Desenvolvimento de API REST de autenticação utilizando Spring para arquiteturas de microsserviços

Zaqueu Moura da Silva

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

2023

Zaqueu Moura da Silva

Desenvolvimento de API REST de autenticação utilizando Spring para arquiteturas de microsserviços

Trabalho de Conclusão de Curso (TCC) apresentado ao curso de Engenharia de Computação do Centro de Informática, da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Dr. Alisson Vasconcelos De Brito

Junho de 2023

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

Catalogação na publicação
Seção de Catalogação e Classificação

S586d Silva, Zaqueu Moura da.
Desenvolvimento de API REST de autenticação utilizando Spring para arquiteturas de microsserviços / Zaqueu Moura da Silva. - João Pessoa, 2023.
47 f.

Orientação: Alisson Vasconcelos De Brito.
TCC (Graduação) - UFPB/CI.

1. API de Autenticação. 2. Java. 3. Microsserviços. 4. Spring Security. 5. JWT. I. Brito, Alisson Vasconcelos De. II. Título.

UFPB/CI CDU 004.43

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

RESUMO

Este trabalho tem como principal objetivo o desenvolvimento de uma API REST de autenticação utilizando o Spring para arquiteturas de microsserviços segura e eficiente. A principal motivação para o trabalho é o crescimento das aplicações empresariais que utilizam arquiteturas de microsserviços pela sua flexibilidade, escalabilidade. Com isso, há uma maior procura por profissionais que entendam das tecnologias. É importante ressaltar que a utilização do *framework* Spring boot é bastante viável, prática, segura e muito bem aceita no mercado.

Para isso, utilizaremos o *framework* Spring MVC com a linguagem de programação Java, desenvolvendo as camadas de controladores que serão acessíveis via endpoints. Os modelos serão as entidades de persistência utilizando o Hibernate. Com auxílio das bibliotecas do *Spring Security* para geração e validação de *tokens* JWT a partir de um usuário e senha pré-definidos, salvos no banco de dados MySQL. Posteriormente, esse token será validado a partir de um *gateway* que irá encaminhar a requisição para o endpoint definido na URL de requisição, tornando o sistema escalável e flexível para funcionar com diversas tecnologias. Bastando apenas a comunicação entre os endpoints no formato JSON.

Palavras-chave: API de Autenticação, Java, Microsserviços, Spring Security, JWT.

ABSTRACT

The main objective of this work is to develop a secure and efficient REST API authentication using Spring for microservice architectures. The main motivation for this work is the growth of enterprise applications using microservice architectures for their flexibility, scalability. As a result, there is a greater demand for professionals who understand the technologies used. It is important to note that the use of the Spring boot framework is very viable, practical, secure, and highly accepted in the market.

To achieve this, we will use the Spring MVC framework with the Java programming language, developing controller layers that will be accessible via endpoints. The models will be persistence entities using Hibernate. With the help of Spring Security libraries for generating and validating JWT tokens from a pre-defined username and password saved in the MySQL database. Later, this token will be validated from a gateway that will forward the request to the endpoint defined in the request URL, making the system scalable and flexible to work with various technologies. Just requiring communication between the endpoints in JSON format.

Key-words: API authentication, Java, Microservices, Spring Security, JWT.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

JWT - JSON Web Token

JSON - JavaScript Object Notation

MVC - Model, View, Controller

URL - Uniform Resource Locator

SQL - Structured Query Language

API - Application Programming Interface

IDE - Integrated Development Environment

HTTP - Hypertext Transfer Protocol

REST - Representational State Transfer

JVM - Java Virtual Machine

JPA - Java Persistence API

CSRF - Cross-Site Request Forgery

XSS - Cross-Site Scripting

UUID - Universally Unique Identifier

Listings

Sumário

INTRODUÇÃO

Nos últimos anos, as arquiteturas de microsserviços têm se tornado cada vez mais populares na indústria de desenvolvimento de *software*. Essa abordagem permite que sistemas complexos sejam divididos em partes menores e independentes, o que traz benefícios como maior escalabilidade e flexibilidade, além de facilitar a integração entre diferentes tecnologias (LARRUCEA et al., 2018). No entanto, um dos desafios das arquiteturas de microsserviços é garantir a segurança e a autenticação dos usuários que acessam os diversos serviços da aplicação como relata o (BARABANOV; MAKRUSHIN, 2020).

As APIs Rest estabelecem princípios para criar serviços *web* que utilizam o protocolo HTTP, garantindo que eles sejam escaláveis, modulares, fáceis de entender e implementar (SONI; RANGA, 2019). Essas APIs expõem os recursos e funcionalidades de uma aplicação, permitindo que outras aplicações acessem e interajam com eles de forma padronizada e independente da plataforma. O uso de um padrão bem definido para requisições e respostas, o formato JSON facilita a integração e o consumo dos serviços oferecidos pelas APIs Rest pelos desenvolvedores(SONI; RANGA, 2019).

Em sistemas distribuídos, a segurança e a autenticação são fatores cruciais a serem considerados, juntamente com a separação clara de responsabilidades. No presente trabalho, optou-se por utilizar o ecossistema Spring para implementar a estratégia de autenticação, empregando o Spring Security como principal recurso para assegurar a integridade do sistema (NGUYEN; BAKER, 2019). Através do Spring Security, tornou-se viável implementar diversas funcionalidades de segurança, tais como controle de acesso, autenticação de usuários, gerenciamento de sessões, entre outras. Permitindo que apenas usuários autorizados pudessem acessá-lo (ARMANDO et al., 2014).

A arquitetura monolítica possui certamente algumas vantagens, como a simplicidade, a facilidade de desenvolvimento e manutenção, o custo mais baixo e a facilidade de implantação em produção, já que todas as partes da aplicação funcionam em um único pacote.

No entanto, apesar das vantagens, existem também muitas desvantagens, como a baixa flexibilidade, já que a aplicação como um todo funciona em um único pacote, a falta de resiliência, já que se uma parte do serviço falhar, isso afeta todo o resto, e as limitações tecnológicas, já que se trata de um sistema único para toda a aplicação, restringindo o uso de diversas tecnologias (LUCIO et al., 2017).

Com o passar dos anos e o avanço das tecnologias, novas arquiteturas surgiram e vêm se aprimorando, como a de microsserviços (BUSHONG et al., 2021). Essa arquitetura consegue contornar a maioria das desvantagens dos sistemas monolíticos, pois permite que cada serviço seja desenvolvido de forma individual, testado e implantado separadamente,

além de permitir a escalabilidade apenas dos serviços que necessitam de mais recursos, em vez de todo o sistema (LARRUCEA et al., 2018), o uso de diversas tecnologias para o desenvolvimento, permitindo que vários times possam trabalhar em conjunto, cada um com sua própria linguagem e *frameworks*.

A organização deste trabalho segue a seguinte estrutura: O primeiro capítulo aborda o escopo do projeto, com a descrição do problema a ser estudado e a definição dos objetivos.

No segundo capítulo, são apresentados os recursos utilizados, incluindo documentação, ambientes de desenvolvimento, tecnologias, protocolos, algoritmos, *frameworks*, bem como detalhes sobre os algoritmos de criptografia presentes nas bibliotecas e sobre a arquitetura de sistemas escaláveis baseados em microsserviços, que é o foco central desta proposta.

O terceiro capítulo tem como propósito explicar a metodologia adotada ao longo do trabalho, desde o cadastro de usuários até a estratégia de autenticação utilizando o ecossistema Spring, além de detalhes sobre o modelo de banco de dados empregado, incluindo um banco de dados dinâmico para cache e um gateway para gerenciar e direcionar as requisições.

O quarto e último capítulo discute os resultados obtidos no trabalho, as dificuldades encontradas e as conclusões, apresentando ainda sugestões para futuras pesquisas.

1 ESCOPO DO TRABALHO

1.1 Definição do Problema

Para se ter acesso a algo ou alguma coisa, é necessário autenticar-se, podendo ser por meio de chaves, senhas, entre outros (ALOUL et al., 2009). A segurança é um assunto crucial no desenvolvimento de sistemas web e deve ser uma prioridade em todas as etapas do desenvolvimento. Com o número elevado de pessoas acessando conteúdos, é comum que esses sistemas restrinjam o acesso de usuários ou processos a recursos específicos. No entanto, implementar esses controles pode ser uma tarefa desafiadora (BIANCHI; FONSECA,).

Algumas das dificuldades enfrentadas podem ser a escolha das tecnologias para a implementação, pois cada uma possui sua especificidade; a forma como armazenamos as credenciais dos usuários é importante para garantir a segurança desses dados e garantir que o acesso a essas informações seja somente por pessoal autorizado; é necessário garantir que a aplicação seja capaz de integrar-se com outros sistemas; o gerenciamento de sessões e tokens é importante para garantir que sejam gerados de forma segura e que expirem automaticamente após um período de tempo determinado(BALAJ, 2017).

A necessidade de uma API de autenticação, utilizando *frameworks* bem difundidos e de fácil integração com diversas tecnologias (NGUYEN; BAKER, 2019). O presente trabalho apresenta uma solução de desenvolvimento que utiliza diversas tecnologias.

1.2 Objetivo geral

O objetivo deste trabalho é desenvolver uma API REST de autenticação, utilizando o ecossistema Spring como base tecnológica.

1.3 Objetivos específicos

- Criação de um banco de dados SQL para armazenamento de credenciais criptografadas.
- Utilização do framework Spring MVC juntamente com práticas seguras de desenvolvimento para a criação das APIs.
- Construção de uma API REST para o cadastro de usuários, processando requisições HTTP.
- Construção de uma API REST de autenticação de usuários utilizando o framework Spring Security.

- Utilização do framework Redis como banco de dados de cache
- Construção de um Gateway para centralizar as requisições via API, validando a autenticação e roteando para o endpoint desejado.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentada, uma revisão da literatura e temas abordados neste trabalho.

2.1 Autenticação

A autenticação é um dos pilares da segurança da informação e é amplamente utilizada em sistemas de computação para garantir que somente usuários autenticados tenham acesso a recursos e dados. A autenticação pode ser implementada através de senhas, *tokens* ou certificados digitais, que são mecanismos para confirmar a identidade do usuário. Esse processo é essencial para garantir a segurança de informações confidenciais e proteger contra ameaças de invasão e roubo de dados. É um processo de validação da identidade do usuário que acessa um sistema ou recurso, com o objetivo de garantir que a pessoa ou entidade que está acessando o sistema seja realmente quem afirma ser (FERRAG et al., 2017). (SANTOS, 2015) destaca que a autenticação é um processo crítico que deve ser implementado adequadamente para proteger dados e sistemas contra ataques mal-intencionados. Existem diversos métodos de autenticação, cada um com suas vantagens e limitações. A autenticação baseada em senha é um dos métodos mais comuns e simples, porém, é suscetível a ataques de força bruta e *phishing*.

2.2 API Rest

API REST é uma interface de programação que permitem que diferentes sistemas se comuniquem de maneira padronizada e independente de plataforma, linguagem ou tecnologia utilizada.

Essa interface é baseada em um conjunto de princípios que definem como os recursos de uma aplicação devem ser expostos através de URLs e como esses recursos podem ser manipulados por métodos HTTP, existem alguns princípios básicos para a construção de uma API REST:

- O primeiro princípio é a separação entre cliente e servidor, o que permite que cada parte seja desenvolvida de forma independente e sem que o cliente precise conhecer a implementação do servidor. Além disso, a comunicação entre o cliente e o servidor é realizada sem armazenar informações entre as requisições.
- O segundo princípio importante é que os recursos devem ser identificáveis e armazenáveis em cache, permitindo que o cliente possa reutilizá-los em requisições

futuras e diminuindo o tráfego de rede. A arquitetura REST deve permitir a inclusão de camadas intermediárias entre cliente e servidor, como *gateways*, *caches* e *load balancers*, sem afetar a comunicação entre as partes.

- O terceiro princípio é que a interface deve ser uniforme e consistente, definindo um conjunto de operações básicas que podem ser realizadas nos recursos através de métodos HTTP alguns tipos são GET, POST, PATCH, PUT, DELETE que em sequência servem para consultar, publicar, realizar uma pequena alteração, realizar uma alteração completa no modelo, e deletar. Essas operações são identificadas por meio de URLs bem definidas, que representam os recursos de forma hierárquica.

Em resumo, a API REST é uma das tecnologias mais utilizadas atualmente para integrar sistemas e criar aplicações distribuídas e escaláveis na *web* (BALACHANDAR, 2017). Sua implementação permite que diferentes sistemas se comuniquem de forma padronizada, o que torna a integração mais fácil e eficiente, além de oferecer diversas vantagens técnicas que a tornam uma escolha popular para projetos de desenvolvimento de *software*.

2.3 Arquitetura de microsserviços

A arquitetura de microsserviços é uma abordagem de arquitetura de *software* que divide um aplicativo em muitos pequenos serviços independentes que se comunicam entre si por meio de APIs. Cada serviço é responsável por uma única função de negócio e pode ser implantado, gerenciado e escalado independentemente dos outros serviços. Essa abordagem tem sido cada vez mais adotada por empresas que buscam construir aplicativos escaláveis, flexíveis e resilientes.

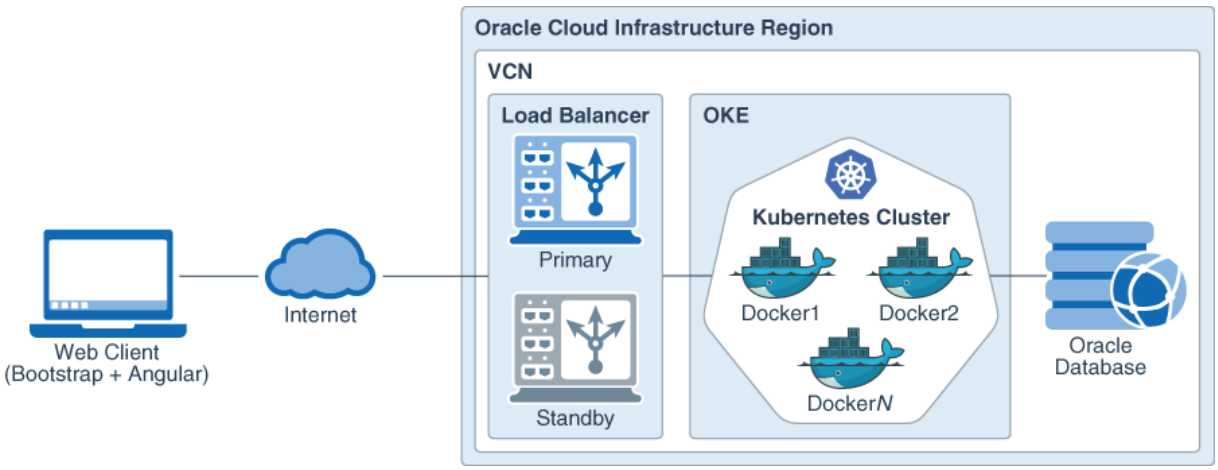
Uma das principais vantagens oferecidas pela arquitetura de microsserviços é a escalabilidade horizontal, possibilitando que cada serviço seja escalonado de maneira independente, de acordo com a demanda. Ademais, essa abordagem favorece a agilidade no desenvolvimento de software, permitindo que novos serviços sejam acrescentados ou removidos sem afetar os demais. (BUSHONG et al., 2021).

Para implementar uma arquitetura de microsserviços, é essencial utilizar ferramentas e tecnologias específicas, como contêineres *Docker*, que oferecem a possibilidade de encapsular um serviço inteiro em um ambiente isolado, facilitando a implantação e o gerenciamento do mesmo. Além disso, podem ser escalonados com facilidade, tornando mais simples o gerenciamento de serviços com alta demanda.

Utilizar gerenciadores de contêineres, como o *Kubernetes* é fundamental para a implantação, fornecendo recursos avançados de gerenciamento de contêineres (LARRUCEA et al., 2018), como escalonamento automático, balanceamento de carga, gerenciamento

de armazenamento e gerenciamento de rede, assim como descrito na figura 1 onde podemos observar a presença do balanceador de cargas na entrada das requisições do cliente, administrando o fluxo entre os *containers*.

Figura 1: Arquitetura da Aplicação de Microsserviços



Fonte: https://docs.oracle.com/pt-br/solutions/build-rest-java-application-with-oke/

APIs REST são outro elemento crucial para a comunicação entre serviços, pois fornecem uma maneira padronizada e flexível de expor serviços e dados entre diferentes sistemas e dispositivos, permitindo que serviços diferentes se comuniquem de forma simples e eficiente.

2.4 Java

Java é uma linguagem de programação orientada a objetos criada em 1991 por James Gosling e sua equipe na Sun Microsystems, e posteriormente adquirida pela Oracle Corporation. Uma das principais vantagens do Java é a sua portabilidade, pois programas escritos em Java podem ser executados em qualquer plataforma que possua uma máquina virtual Java (JVM). Além disso, a linguagem foi projetada para ser segura, com recursos de controle de acesso e execução em sandbox, que evitam o acesso não autorizado aos recursos do sistema (KALIBERA et al., 2009).

O Java é amplamente utilizado no desenvolvimento de aplicativos empresariais, como sistemas de gerenciamento de conteúdo, bancos, comércio eletrônico e gerenciamento de recursos corporativos, devido à sua compatibilidade com bibliotecas e *frameworks*, como Spring e Hibernate. Em resumo, o Java é uma linguagem versátil, segura, escalável e de alto desempenho, e é uma das mais populares e confiáveis do mercado atualmente.

2.5 Spring

O Spring é um ecossistema de desenvolvimento que reúne diversos *frameworks* a fim de facilitar a criação de aplicações complexas, oferece uma abordagem modular para o desenvolvimento de aplicativos, permitindo que você escolha e use apenas os componentes necessários. O Spring oferece uma ampla gama de recursos, incluindo injeção de dependência, acesso a banco de dados, MVC, segurança e testes. O Spring Boot é um subprojeto popular do Spring que fornece um ambiente de execução completo para aplicativos Java, Kotlin.

O Spring é altamente configurável e pode ser usado em uma ampla variedade de cenários de desenvolvimento, desde aplicativos empresariais complexos até aplicativos da web simples (WALLS, 2022). Criado em 2003 por Rod Johnson, o Spring evoluiu e se expandiu para incluir uma série de subprojetos que tem por finalidade garantir um desenvolvimento rápido e seguro.

2.5.1 Spring boot

O Spring Boot é um framework Java que possibilita maior produtividade e menos configuração no desenvolvimento de aplicativos. Sua alta modularidade o torna uma escolha popular para arquiteturas de microsserviços e sua fácil incorporação em outros aplicativos ou servidores de aplicativos é um diferencial.

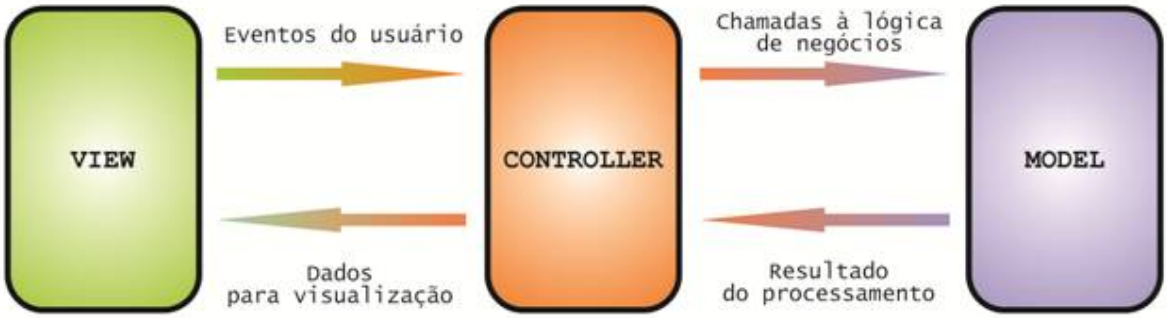
O *framework* simplifica a configuração através do arquivo **application.properties** ou **application.yml** e disponibiliza diversas ferramentas de automação, facilitando a criação, implantação e gerenciamento do aplicativo. O *framework* também oferece recursos de monitoramento e uma estrutura de teste integrada (REDDY, 2017), tornando mais simples para os desenvolvedores escreverem testes automatizados.

Suporta vários bancos de dados, tanto relacionais quanto NoSQL. Com todas essas características técnicas, o Spring Boot é uma escolha popular entre os desenvolvedores que buscam desenvolver aplicativos de forma mais rápida e eficiente.

2.5.2 Spring MVC

O Spring MVC é um *framework* do ecossistema Spring que permite o desenvolvimento de aplicações web de forma robusta e flexível, isolando as regras de lógica e negócio da interface com o usuário. Segundo (MAK, 2008) a arquitetura MVC divide a aplicação em três camadas:

Figura 2: Diagrama de fluxo MVC



Fonte: https://www.devmedia.com.br/padrao-mvc-java-magazine/21995

- *Model*, Responsável por executar as regras de negócio da aplicação, processando as informações e acessando e persistindo as entidades no banco de dados.
- *View*, É a interface que interage com o usuário, sem conter regras de negócio.
- *Controller*, Serve como intermediador entre as duas camadas descritas acima, model e view, com o propósito de garantir uma comunicação mais organizada e manter as duas camadas independentes, como ilustrado na figura 2."

2.5.3 Spring Security

O Spring Security é um *framework* de segurança para aplicações Java, projetado para oferecer proteção contra ameaças comuns de segurança, mostrando-se eficaz em alguns cenários, segundo (NGUYEN; BAKER, 2019) contra ataques de CSRF, XSS e de força bruta.

Para contextualizar o tipo de ataque CSRF que posteriormente iremos realizar um teste habilitando o filtro contra esse tipo de ataque e desabilitar para ver os resultados, consiste em que um *software* malicioso ou uma pessoa gera uma página *web* falsa. Esse código malicioso faz com que o navegador do usuário execute uma solicitação HTTP legítima para o site alvo sem o conhecimento ou consentimento do usuário. Como a solicitação é enviada a partir do navegador do usuário autenticado, o site alvo a trata como uma solicitação legítima e executa a ação desejada pelo invasor, na tentativa de contornar esses ataques é possível a ativação ou desativação como descritos nos Listing 1 e Listing 2.

Listing 1: Exemplo de configuração do Spring Security com proteção contra ataques CSRF

```
1       @Override
2       protected void configure(HttpSecurity http) throws Exception {
3       http.csrf()
4       .csrfTokenRepository(CookieCsrfTokenRepository
5           .withHttpOnlyFalse())
6           .csrfTokenParameterName("_csrf");
7       }
```

Listing 2: Exemplo de configuração do Spring Security sem proteção contra ataques CSRF

```
1       @Override
2       protected void configure(HttpSecurity http) throws Exception {
3           http.csrf().disable().authorizeRequests()
4                   .anyRequest().permitAll();
5       }
```

O Spring security utiliza filtros do *Servlet* para proteger a aplicação, adicionando uma série de filtros de segurança ao pipeline de filtro padrão.

Para atender às necessidades específicas da aplicação, o Spring Security é altamente configurável e integra-se facilmente com outros *frameworks* do Spring, como o MVC e o Spring Boot. Amplamente utilizado na indústria para proteger aplicações web críticas, é considerado uma das melhores opções para segurança em aplicações Java.

2.6 Banco de dados SQL

SQL é uma linguagem padrão que permite gerenciar bancos de dados relacionais de forma eficiente. Com ela, é possível criar, modificar e consultar tabelas, bem como definir a estrutura dos dados a serem armazenados e realizar operações como busca, filtragem, ordenação e cálculos matemáticos e estatísticos (GROFF et al., 2002).

É uma linguagem padronizada e compatível com a maioria dos sistemas de gerenciamento de bancos de dados relacionais, como MySQL, PostgreSQL, Oracle, Microsoft SQL , sendo amplamente utilizada em aplicações empresariais para gerenciamento de dados, geração de relatórios e análise de informações.

2.7 Redis

O Redis é um sistema de cache escalável e de alto desempenho, ideal para aplicativos que precisam de acesso rápido aos dados. É um banco de dados na memória que armazena vários tipos de estruturas de dados, como *strings*, *hashes*, listas, conjuntos e

muito mais. É amplamente usado em aplicativos web para armazenar dados temporários, como sessões do usuário e informações do carrinho de compras. Além disso, oferece persistência, operações atômicas e suporte a mensagens, tornando-o uma excelente opção para implementação de mensagens em tempo real e filas de tarefas assíncronas.

Pode ser executado em vários nós e é usado em aplicativos de alta demanda, jogos em tempo real, análise de dados em tempo real e outros aplicativos que exigem alto desempenho e gerenciamento de dados eficiente. Em resumo, o Redis é uma solução robusta e confiável que atende às necessidades de aplicativos que exigem alta velocidade, baixa latência e gerenciamento de dados eficiente.

2.8 Gateway

O uso de um *gateway* é fundamental para garantir a segurança, escalabilidade e eficiência de uma arquitetura de microsserviços. Este componente de acordo com (ZHAO et al., 2018) é capaz de implementar diversas funcionalidades, roteamento de requisições, balanceamento de carga, monitoramento de performance, cache de dados, entre outras que contribuem para melhorar a comunicação entre os microsserviços e seus clientes.

Um gateway também pode ser usado para simplificar a complexidade de uma arquitetura de microsserviços, fornecendo uma única API coerente para usuários externos, independentemente do número de microsserviços que compõem o sistema. Essa abordagem torna mais fácil para os desenvolvedores criar, testar e manter seus aplicativos (ZHAO et al., 2018).

Em resumo, um gateway é parte de uma arquitetura de microsserviços que fornece uma interface consistente para usuários externos e permite um controle da comunicação entre microsserviços.

3 METODOLOGIA

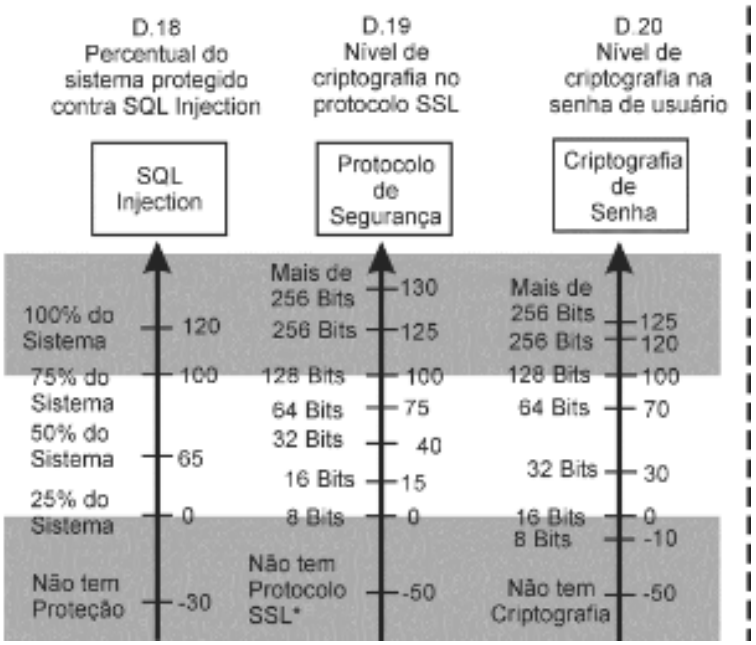
Com o passar dos anos, a internet se tornou uma parte indispensável da nossa vida cotidiana, facilitando a comunicação, a realização de negócios, a educação, entre outras atividades. Com isso, também ocorreu um aumento massivo do uso de dados sensíveis, como informações bancárias, dados de saúde, informações pessoais e outras informações confidenciais.

No entanto, o uso cada vez mais frequente desses dados também aumentou a ameaça de ataques cibernéticos, que podem resultar em roubo de identidade, fraude financeira, perda de privacidade e outros danos significativos. Por isso a grande importância de preservar esses dados fornecendo acesso só a quem realmente pode ter, por meio de autenticação.

Além disso, as organizações também desempenham um papel fundamental na proteção dos dados sensíveis de seus clientes, implementando medidas de segurança adequadas em suas plataformas e redes, como criptografia, autenticação de usuários e monitoramento constante de atividades suspeitas.

A Figura 3 refere-se a uma avaliação sobre segurança de *software* e as melhores práticas para garantir a integridade dos dados apresentando alguns níveis de cobertura do sistema e a utilização de níveis de segurança do tipo SSL, criptografia na senha do usuário o qual serve para justificar a prática de criptografar a senha na aplicação desenvolvida.

Figura 3: Identificação das Alternativas para determinar as taxas de substituição



Fonte: (CHAVES et al., 2013)

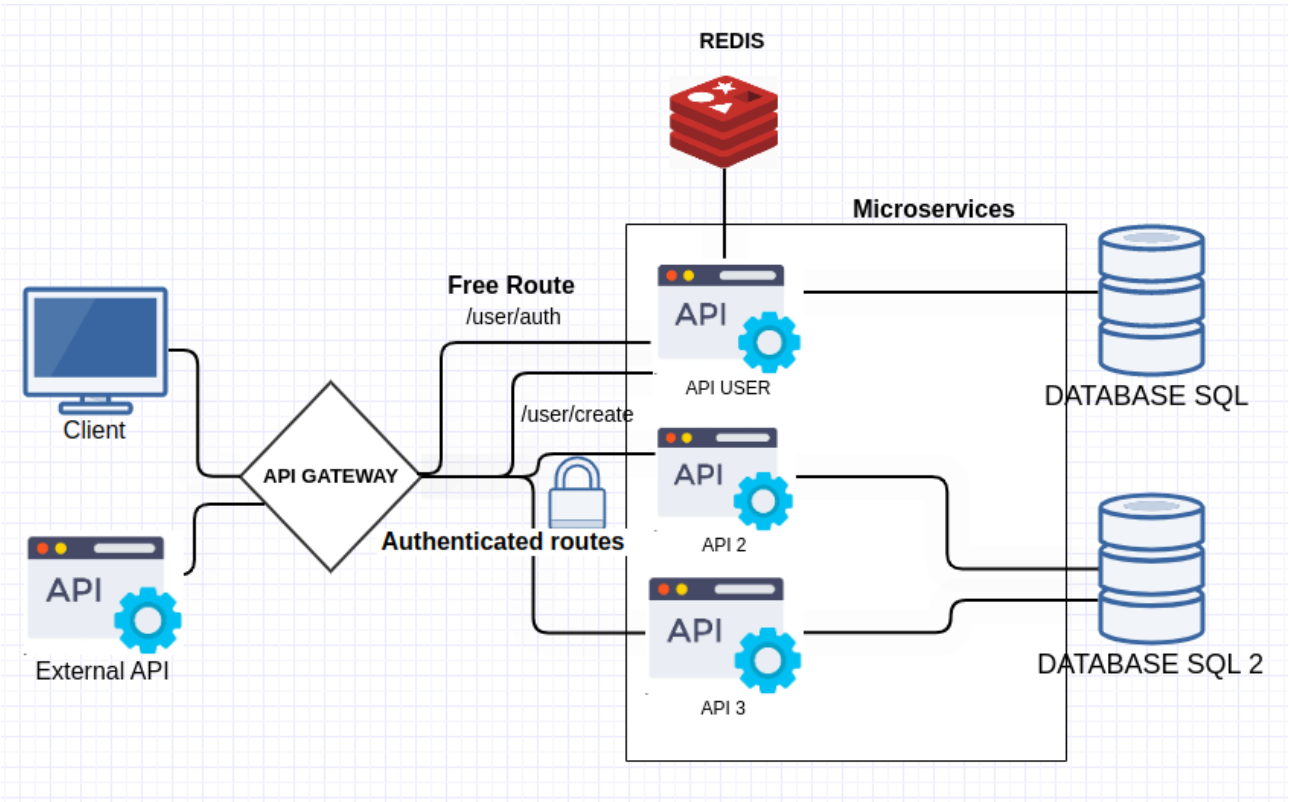
3.1 Arquitetura do sistema

A arquitetura consiste em um *gateway* que é o centralizador, todas as requisições de clientes ou API's Rest terão que passar por ele, ele fica responsável por fazer o roteamento dessas requisições para os microsserviços correspondentes e os mesmos podem trocar mensagens entre si, por meio de filas ou de seus *endpoints*.

Posteriormente temos uma API Rest de autenticação chamada de *user* com a qual o *gateway* vai se comunicar para realizar algumas funções. Ela é responsável por criar, autenticar, e consultar os usuários que possuem permissão. Irei descrever o processo mais detalhadamente.

Utiliza o uso de um banco de dados relacional MySQL para guardar as informações dos usuários tais como senhas e identificadores. Uma vez o usuário com acesso ao JWT que receberá ao realizar a autenticação ele conseguirá consumir todas as API's da arquitetura que forem publicadas. Dessa forma garantindo mais segurança e integridade das informações dos usuários. A API utiliza de uma configuração de cache para respostas mais rápidas e diminuição do acesso ao banco de dados a figura 4 ilustra a arquitetura no geral.

Figura 4: Arquitetura do sistema desenvolvido



Fonte: Autoria Própria

No desenvolvimento deste projeto, utilizou-se a linguagem de programação Java versão 17, a IDE *IntelliJ Ultimate*, o *framework* Spring e as dependências do projeto gerenciadas pelo *Maven*. Desta forma, o gerenciador de dependências é responsável pelo *download* e inclusão das dependências no projeto, o que torna o desenvolvimento mais ágil.

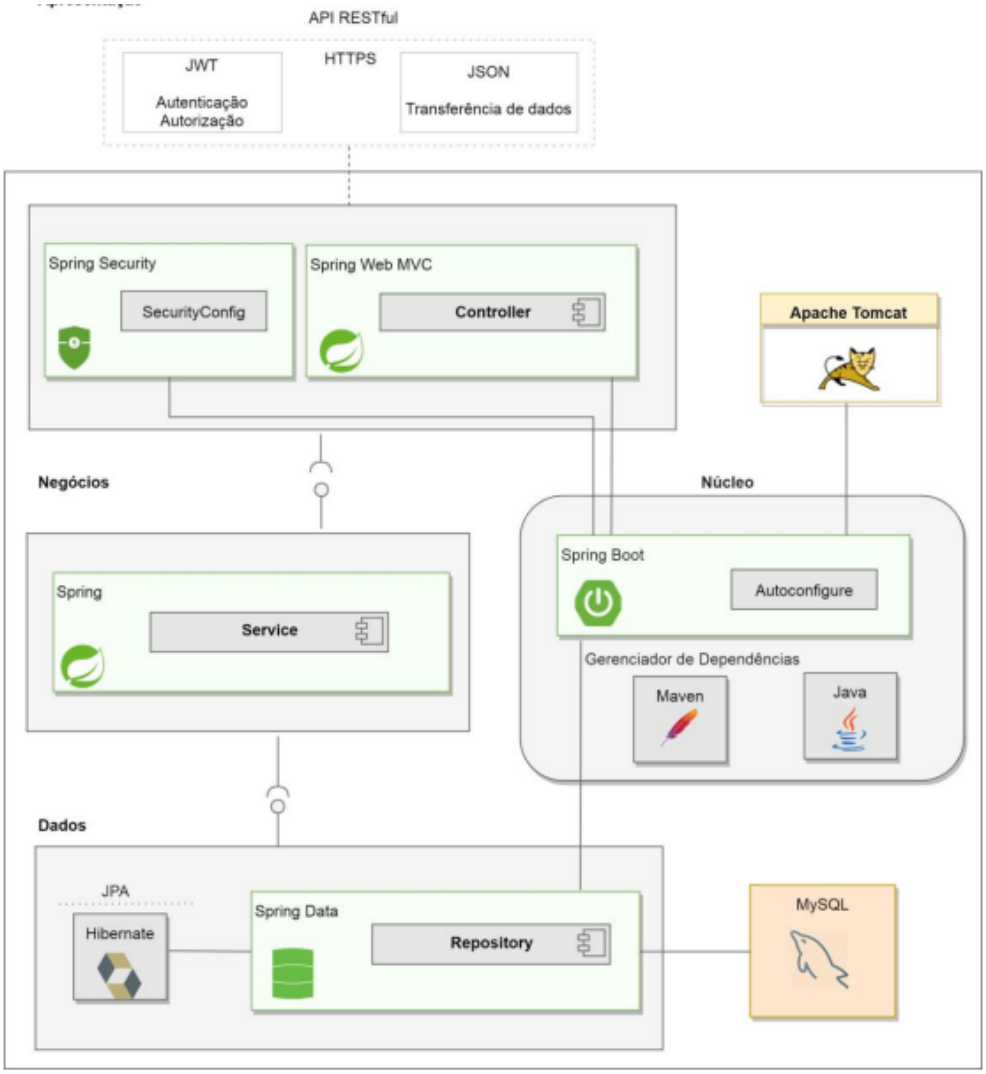
3.1.1 Bibliotecas Utilizadas

O Spring boot é um *framework* do Spring desenvolvido para facilitar a configuração de aplicações no *back-end*, fazendo uso de anotações que reduzem bastante a complexidade do desenvolvimento, precisando muitas das vezes poucas linhas para estar tudo pronto para a publicação da API.

Spring Data é uma coleção de projetos baseados no Spring *Framework* para manipular dados de várias maneiras, incluindo bancos de dados relacionais como MySQL e PostgreSQL, bem como bancos de dados NoSQL como MongoDB e bancos dinâmicos de rápido acesso como o Redis. Um dos projetos Spring Data mais populares é o Spring Data JPA. Além disso, a estrutura oferece métodos de acesso a dados sendo necessário utilizar apenas algumas anotações, tornando o processo de aplicação da arquitetura de banco de dados desejada bem mais rápido.

Utilizamos o Spring Gateway que é um *framework* que nos possibilita com algumas bibliotecas realizarmos o roteamento de requisições e também filtrar de forma segura solicitações indesejadas que podem ser ataques ao nosso sistema. Dessa forma conseguimos centralizar em uma única API as requisições e distribuir para os microsserviços.

Figura 5: Arquitetura Spring



Fonte: (ARAÚJO, 2018)

3.2 Construção das APIS

Para implementar essa arquitetura, é importante criar APIs bem definidas e com documentação clara, para que cada serviço possa ser acessado de forma consistente por outros microsserviços ou clientes externos. Uma boa prática é seguir padrões de nomenclatura e versionamento de APIs, garantindo a manutenção e evolução dos serviços.

Além disso, é importante garantir a segurança das APIs, utilizando autenticação e autorização para controlar o acesso aos serviços. Segundo (JONES et al., 2015) a utilização de tokens de acesso, como o JWT, pode ajudar a implementar essa segurança de forma eficiente.

Em resumo, a construção de APIs para uma arquitetura de microsserviços requer

uma abordagem cuidadosa e bem planejada, com foco na definição clara das interfaces de comunicação, segurança, desempenho e testes. Com a adoção dessa estratégia, é possível criar sistemas mais escaláveis, flexíveis e fáceis de manter, permitindo a evolução contínua da aplicação de forma ágil e eficiente.

3.3 API Gateway

Neste capítulo iremos abordar a construção da API gateway com suas camadas e estratégias de roteamento.

3.3.1 Configurações da API

Para a configuração do projeto Spring Boot, foi utilizado um arquivo chamado **application.yml**, localizado no diretório **"src/main/resources"**, a fim de definir diversas configurações que permitem a representação de dados em uma estrutura hierárquica. Nele configuramos o servidor e e as caracteristicas da aplicação Spring, um exemplo é o Listing 3.

O arquivo **pom.xml** localizado na raíz do projeto é muito importante em projetos que utilizam o gerenciador de dependencias "Maven"é nele que vamos inserir as dependências nas quais ele automaticamente irá realizar o *download* das bibliotecas utilizadas no projeto, nele configuramos a versão desejada do Java, podemos fazer versionamento do código.

Foi utilizada a biblioteca chamada "Lombok"responsável pela geração automática de códigos como contrutores, "getters", "setters"entre outros.

Listing 3: Arquivo de configuração da API Gateway

```
1 server:
2   port: 8180
3
4 spring:
5   profiles.active: dev
6   application.name: gateway-project
7   main:
8     web-application-type: reactive
```

3.3.2 Implementando as classes do sistema

O Gateway possui três implementações de classes principais sendo eles:

- GatewayConfig, responsável por rotear as requisições recebidas através de rotas pré-configuradas. O Listing 4 mostra que a classe está anotada com "@Configuration"e o método com "@Bean". O atributo "id"serve como um apelido para aquela rota, "path"é a rota pela qual o gateway recebe a requisição, "uri"é a rota de destino e, por fim, "filters"são os filtros onde o gateway decide se a rota precisa de autenticação ou não.

Listing 4: Classe de roteamento do gateway

```
1  @Configuration
2  public class GatewayConfig {
3
4      @Bean
5      public RouteLocator gatewayRoutes(RouteLocatorBuilder
           builder) {
6          return builder.routes()
7              .route("user", r -> r
8                      .path("/user/auth")
9                      .uri("http://localhost:8181/user/auth"))
10             .route("user-get-one", r -> r
11                 .path("/user/get-one")
12                 .filters(f -> f.filter(new
                       AuthenticationFilter()))
13                 .uri("http://localhost:8181/user/get-one"))
14             .route("user-create", r -> r
15                 .path("/user/create")
16                 .filters(f -> f.filter(new
                       AuthenticationFilter()))
17                 .uri("http://localhost:8181/user/create"))
18             .route("user-delete", r -> r
19                 .path("/user/delete/**")
20                 .filters(f -> f.filter(new
                       AuthenticationFilter()))
21                 .uri("http://localhost:8181/user/delete"))
22             .build();
23     }
24 }
```

- AuthenticationFilter ilustrada no Listing 5 é responsável por filtrar as requisições e permitir apenas aquelas que estejam autenticadas. Ele valida o token "Bearer"informado no cabeçalho da requisição e usa a "jwtSecret", que é armazenada em

uma variável de ambiente e é a mesma utilizada na geração do token, para tentar validá-lo. Se a chave não for válida, ele lança uma exceção com o código HTTP 401 de não autorizado.

Listing 5: Classe de filtragem do gateway

```
1  public class AuthenticationFilter implements GatewayFilter {
2
3      @Value("$jwt.secret")
4      private String jwtSecret;
5
6      @Override
7      public Mono<Void> filter(ServerWebExchange exchange,
8      GatewayFilterChain chain) {
9          String header = exchange.getRequest().getHeaders()
10         .getFirst("Authorization");
11
12         if (header == null || !header.startsWith("Bearer
               ")) {
13             throw new UnauthorizedException();
14         }
15
16         String token = header.substring(7);
17
18         try {
19             Claims claims =
                   Jwts.parser().setSigningKey(jwtSecret)
20             .parseClaimsJws(token).getBody();
21             exchange.getRequest().mutate().header("documentNumber"
                   ,claims.getSubject());
22         } catch (ExpiredJwtException e){
23             throw new UnauthorizedException();
24         }
25         catch (JwtException | IllegalArgumentException e) {
26             throw new UnauthorizedException();
27         }
28
29         return chain.filter(exchange);
30     }
31 }
```

- AuthenticationException ilustrado no Listing 6, Representa um erro que pode ocorrer durante a execução do programa na validação de autenticação e retorna um código HTTP 401 de não autorizado.

Listing 6: Classe de erro não autorização

```
1  @ResponseStatus(HttpStatus.UNAUTHORIZED)
2  public class UnauthorizedException extends RuntimeException{
3
4      public UnauthorizedException(String message) {
5          super(message);
6      }
7
8      public UnauthorizedException(String message, Throwable
           cause) {
9          super(message, cause);
10     }
11 }
```

As rotas ilustradas no Listing 4 são acessíveis por meio de requisições REST utilizando o protocolo HTTP e seus verbos, tais como POST, GET, PUT, DELETE, entre outros. A tabela 1 representa as requisições aceitas pelo gateway e a resposta sobre as mesmas estarem autenticadas ou não. É considerado que a requisição está no modelo correto, variando apenas a autenticação.

Tabela 1: Métodos de requisição para API Gateway

URI	HTTP	Retorno c/ autenticação	Retorno s/ autenticação
http://localhost:8180/user/auth	POST	200 OK	200 OK
http://localhost:8180/user/create	POST	201 CREATED	401 UNAUTHORIZED
http://localhost:8180/user/get-one	GET	200 OK	401 UNAUTHORIZED
http://localhost:8180/user/delete/**	DELETE	200 OK	401 UNAUTHORIZED

3.4 API de autenticação

Neste capítulo iremos abordar a construção da API de autenticação com suas camadas e estratégias de segurança.

3.4.1 Configurações da API

Para a configuração do projeto Spring Boot, foi utilizado um arquivo chamado **application.yml**, localizado no diretório **"src/main/resources"**, a fim de definir diversas configurações que permitem a representação de dados em uma estrutura hierárquica. Nele configuramos o servidor e e as caracteristicas da aplicação Spring no caso dessa API temos que configurar também o banco de dados SQL e o cache Redis.

Listing 7: Arquivo de configuração da API de autenticação

```
server:
  port: 8181

spring:
  profiles.active: dev
  application.name: user

  spring:
    redis:
      host: localhost
      port: 6379
      password: root
  ##Mysql
  datasource:
    url: jdbc:mysql://${MYSQL_HOST:localhost}:${MYSQL_PORT
      :3306}/user-project
    username: ${MYSQL_USER:root}
    password: ${MYSQL_PASSWORD:root}
    hikari:
      connectionTimeout: 30000
      idleTimeout: 10000
      maxLifetime: 60000
      maximumPoolSize: 4
  jpa:
    show-sql: false
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL5InnoDBDialect
---
```

```
31 management:
32   endpoints:
33     web:
34       exposure:
35         include: loggers
36   endpoint:
37     loggers:
38       enabled: true
39 jwt:
40   secret: ${SECRET_JWT}
41   expiration: 100000
```

Por questões de segurança alguns valores são omitidos e salvos em variáveis de ambiente que posteriormente serão injetados na aplicação como vemos no Listing 7, por exemplo REDIS_PASSWORD"é a senha para ter acesso ao banco de dados redis, "MYSQL_PASSWORD"a senha para se ter acesso ao banco de dados SQL, variável "SECRET_JWT"é o valor da chave responsável por gerar o JWT.

3.4.2 Camada de serviço

A camada de serviço em uma API é responsável por implementar a lógica de negócio e coordenar as operações entre a camada de controle e a camada de acesso ao repositório. A camada de serviço geralmente contém a lógica de negócio principal da aplicação, como validações, processamento de dados e interações com outros componentes.

Listing 8: Implementação da camada de serviço 1

```
1  @Service
2  @RequiredArgsConstructor
3  public class UserServiceImpl implements IUserService{
4
5      private final UserRepository userRepository;
6
7      private final IJwtTokenService jwtTokenService;
8
9      private final ICacheToken cacheToken;
10
11     @Override
12     public UserCreateResponseDTO create(UserCreateDTO
           userCreateDTO, String documentNumber){
```

```
13          validateCreateUser(documentNumber, userCreateDTO.
                getDocumentNumber());
14          userRepository.save(new User(userCreateDTO,
                generateHashPassword(userCreateDTO.getPassword())));
15          return new UserCreateResponseDTO(userCreateDTO);
16      }
17
18      private String generateHashPassword(String password){
19          BCryptPasswordEncoder encoder = new
                BCryptPasswordEncoder();
20          return encoder.encode(password);
21      }
22
23      private void descryptoHashPassword(String password, User
            user){
24          BCryptPasswordEncoder encoder = new
                BCryptPasswordEncoder();
25          if (!encoder.matches(password, user.getPassword())){
26              if (user.getAttempts() >= 8){
27                  user.setStatus(Status.BLOCKED);
28                  cacheToken.save(user);
29                  throw new UnauthorizedException("Usuario ou
                        senha incorretos");
30              }
31              user.setAttempts(user.getAttempts() +1);
32              cacheToken.save(user);
33              throw new UnauthorizedException("Usuario ou senha
                    incorretos");
34          }
35          user.setAttempts(0);
36          cacheToken.save(user);
37      }
38
39      @Override
40      public UserAuthenticatedDTO autentication(UserDTO userDTO){
41          var user = cacheToken.getOne(userDTO.getDocumentNumber())
                ;
42          if (!Status.ACTIVE.equals(user.getStatus())){
```

```
43                 throw new UnauthorizedException("Usuario inativo ou
                       bloqueado");
44             }
45
46             descryptoHashPassword(userDTO.getPassword(), user);
47
48             return jwtTokenService.generateToken(user);
49         }
50 }
```

Assim como visto no Listing 8, a camada de serviço é uma implementação de uma inteface chamada de "IUserService", nela contém um contrato qual a API deve seguir ao implementar seus métodos, abordaremos os mais importantes.

3.4.2.1 Método *create()*

É responsável pela criação de um usuário no banco de dados, ele realiza a validação se o usuário já foi criado ou se o usuário que está tentando cria-lo tem autorização de administrador, a anotação "@Override"serve para sobrescrever o método, e retorna um DTO com algumas informações e se o usuário foi criado com sucesso. Após a sua criação o usuário pode usar as suas credenciais informadas no momento da criação para se autenticar e consumir outros endpoints da API.

3.4.2.2 Método *generateHashPassword()*

Serve para criptografar a senha que foi escolhida no momento da criação do usuário antes de ser salva no banco, garantindo maior segurança. O "BCryptPasswordEncoder"é uma biblioteca de segurança fornecida pelo Spring Security que utiliza um algoritmo de hash para senhas de forma segura, sendo amplamente utilizado em aplicações devido à sua robustez e capacidade de resistir a ataques de quebra de senha o mesmo utilizado por (NGUYEN; BAKER, 2019) .

Ao chamar o método *encode()* do "BCryptPasswordEncoder", a senha é convertida em um hash criptografado usando o algoritmo "BCrypt". Esse hash resultante é o valor que é armazenado no banco de dados. Quando um usuário faz login, a senha inserida é novamente codificada usando o mesmo algoritmo e o hash resultante é comparado ao hash armazenado no banco de dados para autenticação.

3.4.2.3 Método *descryptoHashPassword()*

Realiza o oposto do método anterior, a partir da senha digitada na autenticação ele converte usando o mesmo algoritmo e tenta comparar se as duas são iguals, dessa forma validamos se o usuário informou a senha que foi cadastrada na criação.

3.4.2.4 Método *autentication()*

Esse método é chamado no controlador e a rota de acesso é liberada pelo gateway, permitindo o acesso de requisições que ainda não estão autenticadas. Inicialmente, ele busca o usuário pelo número de documento, que é um campo obrigatório no momento do credenciamento. Se o usuário não existir no banco de dados, retornamos uma resposta indicando que o usuário é inválido, com um código HTTP 401.

Caso o usuário exista, as informações são armazenadas em um cache, e tentamos validar a senha usando o método anterior. Se a senha não corresponder à senha cadastrada, retornamos uma mensagem de erro "Usuário ou senha incorretos", juntamente com o código HTTP 401, e adicionamos uma tentativa sem sucesso para que caso se repita 8 vezes realizar o bloqueio do usuário. Caso tudo esteja correto, é retornado um JWT do tipo "Bearer"autorizando o acesso às APIs por um período de 100 segundos.

Na próxima autenticação desse usuário, se estiver dentro do limite de 500 segundos, suas informações serão recuperadas da memória cache, resultando em um ganho de desempenho na resposta da solicitação.

Alguns componentes foram desenvolvidos para o auxílio de algumas funções o primeiro é o "JwtTokenService"ilustrado no Listing 9.

Listing 9: Componente de segurança JWT

```
1  @Component
2  @RequiredArgsConstructor
3  public class JwtTokenService implements IJwtTokenService{
4
5      @Value("${jwt.secret}")
6      private String jwtSecret;
7
8      @Value("${jwt.expiration}")
9      private int jwtExpiration;
10
11     @Override
12     public UserAuthenticatedDTO generateToken(User
           authentication) {
```

```
13            return UserAuthenticatedDTO.builder()
14                    .token(Jwts.builder()
15                            .setSubject(authentication.getDocumentNumber
                                ())
16                            .setIssuedAt(new Date())
17                            .setExpiration(new Date((new Date()).getTime
                                () + jwtExpiration))
18                            .signWith(SignatureAlgorithm.HS512,
                                jwtSecret)
19                            .compact())
20                    .expirationTime(this.jwtExpiration)
21                    .name(authentication.getName())
22                    .build();
23        }
24 }
```

A anotação "@Value"injeta o valor nas variáveis que foi configurado no arquivo **application.yml**, dessa forma podendo ficar escondidos em uma variável de ambiente por questões de segurança, o "jwtSecret"é a chave utilizada para a geração do "token", e o "jwtExpiration"é o tempo de expiração em milisegundos.

O método *generateToken()* é responsável pela geração do JWT com o auxílio da biblioteca jjwt. Com ele, conseguimos adicionar um "subject", que seria um assunto, identificando assim um possível objetivo para aquele "token", a data de geração e a data de expiração. Dessa forma, na função de "quebra"dessa chave podemos identificar se o tempo de validade expirou.

O método *signWith()* é utilizado para adicionar uma assinatura digital, garantindo a integridade do token. Ao chamar esse método, são necessários o algoritmo de assinatura e uma chave secreta, utilizamos o algoritmo HS512 de acordo com o artigo (CHAVES et al., 2013) algoritmos de mais de 256 bits são potencialmente mais seguros.

A assinatura permite que o receptor verifique se o token foi gerado pelo emissor autorizado e se não foi alterado. Isso fornece segurança e confiança na integridade dos dados na chave. Ao validar o JWT, o receptor verifica a assinatura usando a mesma chave e algoritmo de assinatura. Se a assinatura for válida, a chave é considerada autêntica e não sofreu alterações.

Outro componente desenvolvido é o que trata o uso do Redis, por ser um armazenamento em memória cache extremamente rápido é possível reduzir o tempo de resposta a consultas muito repetitivas draticamente.

Listing 10: Componente de memória cache

```
1  @Component
2  @RequiredArgsConstructor
3  @RedisHash(timeToLive = 200)
4  public class CacheToken implements ICacheToken {
5
6      private final UserRepository userRepository;
7
8      @Cacheable(value = "user", key = "#documentNumber")
9      public User getOne(String documentNumber) {
10         return userRepository.findByDocumentNumber(
               documentNumber)
11                 .orElseThrow(() -> new UnauthorizedException("
                       Usuario incorreto"));
12
13     }
14 }
```

Quando uma classe é anotada com "@RedisHash", o Spring Data Redis assume a responsabilidade de armazenar e recuperar objetos dessa classe no Redis. O atributo "timeToLive"é usado para definir o tempo em que essa informação será armazenada em cache. A anotação "@Cacheable"identifica qual método irá armazenar o cache, juntamente com um valor "value"e uma chave "key", que serão usados para buscar essa informação novamente.

Se o cache não existir, a consulta *findByDocumentNumber()* será feita novamente no banco MySQL.

Além dos componentes temos também a implementação de "exceptions"para tratamento de erros utilizando uma resposta adequada para as demais situações, são elas "BadRequestException", "NotFoundException"e "UnauthorizedException"retornando os códigos HTTP respectivamente 400, 404, 401.

3.4.3 Camada de repositório e entidades

Para permitir a persistência das nossas classes, utilizamos o Spring Data, um módulo do Spring que implementa o JPA. O Spring Data simplifica o acesso ao banco de dados ao fornecer uma abstração de alto nível para operações de persistência.

Para utilizar o Spring Data, cada classe que desejamos persistir deve ter a sua interface de acesso ao banco de dados. Essa interface deve estender a classe "JpaRepository"e especificar dois parâmetros genéricos. O primeiro parâmetro é a classe que será

persistida e o segundo parâmetro é o tipo da chave primária dessa classe como ilustrado no Listing 11.

Listing 11: Acesso a dados com Spring Data

```
1  @Repository
2  public interface UserRepository extends JpaRepository<User, UUID
       > {
3
4      Optional<User> findByDocumentNumber(String documentNumber);
5
6  }
```

O modelo orientado a objetos ilustrado no Listing 12 representa a entidade a ser persistida.

Listing 12: Entidade User

```
1  @Entity
2  @Table(name = "db_user")
3  @Getter
4  @Setter
5  @NoArgsConstructor
6  public class User {
7
8      @Id
9      @Column(columnDefinition = "BINARY(16)", updatable = false)
10     private UUID identifier = UUID.randomUUID();
11
12     @Column(unique = true)
13     private String documentNumber;
14
15     private String password;
16
17     private String name;
18
19     private boolean admin;
20
21     private int attempts;
22
23     @Enumerated(EnumType.STRING)
24     private Status status;
```

```
25
26      public User(UserCreateDTO userCreateDTO, String password) {
27          this.setName(userCreateDTO.getName());
28          this.setDocumentNumber(userCreateDTO.getDocumentNumber()
                );
29          this.setPassword(password);
30          this.setStatus(Status.ACTIVE);
31      }
32  }
```

Os atributos "identifier"e "documentNumber"possuem anotações do JPA, que é uma especificação do Java para mapeamento objeto-relacional. Essas anotações são usadas para definir o comportamento de persistência desses atributos no banco de dados.

UUID é um identificador único universalmente reconhecido é uma sequência de caracteres de 128 bits. Em Java, a classe **java.util.UUID** é usada para trabalhar com UUIDs. Ela fornece métodos para criar, manipular e converter, o atributo identifier vai receber e persistir esse identificador.

Os atributos "documentNumber", "password", "name"e "admin"representam, respectivamente, o número de documento do usuário, a senha cadastrada no momento do credenciamento, o nome do usuário para identificação e uma flag lógica booleana que indica se o usuário possui privilégios de administrador ou não, "attempts"indica a quantidade de tentativas de acesso mau-suscedidas e "status"o status do usuário se está ativo, bloqueado ou inativo.

3.4.4 Camada do controlador

A camada do controlador é onde são implementados os *endpoints* que seriam porta de acesso para quem utiliza a API. Para a sua criação é necessário utilizar uma anotação chamada de "@RestController"do Spring Mvc e outra definindo o caminho desse controlador "@RequestMapping", com essas duas anotações o controlador já está pronto para receber requisições, os endpoints destacados abaixo utilizam as anotações de "@PostMapping", "@GetMapping", que servem para configurar o tipo de métodos HTTP "@ResponseStatus"que define o retorno caso a solicitação seja realizada com sucesso. As anotações "@RequestBody"e "@RequestHeader"são utilizadas para receber dados no corpo da requisição e no cabeçalho, respectivamente.

3.4.4.1 Endpoint de criação de usuário

O endpoint ilustrado no Listing 13 é responsável por criar um novo usuário. O objeto "UserCreateDTO", ilustrado no Listing 14, contém as informações necessárias para criar a conta, como "documentNumber", "password"e "name". Para utilizar esse endpoint, é necessário estar autenticado como um usuário "admin".

Listing 13: Endpoint de criação de usuário

```
1 @PostMapping("/create")
2 @ResponseStatus(HttpStatus.CREATED)
3 public UserCreateResponseDTO create(@RequestBody UserCreateDTO
    userDTO,
4             @RequestHeader String documentNumber){
5     return userService.create(userDTO, documentNumber);
6 }
```

Listing 14: Objeto de requisição para criação de usuários

```
1 @PostMapping("/create")
2 @ResponseStatus(HttpStatus.CREATED)
3     public UserCreateResponseDTO create(@RequestBody
        UserCreateDTO userDTO,
4             @RequestHeader String documentNumber){
5     return userService.create(userDTO, documentNumber);
6 }
```

3.4.4.2 Endpoint de autenticação

O objetivo desse endpoint é autenticar as credenciais do usuário que já estão cadastradas na base de dados. O Listing 15 ilustra a chamada do método na camada de serviço e o objeto de requisição "UserDTO"ilustrado no Listing 16, onde o usuário informa suas credenciais. O retorno é um "token"temporário válido para acessar outras rotas da API.

Listing 15: Endpoint de autenticação de usuários

```
1 @PostMapping("/create")
2 @ResponseStatus(HttpStatus.CREATED)
3 public UserCreateResponseDTO create(@RequestBody UserCreateDTO
    userDTO,
4             @RequestHeader String documentNumber){
5     return userService.create(userDTO, documentNumber);
6 }
```

Listing 16: Objeto de requisição para autenticação das credenciais

```
1 @Getter
2 public class UserDTO {
3
4     private String documentNumber;
5
6     private String password;
7
8 }
```

3.4.4.3 Endpoint de consulta de usuário

O objetivo do endpoint é retornar os dados do usuário que já está autenticado. Quando o gateway encaminha a requisição para a API de usuários, ele quebra o token e inclui a informação do documento do usuário no cabeçalho da requisição. Com base nessa informação, podemos buscar os dados correspondentes no banco de dados e retorná-los, como ilustrado no Listing 17. As informações retornadas incluem o nome e o número do documento.

Listing 17: Endpoint de consulta de usuário autenticado

```
1 @GetMapping("/get-one")
2 @ResponseStatus(HttpStatus.OK)
3 public UserCreateResponseDTO getOne(@RequestHeader String
    documentNumber){
4     return userService.getOne(documentNumber);
5 }
```

4 DISCUSSÃO E ANALISE DOS RESULTADOS

Neste capítulo, serão apresentados os resultados obtidos com a metodologia desenvolvida. As APIs foram executadas em um ambiente local, em containers gerenciados pelo docker-compose, onde a máquina possuía uma capacidade de processamento de 1,6 GHz e 12 GB de memória RAM.

4.1 Api Gateway

A API Gateway se comportou conforme esperado, atuando como a porta de entrada para todas as requisições e validando aquelas que requerem autenticação antes de se comunicarem com os microsserviços de destino.

4.2 Api User

Foi possível realizar a criação, listagem e exclusão de usuários, bem como a validação de credenciais e autenticação, utilizando o Spring Security. Utilizamos o banco de dados MySQL para armazenar essas informações e implementamos um cache de memória dinâmico para intermediar a comunicação e evitar múltiplos acessos ao banco.

Foi utilizada a ferramenta Postman, que serve para realizar testes via API. Realizamos 10 mil requisições de autenticação de usuário com as credenciais válidas e com o cache das informações do usuário habilitado. Replicamos o mesmo teste com o cache desabilitado. Os resultados podem ser observados na tabela 2. Com a máquina e cenários relatados, obtivemos os seguintes valores, tendo um possível ganho no tempo de resposta ao utilizar o cache.

Tabela 2: Resultados obtidos na autenticação com e sem cache

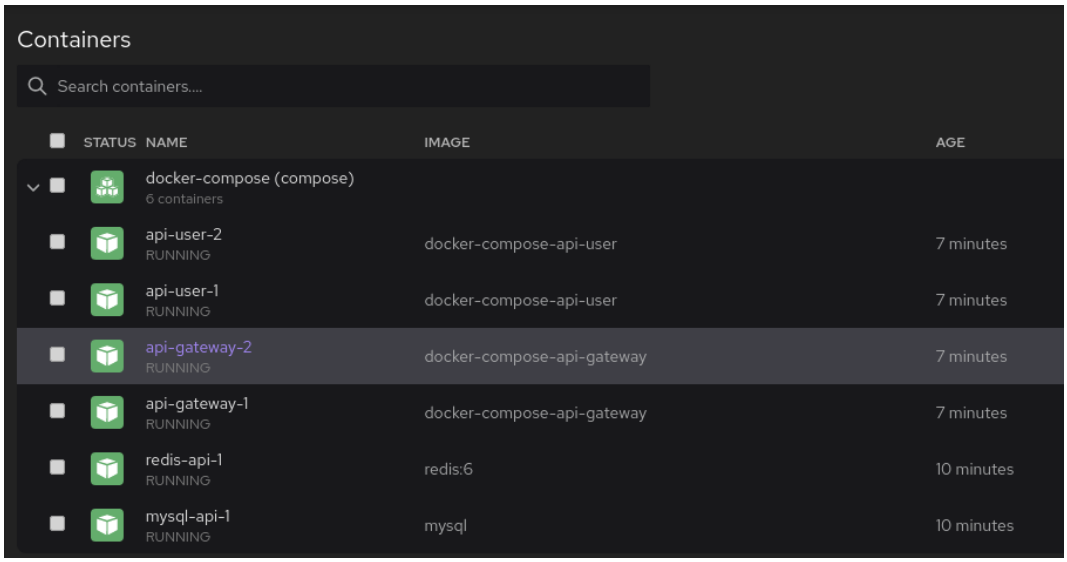
Utilização de cache	**Quantidade de requisições**	**Tempo médio de Resposta**	**Retorno**
Sim	10000	157ms	200 OK
Não	10000	168ms	200 OK

4.3 Escalabilidade

Com o objetivo de demonstrar a escalabilidade do sistema mesmo que mínima, realizamos o dimensionamento dos microsserviços User e Gateway utilizando o Docker Compose. Configuramos o número de réplicas dos microsserviços, permitindo escalá-los algumas vezes mas de forma não dinâmica, precisando realizar configuração manual a fim da criação de uma nova instância do microsserviço, não foi desenvolvido balanceamento de carga.

A imagem 6 foi obtida a partir do Podman Desktop, uma interface gráfica para o gerenciamento de containers. Após o escalonamento podemos ver que existem duas instâncias da api-gateway e da api-user dentro de um gerenciador do docker-compose, pudemos utilizar as APIs normalmente, juntamente com uma única instância do MySQL e do Redis.

Figura 6: Visão do podman desktop com microsserviços replicados



Fonte: Autoria Própria

4.4 Tentativas de invasão

Realizamos um teste de tentativa de invasão utilizando o ataque CSRF, que foi descrito anteriormente. Foram realizadas 10 mil requisições com o filtro CSRF do Spring Security ativado, e a mesma quantidade com o filtro desativado, conforme as configurações mencionadas anteriormente. Fomos motivados a realizar esse teste com base no trabalho de (NGUYEN; BAKER, 2019), e chegamos aos resultados descritos na tabela 3.

Dessa forma, sem a ativação do filtro não é verificado a presença de um token a mais de segurança conseguindo assim autenticar 10 mil vezes sendo necessário apenas o uso das credenciais corretas, já com a ativação não foi possível autenticar em nenhuma das tentativas mesmo com as credenciais corretas.

Tabela 3: Resultados obtidos em testes na tentativa de invasão

Cenário de teste	Caso de teste	Resultado
CSRF configuração do spring desabilitada	Credenciais válidas, sem token CSRF	Autenticou
CSRF configuração do spring habilitada	Credenciais válidas, sem token CSRF	Não autenticou

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve como principal objetivo o desenvolvimento de uma API de autenticação para uma arquitetura de microsserviços. Para atingir esse objetivo, utilizamos uma metodologia que se baseou no emprego de tecnologias amplamente reconhecidas no mercado, pertencentes ao ecossistema Spring. Entre essas tecnologias, destacam-se os frameworks Spring MVC, Security, Gateway e Boot. Ao utilizar essas ferramentas, foi possível implementar uma solução, capaz de fornecer serviços de autenticação para a arquitetura de microsserviços. Através da API desenvolvida, os usuários podem se autenticar e utilizar as funcionalidades da API

Para trabalhos futuros, implementar autenticação de múltiplos fatores, a fim de aumentar a segurança das aplicações. É importante considerar também a comunicação da API por meio do protocolo TLS, com a adição de certificados digitais para garantir maior segurança no transporte dos dados, implementar um balanceador entre os microsserviços para equalizar a utilização de recursos dentro da arquitetura de microsserviços.

REFERÊNCIAS

ALOUL, F.; ZAHIDI, S.; EL-HAJJ, W. Two factor authentication using mobile phones. In: IEEE. *2009 IEEE/ACS international conference on computer systems and applications*. [S.l.], 2009. p. 641–644.

ARAÚJO, C. P. d. Projeto e implementação de um serviço web restful com técnicas de segurança. Universidade Federal da Paraíba, 2018.

ARMANDO, A. et al. Attribute based access control for apis in spring security. In: *Proceedings of the 19th ACM symposium on Access control models and technologies*. [S.l.: s.n.], 2014. p. 85–88.

BALACHANDAR, B. M. *RESTful Java Web Services: A pragmatic guide to designing and building RESTful APIs using Java*. [S.l.]: Packt Publishing Ltd, 2017.

BALAJ, Y. Token-based vs session-based authentication: A survey. *no. September*, p. 1–6, 2017.

BARABANOV, A.; MAKRUSHIN, D. Authentication and authorization in microservice-based systems: survey of architecture patterns. *arXiv preprint arXiv:2009.02114*, 2020.

BIANCHI, L. H. T.; FONSECA, J. Java authentication e authorization service como mecanismo de segurança e controle de acesso em aplicações web.

BUSHONG, V. et al. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, MDPI, v. 11, n. 17, p. 7856, 2021.

CHAVES, L. C. et al. Segurança de software: uma abordagem multicritério para avaliação de desempenho. *Pesquisa Operacional para o Desenvolvimento*, v. 5, n. 2, p. 136–171, 2013.

FERRAG, M. A. et al. Authentication protocols for internet of things: a comprehensive survey. *Security and Communication Networks*, Hindawi, v. 2017, 2017.

GROFF, J. R.; WEINBERG, P. N.; OPPEL, A. J. *SQL: the complete reference*. [S.l.]: McGraw-Hill/Osborne, 2002. v. 2.

JONES, M.; CAMPBELL, B.; MORTIMORE, C. *JSON Web Token (JWT) profile for OAuth 2.0 client authentication and authorization Grants*. [S.l.], 2015.

KALIBERA, T. et al. Real-time java in space: Potential benefits and open challenges. *Proceedings of Data Systems in Aerospace (DASIA)*, Citeseer, 2009.

LARRUCEA, X. et al. Microservices. *IEEE Software*, IEEE, v. 35, n. 3, p. 96–100, 2018.

LUCIO, J. P. D. et al. Análise comparativa entre arquitetura monolítica e de microsserviços. Florianópolis, SC, 2017.

MAK, G. Spring mvc framework. In: *Spring Recipes: A Problem-Solution Approach*. [S.l.]: Springer, 2008. p. 321–393.

NGUYEN, Q.; BAKER, O. F. Applying spring security framework and oauth2 to protect microservice architecture api. *J. Softw.*, v. 14, n. 6, p. 257–264, 2019.

REDDY, K. S. P. *Beginning Spring Boot 2: Applications and microservices with the Spring framework*. [S.l.]: Apress, 2017.

SANTOS, O. *Network Security with NetFlow and IPFIX: Big Data Analytics for Information Security*. [S.l.]: Cisco Press, 2015.

SONI, A.; RANGA, V. Api features individualizing of web services: Rest and soap. *International Journal of Innovative Technology and Exploring Engineering*, v. 8, n. 9, p. 664–671, 2019.

WALLS, C. *Spring in action*. [S.l.]: Simon and Schuster, 2022.

ZHAO, J.; JING, S.; JIANG, L. Management of api gateway based on micro-service architecture. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2018. v. 1087, n. 3, p. 032032.

ANEXOS

A implementação da API gateway, na versão em que este trabalho foi escrito, encontra-se disponível em: https://github.com/zaqueumoura/Gateway-Project. Acesso em: 31/05/2023.

A implementação da API user, na versão em que este trabalho foi escrito, encontra-se disponível em: https://github.com/zaqueumoura/User-Project. Acesso em: 31/05/2023.

A Implementação da configuração do docker-compose para gerenciamento das APIs em containers na versão em que este trabalho foi escrito, encontra-se disponível em: https://github.com/zaqueumoura/configuration-docker-compose. Acesso em: 31/05/2023