

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Segurança no Desenvolvimento de Microsserviços: Práticas e Padrões Essenciais com OAuth2

Theo Marques da Rocha

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Daniel de Paula Porto

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Segurança no Desenvolvimento de Microsserviços: Práticas e Padrões Essenciais com OAuth2

Theo Marques da Rocha

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Daniel de Paula Porto (Orientador)
CIC/UnB

Prof.a Dr.a Genaína Nunes Rodrigues
CIC/UnB

Prof.a Dr.a Aletéia Patricia Favacho de Araújo
CIC/UnB

Prof. Dr. Guilherme Novaes Ramos
Coordenador do Curso de Engenharia da Computação

Brasília, 19 de fevereiro de 2025

Dedicatória

Toda honra e toda a glória a Deus, por toda a eternidade, amém!

Agradecimentos

Agradeço primeiramente a Deus por abençoar minha vida e proporcionar tantas experiências gratificantes e construtivas nestes anos universitários. Gostaria de agradecer meus pais por me fornecerem educação de qualidade e com valores cristãos desde minha infância, o que me ajudou a ingressar na Universidade de Brasília. Obrigado por sempre me orientarem e guiarem com todo amor e carinho, eu sinto o cuidado de Deus comigo por meio da vida de vocês. Obrigado pelos puxões de orelha que me tornaram o homem que sou. Agradecimentos especiais para minha noiva Giulia Riccardi que sempre acreditou no meu potencial e me incentiva a crescer e prosperar cada vez mais. Gostaria de agradecer também ao meu professor orientador Daniel Porto por toda atenção e paciência com meu trabalho, pelo esforço dedicado e pelo ensino de excelência. Por fim, não poderia me esquecer de deixar meus agradecimentos aos colegas da Coordenadoria de Desenvolvimento de Sistemas (CDS) do Tribunal Superior do Trabalho (TST), que contribuiram tanto para minha evolução no âmbito profissional e relacional.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A tecnologia moderna envolve um alto tráfego de dados e requisições na Internet, o que requer uma maior segurança no tratamento das informações e na segurança ao acesso de recursos. O presente trabalho consiste na análise dos principais padrões de segurança recomendados pelo protocolo OAuth 2.0, aplicados na arquitetura pautada em microsserviços. O uso e relevância da criptografia também foram abordados, almejando um escopo de segurança computacional mais abrangente. A solução desenvolvida propõe um modelo de controle e de segurança para sistemas que utilizam a arquitetura de microsserviços, adotando o fluxo *Resource Owner Password Credentials Grant*. Com o intuito de demonstrar e aplicar tais conceitos, foi desenvolvido um servidor *backend* que implementa questões medulares defendidas pelo *framework* OAuth 2.0, aliado a técnicas de criptografia e à emissão de *JSON Web Tokens* para controle de acesso. Para isso, foi utilizada a linguagem de programação Java juntamente à ferramenta Spring Boot para implementação, pois essas tecnologias apresentam robustez, segurança e suporte do *Spring Security*, que facilita a autenticação e o uso de JWT em microsserviços. Como contribuição, destaca-se a implantação de um *frontend* para aprimorar a experiência do usuário no uso do serviço, além do desenvolvimento de módulos de microsserviços para agregar à solução proposta.

Palavras-chave: Microsserviços, OAuth2, Segurança, Autenticação, Autorização, Criptografia, *Spring Boot*

Abstract

Modern technology involves high data traffic and internet requests, requiring greater security in information handling and resource access control. This work analyzes the main security standards recommended by the OAuth 2.0 protocol, applied to a microservices-based architecture. The use and relevance of cryptography were also addressed, aiming for a broader scope of computational security. The developed solution proposes a security and control model for systems using a microservices architecture, adopting the Resource Owner Password Credentials Grant flow. To demonstrate and apply these concepts, a backend server was developed, implementing core aspects of the OAuth 2.0 framework, combined with encryption techniques and the issuance of JWTs for access control. Java and the Spring Boot framework were used for implementation, as these technologies offer robustness, security, and support from Spring Security, which simplifies authentication and JWT usage in microservices. As an expected contribution, the implementation of a frontend is highlighted to enhance the user experience in using the service, along with the development of microservices modules to complement the proposed solution.

Keywords: Microservices, OAuth2, Security, Authentication, Authorization, Encryption, Spring Boot

Sumário

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 2

1.3 Organização deste trabalho 3

2 Microsserviços **4**

2.1 Monolítico x Microsserviços 4

2.2 Arquitetura 5

2.2.1 Comunicação entre Microsserviços 6

2.2.2 *Containers* 7

2.3 Desafios de Segurança 8

2.4 Boas Práticas de Segurança 8

3 Padrão OAuth2 **10**

3.1 Visão Inicial 10

3.1.1 Papéis Essenciais 11

3.1.2 Principais Fluxos de Implementação 11

3.2 Autenticação e Autorização 13

3.3 *JSON Web Token* (JWT) 14

3.4 Princípios Essencias na Segurança do OAuth 2.0 14

4 Criptografia e Segurança **16**

4.1 Conceitos Iniciais 16

4.2 Criptografia Simétrica e Assimétrica 17

5 Implementação **20**

5.1 Considerações Iniciais 20

5.2 Solução Desenvolvida 20

5.3 Funcionamento 24

5.3.1 Fluxo do Estudante 24

5.3.2 Fluxo do Professor 25
5.3.3 Fluxo do *Super Admin* 25
5.4 Considerações finais 26

6 Conclusão **27**

Referências **29**

Anexo **30**

I Imagens obtidas na utilização do sistema **31**

Lista de Figuras

2.1 Monolitíco x Microsserviços. Imagem adaptada de `https://medium.com/`. 5
2.2 Modelo *Request-Reply*. Imagem retirada de `https://medium.com/`. 7
2.3 Modelo *Publish-subscribe*. Imagem adaptada de `https://inside.contabilizei.com.br/`. 7

3.1 Fluxo padrão do OAuth2.0. Imagem adaptada de `https://medium.com/`. . 12
3.2 Fluxo do *Resource Owner Password Credentials Grant*. Imagem retirada de `https://is.docs.wso2.com/en/latest/`. 13
3.3 Estrutura interna de um *token* JWT. 15

4.1 Funcionamento da criptografia simétrica. Imagem retirada de `https://www.universidadejava.com.br/`. 18
4.2 Funcionamento da criptografia assimétrica. 19

I.1 Requisição de registro do aluno. 31
I.2 Aluno registrado persistido no banco de dados. 32
I.3 Solicitação e resposta do *login* do aluno. 32
I.4 Conteúdo do *token* do aluno logado. 33
I.5 Requisição para obter a própria nota. 34
I.6 Proibição de excluir uma nota. 34
I.7 Lançamento da nota pelo professor. 35
I.8 Persistência da nota no banco de dados. 35
I.9 Solicitação para ver todas as notas. 36
I.10 Exclusão da nota pelo professor. 36
I.11 Listagem de todos os usuários pelo professor. 37
I.12 Professor com acesso negado para visualizar as *roles*. 37
I.13 *Super admin* listando todos os usuários. 38
I.14 *Super admin* com privilégio exclusivo de ver todas as roles. 39
I.15 *Super admin* sem permissão para deletar uma nota. 39

Lista de Tabelas

2.1 Modelo de ameaças STRIDE (traduzidas) 9

Lista de Abreviaturas e Siglas

API Application Programming Interface.

HTTP Hypertext Transfer Protocol.

HTTPS Secured Hypertext Transfer Protocol.

JSON Javascript Object Notation.

JWT JSON Web Token.

REST Representational State Transfer.

SOAP Simple Object Access Protocol.

URI Uniform Resource Identifier.

Capítulo 1

Introdução

Com o avanço da globalização e a modernização tecnológica, a dinâmica mundial está sendo alterada. Assim, são perceptíveis as mudanças que o mundo atual está passando: documentos e procedimentos judiciais e administrativos que a priori eram burocráticos e demorados, atualmente são acessíveis online e com poucos cliques é possível resolver algo que demoraria dias, até mesmo semanas para ser solucionado [1]. Com essa inclusão digital e com o alto volume de dados transitando pela internet, existe o outro lado da moeda: a dificuldade de garantir a segurança e proteção de dados e informações que estão trafegando na rede. Vários ataques cibernéticos acontecem diariamente e visam atingir instituições e tirar vantagens ilícitas a partir de documentos confidenciais [2]. Em decorrência desse quadro, surge a necessidade da harmonia entre esses dois campos, o primeiro é a cultura do compartilhamento, globalização, socialização e transparência, e o segundo é a garantia da confidencialidade, segurança e privacidade das informações [3].

O modelo de desenvolvimento de software tradicional segue a arquitetura monolítica, que consiste em um sistema unificado para a aplicação, fortemente acoplado e com uma implementação muitas vezes robusta e complexa, podendo dificultar a manutenção do código e a execução de metodologias ágeis [4]. Estes fatores limitam a escalabilidade do sistema, e podem também prejudicar o desempenho, uma vez que, caso um serviço do sistema apresente algum erro, o acoplamento o reflete para toda a aplicação. Uma alternativa é a arquitetura baseada em microsserviços, constituída por múltiplas entidades com um alto grau de desacoplamento, com alguns elementos característicos essenciais: responsabilidade única, em que cada módulo é responsável por uma funcionalidade específica, alta escalabilidade, alto escalonamento e desenvolvimento independente de outros microsserviços [5]. Esses aspectos possibilitam um desenvolvimento e uma manutenção do software mais eficientes, pois há uma separação lógica e física dos serviços, permitindo desenvolvimento paralelo e um eficiente incremento de funcionalidades [5].

Com esse modelo de sistema, uma maneira de comunicação entre os microsserviços

é por meio de *Application Programming Interfaces* (API). Os dois principais protocolos de comunicação para esse formato são *Simple Object Access Protocol* (SOAP) e *Representational State Transfer* (REST). Devido à simplicidade do padrão REST, ele será o utilizado para o estudo e a implementação deste projeto. É de suma importância entender e mapear corretamente as dificuldades no desenvolvimento de fatores de segurança nessa arquitetura, pois, uma vez que um microsserviço seja invadido, o sistema tem a chance de ser exposto. Por essa razão, o estudo de segurança no desenvolvimento de APIs está em uma curva ascendente [3], buscando sempre mitigar falhas e vulnerabilidades do sistema nos locais de maior exposição do serviço. Ademais, é importante manter fluxos de lógica de autenticação, integridade, não-repudiação, confidencialidade, disponibilidade e autorização bem desenhados, porquanto a qualidade de um sistema é diretamente proporcional à qualidade de medidas de segurança desenvolvidas [5].

1.1 Motivação

Com o crescente uso de arquitetura baseada em microsserviços nas empresas mundiais e adoção desse aspecto nos padrões de softwares modernos, também existe o aumento significativo de uso do protocolo OAuth 2.0[1], padrão que permite uma normalização e padronização das soluções de segurança cibernética [6]. O presente trabalho consiste no estudo e na implementação das melhores práticas relacionadas à segurança e confiabilidade de dados neste padrão arquitetural, garantindo sucesso na comunicação entre os serviços, com autenticação e autorização seguras e eficientes pautadas no OAuth 2.0.

1.2 Objetivos

Este trabalho tem como objetivo aplicar o padrão OAuth 2.0 para implementar soluções de segurança de software na arquitetura pautada em microsserviços. Para atingir tal objetivo, faz-se necessário cumprir os seguintes objetivos específicos:

Objetivo 1: Avaliar as dificuldades de segurança no desenvolvimento de microsserviços;

Objetivo 2: Identificar e analisar boas práticas de segurança com OAuth 2.0;

Objetivo 3: Adotar uma solução para implementação robusta, segura e eficaz.

[1] `https://oauth.net/2/`

1.3 Organização deste trabalho

O trabalho está dividido em seis capítulos. O Capítulo 2 mostra os conceitos presentes na arquitetura baseada em microsserviços e os desafios de segurança envolvidos. Em seguida, o Capítulo 3 traz os fundamentos do padrão OAuth2, juntamente com seus principais fluxos e métodos envolvidos no desenvolvimento de soluções de autenticação e de autorização. O Capítulo 4 mostra alguns princípios de criptografia e segurança, juntamente com suas aplicabilidades e características essenciais. O Capítulo 5 apresenta a solução desenvolvida, aplicando os conceitos retratados nos capítulos anteriores e exemplificando as informações discutidas por meio de um sistema. Por fim, o Capítulo 6 traz o encerramento do trabalho, mostrando alguns pontos essenciais discutidos e suas respectivas conclusões.

Capítulo 2

Microsserviços

Neste capítulo será apresentada a arquitetura de microsserviços. No início, a Seção 2.1 mostra um parecer inicial e introduz os conceitos básicos sobre o padrão destes microsserviços. Em seguida, a Seção 2.2 apresenta a arquitetura destes serviços independentes. Dentro da Seção 2.2, alguns aspectos do âmbito comunicativo estão apresentados na Seção 2.2.1, e a Seção 2.2.2 discute a importância da conteinerização. Por fim, a Seção 2.3 mostra algumas ameaças à arquitetura de microsserviços, e a Seção 2.4 lista algumas boas práticas protetivas recomendadas.

2.1 Monolítico x Microsserviços

A arquitetura monolítica tradicionalmente utilizada traz um aspecto unificado e de alto grau de acoplamento [4]. Essa arquitetura pode apresentar uma alta complexidade de manutenção e refatoração, devido ao grande número de linhas de código que o sistema pode atingir devido à centralização em apenas um projeto. Por essas razões, o uso de aplicações monolíticas em um alto grau de escalabilidade pode ser dificultado [5].

Uma alternativa é o desenvolvimento pautado em microsserviços, que traz vários pequenos serviços independentes entre si com um alto grau de desacoplamento [7], permitindo um desenvolvimento autônomo das funcionalidades, sem depender da tecnologia ou linguagem de programação adotada pelos demais serviços. Este padrão arquitetural traz uma autonomia para os sistemas, pois permite uma maior adoção de metodologias ágeis para o desenvolvimento, pois cada serviço é independente. A comunicação entre microsserviços é estabelecida conforme necessário, de forma síncrona ou assíncrona, permitindo assim a troca de dados entre as entidades [5]. A diferença desses padrões de arquitetura pode ser observada na Figura 2.1, enquanto o padrão monolítico concentra tudo em um único componente, a arquitetura de microsserviços traz uma distribuição em diversos módulos.

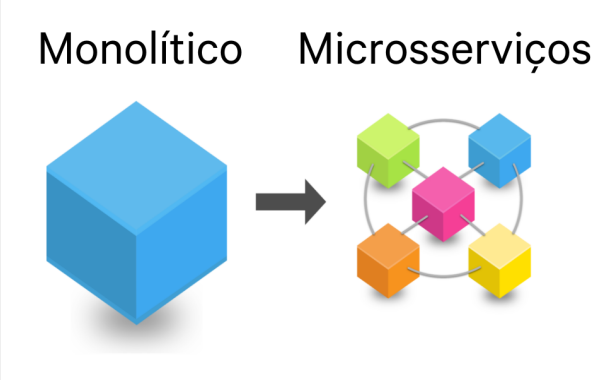


Figura 2.1: Monolitíco x Microsserviços. Imagem adaptada de `https://medium.com/`.

Esses aspectos tornaram a arquitetura pautada em microsserviços uma maneira eficiente para o desenvolvimento de sistemas altamente escaláveis, tendo em vista o uso adequado de recursos alocados, acompanhado da qualidade dos serviços e do bom uso da capacidade computacional [8].

Almejando um sistema robusto e de alto nível, conforme as demandas do mercado atual, não é necessário apenas um alto padrão de servidores alocados, mas também é de suma importância uma arquitetura condizente com a qualidade do software esperado [8]. A complexidade dos sistemas modernos tende a crescer cada vez mais, e com isso, novas funcionalidades são criadas e desenvolvidas. Visando uma aptidão adequada para atender os padrões requisitados, a arquitetura pautada em microsserviços é uma ótima solução, uma vez que restringe e limita o campo de atuação de um setor na área do desenvolvimento, possibilitando um maior desacoplamento das funcionalidades e desenvolvimento independente dos módulos do sistema [8].

2.2 Arquitetura

A arquitetura de *software* estabelece a estrutura, organização e interações de um sistema, e no contexto de microsserviços, existem diretrizes para defini-la corretamente [5]:

- Devem ser gerenciáveis, escalonados, duplicados, alterados e desenvolvidos independentemente de outros microsserviços;
- Devem seguir o princípio de responsabilidade única, em que cada módulo desenvolvido deve ter uma única funcionalidade;
- Devem ser tolerantes a falhas e *bugs*;
- Serviços sensíveis devem ser utilizados para gerenciamento de dados.

Esses aspectos caracterizam a arquitetura de microsserviços como uma solução elegante e eficiente para o desenvolvimento de sistemas altamente escaláveis, carregando um aspecto de tolerância a erros, autonomia entre componentes e um alinhamento adequado que permite a solução técnica para a regra de negócios estabelecida [2].

2.2.1 Comunicação entre Microsserviços

Uma vez que existem serviços desacoplados e independentes, cada um executando seu próprio processo, deve haver alguma maneira para estabelecer comunicação entre eles. Esses microsserviços se comunicam por meio de protocolos leves, como *Hypertext Transfer Protocol* (HTTP), pautados em cima das regras de negócios estabelecidas [8]. É com esta ideia que existem duas principais maneiras de comunicação entre estes módulos: *Request-Reply* e *Publish-Subscribe* [5].

Request-Reply

Esta maneira de comunicação é estabelecida de forma síncrona, na qual um microsserviço expõe informações internas do sistema de maneira segura e organizada, permitindo a visualização de dados por outros serviços através do desenvolvimento e consumo de APIs. Este modelo retrata um modelo de requisição e resposta clássico dos protocolos HTTP em serviços Web, no qual uma solicitação é realizada e uma resposta é esperada [5]. A Figura 2.2 demonstra o fluxo de uma comunicação entre um cliente e um servidor, onde existe uma requisição realizada pelo cliente ao servidor, e uma resposta do servidor ao cliente.

Publish-Subscribe

Diferentemente do padrão *Request-Reply* supracitado, que traz uma sincronia de tempo entre as entidades comunicantes [5], tem-se também o modelo *Publish-subscribe*, utilizado em padrões de integração nos sistemas que adotam filas de mensagerias [9]. Nesta maneira de comunicação assíncrona, o componente *publisher* é responsável por publicar a

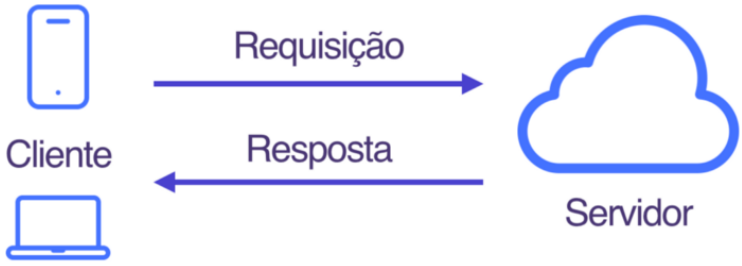


Figura 2.2: Modelo *Request-Reply*. Imagem retirada de `https://medium.com/`.

mensagem em um tópico, e o componente *subscriber* é responsável por se inscrever no tópico conforme necessidade e consumir a mensagem postada pelo *publisher*. Toda vez que há uma postagem ou uma alteração no tópico em que o *subscriber* está inscrito, ele é notificado e a regra de negócio estabelecida é aplicada e os dados disponíveis são tratados conforme demanda [9]. Dois sistemas de mensageria amplamente utilizados são: Apache Kafka[1] e RabbitMQ[2] [9]. A Figura 2.3 mostra uma entidade *publisher* enviando uma mensagem para um tópico, e mostra alguns *subscribers* inscritos no tópico recebendo as mensagens publicadas.



Figura 2.3: Modelo *Publish-subscribe*. Imagem adaptada de `https://inside.contabilizei.com.br/`.

2.2.2 *Containers*

Este conceito é fundamental para o padrão de microsserviços, uma vez que viabiliza a virtualização e o isolamento de processos, o aumento da portabilidade e uso mais eficiente dos recursos alocados e mitigação da complexidade em ambientes UNIX/Linux[10]. Algumas características desses containers são [10]:

[1] `https://kafka.apache.org/`
[2] `https://www.rabbitmq.com/`

- Isolamento em nível de *kernel*;
- Encapsulamento de recursos;
- Processo único.

Com esse artifício, é possível alocar e orquestrar esses containers conforme o modelo de negócios, alocando recursos de maneira eficiente para cada módulo, trazendo um *deploy* mais específico para cada módulo envolvido [10].

2.3 Desafios de Segurança

Para garantir um desenvolvimento seguro e protegido, é importante nomear algumas ameaças em potencial. O *framework* STRIDE[3] da Microsoft, acrônimo em inglês para *Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service e Elevation of Privileges*, destaca alguns dos riscos aos quais a aplicação pode estar sujeita [5]. A Tabela 2.1 apresenta algumas vulnerabilidades que a aplicação pode estar sujeita. E então, é de suma importância atenuar ao máximo essas ameaças durante todo o processo de desenvolvimento, utilizando sempre boas práticas de segurança [5].

Os termos do STRIDE traduzidos para o português são, respectivamente: falsificação, adulteração, repudiação, divulgação de informações, negação de serviço e elevação de privilégios.

2.4 Boas Práticas de Segurança

A gestão e a manutenção do acesso aos recursos e dados de um sistema são essenciais para garantir a segurança, incluindo princípios como confidencialidade, integridade e não-repudiação. A infraestrutura do sistema, composta por redes, bancos de dados e servidores web, deve ser protegida contra acessos não autorizados, pois qualquer violação pode causar danos graves ao sistema e a seus usuários [5].

A seguir estão listadas algumas boas práticas de segurança que visam atenuar os riscos apontados na Tabela 2.1 [5]:

- Cada usuário deve possuir uma conta;
- Usar papéis para os usuários (*Roles*);
- Adotar prática do privilégio mínimo para fornecer permissões aos usuários;

[3] `https://www.microsoft.com/en-us/security/blog/2007/09/11/stride-chart/`

Ameaça	Nome da Ameaça	Propriedade Violada	Definição
S	Falsificação de identidade	Autenticação	Fingir ser algo ou alguém diferente de quem realmente é
T	Adulteração de dados	Integridade	Modificar algo no disco, rede, memória ou em outro local
R	Repúdio	Não repúdio	Alegar que não fez algo ou que não foi responsável; pode ser verdadeiro ou falso
I	Divulgação de informação	Confidencialidade	Fornecer informações a alguém não autorizado a acessá-las
D	Negação de serviço	Disponibilidade	Esgotar os recursos necessários para fornecer um serviço
E	Elevação de privilégio	Autorização	Permitir que alguém faça algo para o qual não tem autorização

Tabela 2.1: Modelo de ameaças STRIDE (traduzidas)

- Encriptar senhas;
- Não salvar senhas em locais abertos.

Capítulo 3

Padrão OAuth2

Este capítulo apresentará uma visão teórica abrangente sobre o padrão OAuth 2.0, começando pela visão inicial na Seção 3.1, seguido pela descrição das entidades envolvidas na Seção 3.1.1, e os fluxos de autorização na Seção 3.1.2. A Seção 3.2 elucida a ideia de autorização e autenticação, ressaltando as diferenças e aplicabilidade. Em seguida, a Seção 3.3 (JWT) introduz a ideia dos *Javascript Object Notations* e sua relevância no padrão OAuth 2.0. Por fim, a Seção 3.4 aborda algumas questões medulares na implementação da segurança do padrão OAuth 2.0.

3.1 Visão Inicial

Com a evolução das aplicações modernas e o aumento do uso da computação em nuvem, novos desafios relacionados à segurança têm surgido, especialmente no que diz respeito à garantia de serviços eficientes de autenticação e autorização. Os sistemas precisam assegurar a integridade e a confiabilidade das informações fornecidas pelos usuários ao realizarem requisições com suas credenciais [6]. O OAuth 2.0[1] surgiu como um *framework* de autorização de alto nível amplamente utilizado pelas empresas, sendo uma referência de protocolo para implementação de soluções seguras e encorpadas [11].

O OAuth2.0 é utilizado em ambientes de computação em nuvem onde existe comunicação entre serviços e módulos do sistema, proporcionando uma atmosfera segura para o acesso de recursos e troca de informações entre as entidades participantes [11]. Com esse recurso, o usuário pode acessar diversos recursos sem a necessidade de compartilhar suas credencias em todas as requisições, haja vista que o mesmo, uma vez autenticado no servidor, recebe um *token* de acesso que será utilizado nas demais solicitações. Estes *tokens* têm sua integridade verificada em toda etapa da comunicação e, assim, uma via segura é estabelecida [2].

[1] `https://oauth.net/2/`

Algumas vantagens acompanham esse protocolo, e uma das principais é sua flexibilidade na implementação, podendo ser usado em aplicações web e também em servidores *back-end*, permitindo uma implementação conjunta de mecanismos de autenticação com a própria autorização [2]. A versatilidade do padrão OAuth 2.0 possibilita alguns fluxos de implementação e variações durante seu desenvolvimento, encaixando-se nos diversos cenários que a computação em nuvem enfrenta, viabilizando uma adaptação do protocolo de acordo com o escopo do projeto que está sendo trabalhado [11].

3.1.1 Papéis Essenciais

A documentação[2] do *framework* OAuth 2.0 apresenta os quatro papéis essenciais em que o protocolo é construído:

- ***Resource Owner*** - Representa a entidade responsável por fornecer acesso a algum recurso protegido, muitas vezes sendo o próprio usuário final [12].
- ***Resource Server*** - É o servidor responsável por armazenar os recursos protegidos, tratando requisições e solicitações de acesso, checando a integridade do *token* de acesso recebido [12].
- ***Client*** - Representa a aplicação que deseja acessar o recurso protegido armazenado no *Resource Server*, ou seja, é o responsável pelas requisições enviadas para o servidor de recursos [12].
- ***Authorization Server*** - Servidor que autentica o dono do recurso e emite *tokens* de acesso para o cliente obter autorização [12].

Todo esse fluxo e o relacionamento entre os papéis está representado na Figura 3.1, mostrando como cada entidade mencionada se relaciona com as demais.

3.1.2 Principais Fluxos de Implementação

Existem quatro principais padrões de implementação para os fluxos de autorização. Compreendê-los e utilizá-los de forma adequada garantirá um alto nível de segurança para a solução desenvolvida. Os quatro padrões são:

- ***Authorization Code Grant*** - É a solução mais comumente adotada por aplicações web, na qual há uma etapa extra que é recebido um código de autorização, o qual é trocado posteriormente pelo *token* de acesso. Esta solução garante uma

[2] `https://datatracker.ietf.org/doc/html/rfc6749`. Disponível em: 31/01/2025. Último acesso: 31/01/2025.

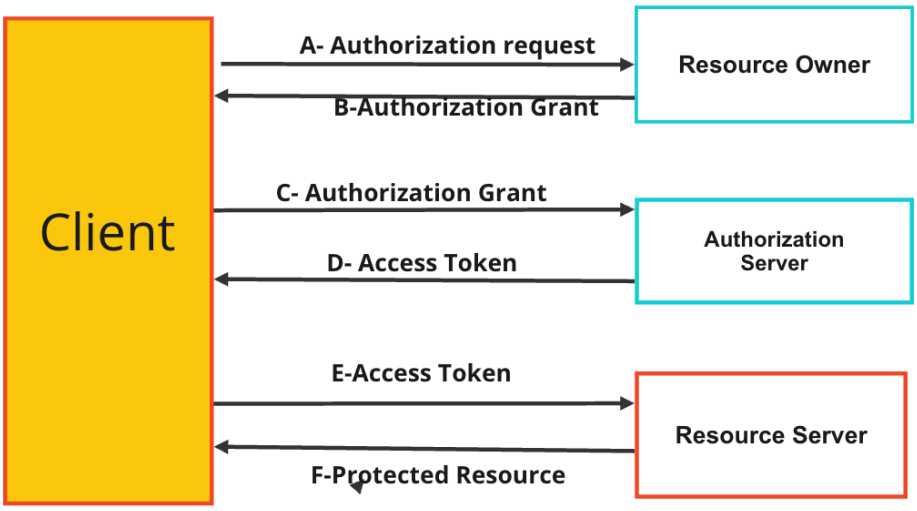


Figura 3.1: Fluxo padrão do OAuth2.0. Imagem adaptada de `https://medium.com/`.

segurança bastante eficiente ao manter o *token* de acesso fora do *browser*, porém tem uma implementação bastante complexa [11];

- ***Implicit Grant*** - Esta solução é utilizada para obter *tokens* de acesso para clientes públicos, como em *Single Page Applications*(SPA) [12]. Diferentemente do fluxo *Authorization Code Grant*, o cliente recebe o *token* no processo de autorização, sem passar pela etapa de autenticação, pois o *token* de acesso está vinculado à *Uniform Resource Identifier*(URI) de redirecionamento do cliente, presente em um fragmento [11];

- ***Client Credentials Grant*** - Este modelo é utilizado para aplicações *machine-to-machine* [11]. Desta forma, o *token* é obtido ao fornecer as credenciais do cliente para o *endpoint* de autorização exposto [11]. O primeiro passo é o cliente se autenticar no servidor de autorização e solicitar um *token*. Caso a autenticação seja bem sucedida, o *token* é emitido. É uma maneira simples de comunicação e não há necessidade do envolvimento do usuário [11];

- ***Resource Owner Password Credentials Grant*** -Este fluxo é válido para situações em que o dono do recurso tem uma relação confiável com o cliente [12], pois o cliente fornece as credenciais do usuário e troca por um *token* de acesso. É uma dinâmica simples e fornece uma experiência satisfatória para o usuário final [11], além de ser muito utilizada para migração de aplicações legadas [12]. Na Figura 3.2

é possível observar uma exemplificação do processo descrito, em que o usuário fornece as credenciais para o cliente. Em seguida, o cliente envia os dados recebidos ao servidor de autorização, solicitando um *token* de acesso e obtendo-o na resposta.

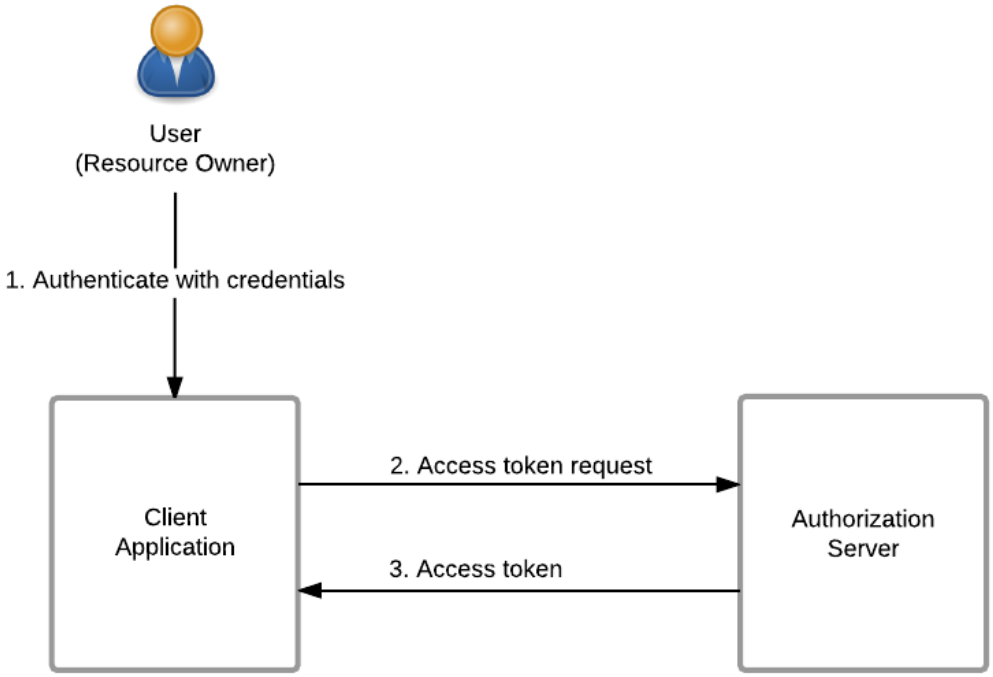


Figura 3.2: Fluxo do *Resource Owner Password Credentials Grant*. Imagem retirada de `https://is.docs.wso2.com/en/latest/`.

3.2 Autenticação e Autorização

Existe uma diferença entre o processo de autenticação e autorização [6]. Autenticação é o processo que valida se o usuário realmente é quem ele diz ser. Existem algumas fraudes que podem ser cometidas em termos de credenciais do usuário. Portanto, é de suma importância garantir que a autenticação de um sistema seja bem elaborada e restrinja as falhas de segurança [6]. Uma vez que se tem conhecimento de quem o usuário realmente é, deve-se analisar qual o limite de ações que um usuário autenticado pode fazer, ou seja, filtrar os recursos que o usuário pode ter acesso. Esta etapa é chamada de autorização [6]. Com essa ideia em mente, é fundamental a estratégia correta e segura desta etapa de desenvolvimento de um sistema [2]. Caso um microsserviço de um sistema grande seja vulnerável nesse quesito, acabará comprometendo toda a aplicação [2].

3.3 *JSON Web Token* (JWT)

Um dos modelos de *token* de acesso emitido pelo servidor de autorização é o *JSON Web Token* (JWT) [6]. O *Bearer token*[13] é um método que utiliza JWT, mais seguro para comunicação *Hypertext Transfer Protocol* (HTTP) e *Secured Hypertext Transfer Protocol* (HTTPS) [14]. Em vez de transmitir as credenciais do usuário em todas as requisições, esse *token* é gerado de forma criptografada após o processo de autenticação e é autossuficiente, sendo enviado nas requisições subsequentes, o que evita o tráfego de informações sensíveis.[14].

O JWT[3] é uma solução eficiente e aberta para transmissão de dados de maneira segura entre as entidades comunicantes. Utiliza-se um objeto *Javascript Object Notation*(JSON) estruturado em três partes que é assinado digitalmente utilizando uma senha. As três partes de um JWT são o *header, payload* e assinatura. O *header* possui dois componentes: o tipo de *token* e o algoritmo utilizado em sua montagem [6]. A estrutura interna está representada na Figura 3.3, obtida após colar o *token* no site oficial do JWT, utilizando a funcionalidade pública fornecida. Alguns benefícios do JWT são:

- Verificação independente do servidor de autenticação [2];
- Possibilidade de conter informações do usuário [2];
- Pode trazer os privilégios do usuário e uma data de expiração [2];
- Integração com o padrão OAuth 2.0 de maneira eficiente [14];

Por estes motivos, o JWT é amplamente utilizado no desenvolvimento de servidores *backend*, pois cria um ambiente *stateless*, ao eliminar a necessidade de armazenamento de sessão. Todavia, ele é enviado no *Header* das requisições sempre que são realizadas [2].

3.4 Princípios Essencias na Segurança do OAuth 2.0

A segurança de uma aplicação é um tema bastante estudado nos últimos anos [15], e representa um aspecto crucial para seu bom funcionamento. A seguir estão alguns itens envolvidos no contexto de segurança do padrão OAuth 2.0, que devem ser levados em consideração no desenvolvimento de uma solução [15]:

- Confidencialidade dos dados;
- Autenticação do usuário;

[3] `https://jwt.io/`

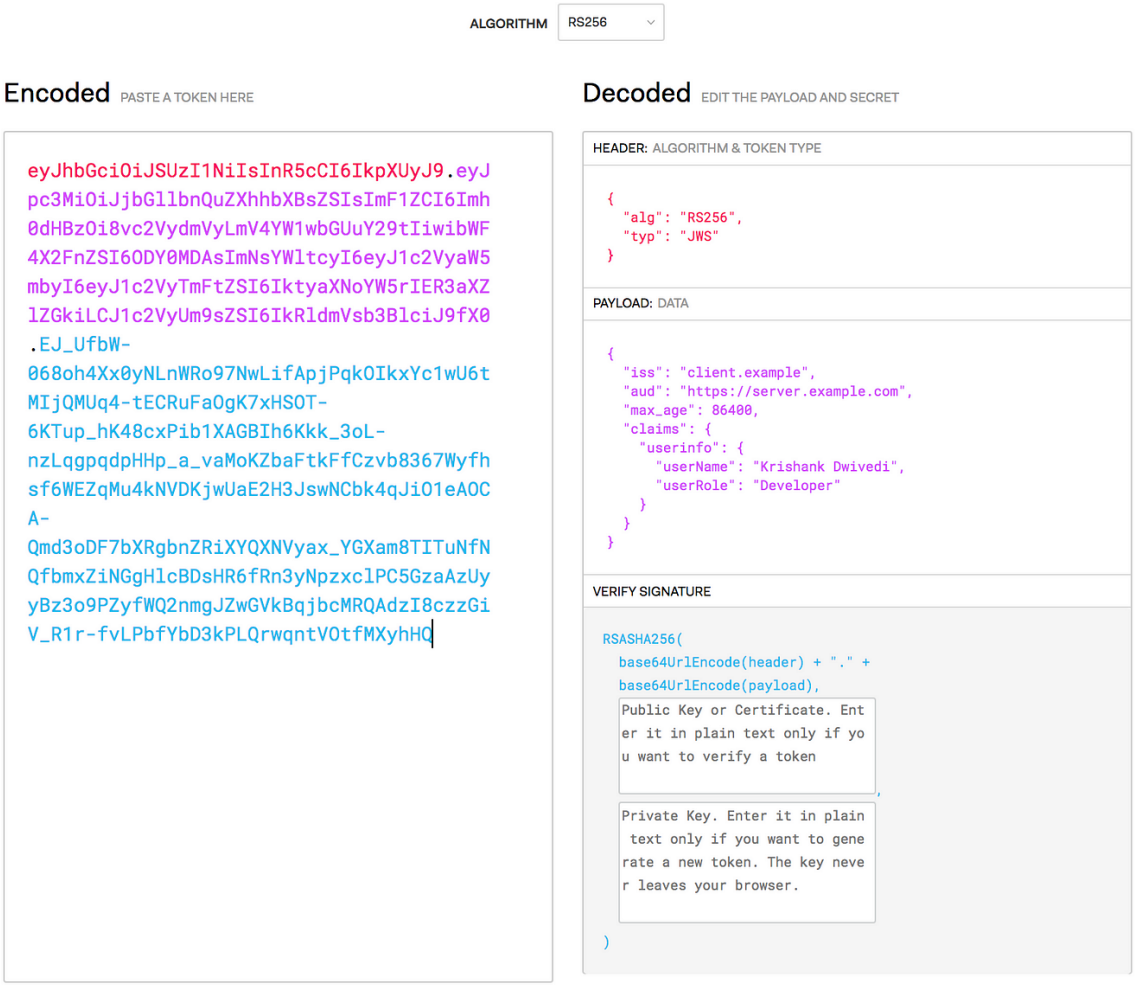


Figura 3.3: Estrutura interna de um *token* JWT.

- Autorização de acesso a recursos;
- Definição do proprietário e consistência dos dados;
- Criptografia;
- *Cross-Origin Resource Sharing*(CORS).

Considerando que a segurança depende de uma implementação sólida dos requisitos do sistema [16], a adoção de boas práticas de desenvolvimento permite a criação de uma solução bem estruturada e eficaz em termos de proteção [15].

Capítulo 4

Criptografia e Segurança

Este capítulo apresenta uma visão abrangente sobre a história, aplicabilidade e alguns conceitos importantes na Seção 4.1. Logo depois, o capítulo aborda na Seção 4.2 a diferença entre a criptografia simétrica e assimétrica.

4.1 Conceitos Iniciais

Existe um grande volume de dados e informações circulando na Internet atualmente [1]. Em redes de computadores públicas como a internet, deve-se estabelecer maneiras de garantir a integridade das informações circuladas, garantindo uma proteção ao dono do recurso e a privacidade eletrônica [17]. É nesse cenário que entra a criptografia, que resumidamente, é um campo de estudo que visa codificar mensagens para que somente entidades autorizadas consigam decifrá-las [18].

O termo criptografia tem sua etimologia proveniente do grego *kritos* (oculto) e *grapho* (escrita)[17]. Modernamente, as fórmulas utilizadas para ocultar as mensagens são chamadas de algoritmos, e entendê-los e estudá-los é essencial para assegurar uma boa qualidade tecnológica na integridade e segurança dos recursos [17]. A privacidade sempre foi um fator valorizado, seja no âmbito militar, político, econômico ou até mesmo social no cenário da comunicação [18].

No decorrer do tempo, existiram diversas situações em que a arte de decifrar algo era considerada uma arte bastante virtuosa exercida por matemáticos e filósofos, como na caça ao tesouro no antigo Egito ou codificação de mensagens importantes no período de guerra [18]. Um dos maiores exemplos do uso da criptografia bélica foi o caso da máquina de criptografia Enigma durante a Segunda Guerra Mundial. Os alemães nazistas utilizaram essa máquina para enviar mensagens codificadas com tamanha complexidade que era praticamente impossível decifrá-las. No entanto, matemáticos e criptógrafos aliados, com o apoio de Alan Turing e sua equipe em Bletchley Park, conseguiram quebrar o algo-

ritmo e decifrar as mensagens [19]. Existem alguns fatores que caracterizaram o cenário criptográfico [17]:

- **Disponibilidade:** O dado criptografado tem que estar pronto para ser acessado no momento pretendido;
- **Integridade:** Deve-se certificar a segurança do conteúdo codificado, evitando sua adulteração;
- **Controle de acesso:** Firmar fluxos para garantir o acesso ao recurso somente por membros autorizados;
- **Autenticidade:** Assegurar a verdadeira identidade do emissor da mensagem;
- **Não-repudiação:** Extinguir a chance da negação da emissão ou recebimento do escopo da mensagem em questão;
- **Privacidade:** Firmar padrões que garantem exclusividade de entidades autorizadas a acessar os conteúdos, garantindo o monopólio do corpo da mensagem.

4.2 Criptografia Simétrica e Assimétrica

O processo de criptografia ocorre quando um emissor deseja enviar uma mensagem garantindo que somente o receptor seja capaz de lê-la corretamente. Em outras palavras, trata-se da transformação de um texto claro, ou seja, informação legível, em um texto ilegível por meio da aplicação de um algoritmo criptográfico. Esse processo é conhecido como codificação. Por outro lado, a decodificação refere-se à conversão do texto ilegível de volta para um texto legível, utilizando um método apropriado [18]. Existem duas formas de criptografia: simétrica (tradicional, utilizando chave privada) e assimétrica (chave pública) [18], cada um dos tipos será discutido a seguir.

Criptografia Simétrica (chave privada)

Representa o modelo mais antigo da criptografia, em que a chave (elemento responsável por conceder acesso à mensagem oculta) é a mesma para as duas entidades comunicantes, e a mesma deve ser mantida escondida, para evitar seu vazamento. A chave normalmente é uma senha, que é usada nas duas pontas da comunicação, tanto no processo de codificação quanto no processo de decodificação [17]. Neste processo é fulcral manter a chave em segredo, para não permitir que interceptadores consigam acesso e pratiquem alguma ação indesejada com o conteúdo trafegado. A segurança da chave é mais essencial do que a do próprio algoritmo usado para gerá-la. Esse método é simples de implementar, mas

exige atenção à segurança da chave e ao escalonamento, pois em uma rede com muitos usuários, seriam necessárias várias chaves exclusivas para cada par emissor-receptor [17]. A Figura 4.1 mostra a lógica descrita, em que a mesma chave privada é usada para criptografar na origem e descriptografar no destino.

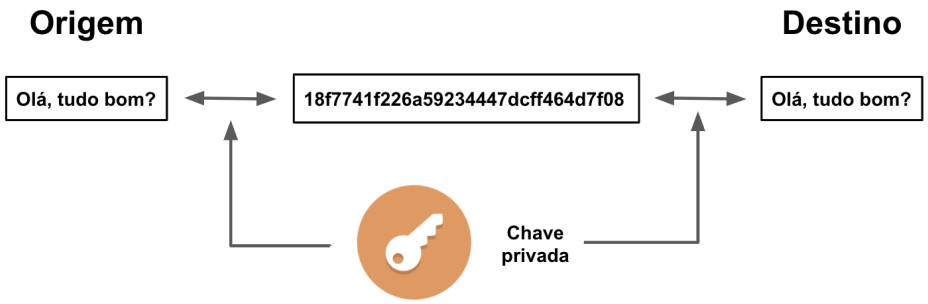


Figura 4.1: Funcionamento da criptografia simétrica. Imagem retirada de `https://www.universidadejava.com.br/`.

Criptografia Assimétrica (chave pública)

Diferentemente do modelo simétrico, a criptografia assimétrica possui um par de chaves relacionadas entre si [18]: uma chave pública e uma chave privada. A chave pública é usada para criptografar a mensagem, e a chave privada é usada para descriptografar. Desta forma, mesmo se a chave pública for exposta, ainda sim o conteúdo da mensagem está seguro pois só é possível interpretá-la com a chave privada [18].

O modelo que utiliza o par de chaves (assimétrica) foi criado na década de 1970 pelo matemático inglês Clifford Cocks e aprimora o conceito de chaves, que no modelo simétrico eram senhas. Agora, as chaves passam a ser componentes digitais muito mais elaborados, o que implica um maior custo computacional para o processamento das mensagens [17]. A lógica envolvida no processo assimétrico é um pouco diferente: cada membro envolvido na comunicação possui uma chave privada e uma chave pública, geradas a partir de um algoritmo que as relaciona internamente, de modo que elas estarão sempre associadas entre si [17].

As chaves públicas de todas as entidades ficam disponíveis publicamente, permitindo que a mensagem seja codificada. Uma vez codificada, somente a chave privada associada à chave pública utilizada para a codificação poderá decifrar a mensagem. Este fluxo traz uma maior fluidez na comunicação, em que qualquer um pode enviar uma mensagem secreta utilizando a chave pública do destinatário, e não há necessidade do envio da chave

privada como havia no modelo simétrico. Mesmo com o avanço de segurança computacional envolvido no processo, a complexidade dos algoritmos que geram as chaves é muito alta, acarretando um maior custo de processamento [17]. A Figura 4.2 representa esse funcionamento, em que a chave pública é usada para criptografar a mensagem e a chave privada é usada para descriptografar.

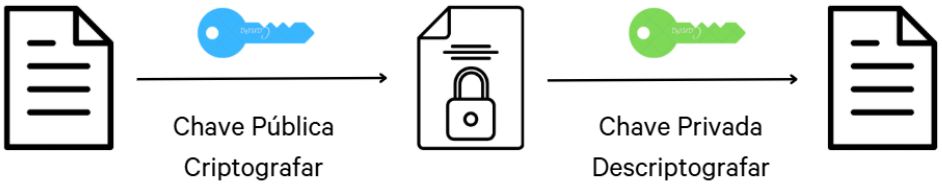


Figura 4.2: Funcionamento da criptografia assimétrica.

No processo de criptografia assimétrica, um algoritmo bastante utilizado é o RSA [18]. A base do funcionamento dele é a utilização de multiplicação de números primos e sua subsequente fatoração. Obter a chave privada a partir da chave pública consiste em fatorar o número, e pelo fato de ser muito grande, não há quem consiga realizar a operação em uma quantidade de tempo decente [17].

Capítulo 5

Implementação

O capítulo de implementação trás a solução desenvolvida para demonstrar todos os tópicos abordados anteriormente. Primeiramente, na Seção 5.1 serão apresentadas as tecnologias empregadas durante a implantação do sistema, assim como seu código fonte. Em seguida, na Seção 5.2 será explicada a solução desenvolvida e como são resolvidos os problemas expostos. A Seção 5.3 apresenta um detalhamento da implementação e os resultados das requisições realizadas. Por fim, a Seção 5.4 fornece um parecer final sobre o trabalho.

5.1 Considerações Iniciais

Para exemplificar os tópicos discutidos neste trabalho e abordar as questões expostas, foi desenvolvido um servidor *backend* de autorização e autenticação utilizando a linguagem de programação Java[1] no ambiente Spring Boot[2]. Foi utilizado um *container* Docker[3] para subir um banco de dados relacional MySQL[4]. Também foi usado o Beekeper[5], um *software* para gerenciamento de bancos de dados multiplataforma. Por fim, foi usado cliente Postman[6] para realizar requisições ao servidor. O código fonte do sistema está disponível no Github[7].

5.2 Solução Desenvolvida

A solução proposta envolve um contexto em que um professor deseja lançar notas para seus alunos. Para isso, cada aluno deve se registrar no sistema para que o professor

[1] `https://www.java.com/pt-BR/`
[2] `https://spring.io/projects/spring-boot`
[3] `https://www.docker.com/products/docker-desktop/`
[4] `https://www.mysql.com/`
[5] `https://www.beekeeperstudio.io/`
[6] `https://www.postman.com/`
[7] `https://github.com/theorocha/tcc`

possa visualizá-lo e, em seguida, atribuir-lhe uma nota. Além do cadastro e *login*, cada aluno tem permissão apenas para visualizar sua própria nota. O professor também pode visualizar todos os usuários do sistema, listar todas as notas lançadas e excluir uma nota. Além disso, há um terceiro tipo de usuário: o *super admin*. Somente esse usuário é capaz de visualizar as *roles* do sistema, privilégio que nem mesmo o professor possui. Este exemplo foi utilizado com o intuito de demonstrar uma abordagem prática dos conceitos discutidos, podendo ser replicado em diversas situações e cenários.

Visão geral

O servidor desenvolvido cumpre dois dos quatro papéis discutidos na Seção 3.1: *Resource Server e Authorization Server*. Já o papel de *Client* é exercido pelo Postman, e o *Resource Owner* é o usuário final que está utilizando o sistema.

Foi utilizado o *Spring Boot* do ecossistema *Spring* no desenvolvimento, pois é uma ferramenta poderosa com bastante documentação e suporte para o desenvolvedor, carregando algumas configurações para que o foco seja principalmente na implantação da solução, ao invés de detalhes burocráticos [15].

Fluxo do Servidor

O sistema simula uma situação em que o professor deve dar notas aos alunos, para isso foram criados dois papéis (*roles*): *basic e admin*. As tabelas no banco de dados MySQL são: *users, grades, roles, users grades*, esta última uma tabela auxiliar para armazenar os usuários e seus respectivos privilégios. O estudante deve se registrar e em seguida realizar o *login*. Quando logados no sistema, um *token* JWT é gerado e enviado no corpo da resposta da requisição com todas as informações e permissões necessárias para tarefas futuras, seguindo o fluxo *Resource Owner Password Credentials Grant*, discutido na Seção 3.1.2, devido a sua simplicidade e eficiência. O aluno só é capaz de ver a própria nota, e o professor é autorizado a acessar algumas APIs para realizar algumas ações:

- Ver todos os usuários cadastrados no sistema;
- Lançar uma nota para um aluno;
- Deletar a nota de um aluno;
- Ver todas as notas lançadas.

Para demonstrar o funcionamento das *roles*, também foi criado um novo perfil, o *Super Admin*, com sua nova respectiva *role* (Super Admin). Ele também consegue ver todos os

usuários do sistema e possui o privilégio exclusivo de ver todas as roles presentes no banco de dados.

Com a regra de negócio clara e desenvolvida, alguns componentes de criptografia, autorização e autenticação foram implementados. Vale ressaltar que só alunos conseguem se cadastrar na API de registro, os demais perfis só podem ser cadastrados de maneira manual diretamente no banco de dados ou por meio de linhas de código durante a implementação.

Criptografia

Para realizar a emissão dos *tokens* JWT após o */login*, foi utilizada a lógica de criptografia assimétrica usando o algoritmo RSA. A chave privada foi gerada executando o seguinte *prompt* no terminal do computador macOS:

```
$ openssl genrsa
```

A saída gera um arquivo chamado `private.pem`. Para gerar a chave pública, foi executado o seguinte comando, passando como parâmetro o nome do arquivo que contém a chave privada:

```
$ openssl rsa -in private.pem -pubout -out public.pem
```

O arquivo *public.pem* contém a chave pública associada à chave privada de *private.pem*. Essas chaves foram salvas no código fonte, nos arquivos chamados *app.pub* e *app.key*, respectivamente.

Além da geração do par de chaves, a criptografia no sistema também se aplica para o armazenamento das senhas no banco de dados, pois foi implementado um codificador de segredos utilizando o algoritmo *bcrypt*, fornecido pelo ecossistema *Spring*. Ao se adicionar o *bean* abaixo, o *Spring* o trata como um objeto gerenciado, segundo a documentação oficial.

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

Autenticação

Após o aluno se registrar com sucesso no sistema através do endpoint */register* e ter seu usuário salvo no banco de dados, ele pode partir para o */login*. Uma vez que o usuário

envia a solicitação para logar no sistema na rota */login*, o servidor recebe o nome e a senha enviados e primeiro verifica se o usuário existe no banco de dados, caso contrário, uma exceção é lançada. Se o usuário consta no banco de dados e a senha enviada, após criptografada, coincide com a senha criptografada presente na base de dados, um *token* JWT é gerado e enviado no corpo da resposta da requisição.

Autorização

Na montagem do *token* JWT, algumas informações são colocadas no conteúdo do *token*, são elas:

- Entidade responsável pela emissão do *token*, no caso o próprio servidor;
- O usuário que recebeu o *token*;
- *Roles* do usuário logado;
- Data de emissäo e data de expiração do *token*.

Estas informações são o suficiente para desenvolver uma solução capaz de restringir e proteger o acesso às APIs conforme os privilégios de cada usuário solicitante.

A documentação oficial do *Spring* mostra que o *framework* oferece duas interfaces que foram utilizadas durante o desenvolvimento: *JwtEncoder* e *JwtDecoder*. Elas foram implementadas e representam a codificação e decodificação discutidas na Seção 4.2. Foi utilizada a biblioteca *Nimbus JOSE+JWT* fornecida pelo Spring para o desenvolvimento do componente responsável pela codificação e decodificação do sistema utilizando o algoritmo RSA, respectivamente, mostradas abaixo:

```
@Bean
public JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(this.publicKey).privateKey(privateKey).build();
    var jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);

        @Bean
public JwtDecoder jwtDecoder() {
    return NimbusJwtDecoder.withPublicKey(publicKey).build();
}
}
```

5.3 Funcionamento

O funcionamento e dinâmica do sistema serão discutidos nesta seção, juntamente com os resultados obtidos em imagens e trechos explicatórios. Para realizar as requisições, o *software* Postman foi utilizado, e o JWT foi enviado, quando necessário, no *Header Authorization Bearer <access token>* . A fim de conseguir visualizar as tabelas do MySQL o *software* Beekeeper foi aplicado. Inicialmente, só existem dois usuários no banco de dados: *admin e super admin*, pois foram inseridos de maneira manual.

5.3.1 Fluxo do Estudante

A Figura I.1 mostra a requisição de registro de um usuário na rota */register*, juntamente com o retorno de sucesso. O usuário cadastrado persistido na tabela de usuários no banco de dados com sua *role basic* está representado na Figura I.2. Em seguida a Figura I.3 mostra a realização do *login* pelo usuário na rota */login* e o retorno do servidor com o *Access Token* (JWT). Na Figura I.4 é colocado o *access token* no próprio site oficial[8], revelando seu conteúdo interno. Este JWT é utilizado para realizar a requisição seguinte no endpoint */my-grade*, como é visto na Figura I.5, e assim a nota do aluno solicitante é retornada, caso exista. Percebe-se que não há nenhuma variável na rota para obter a própria nota, o servidor sabe qual é o aluno solicitante através da informação contida no JWT enviado no *Header Authorization Bearer*. O trecho de código abaixo mostra como o nome do aluno é extraído do *token* no controlador.

```
@GetMapping("/my-grade")
@PreAuthorize("hasAuthority('basic')")
public ResponseEntity<Object> myGrade(JwtAuthenticationToken token) {
    UUID userId = UUID.fromString(token.getName());
    var userOptional = userRepository.findById(userId);
    if (userOptional.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
        .body("User not found.");
    }
    var user = userOptional.get();
    Optional<Grade> gradeOptional = gradeRepository.findByUser(user);
    if (gradeOptional.isEmpty()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND)
        .body("You have no grade.");
```

[8] https://jwt.io/

```
        }
        return ResponseEntity.ok(gradeOptional.get());
    }
```

Nota-se na Figura I.6 que, caso o estudante tente realizar uma ação que sua *role basic* não permita, como deletar a própria nota, receberá de resposta do servidor um *Status Code 403 Forbidden* , que representa a falta de privilégios para realizar a ação.

5.3.2 Fluxo do Professor

Para entender o papel do professor, é importante lembrar que ele não realiza cadastro, tendo seu perfil criado somente pelo *owner* do sistema. Ao enviar suas credenciais para a rota */login*, também recebe o retorno do servidor com o *Access Token* (JWT), desta vez com a *role admin* presente. Utilizando o *token* para realizar as requisições que ele tem permissão, ele conseguirá realizá-las com sucesso, como lançar a nota de um aluno, mostrado na Figura I.7. Para isso, o professor faz a requisição fornecendo a nota e o identificador único do aluno. Percebe-se na Figura I.8 que a nota lançada foi persistida no banco de dados. O professor também pode acessar todas as notas lançadas, como se percebe na Figura I.9. Outro recurso que o professor pode acessar é excluir a nota de um aluno, demonstrado na Figura I.10, em que o identificador único da nota é passado na rota como parâmetro. Por fim, o professor é capaz de ver todos os usuários do sistema ao realizar a requisição *GET* para a rota */users*, mostrado na Figura I.11. Uma requisição que está fora do escopo de atuação do professor terá como resposta um *Status Code 403 Forbidden*, por exemplo a visualização de todas as roles da base de dados, perceptível na Figura I.12. Isto ocorre pois essa rota é exclusiva para o *super admin*.

5.3.3 Fluxo do *Super Admin*

O *super admin* também não realiza cadastro, seu perfil é criado manualmente. O usuário *super admin* consegue também ver todos os usuários do sistema, privilégio que o professor também tem. Esta lógica é possível por causa da notação `@PreAuthorize` do Spring Boot, que permite a implementação do controle de acesso pautado nas *roles*, concatenando as que têm permissão para acessar o recurso. Ao se logar e também receber um JWT de acesso, o mesmo é enviado no *Header* das requisições seguintes. Nota-se na Figura I.13 que o *super admin* é capaz de obter acesso a todos os usuários do sistema ao realizar uma solicitação para a rota */users*. Além desta função, o *super admin* tem um privilégio exclusivo: ver as roles do sistema, acessando a rota */roles*. Negado para os professores, o *super admin* tem acesso a esta função, representada na Figura I.14. Se ele tentar acessar uma rota fora de seu escopo de atuação, como excluir uma nota de um aluno,

recurso exclusivo do professor, receberá um *Status Code 403 Forbidden*, como é visível na Figura I.15.

5.4 Considerações finais

Perante os resultados apresentados, percebe-se que o fluxo de segurança empregado, assim como o padrão criptográfico utilizado, proporcionam uma solução robusta e eficaz para um sistema distribuído pautado em microsserviços, conclusão feita após inúmeros testes com os mais diferentes papéis e cenários na aplicação. As requisições foram realizadas e todas elas foram condizentes com o padrão de segurança esperado e definido na regra de negócio. A utilização de *roles* para acesso a *endpoints* usando a notação `@PreAuthorize` é um protocolo que se mostrou seguro e funcional ao limitar acesso a recursos somente a entidades autorizadas, provando a eficiência das abordagens utilizadas, uma vez que ataques maliciosos sem a devida autorização são bloqueados pelo servidor. Outrossim, o *backend* desenvolvido possui um código fonte bastante amigável e de fácil manutenção, com uma organização em pastas bem elaborada e com os componentes envolvidos na configuração sendo bem intuitivos e limpos, malgrado a verbosidade da linguagem de programação utilizada.

Capítulo 6

Conclusão

Ao final deste trabalho, percebe-se que o padrão OAuth 2.0 é um protocolo muito eficiente e seguro para desenvolver estratégias de segurança e controle de acesso em um sistema pautado na arquitetura de microsserviços, uma vez que estabelece normas protetivas que, ao serem implementadas, criam um ambiente dinâmico e preservado. Foi feito um levantamento teórico com base nas referências expostas que mostra a importância de seguir os padrões adotados para conseguir obter uma solução íntegra e completa, atenuando as possíveis vulnerabilidades e suprindo a necessidade do controle de acesso das *APIs* criadas seguindo a autorização pelas *roles* do usuário. O uso do fluxo *Resource Owner Password Credentials Grant* descrito pelo OAuth 2.0 se revelou consistente, simples e eficiente no controle de acesso aos recursos, pois o *token* de acesso gerado é seguro e descritivo. A criptografia também se mostrou presente e revelou um papel crucial para a segurança das informações trafegadas na internet, sempre protegendo e visando a integridade dos dados.

Foi possível concluir os objetivos descritos na Seção 1.2. A avaliação dos problemas de segurança no desenvolvimento de microsserviços, foi feita na Seção 2.3, com um detalhamento na Tabela 2.1. Dessa forma, o Objetivo 1 foi alcançado. Essa análise expôs alguns pontos importantes que foram levados em consideração na elaboração da solução. Em seguida, a identificação e boas práticas de segurança envolvendo o padrão OAuth 2.0 foram realizadas de forma teórica na Seção 3.4. O entendimento conceitual foi essencial para a aplicação desses princípios durante o desenvolvimento do servidor *backend*, demonstrando a importância de se ter cumprido o Objetivo 2. Por fim, as definições apresentadas e discutidas neste trabalho demonstraram sua importância para uma solução segura e eficaz. Os conceitos abordados foram aplicados durante a implementação, viabilizando uma exemplificação prática e robusta, detalhada na Seção 5.2, permitindo o alcance do Objetivo 3.

A escolha da linguagem de programação Java associada ao *framework* Spring Boot se mostrou uma estratégia adequada para a implementação proposta, tendo em vista que

carrega um ecossistema encorpado e com bastante suporte para os desenvolvedores. Além da popularidade e otimização da tecnologia, também existe uma grande versatilidade, viabilizando a implementação específica pelo programador conforme as regras de negócio requisitadas. A escolha do banco de dados MySQL se mostrou efetiva ao oferecer um banco de dados completo, permitindo que as ações sejam realizadas de maneira idônea. O cliente Postman também foi muito bem utilizado, possibilitando que as requisições sejam realizadas em uma interface amigável e atendendo às expectativas de performance e tempo de resposta.

Como trabalho futuro, é proposto o desenvolvimento de um *frontend* para trazer uma melhor experiência para o usuário e até mesmo externalizar a solução desenvolvida para algum propósito dentro da sala de aula. Vale frisar a importância da adoção de medidas de segurança ao integrar com um *frontend*. Torna-se uma etapa essencial a configuração do servidor para aceitar apenas requisições de um cliente autorizado e também a validação dos dados de entrada, garantindo a sanitização adequada para prevenir injeções de código malicioso. Além disso, é fundamental implementar cabeçalhos *Hypertext Transfer Protocol* (HTTP) de segurança, como *Content-Security-Policy* para restringir a execução de *scripts* não autorizados, *X-XSS-Protection* para mitigar ataques XSS e definir o *Content-Type* correto para evitar interpretações indevidas. Deve-se ainda utilizar cookies com atributos *HttpOnly* e *Secure* para proteger informações sensíveis, evitar o uso de métodos inseguros como *eval()* e *innerHTML*, além de monitorar requisições suspeitas para detectar possíveis tentativas de exploração de vulnerabilidades. Também é válida a ideia de desenvolver alguns microsserviços para integrar com a solução desenvolvida.

Referências

[1] Watanabe, Eduardo e Renato Tarciso Barbosa de Sousa: *Processos judiciais eletrônicos: Desafios para a gestão a partir da arquivologia e da ciência da informação.* Encontro de administração da justiça, 2019. 1, 16

[2] Almeida, Murilo Góes de: *Implementação do protocolo oauth2 e openid connect em uma arquitetura orientada a microsserviços.* Implementação do Protocolo OAuth2 e OpenID Connect em uma Arquitetura Orientada a Microsserviços, 2022. 1, 6, 10, 11, 13, 14

[3] Canongia, Claudia e Raphael Mandarino Junior: *Segurança cibernética: o desafio da nova sociedade da informação.* Parcerias Estratégicas, 14(29), 2009. 1, 2

[4] Nguyen, Quy e Oras F Baker: *Applying spring security framework and oauth2 to protect microservice architecture api.* J. Softw., 14(6):257–264, 2019. 1, 4

[5] Oliveira, Amanda Basso de: *Modelagem de arquitetura baseada em microsserviços com foco em segurança.* Modelagem de arquitetura baseada em microsserviços com foco em segurança, 2022. 1, 2, 4, 5, 6, 8

[6] Gibbons, Kevin, John O Raw e Kevin Curran: *Security evaluation of the oauth 2.0 framework.* Information Management and Computer Security, 22(3):01–23, 2014. 2, 10, 13, 14

[7] Pahl, Claus e Pooyan Jamshidi: *Microservices: A systematic mapping study.* Closer (1), páginas 137–146, 2016. 4

[8] Fernandes, Eduardo Mioto de Oliveira dos Santos: *Sugestão de criticidade baseada em multicritério para arquiteturas estabelecidas orientadas a microsserviços.* Tese de Doutoramento, Universidade Federal do Rio de Janeiro, 2020. 5, 6

[9] Amorim, JP e BBN França: *Análise de dependências via comunicação assíncrona em arquiteturas baseadas em microsserviços.* 2020. 6, 7

[10] Brinza, Andrei: *Metodologia para conteinerização de micro-serviços.* Tese de Mestrado, Universidade de Évora, 2022. 7, 8

[11] Shekhar, Gaurav: *Oauth 2.0 patterns implementation for cloud architecture.* 3, 2024. 10, 11, 12

[12] Hardt, Dick: *The OAuth 2.0 Authorization Framework.* RFC 6749, outubro 2012. 11, 12

[13] Jones, Michael e Dick Hardt: *The oauth 2.0 authorization framework: Bearer token usage*. Relatório Técnico, 2012. 14

[14] Bucko, Ahmet, Kamer Vishi, Bujar Krasniqi e Blerim Rexha: *Enhancing jwt authentication and authorization in web applications based on user behavior history*. Computers, 12(4):78, 2023. 14

[15] Chatterjee, Ayan e Andreas Prinz: *Applying spring security framework with keycloak-based oauth2 to protect microservice architecture apis: a case study*. Sensors, 22(5):1703, 2022. 14, 15, 21

[16] McGraw, Gary: *Software security*. Building security in, 2006. 15

[17] Oliveira, Ronielton Rezende: *Criptografia simétrica e assimétrica-os principais algoritmos de cifragem*. Segurança Digital [Revista online], 31:11–15, 2012. 16, 17, 18, 19

[18] Bonfim, Daniele Helena: *Criptografia RSA*. Tese de Doutoramento, Universidade de São Paulo, 2017. 16, 17, 18, 19

[19] Scaquett, Willian e Célia de Oliveira de Santana: *A importância da computação na segunda guerra mundial*. REGRAD-Revista Eletrônica de Graduação do UNIVEM-ISSN 1984-7866, 17(1):238–251, 2024. 17

Anexo I

Imagens obtidas na utilização do sistema

Neste anexo estão presentes as imagens referenciadas na Seção 5.3.

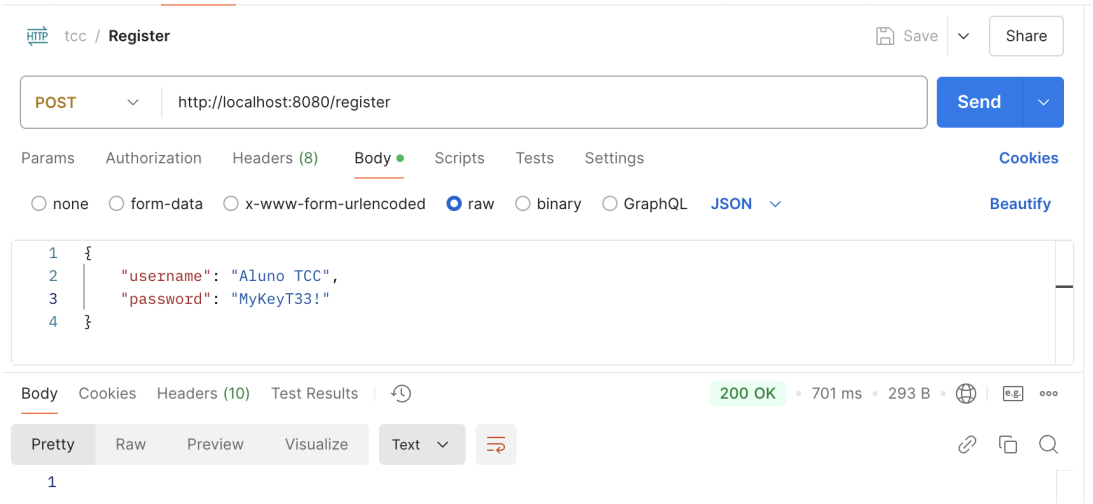


Figura I.1: Requisição de registro do aluno.

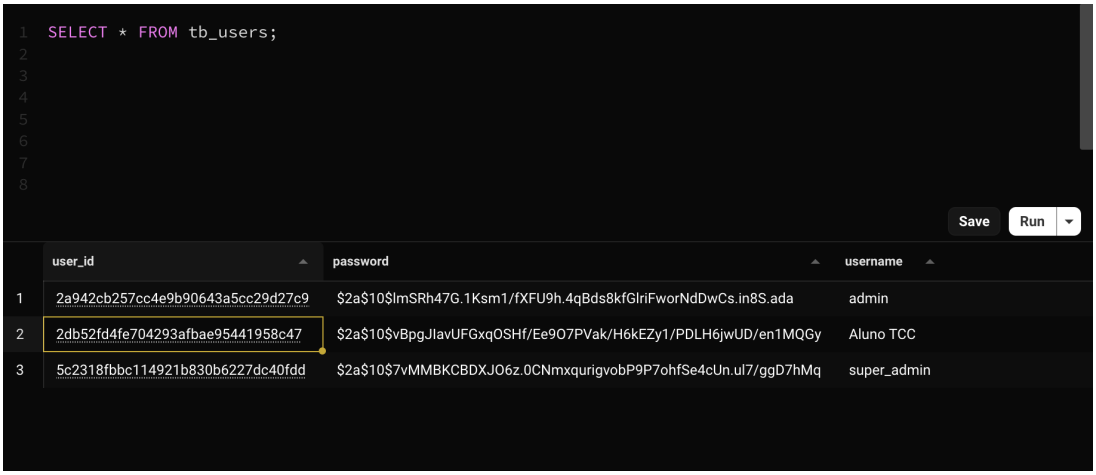


Figura I.2: Aluno registrado persistido no banco de dados.

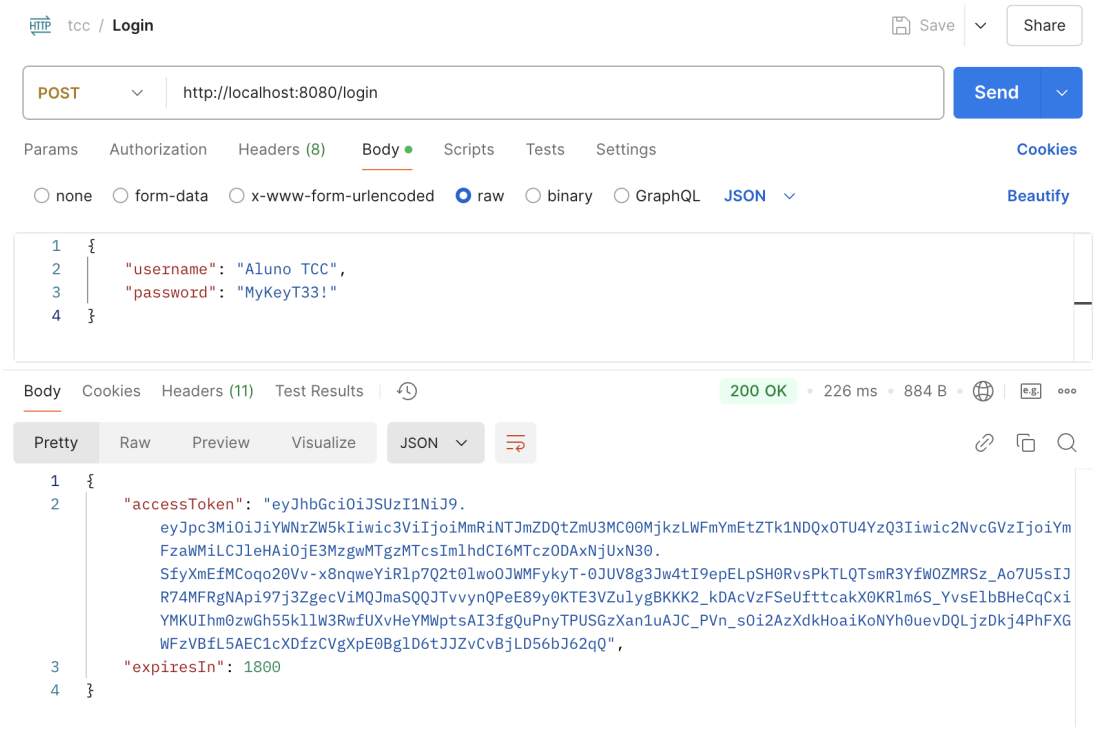


Figura I.3: Solicitação e resposta do *login* do aluno.

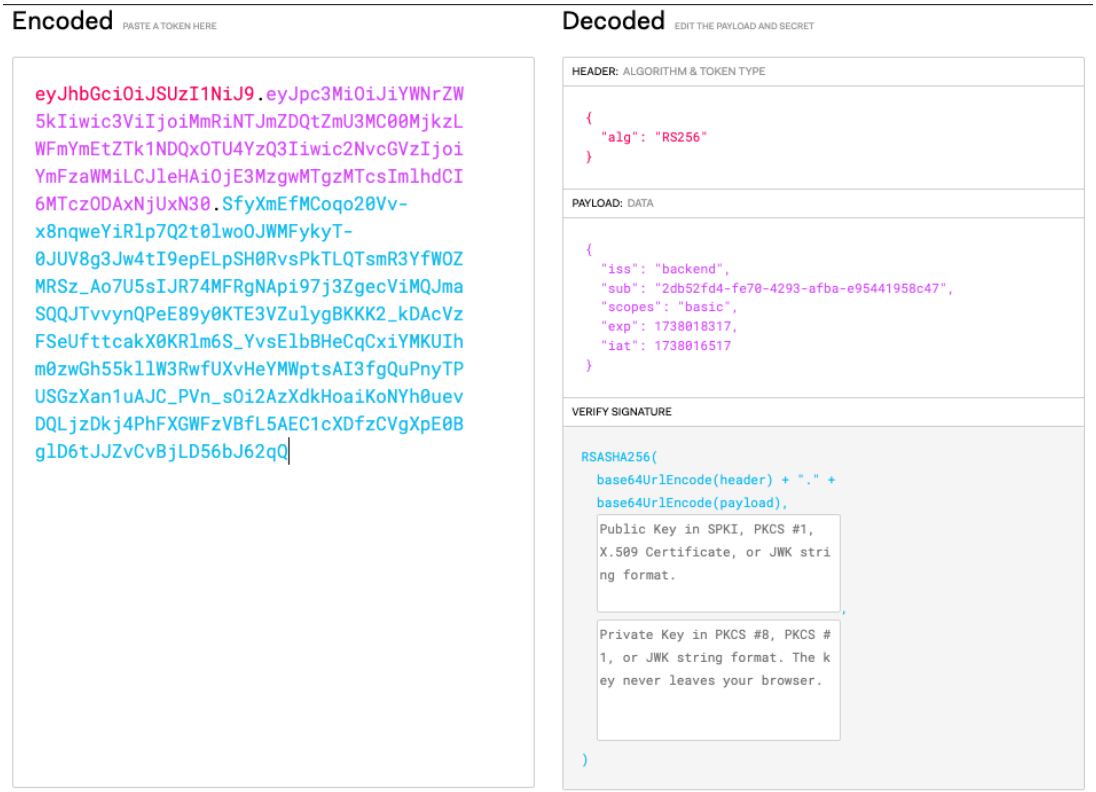


Figura I.4: Conteúdo do *token* do aluno logado.

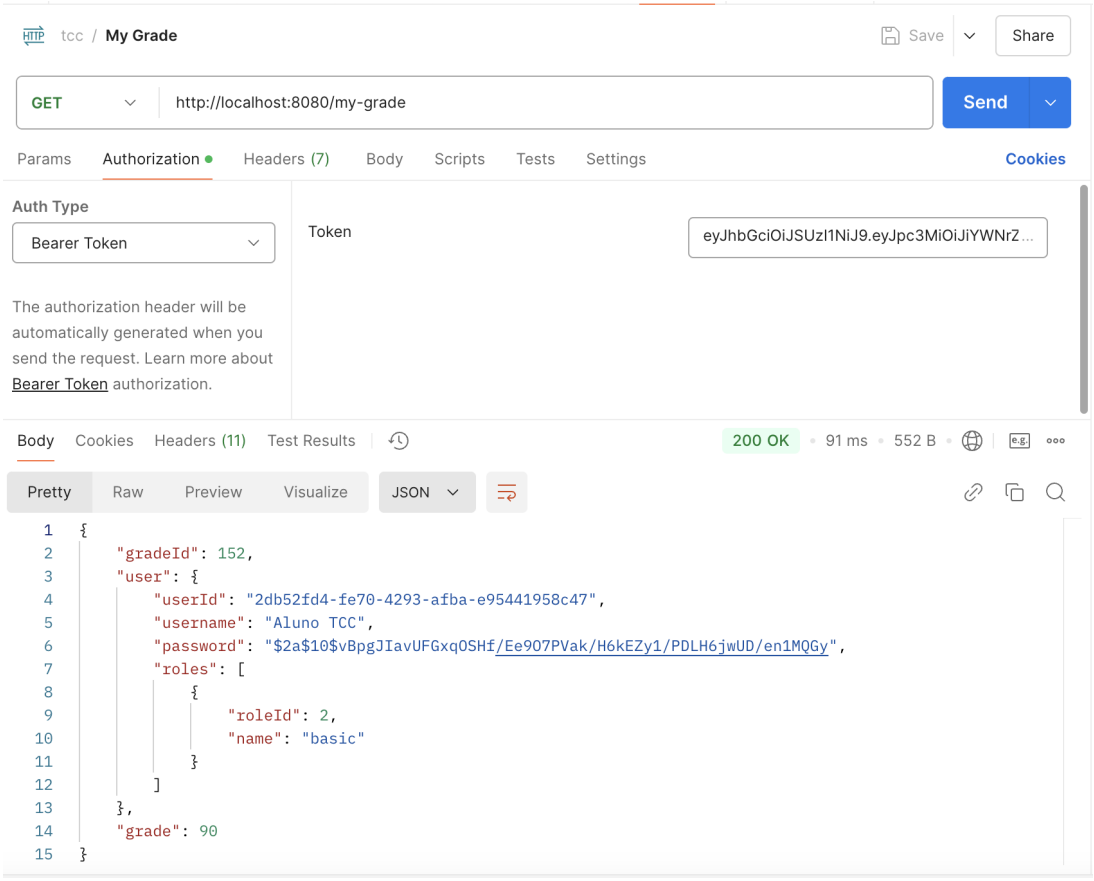


Figura I.5: Requisição para obter a própria nota.

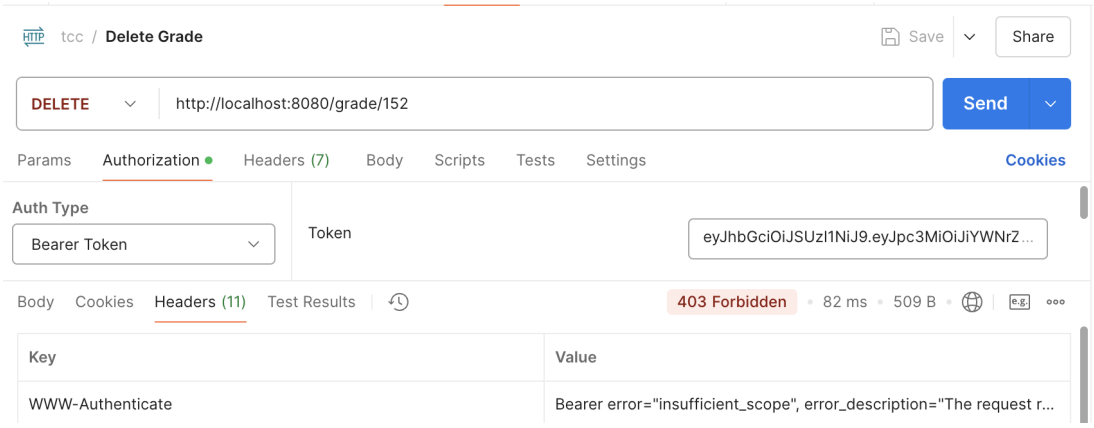


Figura I.6: Proibição de excluir uma nota.

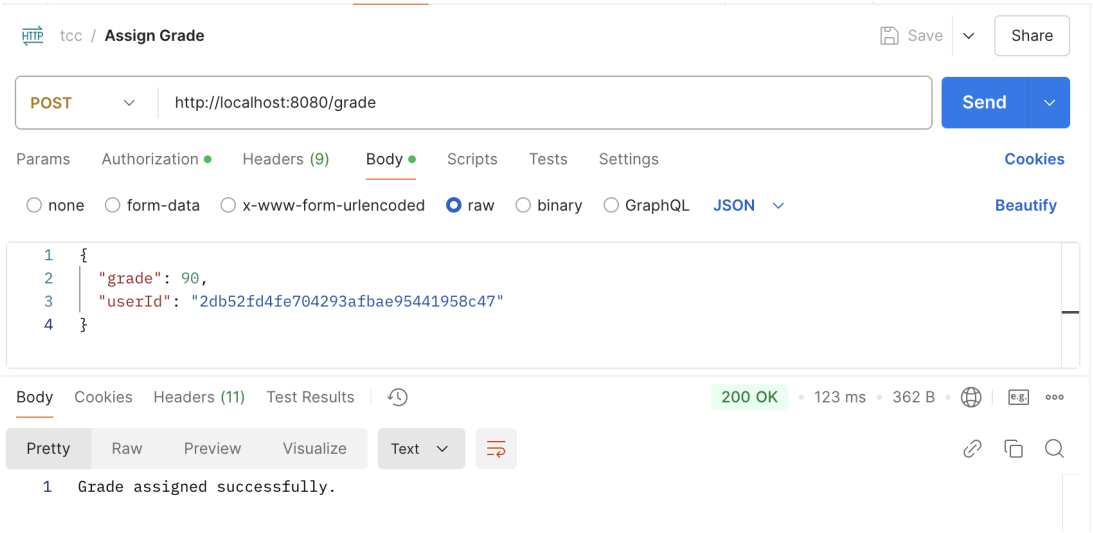


Figura I.7: Lançamento da nota pelo professor.

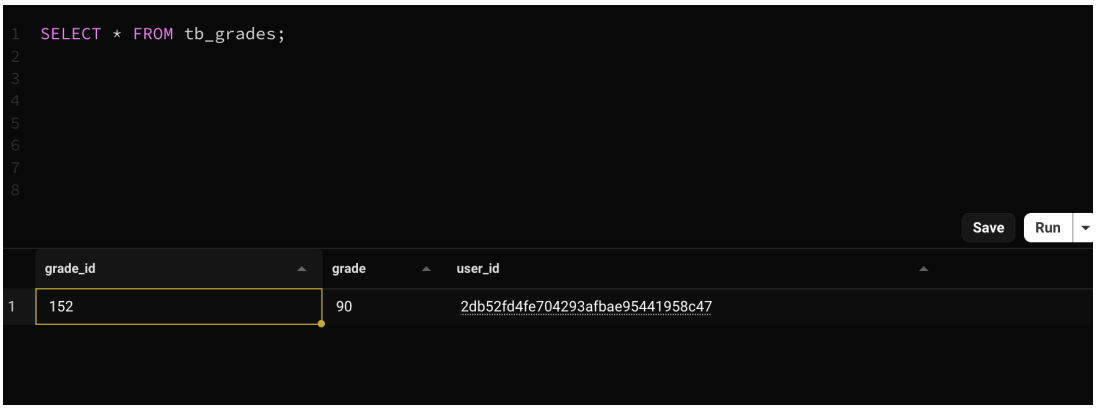


Figura I.8: Persistência da nota no banco de dados.

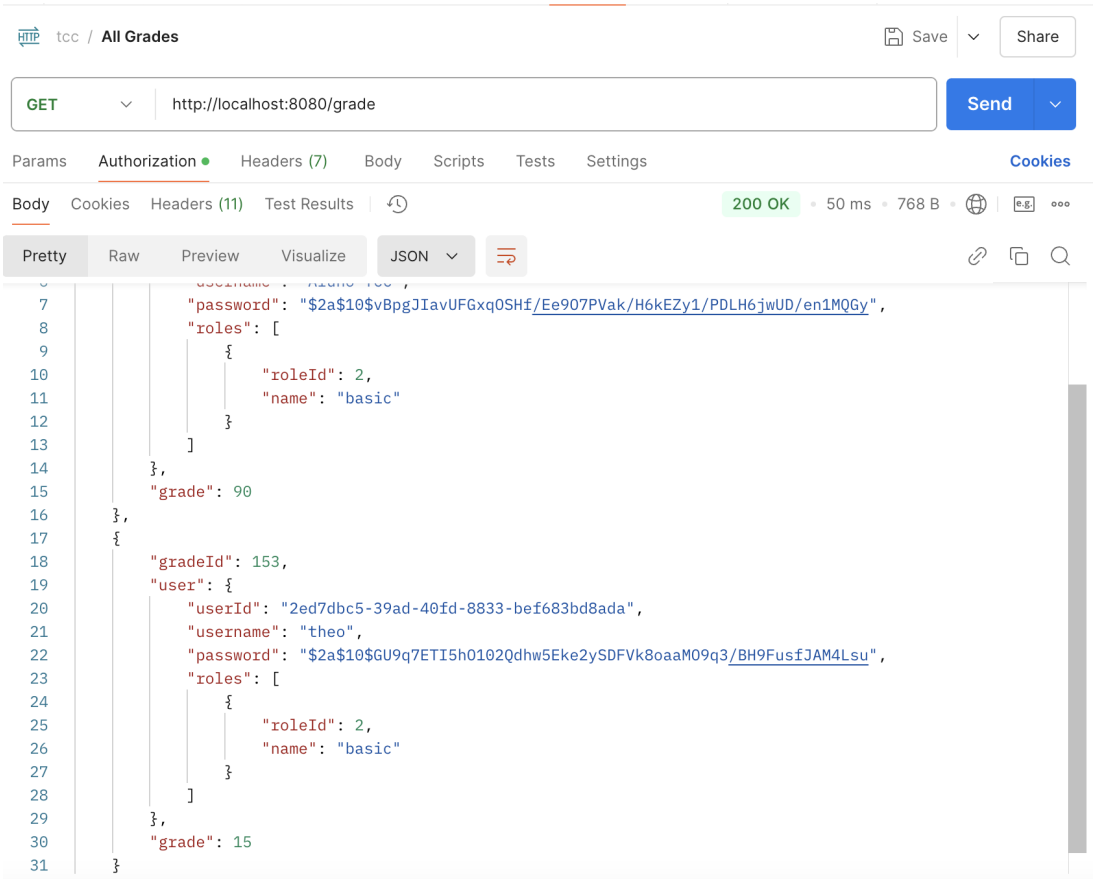


Figura I.9: Solicitação para ver todas as notas.

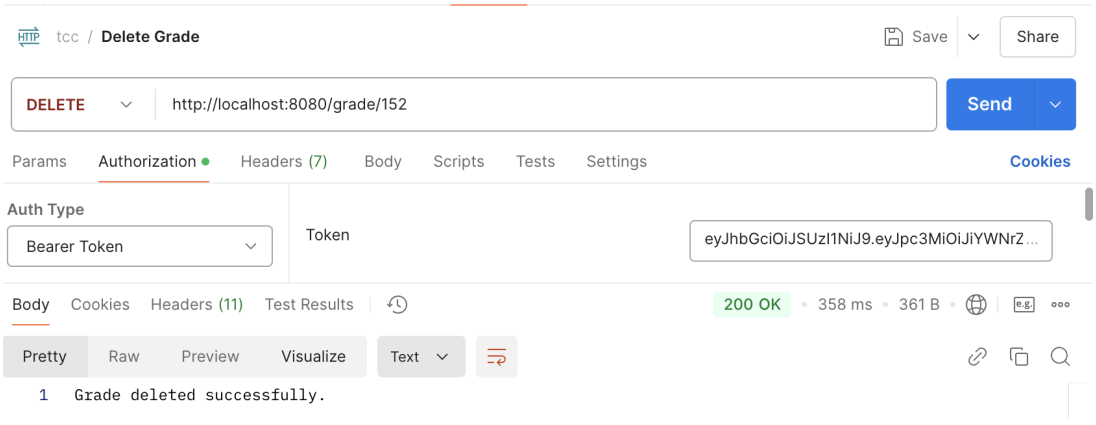


Figura I.10: Exclusão da nota pelo professor.

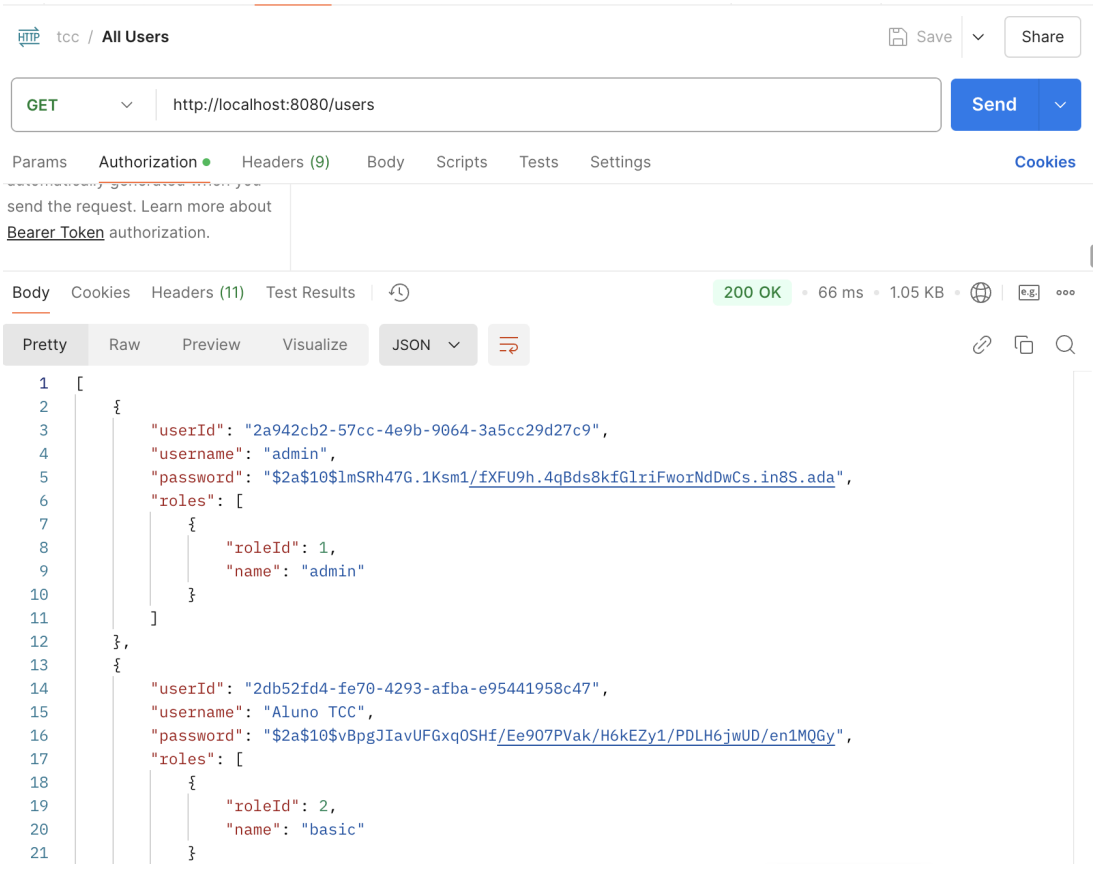


Figura I.11: Listagem de todos os usuários pelo professor.

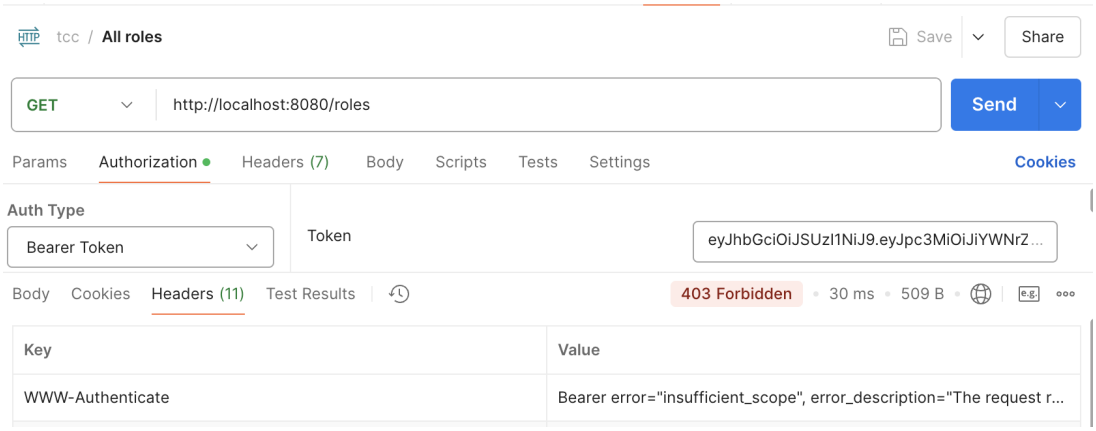


Figura I.12: Professor com acesso negado para visualizar as *roles*.

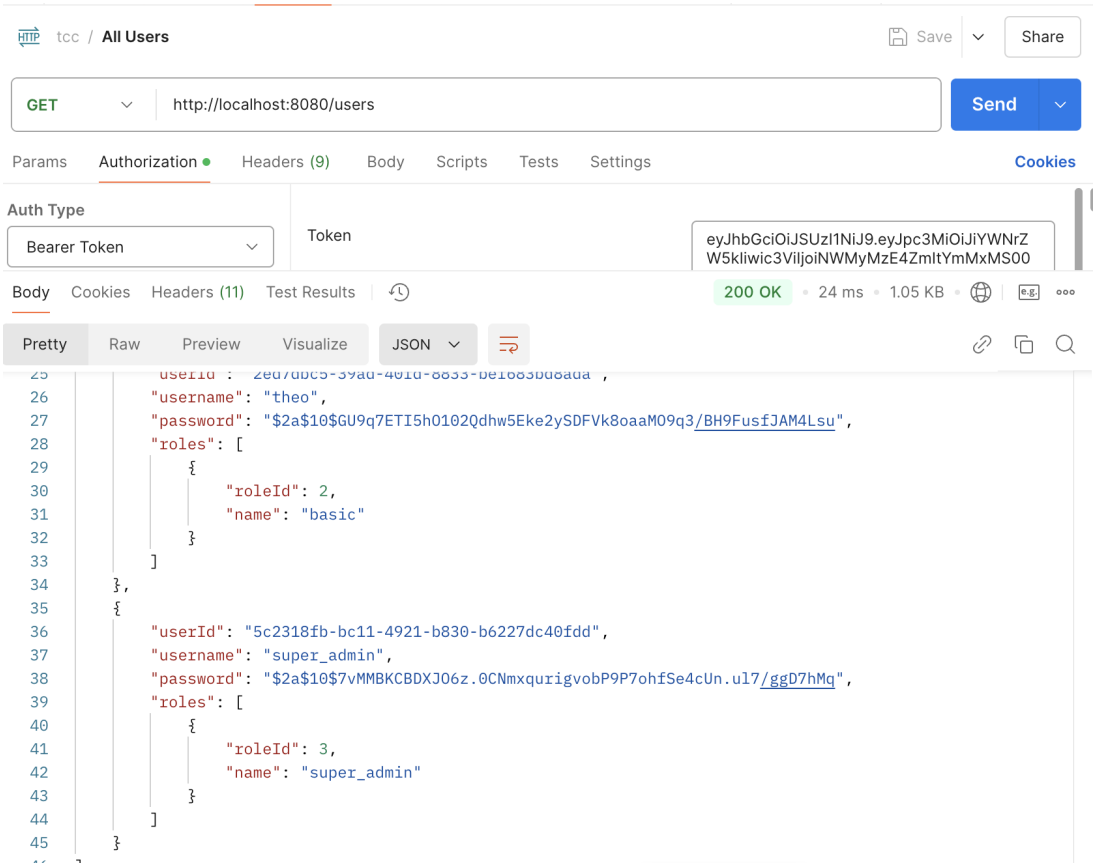


Figura I.13: *Super admin* listando todos os usuários.

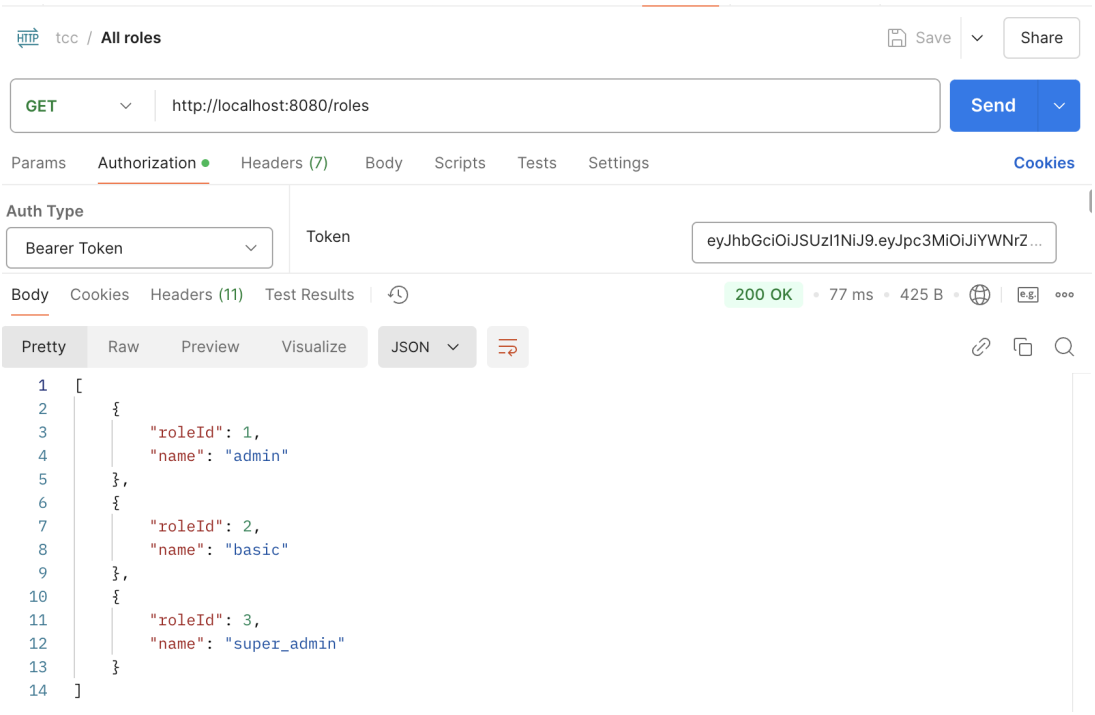


Figura I.14: *Super admin* com privilégio exclusivo de ver todas as roles.

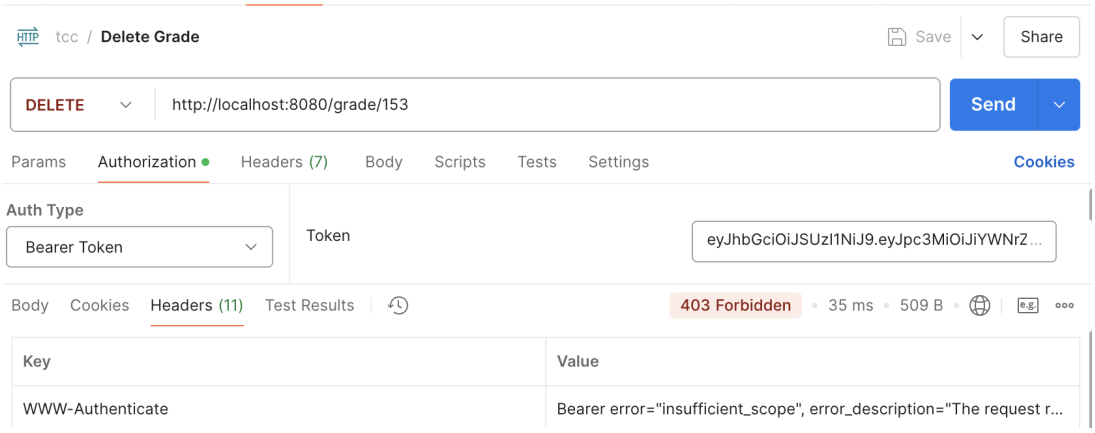


Figura I.15: *Super admin* sem permissão para deletar uma nota.