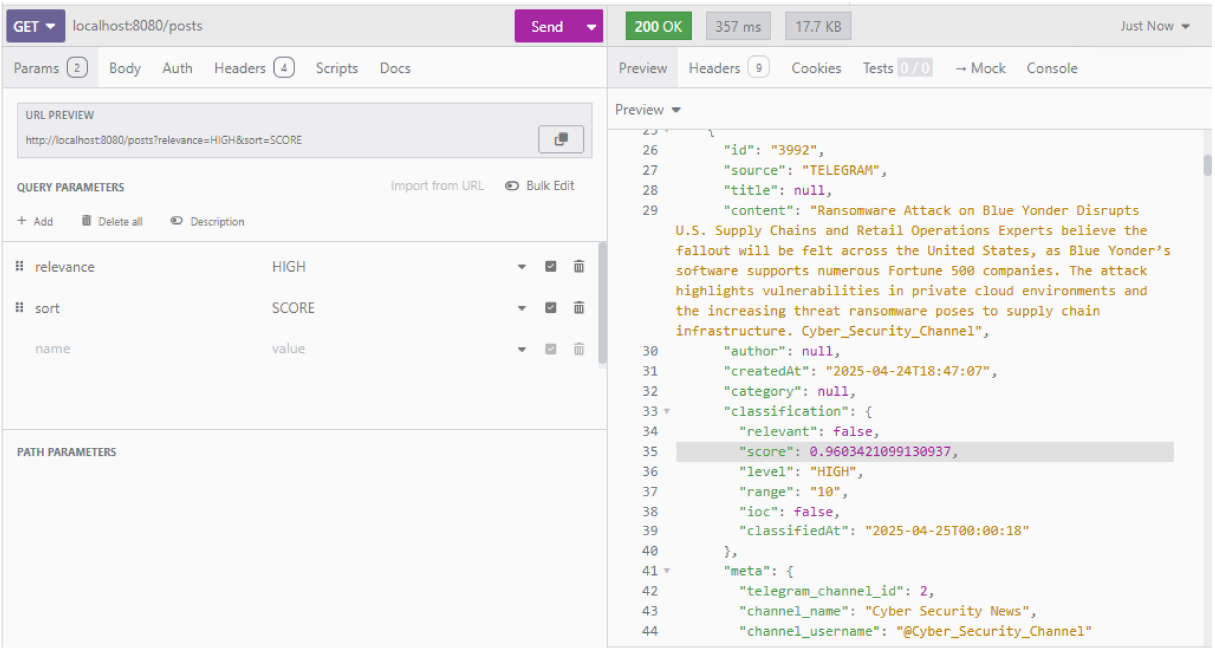
```
GET /posts?relevance=HIGH&sort=SCORE&order=DESC&page=0&size=20
```

Figura 50 – GET /posts ordenado por score decrescente com filtro HIGH — requisição e resposta



Fonte: Autoria própria

Figura 51 – GET /posts ordenado por score decrescente com filtro HIGH — requisição e resposta



Fonte: Autoria própria

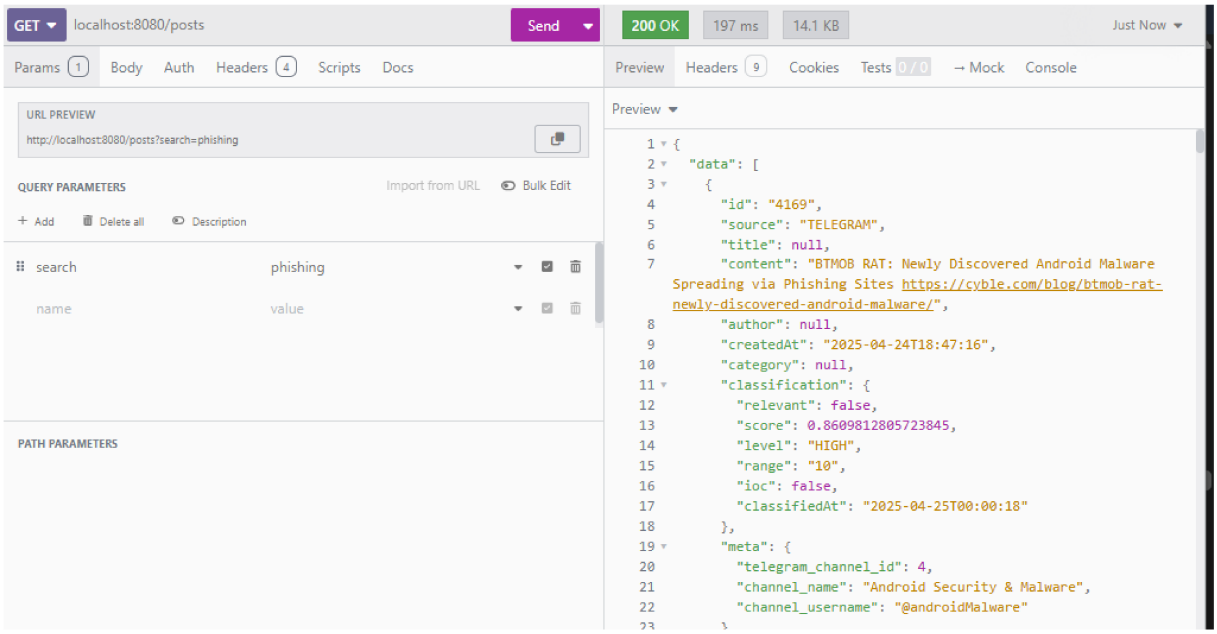
A.8 Cenário 8 — Busca por texto livre (`search`)

O analista utiliza o parâmetro `search` para restringir a listagem de *posts* pelo conteúdo da mensagem. As três subseções demonstram, respectivamente, a busca isolada, a combinação com filtro de relevância e a combinação com ordenação – todas exclusivas do endpoint `GET /posts`.

A.8.1 Busca isolada

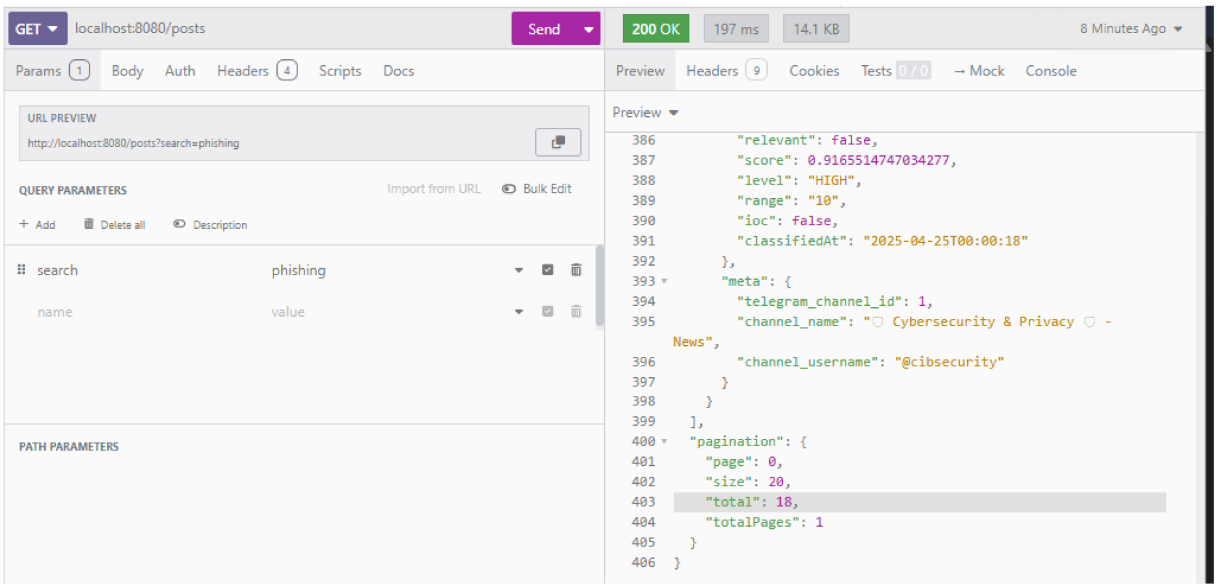
```
GET /posts?search=phishing&page=0&size=20
```

Figura 52 – GET /posts com busca textual `search=phishing` — requisição e resposta



Fonte: Autoria própria

Figura 53 – GET /posts com busca textual `search=phishing` — corpo da resposta

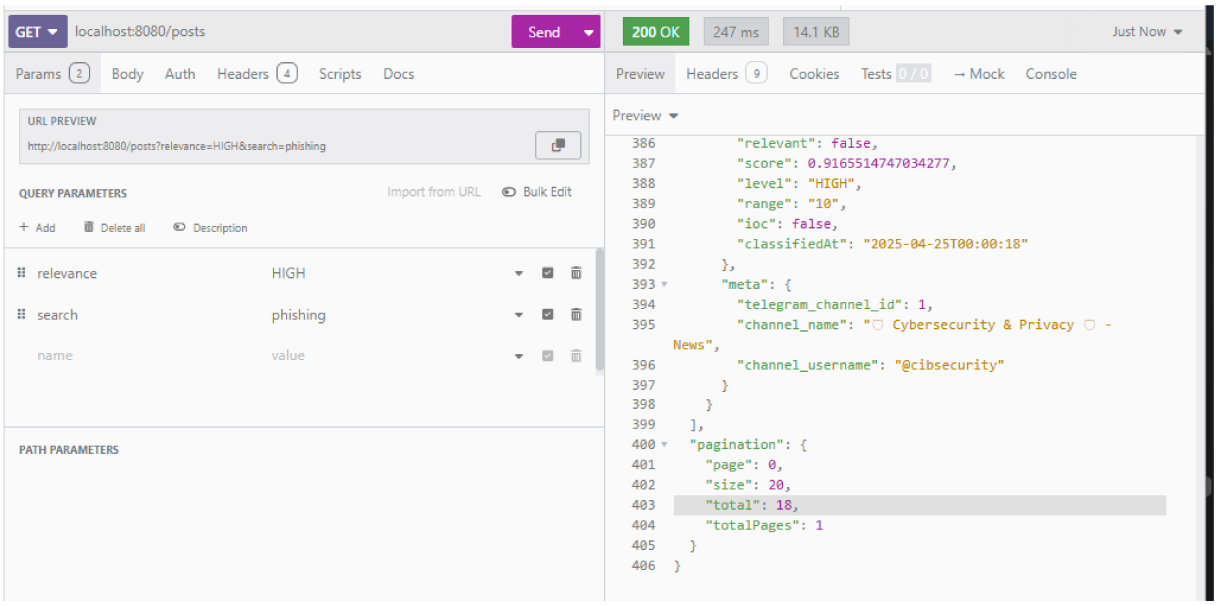


Fonte: Autoria própria

A.8.2 Busca combinada com filtro de relevância

```
GET /posts?search=phishing&relevance=HIGH&page=0&size=20
```

Figura 54 – GET /posts com `search=phishing` e `relevance=HIGH` — requisição e resposta

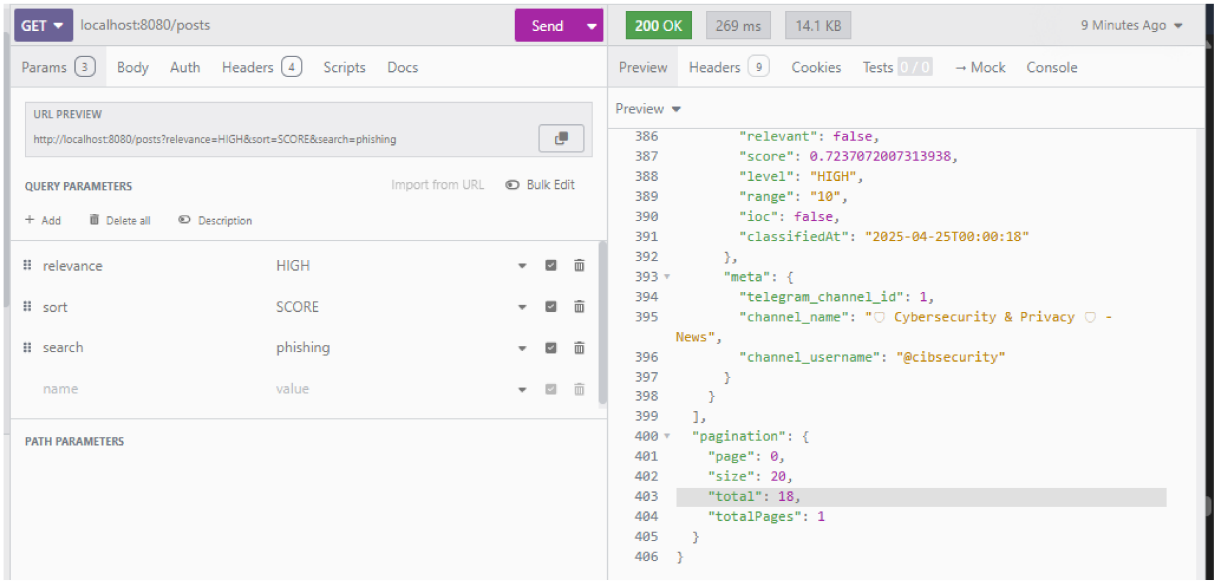


Fonte: Autoria própria

A.8.3 Busca combinada com ordenação

```
GET /posts?search=phishing&sort=SCORE&order=DESC&page=0&size=20
```

Figura 55 – GET /posts com `search=phishing` ordenado por *score* decrescente — requisição e resposta



Fonte: Autoria própria

Figura 56 – GET /posts com `search=phishing` ordenado por *score* decrescente — corpo da resposta


```
{
  "data": [
    {
      "id": "3822",
      "source": "TELEGRAM",
      "title": null,
      "content": "👮 FBI: Cybercrime Losses Rocket to $16.6B in 2024 👮 The losses are 33 higher than the year before, with phishing leading the way as the mostreported cybercrime last year, and ransomware was the top threat to critical infrastructure, according to the FBI Internet Crime Report. 📖 Read more. 🔗 Via \"Dark Reading\" ---------- 🧿 Seen on @cibsecurity",
      "author": null,
      "createdAt": "2025-04-24T18:46:59",
      "category": null,
      "classification": {
        "relevant": false,
        "score": 0.9165514747034277,
        "level": "HIGH",
        "range": "10",
        "ioc": false,
        "classifiedAt": "2025-04-25T00:00:18"
      },
      "meta": {
```


Fonte: Autoria própria

Figura 57 – GET /posts com `search=phishing` ordenado por *score* decrescente — paginação


```
      "id": "3871",
      "source": "TELEGRAM",
      "title": null,
      "content": "🖊️ APT29 Deploys GRAPELOADER Malware Targeting European Diplomats Through Wine-Tasting Lures 🖊️ The Russian statesponsored threat actor known as APT29 has been linked to an advanced phishing campaign that's targeting diplomatic entities across Europe with a new variant of WINELOADER and a previously unreported malware loader codenamed GRAPELOADER. \"While the improved WINELOADER variant is still a modular backdoor used in later stages, GRAPELOADER is a newly observed initialstage tool. 📖 Read more. 🔗 Via \"The Hacker News\" ---------- 👁️ Seen on @cibsecurity",
      "author": null,
      "createdAt": "2025-04-24T18:47:01",
      "category": null,
      "classification": {
        "relevant": false,
        "score": 0.9013968876880168,
        "level": "HIGH",
        "range": "10",
        "ioc": false,
        "classifiedAt": "2025-04-25T00:00:18"
      },
```


Fonte: Autoria própria