DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Estudo Comparativo de Soluções IAM Para Arquitetura Zero Trust Sobre Microsserviços

André Luiz Lourenço de Andrade

Orientador: Rafael Timóteo de Sousa Júnior

Coorientador: Robson de Oliveira Albuquerque

Programa de Pós-Graduação Profissional em Engenharia Elétrica

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Estudo Comparativo de Soluções IAM Para Arquitetura Zero Trust Sobre Microsserviços

André Luiz Lourenço de Andrade

Orientador: Rafael Timóteo de Sousa Júnior

Coorientador: Robson de Oliveira Albuquerque

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Rafael Timóteo de Sousa Júnior, Ph.D, FT/UnB _____
Orientador

Prof. Robson de Oliveira Albuquerque, Ph.D, FT/UnB _____
Coorientador

Prof. João José Costa Gondim, Ph.D, FT/UnB _____
Examinador Interno

Prof. Dino Macedo Amaral, Banco do Brasil _____
Examinador externo

Prof. Fábio Lúcio Lopes de Mendonça, Ph.D, FT/UnB
Examinador suplente _____

FICHA CATALOGRÁFICA

ANDRADE, ANDRÉ LUIZ
Estudo Comparativo de Soluções IAM Para Arquitetura Zero Trust Sobre Microsserviços [Distrito Federal] 2024.
xvi, 68 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2024).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Autorização	2. APIs REST
3. JSON Web Token	4. Segurança
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ANDRADE, A. (2024). *Estudo Comparativo de Soluções IAM Para Arquitetura Zero Trust Sobre Microsserviços*. Dissertação de Mestrado Profissional, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 68 p.

André Luiz Lourenço de Andrade
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

AGRADECIMENTOS

Quero agradecer primeiramente a Deus por ser infinitamente gracioso em sua bondade comigo. Aos meus pais, por fazerem o impossível para me proporcionarem a melhor educação disponível. Ao meu co-orientador Robson, peça fundamental para que eu não desistisse desse projeto, ao meu orientador Rafael Timóteo por me possibilitar ter trilhado essa trajetória e a minha noiva Maria Luiza que sempre me apoia em todas as minhas ideias.

RESUMO

Como consequência da evolução do desenvolvimento de sistemas monolíticos para os microsserviços, distribui-se com a aplicação a responsabilidade de garantir chamadas de rede que sejam devidamente autenticadas e autorizadas. A arquitetura *Zero Trust* institui um modelo de segurança que não presume a segurança intrínseca de qualquer rede, exigindo autenticação, autorização e criptografia para todas as solicitações de serviço, independentemente da origem. Esta abordagem minimiza a superfície de ataque e fortalece a segurança ao aplicar rigorosamente a verificação de identidade e o princípio de menor privilégio para cada requisição que possui como destino um microsserviço. Para este trabalho, foi analisado o processo de implementação de autorização utilizando 3 arquiteturas de microsserviços: *mesh*, com autorização descentralizada, gateways de API e, por último, uma arquitetura associada ao serviço de nuvem authorization as a service, destacando as principais vantagens e limitações na implementação de cada cenário.

Palavras-chave: Segurança cibernética, microsserviços, autorização, JSON Web Token, APIs REST

ABSTRACT

As a consequence of the evolution from monolithic system development to microservices, the responsibility for ensuring properly authenticated, and authorized network calls is distributed along with the application. The Zero Trust architecture establishes a security model that does not assume the inherent security of any network, requiring authentication, authorization, and encryption for all service requests, regardless of their origin. This approach minimizes the attack surface and strengthens security by rigorously applying identity verification and the principle of least privilege for each request targeting a microservice. In this work, the process of implementing authorization using three microservices architectures was analyzed: mesh with decentralized authorization, API gateways, and, finally, an architecture associated with the authorization-as-a-service cloud service. This analysis highlights the main advantages and limitations in the implementation of each scenario.

Keywords: cybersecurity, microservices, authorization, JSON Web Token, REST APIs

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Embora medidas de segurança baseadas em perímetro possam fornecer proteção contra ameaças cibernéticas, elas são insuficientes por si só, uma vez que dependem da suposição de que todas as ameaças virão de fora da rede em administração [6]. A fim de conter os riscos de maneira adequada, as organizações tiveram que adotar uma abordagem de defesa baseada em camadas, que incorpora uma variedade de medidas para proteger contra perigos que se originam internamente e externamente à rede. Esta abordagem tem sido amplamente reconhecida como essencial na proteção eficaz dos ativos e dados de uma organização [7].

No contexto da modernização de aplicativos, a atenção dedicada aos processos de autenticação e autorização tornou-se indispensável no *design* de serviços seguros que demandam a identificação precisa de seus usuários. A evolução dessas tecnologias ocorre simultaneamente com o aumento da sofisticação dos ataques que visam contornar os processos de identificação e obter privilégios indevidos. Portanto, a obtenção de controle de identidade e níveis de permissão para cada requisição que consomem serviços WEB ganhou relevância substancial, tornando-se necessário a inclusão de *tokens* de autorização em cada requisição *HTTP* realizada pelos clientes [8].

Dentre os novos desafios embutidos com a migração de sistemas monolíticos para microsserviços, surge a possibilidade de implantações independentes que fomentam a cultura *DevOps*. No entanto, os mesmos mecanismos que possibilitam a maior agilidade no desenvolvimento de código entre as equipes também podem trazer consigo sérias irregularidades sobre a padronização do controle de acesso às informações providas pelos microsserviços. Em uma arquitetura de microsserviços, os desenvolvedores se encontram em uma nova posição mais desafiadora: garantir a segurança não apenas de um único *gateway* de API externo, mas de cada API de microsserviço individual com uma etapa de autenticação e autorização segura. De fato, um princípio fundamental de uma arquitetura de confiança zero é que cada solicitação deve ser autenticada e autorizada antes de sua resposta com dados potencialmente sensíveis.

No cenário contemporâneo dos centros de operações de segurança modernos, a superfície de ataque das organizações expandiu-se, onde cada *endpoint* pode representar uma entidade autônoma. Como resultado, uma violação de dados causada por uma vulnerabilidade conhecida pode potencialmente incorrer em custos que chegam a US$ 4,45 milhões em 2023, refletindo a média global de despesas em situações de ataques que causam vazamento de dados [9]. Atualmente, os impactos das ameaças cibernéticas extrapolam o contexto de prejuízo financeiro das organizações. Países em todo o mundo, continuamente promulgam leis de proteção de dados. No início de 2021, 145 países o fizeram e no ano subsequente, outros doze países promulgaram tais leis, totalizando 157 nações até meados de março de 2022 [10], sendo a maioria delas substancialmente influenciada pelo Regulamento Geral de Proteção de Dados (GDPR) da União Europeia.

1.1 MOTIVAÇÃO E JUSTIFICATIVA

A crescente demanda da inserção de tecnologia em processos anteriormente realizados manualmente, faz com que as aplicações tenham que se adequar em lidar com as nuances das regras de negócios que elas são submetidas, tornando-as complexas e com excessivas normas necessárias para entregarem o conteúdo esperado, conforme as limitações, ou atributos, de quem está realizando a requisição.

Nesse cenário, as APIs são um dos principais facilitadores da digitalização nas empresas, visto que elas interconectam aplicações para aparecerem como um único fluxo de trabalho. No entanto, à medida que as APIs oferecem uma experiência contínua aos usuários e empresas, elas também expõem oportunidades para cibercriminosos, que podem explorar *endpoints* para o estabelecimento de controle, mantendo persistência e movendo-se lateralmente através das redes [11] com o potencial de acessar dados de forma não autorizada.

Por esta razão, técnicas e ferramentas que lidam com o gerenciamento de identidades de acesso (IAM), torna-se parte fundamental do controle e da segurança de APIs, fazendo com que o conteúdo entregue ao solicitante seja de acordo com seu nível de permissão, podendo ajudar com a *compliance* demandada pelos órgãos regulatórios, entregando mais segurança aos dados e reduzindo as chances de erros humanos. A falta de um mecanismo eficiente de IAM pode criar múltiplas adversidades em uma infraestrutura de TI, incluindo gestão de identidade, gestão de riscos, gestão de confiança, conformidade, segurança de dados, privacidade, transparência e vazamento de dados [12].

Diferente da arquitetura monolítica, ambientes modernos que utilizam microsserviços possuem desafios particulares na concessão de acesso à informação. Tendo em vista a sua premissa de oferecer um projeto dinâmico e fragmentado, a validação de credenciais pode ter que ser realizada sistematicamente, o que, por sua vez, é uma dificuldade considerando que várias equipes devem trabalhar separadamente e implementar restrições de autorização padronizadas e consistentes [13].

Em contraste, uma aplicação de microsserviços possui múltiplos componentes independentes integrados por meio de APIs. Sempre que um microsserviço se comunica com outros microsserviços, é necessário para sua segurança garantir que ele esteja autenticado. A autenticação assegura que apenas serviços legítimos e usuários tenham acesso a cada microsserviço. Além disso, assim como em uma aplicação monolítica, é necessário autenticar os usuários finais.

Quando implementada corretamente, a autenticação e a autorização são ativos essenciais de uma aplicação de microsserviços, atuando como uma verificação de segurança adicional para todos os recursos acessados, evitando lacunas de segurança e pontos cegos. Entre os desafios de autenticação em microsserviços, estão:

1. **Dependência centralizada:** a lógica de autenticação e autorização deve ser tratada separadamente por cada microsserviço. É possível utilizar o mesmo código em todos os microsserviços, mas isso requer que todos os microsserviços suportem uma estrutura específica.

2. **Violação do princípio da responsabilidade única:** os microsserviços devem cumprir apenas uma função. Se você adicionar lógica de autenticação e autorização global aos microsserviços, eles passarão a desempenhar uma função adicional, tornando-os menos confiáveis e mais difíceis de gerenciar.

3. **Complexidade:** a autenticação e autorização em microsserviços podem levar a cenários muito complexos. Considerando que pode haver usuários, microsserviços e sistemas de terceiros acessando cada microsserviço, essa complexidade pode dificultar a implementação e a manutenção.

Em artigos que antecedem esta pesquisa, é possível observar a comparação de arquiteturas de microsserviços sob o ponto de vista de performance e executabilidade entre a equipe de desenvolvimento considerando a cultura *DevOps*, no entanto há uma escassez bibliográfica na avaliação dessas disposições sob o ponto de vista de segurança e autorização dessas requisições em um cenário que prioriza *Zero Trust Architecture*.

Considerando a metodologia de *Zero-trust* em um cenário comum em que um microsserviço necessite realizar uma chamada a outro microsserviço, é necessário de alguma forma que cada microsserviço realize a validação das requisições realizadas para si. Portanto, cada requisição deve carregar consigo uma informação que prove que o solicitante está devidamente autenticado e autorizado para pedir por aquela informação, conforme evidenciado na figura 1.1. Este contexto impõe alguns desafios na validação dessa informação, tendo em vista que se deve garantir a integridade da informação que prova a concessão de permissão.

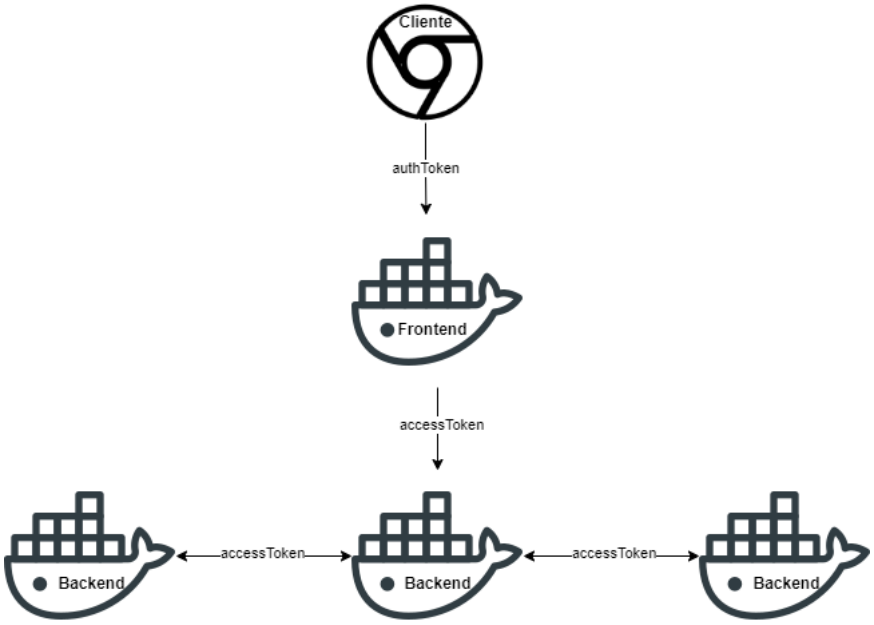

Figura 1.1: *Token* gerado para uma requisição específica e passado para os microsserviços *downstream*

O primeiro cuidado que deve ser considerado está relacionado ao gerenciamento de chaves. Para o caso de *tokens* que são assinados, os microsserviços necessitam consultar uma chave pública para poderem validar a integralidade da informação. Questões relacionadas a como o microsserviço obtém a chave pública e o que acontece caso a chave pública mude devem ser analisadas na implementação de microsserviços em arquitetura *zero trust*.

Outra preocupação atribuída a esse tipo de sistema é a definição do prazo de validade adequado para o *token*, o que pode ser uma tarefa complexa se forem considerados processamentos de longa duração. Um conjunto de processos assíncronos pode demorar horas para ser concluído. Desse modo, é necessária

a discussão por escolher gerar *tokens* de maior duração ou mesmo considerar a possibilidade de não se utilizar *tokens* e partir para um método mais eficaz para a situação.

Por fim, outra questão a ser observada é que, após a validação de autenticidade e integridade do valor passado junto às requisições aos microsserviços, é necessário ser avaliado o nível de autorização da requisição. Em sistemas robustos e complexos, a própria quantidade de informações contidas no *token* pode se tornar um problema de gerenciamento de permissões, tendo em vista o número de possibilidades de retornos, o que pode trazer prejuízos em relação à confiabilidade dos sistemas.

1.2 OBJETIVO

Esta dissertação propõe como objetivo principal analisar e comparar a implementação de controle de acesso em três cenários distintos de arquiteturas de microsserviços. No primeiro cenário, adota-se uma arquitetura *mesh*, na qual os microsserviços estabelecem comunicação direta. O segundo cenário introduz um *gateway* de API, agindo como intermediário na comunicação norte-sul entre os microsserviços. No terceiro cenário, a arquitetura incorpora um serviço de nuvem do tipo *authorization as a service*, no qual os contêineres enviam tokens de autorização para um serviço gerenciado em nuvem, separando ainda mais o controle de acesso da equipe de desenvolvimento. O estudo visa fornecer visões sobre as vantagens, desafios e características distintivas de cada cenário, contribuindo para a compreensão aprofundada das práticas de controle de acesso em ambientes de microsserviços segundo a arquitetura empregada.

1.2.1 Objetivos específicos

Esta dissertação possui os seguintes objetivos específicos:

- **Avaliar IAM em arquitetura de microsserviços distribuídos.** Analisar diferentes arquiteturas de IAM para microsserviços distribuídos, como *OAuth2, OpenID Connect, JWT,* e verificar sua adequação para ambientes escaláveis e distribuídos.
- **Analisar modelos de autorização em microsserviços.** Examinar e comparar diferentes modelos de autorização, como *Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC) e Policy-Based Access Control*, para identificar o mais adequado para microsserviços.
- **Avaliar segurança e gestão de *tokens* de autenticação.** Investigar práticas seguras para a geração, distribuição e gerenciamento de tokens de autenticação em ambientes de microsserviços para evitar ataques de token e garantir a autenticação segura.

1.3 CONTRIBUIÇÕES DA PESQUISA

Esta pesquisa tem como contribuição a realização de um estudo sobre autorização em uma infraestrutura de microsserviços, colocando em evidência 3 arquiteturas possíveis para a implementação de sistemas

conteinerizados: a primeira arquitetura é dada pela comunicação *mesh* entre microsserviços e é caracterizada pela chamada direta entre as APIs dos microsserviços, fazendo com que cada microsserviço seja o responsável direto pelo controle de acesso, inserindo no código de aplicação os controles de autorização necessários para a concessão de dados. No segundo cenário é adicionado à arquitetura um *gateway* de API que realiza a intermediação da comunicação entre os microsserviços, centralizando responsabilidades de roteamento na camada de aplicação e controle de acesso, sendo possível desta forma, uma padronização mais precisa das políticas de controle de acesso, além de ser possível terceirizar completamente a responsabilidade para uma equipe de seguraça erspecializada. No terceiro cenário o *gateway* de API sai de cena para a validação de autorização e entra em seu lugar o serviço de nuvem *permit.io* que realiza a validação dos *tokens* transmitidos junto às requisições para os microsserviços, identificando se a chamada em questão é autorizada a consumir o recurso pedido.

O trabalho realizado que antecede este artigo [14] proporciona sua contribuição ao campo de segurança de APIs REST ao realizar uma análise comparativa entre JWT *(JSON Web Tokens)*, PASETO (*Platform-Agnostic Security Tokens*) e *Macaroons*. Ao mergulhar nas características intrínsecas, vantagens e desvantagens de cada token de autorização, a pesquisa destaca aspectos fundamentais como segurança, flexibilidade, facilidade de implementação e interoperabilidade. JWT, sendo amplamente adotado, oferece uma estrutura compacta e eficiente para a transmissão de informações entre partes, garantindo a integridade e a autenticidade dos dados. Por outro lado, PASETO busca endereçar algumas vulnerabilidades conhecidas do JWT, propondo um modelo mais robusto e seguro que não depende da complexidade de escolha dos algoritmos de criptografia para a sua segurança. *Macaroons*, embora menos conhecidos, introduzem um mecanismo flexível e granular de delegação de permissões, permitindo construções de autorização mais sofisticadas e adaptáveis às necessidades específicas das aplicações.

1.4 ORGANIZAÇÃO DO TRABALHO

A organização do trabalho está dividida em cinco partes essenciais que desempenham um papel fundamental na estruturação e compreensão do conteúdo. A primeira parte, a Introdução, tem o propósito de apresentar o contexto e o problema de pesquisa, além de fornecer uma visão geral do que o leitor pode esperar encontrar ao longo do trabalho, assim como os objetivos gerais e específicos dos temas que o trabalho se propõe a debater. Em seguida, a seção de “Trabalhos Relacionados e Estado da Arte” explora as pesquisas e desenvolvimentos prévios relevantes, contextualizando o projeto em relação ao conhecimento existente na área e às tecnologias modernas que serão exploradas. O “Método de Pesquisa” detalha a abordagem metodológica e os procedimentos empregados para conduzir o estudo, proporcionando informações cruciais sobre a metodologia e ferramentas implementadas nos testes realizados. A “Descrição da Arquitetura e Resultados” é onde os aspectos técnicos do sistema ou modelo são explicados em detalhes, demonstrando como a pesquisa foi implementada, evidenciando as nuances, benefícios e desvantagens dos padrões de arquitetura abordados. Por fim, a seção de Conclusão apresenta os achados da pesquisa, discute suas implicações e oferece uma síntese das principais conclusões. A organização explicada das partes do trabalho acadêmico permite que os leitores naveguem eficazmente pelo conteúdo, compreendam o processo de pesquisa e avaliem as contribuições da pesquisa de maneira significativa.

2 TRABALHOS RELACIONADOS E ESTADO DA ARTE

Nesta seção, conceitos fundamentais e referências teóricas são discutidos tomando-se por base trabalhos relevantes para o tema da pesquisa. Elementos considerados basilares à compreensão do Modelo proposto serão explicitados nas seções seguintes. Alguns campos de pesquisa foram especialmente importantes para o desenvolvimento deste trabalho e, a cada um deles, foi dedicada uma seção ou subseção própria.

2.1 TRABALHOS RELACIONADOS

Na literatura acadêmica, uma análise aprofundada dos trabalhos relacionados desempenha um papel fundamental na construção de uma base sólida para qualquer pesquisa. Os trabalhos relacionados fornecem uma visão abrangente das contribuições anteriores ao campo de estudo, identificando lacunas de conhecimento e oportunidades para avanço. Uma revisão crítica da literatura também permite aos pesquisadores contextualizar seu trabalho no contexto mais amplo e avaliar como suas descobertas se comparam com estudos anteriores. Portanto, a revisão dos trabalhos relacionados é uma etapa crucial na pesquisa, ajudando a fundamentar a pesquisa atual em uma base sólida e a contribuir para o desenvolvimento contínuo do conhecimento na área de estudo.

Equilibrar os desafios concorrentes da colaboração e segurança é uma tarefa complexa e multidimensional. Enquanto os sistemas colaborativos concentram-se na criação de conexões úteis entre pessoas, ferramentas e informações, a segurança visa garantir a disponibilidade, confidencialidade e integridade desses mesmos elementos. Para a parte de estudos sobre controle de acesso foi utilizado a bibliografia [4] onde foi fornecido um estudo abrangente dos mecanismos de autorização para ambientes colaborativos. Fundamentalmente para este estudo foi realizado uma análise tanto dos méritos quanto das fraquezas de cada abordagem, bem como a identificação de tendências emergentes para modelos de autorização em colaboração.

A análise bibliográfica foi orientada segundo o trabalho de ALMEIDA, 2022 [15], onde é realizada uma revisão da literatura sobre a implementação de autenticação e autorização em microsserviços. O trabalho conclui haver uma carência de estudos relacionados à segurança na arquitetura de microsserviços e esta falta se torna mais evidente quando o estudo se concentra especificamente na autenticação e autorização, especialmente em uma abordagem prática. Desta forma, finaliza que é importante que o tema seja explorado de forma mais abrangente, pois, em um ambiente de microsserviços, é necessário se preocupar com aspectos de segurança em cada serviço individualmente.

O estudo de microsserviços e suas arquiteturas foi revisado pelo livro NEWMAN, 2021 [16] onde contém uma sustentação teórica robusta de migração de sistemas monolíticos para microsserviços. Além de definições de microsserviços e orientações da comunicação entre microsserviços utilizando APIs do tipo REST. Foi resgatado deste exemplar os principais benefícios da adoção de microsserviços em sistemas complexos e as principais tecnologias que o torna possível.

PEREIRA, 2019 [17] conduziu um mapeamento sistemático para revelar os mecanismos de segurança adotados em sistemas baseados em microsserviços, concentrando-se exclusivamente em mecanismos de segurança e examinaram 26 artigos publicados de novembro de 2018 a março de 2019. HANNOUSSE, 2021 [18] realizou uma investigação semelhante ao estudo [17], no entanto, abrangendo sua pesquisa de algumas maneiras, incluindo artigos publicados desde 2011 e, além de mecanismos de segurança, também se concentra em identificar ameaças de segurança e na aplicabilidade das soluções propostas em relação às plataformas de execução e camadas arquiteturais. YU, 2019 [19] realizou uma revisão do trabalho relacionado a riscos de segurança para aplicações baseadas em microsserviços em arquitetura *fog*, onde foi argumentado que questões de segurança surgem em quatro aspectos do sistema: contêineres, dados, permissões e segurança de rede.

Para a implementação das configurações realizadas no servidor de autorização, foi utilizado como principal fonte o livro THORGERSEN, SILVA, 2021 [20] onde é relatado com riquezas de detalhes as especificações utilizadas no servidor de autorização *Keycloak*. Com esta fonte foi possível inicializar todo o ambiente em contêineres *docker*, assim como realizar o correto *setup* das configurações de rotacionamento de chaves para a validação dos *JWTs* por meio dos *JWKS endpoints*. A obra foca na parte prática do desenvolvimento de IAM para aplicações modernas, respeitando todas as melhores práticas de segurança para a sua realização.

BARABANOV, 2020 [21] alertou que a implementação da autorização diretamente no código-fonte de cada microsserviço pode resultar em problemas futuros, especialmente em equipes diferentes trabalhando em microsserviços independentes, pois as novas atualizações de segurança devem ser realizadas em todos os projetos individualmente. Ele e HE, 2017 [22] seguiram na mesma linha, alertando que imitar a estrutura monolítica em cada microsserviço possui várias deficiências, principalmente quando um novo serviço é incorporado ao sistema, sendo necessário implementar a função de segurança nesse caso. Propuseram uma solução criando um serviço específico focado em autenticação e autorização, resultando em cada serviço concentrado em seu próprio negócio, garantindo melhor escalabilidade e desvinculando o sistema. JANDER, 2018 [23] mencionaram que diferentes equipes podem implementar sua própria abordagem interna de segurança no microsserviço pelo qual são responsáveis, mas isso pode exigir conhecimento mais especializado das equipes. TORKURA, 2017 [24] expressaram a preocupação de que o desenvolvimento por diferentes equipes pode resultar em microsserviços usando não apenas padrões diferentes no desenvolvimento, mas também tecnologias diferentes, o que pode dificultar a implementação dos mesmos padrões de segurança.

2.2 ARQUITETURA *ZERO TRUST*

Outra pesquisa basilar utilizada como fonte de informação para este estudo foi o documento do NIST [6], que é resultado de uma colaboração entre várias agências federais, supervisionada pelo Conselho Federal de CIO (*Chief Information Officers*). Neste documento, pode-se observar que há premissas básicas para a conectividade de rede para qualquer organização que utiliza *ZTA* no planejamento e implantação de rede. Algumas dessas premissas se aplicam à infraestrutura de rede de propriedade da empresa, e outras se aplicam aos recursos de propriedade da empresa que operam em infraestrutura de rede não pertencente

à empresa (por exemplo, redes *Wi-Fi* públicas ou provedores de nuvem pública). Essas premissas são usadas para orientar a formação de uma Arquitetura de Confiança Zero (ZTA). A rede em uma empresa que implementa ZTA deve ser desenvolvida com as seguintes premissas:

1. **Não se considera toda a rede privada da empresa uma zona de confiança implícita.** Os ativos devem sempre agir como se um atacante estivesse presente na rede da empresa, e a comunicação deve ser feita da maneira mais segura disponível. Isso envolve ações como autenticar todas as conexões e criptografar todo o tráfego.

2. **Dispositivos na rede podem não ser de propriedade ou configuráveis pela empresa.** Visitantes e/ou serviços contratados podem incluir ativos não pertencentes à empresa que precisam de acesso à rede para desempenhar seu papel. Isso inclui políticas de *Bring Your Own Device* (BYOD) que permitem que os sujeitos da empresa usem dispositivos não pertencentes à empresa para acessar recursos da empresa.

3. **Nenhum recurso é inerentemente confiável.** Cada ativo deve ter sua postura de segurança avaliada por um *Policy Enforcement Point* (PEP) antes que uma solicitação seja concedida a um recurso de propriedade da empresa. Essa avaliação deve ser contínua enquanto a sessão durar. Dispositivos de propriedade da empresa podem ter artefatos que permitem autenticação e fornecem um nível de confiança mais alto do que a mesma solicitação proveniente de dispositivos não pertencentes à empresa. Credenciais de sujeitos sozinhos não são suficientes para a autenticação de dispositivos a um recurso da empresa.

4. **Nem todos os recursos da empresa estão em infraestrutura de propriedade da empresa.** Os recursos incluem sujeitos da empresa remotos, bem como serviços em nuvem. Ativos de propriedade ou gerenciados pela empresa podem precisar utilizar a rede local (ou seja, não pertencente à empresa) para conectividade básica e serviços de rede (por exemplo, resolução de DNS).

5. **Sujeitos e ativos da empresa remotos não podem confiar completamente em sua conexão de rede local.** Sujeitos remotos devem assumir que a rede local (ou seja, não pertencente à empresa) é hostil. Os ativos devem assumir que todo o tráfego está sendo monitorado e potencialmente modificado. Todas as solicitações de conexão devem ser autenticadas e autorizadas, e todas as comunicações devem ser feitas da maneira mais segura possível.

6. **Ativos e fluxos de trabalho que se deslocam entre infraestrutura de propriedade da empresa e não pertencente à empresa devem ter uma política de segurança e postura consistentes.** Ativos e dados devem manter sua postura de segurança ao se deslocar para ou a partir da infraestrutura de propriedade da empresa. Isso inclui dispositivos que se deslocam das redes da empresa para redes não pertencentes à empresa. Isso também inclui dados que migram de *data centers* locais para instâncias de nuvem não pertencentes à empresa.

Neste contexto, o trabalho se prende aos princípios de ZTA realizados pelo NIST para avaliar a adequabilidade dos controles de segurança sugeridos na validação de *tokens* em microsserviços ao nível de requisição.

2.2.1 Autorização ao nível de requisição

Para o respeito dos princípios da *ZTA*, deve ser levado em consideração que toda a requisição, independente da sua origem, que chega a um microsserviço, deve ser inicialmente considerada maliciosa até ser realizada a devida validação de autenticação e autorização da solicitação. A autorização ao nível de requisição garante que todas as requisições que chegam a uma API em um microsserviços são devidamente validadas antes de sua resposta. Isto é importante tendo em vista a objeção de se manter confiança implícita entre os microsserviços.

Em um paradigma de microsserviços, onde serviços independentes colaboram para fornecer funcionalidades mais amplas, a confiança implícita é minimizada. A abordagem *zero trust* pressupõe que nenhum componente, mesmo internamente, é totalmente confiável, exigindo uma verificação rigorosa em cada solicitação. A autorização ao nível de solicitação permite avaliar, em tempo real, se um microsserviço específico tem a permissão necessária para acessar recursos ou executar ações. Isso é essencial para reforçar a política de menor privilégio, garantindo que cada solicitação seja avaliada individualmente, independentemente do histórico ou contexto anterior. Dessa forma, a arquitetura *zero trust* em microsserviços se beneficia significativamente da granularidade da autorização ao nível de solicitação, proporcionando uma camada adicional de segurança e controle em um ecossistema dinâmico e distribuído.

2.3 *IDENTITY ACCESS MANAGEMENT (IAM)*

Gestão de identidades e acessos, também conhecido por *IAM*, garante que as pessoas, os computadores e os componentes de *software* válidos tenham acesso aos recursos permitidos no momento certo. Primeiro, a pessoa, o computador ou o componente de *software* prova que é quem ou o que afirma ser (autenticação). Em seguida, a entidade recebe permissão ou negação de acesso ao uso de determinados recursos (autorização) [25]. Para um gerenciamento de Identidade que segue as boas práticas, é necessário uma visão holísticas dos processos que as ferramentas de *IAM*, com as funcionalidades evidenciadas na figura 2.1, se utilizam para manter a governança sobre o controle de gestão de acessos implementados nas organizações.

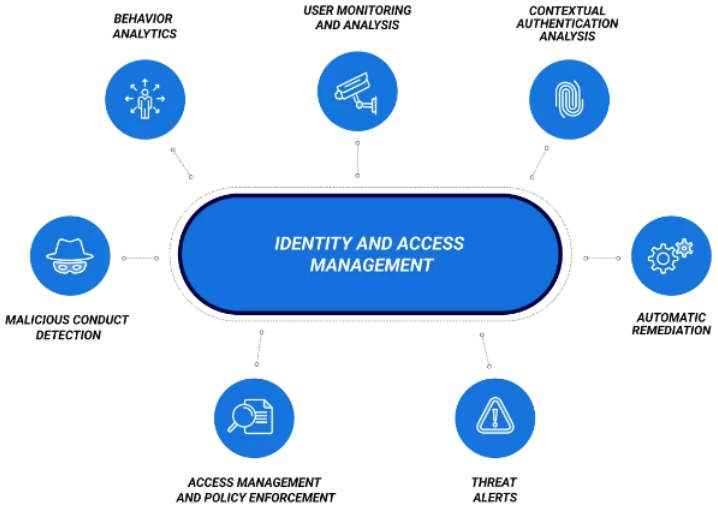

Figura 2.1: Funcionalidades de IAM [1].

A gestão de identidade e acesso tornou-se fundamental para as estratégias de cibersegurança de muitas empresas. Ferramentas e estruturas de *IAM* podem ajudar em:

- **Conformidade regulatória:** Normas como GDPR e PCI-DSS exigem políticas rigorosas sobre quem pode acessar dados e para quais finalidades. Sistemas de *IAM* permitem que empresas estabeleçam e apliquem políticas formais de controle de acesso que atendam a essas normas. As empresas também podem rastrear a atividade do usuário para comprovar conformidade durante uma auditoria.
- **Segurança de dados:** De acordo com o relatório Custos de uma Violação de Dados da IBM [9], roubo de credenciais é a principal causa de violações de dados. Sistemas de *IAM* podem adicionar camadas extras de autenticação, exigindo que *hackers* tenham mais do que apenas uma senha para acessar dados sensíveis. Políticas RBAC (*Role-Based Access Control*) podem limitar o movimento lateral de agentes maliciosos, incluindo ameaças internas.
- **Transformação digital:** Com o aumento de ambientes de nuvem múltipla, dispositivos *IoT*, trabalho remoto e *BYOD* (*Bring Your Own Device*), as empresas precisam facilitar o acesso seguro para mais tipos de usuários a mais tipos de recursos. Sistemas de *IAM* podem centralizar a gestão de acesso para todos os usuários e recursos em uma rede, mantendo a segurança da rede sem interromper a experiência do usuário.

Sendo assim, ferramentas de IAM podem trazer para as organizações mais segurança, maior produtividade dos funcionários, tendo em vista também a utilização de *SSO* (*Single Sign on*), conformidade mais sólidas e redução de custos.

2.3.1 Autorização

A autorização é um conceito fundamental no contexto da governança de dados e segurança da informação. Refere-se ao ato deliberado de conceder permissão ou consentimento para que indivíduos, sistemas ou

entidades acessem, modifiquem ou compartilhem recursos, ou informações específicas [26]. A autorização desempenha um papel crítico na proteção da privacidade, na conformidade regulatória e na mitigação de riscos, uma vez que ajuda a garantir que apenas indivíduos ou entidades autorizadas tenham acesso a dados sensíveis. Ela também é um componente essencial na garantia de que os sistemas de informação operem conforme os princípios de *least privilege*, limitando o acesso aos recursos somente ao necessário para a realização de tarefas específicas.

2.3.1.1 OAuth2.0

Em algumas situações, é necessário que os direitos de autorização sejam concedidos para fornecedores terceiros, fazendo com que essas aplicações externas possam ser autorizadas a acessar informações privadas específicas. Isto é o que possibilita, por exemplo, nas redes sociais, aplicações hospedadas em nuvem serem acessadas por aplicações externas sempre que o usuário autoriza essa aplicação [27]. Para a resolução desse problema, era comum que aplicações terceiras requisitassem a senha dos sistemas que elas necessitavam o acesso.

As implicações desfavoráveis dessa abordagem são diversas. Em primeiro lugar, as aplicações de terceiros seriam forçadas a armazenar as credenciais do proprietário dos recursos para uso futuro, muitas vezes em formato de senha em texto simples. Isso não apenas representa um risco significativo de segurança, mas também levanta preocupações quanto à privacidade e à integridade dos dados. Adicionalmente, os servidores seriam obrigados a suportar a autenticação por senha, apesar das vulnerabilidades inerentes a esse método. Além disso, as aplicações de terceiros, ao utilizar esse modelo, obteriam acesso excessivamente amplo aos recursos protegidos do proprietário, resultando na perda da capacidade do usuário de restringir a duração ou o acesso a um subconjunto limitado de recursos. Ademais, os proprietários de recursos enfrentariam a limitação de não poder revogar o acesso a um terceiro individual sem revogar o acesso a todos os terceiros, sendo obrigados a fazê-lo por meio da alteração da senha do terceiro. Por último, mas não menos importante, qualquer comprometimento de uma aplicação de terceiros resultaria no comprometimento direto da senha do usuário final e, por consequência, de todos os dados protegidos por essa senha. Essas implicações destacam a necessidade premente de estratégias mais seguras e eficientes no contexto de autenticação e controle de acesso.

Para endereçar este problema é utilizado o *OAuth2.0* que é um *framework* de autorização onde o cliente solicita acesso a recursos controlados pelo proprietário do recurso *(RO)* e hospedados pelo servidor de recursos, e recebe um conjunto diferente de credenciais em relação ás do proprietário do recurso. Em vez de usar as credenciais do proprietário do recurso para acessar dados protegidos, o cliente obtém um *token* de acesso, sendo uma cadeia de caracteres que denota um escopo específico, vida útil e outros atributos de acesso [28].

2.3.1.2 Papéis no *OAuth2.0*

O *framework* estabelece 4 papéis fundamentais para a concessão de privilégios, sendo esses:

- **Resource Owner (RO)**: Uma entidade capaz de conceder acesso a um recurso protegido. Quando o

dono de recurso é uma pessoa, é referido como um usuário final.

- **Servidor de Recurso:** O servidor que hospeda os recursos protegidos, capaz de aceitar e responder a solicitações de recursos protegidos usando tokens de acesso.
- **Cliente:** Uma aplicação que faz solicitações de recursos protegidos em nome do dono de recurso e com sua autorização. O termo "cliente"não implica em características de implementação específicas (por exemplo, se a aplicação é executada em um servidor, em um computador desktop ou em outros dispositivos).
- **Servidor de Autorização:** O servidor que emite tokens de acesso para o cliente após autenticar com sucesso o dono de recurso e obter autorização.

No *framework OAuth2* é possível utilizar vários fluxos possíveis para o manejamento de credenciais e concessão de acesso a recursos privados, para este trabalho, optamos por utilizar o *authorization code flow* que foi devidamente implementado com todas as suas etapas na seção 3.5.1.

Com o *authorization code flow* o código de autorização é obtido mediante o uso de um servidor de autorização como intermediário entre o cliente e o proprietário de recursos (*resource owner*) . Em vez de solicitar autorização diretamente ao proprietário de recursos, o cliente direciona o proprietário de recursos a um servidor de autorização, que, por sua vez, redireciona o proprietário de recursos de volta ao cliente com o código de autorização.

Antes de direcionar o proprietário de recursos de volta ao cliente com o código de autorização, o servidor de autorização autentica o proprietário de recursos e obtém a autorização. Como o proprietário de recursos somente se autentica com o servidor de autorização, as credenciais do proprietário de recursos nunca são compartilhadas com o cliente. O fluxo utilizando authorization code flow pode ser observado na figura 2.2.

O código de autorização oferece alguns benefícios importantes em termos de segurança, como a capacidade de autenticar o cliente, bem como a transmissão do *token* de acesso diretamente ao cliente, sem passá-lo pelo agente de usuário do proprietário de recursos e potencialmente expô-lo a terceiros, incluindo o proprietário de recursos (RO).

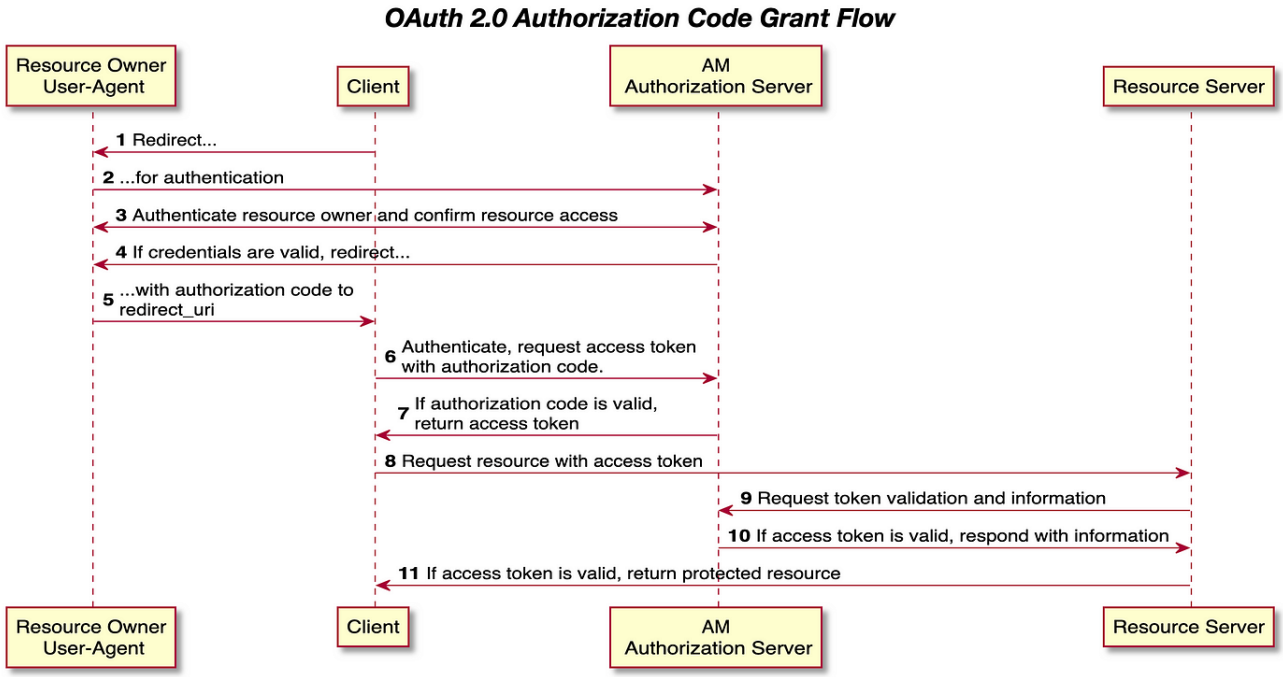

Figura 2.2: *Authorization code flow [2]*

2.3.2 Autenticação

A autenticação é o processo de confirmação da identidade por meio de algum tipo de credencial. A autenticação, quando bem implementada, consiste em provar que você é quem diz ser [29]. Credenciais, por sua vez, são objetos digitais que formam uma combinação de informações que serve para autenticar a identidade de um usuário em um sistema ou aplicação. Essas informações podem incluir um nome de usuário (ou identificador), como um endereço de e-mail, nome de usuário único, uma senha, PIN ou outro tipo de segredo compartilhado, como um token. [30].

As credenciais fornecidas pelos usuários são divididas em alguns tipos, sendo as mais conhecidas: *something you know*, que é amplamente utilizado e consiste no fornecimento de informações que apenas aquele usuário deve saber, como uma senha. *something you are* que é fornecido através de uma informação particular da anatomia corporal do usuário que queira se autenticar [31], um exemplo é o *Optic ID* implementado no *Apple Vision Pro* que identifica seus usuário através da íris [32] dos olhos e *something you have* que consiste em provar a autenticidade baseado em algo que você tem como um *token* ou *smart card* [33].

2.3.2.1 FIDO

Apesar dos esforços realizadas pela aliança FIDO (*Fast IDentity Online*) que, por sua vez, foi fundada com o intuito de promover a padronização dos processos de autenticação, seu protocolo se encontra na sua segunda versão (FIDO2) que uniformiza o reconhecimento de identidades reduzindo dependência de credencias do tipo *something you know* (senhas). Para alcançar isso, o protocolo do FIDO2 utiliza tanto a

criptografia quanto a autenticação tradicional [34] no seguinte processo:

1. O usuário se registra em um serviço de gerenciamento de identidade como um usuário FIDO2, e o serviço gera um par de chaves criptográficas.
2. A chave privada é armazenada em um dispositivo, e a chave pública é registrada no serviço.
3. A autenticação é mapeada para um ou mais serviços. Essas abordagens podem incluir qualquer biometria, PINs ou chaves físicas. O mais importante é que essas formas de verificação de identidade não são enviadas ao serviço em si. Em vez disso, a autenticação permanece no dispositivo, e o dispositivo autentica o usuário com a chave secreta

O que torna esse padrão seguro é que as informações de verificação nunca deixam o dispositivo. A chave permanece no equipamento, e a autenticação só pode ocorrer por meio da posse física desse dispositivo. Ainda é possível observar dificuldades de padronizar implementações de autenticação que sejam *passwordless*, este fato se deve às principais barreiras de adoção serem recuperação de contas, mudança na experiência dos usuários, complexidades técnicas, requisitos regulatórios e cultura de segurança [35].

2.3.2.2 OIDC

As especificações do *framework* de Autorização *OAuth 2.0* fornecem um quadro geral para que aplicativos de terceiros obtenham e utilizem acesso limitado a recursos HTTP. Elas definem mecanismos para obter e utilizar *tokens* de acesso para acessar recursos, mas não definem métodos padrão para fornecer informações de identidade. Notavelmente, sem a criação de perfis específicos para o *OAuth 2.0*, ele não é capaz de fornecer informações sobre a autenticação de um usuário final.

O *OpenID Connect* (OIDC) é uma camada de identidade construída sobre o *framework OAuth2.0*, permitindo que aplicações de terceiros verifiquem a identidade do usuário final e obtenham informações básicas do perfil do usuário. O OIDC utiliza JSON Web Tokens (JWTs), que podem ser obtidos por meio de fluxos que seguem as especificações do *OAuth2.0* [36]. O uso dessa extensão é solicitado pelos clientes ao incluir o valor do escopo "openid"na solicitação de autorização. As informações sobre a autenticação realizada são retornadas em um *JWT* chamado Token de Identificação (*ID Token*). Servidores de Autenticação do *OAuth 2.0* que implementam o *OpenID Connect* também são referidos como Provedores *OpenID*.

O Fluxo de código de autorização (*authorization code flow*) retorna um código de autorização ao cliente, que pode então ser trocado diretamente por um *ID Token* e um *Access Token*. Isso oferece a vantagem de não expor nenhum token ao *user agent* e, possivelmente, a outras aplicações maliciosas com acesso ao Agente de Usuário. O servidor de autorização também pode autenticar o cliente antes de trocar o código de autorização por um Token de Acesso. O fluxo de código de autorização é adequado para clientes que podem manter de forma segura um segredo de cliente entre eles e o servidor de autorização. Na autenticação utilizando o *authorization code flow* especificado na RFC do OAuth2.0, temos os seguintes passos:

1. O cliente prepara uma solicitação de autenticação contendo os parâmetros de solicitação desejados.
2. O cliente envia a solicitação para o servidor de autorização.

3. O servidor de autorização autentica o Usuário Final.
4. O servidor de autorização obtém o consentimento/autorização do usuário final.
5. O servidor de autorização envia o usuário final de volta ao cliente com um código de autorização.
6. O cliente solicita uma resposta usando o código de autorização no *endpoint* do token.
7. O cliente recebe uma resposta que contém um token de Identificação (ID Token) e um token de acesso no corpo da resposta.
8. O cliente valida o token de identificação e recupera o identificador do usuário Final.

2.4 MODELOS DE CONTROLE DE ACESSO

Nesta seção será enfatizado alguns modelos de controle de acesso em ambientes colaborativos, evidenciando suas características, vantagens e desvantagens em sua implementação.

2.4.1 Modelo MAC (*Mandatory Access Control*)

O modelo MAC é um método de limitar o acesso a recursos com base na sensibilidade das informações contidas no recurso e na autorização do usuário para acessar informações com esse nível de sensibilidade. Neste modelo, é possível definir a sensibilidade do recurso por meio de um rótulo de segurança. O rótulo de segurança, por sua vez, indica um nível ou classificação hierárquica das informações (por exemplo, restrito, confidencial ou Interno). A categoria de segurança define a categoria ou grupo ao qual as informações pertencem (como Projeto A ou Projeto B). Os usuários só podem acessar as informações em um recurso para o qual seus rótulos de segurança os autorizem. Se o rótulo de segurança do usuário não tiver autoridade suficiente, o usuário não poderá acessar as informações no recurso.

2.4.2 Modelo de Matriz de Acesso

O modelo de matriz de acesso é um padrão conceitual que especifica os direitos que cada sujeito possui para cada objeto. A matriz de acesso [37] fornece um quadro útil para descrever a proteção de recursos em sistemas operacionais. Este modelo define três tipos de entidades de controle de acesso: a) objetos protegidos — as entidades/recursos que podem ser acessados; b) sujeitos — as entidades ativas que acessam objetos; e c) direitos de acesso que associam o sujeito aos objetos protegidos, especificando as operações que os sujeitos estão autorizados a realizar nos objetos. Uma matriz de acesso A, com linhas representando sujeitos e colunas representando objetos, é usada para definir o estado de proteção. A[s, o] denota os direitos de acesso que um sujeito **s** tem sobre um objeto **o**. A regra de verificação de acesso do modelo afirma que uma solicitação feita pelo sujeito **s** para acessar o objeto **o** só é concedida se A[s, o] contiver o direito necessário. Isso é alcançado por meio de um monitor de referência responsável por mediar todas as operações tentadas pelos sujeitos em objetos. Um exemplo de matriz de acesso pode ser observado na figura 2.3.

	File 1	File 2	File 3	File 4
John	Own R W		Own R W	
Alice	R	Own R W	W	R
Bob	R W	R		Own R W

Figura 2.3: Modelo de Matriz de Acesso [3].

Dentre as desvantagens do modelo por matriz de acesso estão:

- Políticas de acesso mais sofisticadas, como acesso baseado em competência, princípio do mínimo privilégio ou regras de conflito de interesses, são difíceis de serem fornecidas sem direitos de acesso associados às credenciais de um sujeito ao realizar uma operação.
- Os usuários podem mudar de responsabilidades a qualquer momento. Abordagens baseadas em ACL (Listas de Controle de Acesso) e baseadas em lista de capacidades (C-lists) carecem da capacidade de suportar mudanças dinâmicas nos direitos de acesso. Para a ACL de um objeto, é fácil determinar a quais modos de acesso os sujeitos têm autorização para esse objeto. Ou seja, as ACLs permitem uma revisão conveniente do acesso em relação a um objeto. Também é fácil revogar todos os direitos de acesso a um objeto substituindo a ACL existente por um modo de acesso vazio. Por outro lado, determinar todos os acessos que um sujeito possui é difícil em um sistema baseado em ACL. É necessário examinar a ACL de cada objeto no sistema para revisar os privilégios de acesso em relação a um sujeito. Da mesma forma, se todos os acessos de um sujeito precisarem ser revogados, todas as ACLs devem ser verificadas uma por uma. Em uma abordagem de lista de capacidades, é fácil revisar todos os acessos que um sujeito está autorizado a executar, revisando a lista de capacidades do sujeito. No entanto, determinar todos os sujeitos que têm permissão para acessar um objeto específico requer a revisão de cada lista de capacidades de cada sujeito.
- Em um cenário de fluxo de trabalho colaborativo ou organizacional, a propriedade pode não estar a critério do usuário, ou seja, o sistema pode ser o proprietário dos recursos. ACLs e listas de capacidades abordam inadequadamente essa questão. Os direitos de acesso podem estar relacionados ao conteúdo, atributo dos recursos ou outras informações contextuais. As matrizes de acesso não desconsideram essa situação.

2.4.3 Modelo DAC (*Discretionary Access Control*)

A ideia central do DAC é que o proprietário de um objeto, geralmente seu criador, tem autoridade discricionária sobre quem mais pode acessar esse objeto [3]. Em outras palavras, a política central do DAC é a administração baseada no proprietário dos direitos de acesso. Existem muitas variações da política do DAC, especialmente no que diz respeito a como o poder discricionário do proprietário pode ser delegado a outros usuários e como o acesso é revogado [38].

Embora o DAC seja simples e amplamente utilizado em muitos sistemas, o modelo apresenta algumas limitações em termos de administração, segurança e escalabilidade em ambientes complexos, como:

- **Falta de Centralização e Consistência:** No DAC, a autoridade para conceder ou revogar direitos de acesso é deixada nas mãos dos proprietários dos objetos. Isso pode levar a uma falta de centralização e consistência na administração de direitos. Cada proprietário pode tomar decisões de acesso de maneira independente, o que pode resultar em políticas de acesso variadas e desorganizadas em todo o sistema. Isso torna a administração de direitos mais difícil de controlar e auditar.
- **Fragilidade da Segurança:** A segurança no DAC depende da capacidade dos proprietários em tomar decisões de acesso apropriadas. Se um proprietário negligenciar a configuração de permissões corretas ou se o sistema for comprometido por um atacante que ganha controle sobre a conta de um proprietário, a segurança do sistema pode ser comprometida. Isso torna o DAC vulnerável a erros humanos e a ameaças internas.
- **Dificuldade de Implementação em Grandes Organizações:** Em organizações de grande porte, a administração do DAC pode se tornar complexa. À medida que o número de objetos e usuários aumenta, a administração de permissões de maneira descentralizada se torna um desafio. A delegação de autoridade para proprietários individuais pode levar a uma sobrecarga administrativa significativa, e a manutenção de políticas de controle de acesso coerentes pode ser difícil.

2.4.4 Modelo RBAC (*Role Based Access Control*)

A essência do RBAC [39] é que as permissões são atribuídas as funções (ou papéis) em vez dos usuários individuais. Os papéis são criados para várias funções de trabalho, e os usuários são atribuídos a papéis com base em suas qualificações e responsabilidades. Dessa forma, a tarefa de especificar a autorização do usuário é dividida em duas partes logicamente independentes: uma que atribui usuários a papéis e outra que atribui direitos de acesso a objetos aos papéis, como ilustrado na 2.4.

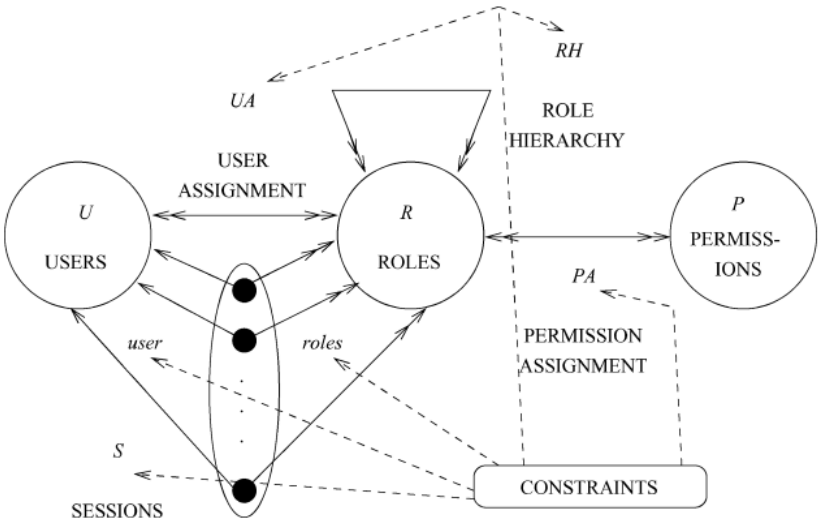

Figura 2.4: Modelo RBAC. [4]

A família de modelos RBAC suporta a noção de ativação de papéis em sessões, onde a sessão é um conceito ligado a um único usuário e permite que o usuário ative as permissões de um subconjunto de papéis aos quais ele pertence. Do ponto de vista de políticas, a capacidade dentro do RBAC de impor restrições à adesão de usuários atribuindo-os a papéis fornece um meio poderoso de fazer cumprir regras de conflito de interesses e regras de cardinalidade para papéis, à medida que se aplicam exclusivamente a um ambiente colaborativo. Os usuários podem ser facilmente reatribuídos de um papel para outro sem modificar a estrutura de acesso subjacente. Portanto, o RBAC é mais escalável do que as especificações de segurança baseadas em usuários e reduz significativamente o custo e a sobrecarga administrativa associados à administração de segurança detalhada no nível de usuários individuais, objetos ou permissões.

Dentre as desvantagens do modelo RBAC estão:

- A maioria das implementações iniciais do RBAC determina o conjunto de papéis em uso, bem como a adesão a esses papéis no início da sessão. Mudanças não eram bem suportadas e a natureza desses papéis poderia ser chamada de estática, pois faltavam flexibilidade e capacidade de resposta ao ambiente em que eram usados.
- O RBAC tradicional não possui a capacidade de especificar um controle detalhado sobre usuários individuais em determinados papéis e sobre instâncias individuais de objetos. Para ambientes colaborativos, é insuficiente ter permissões de papel baseadas em tipos de objetos. Muitas vezes, o usuário em uma instância de um papel pode precisar de uma permissão específica em uma instância de um objeto.
- Outra questão importante na implementação do modelo RBAC é o poder de especificação de restrições. Restrições são um aspecto importante do controle de acesso baseado em papéis e um mecanismo poderoso para estabelecer políticas organizacionais em níveis mais altos.

2.4.5 Modelo ABAC (*Attribute-Based Access Control*)

ABAC [40] é um modelo de controle de acesso baseado nos atributos das entidades. As entidades incluem sujeito, objeto, atributo, operação, política e condição do ambiente. Um sujeito representa um usuário humano ou um dispositivo que solicita uma operação a ser realizada em um objeto. Um objeto representa um recurso do sistema (por exemplo, arquivos, processos) em que uma operação é executada. Um atributo captura uma característica de um sujeito ou de um objeto. Uma operação é uma função a ser executada em um objeto. As operações incluem ler, escrever, editar, excluir, copiar, executar e modificar. Uma condição do ambiente representa um contexto situacional em que ocorrem solicitações de acesso. Ela captura uma característica ambiental detectável independente de sujeitos e objetos e pode incluir informações de tempo, localização do usuário e nível de ameaça. Uma política é um conjunto de regras para determinar uma decisão de acesso com base nos valores dos atributos do sujeito, objeto e condições do ambiente. O administrador ou proprietário de um objeto é responsável por definir as regras de controle de acesso para o objeto.

O ABAC permite a gestão dinâmica do controle de acesso alterando os valores dos atributos sem alterar as relações entre sujeitos e objetos que definem as políticas de controle de acesso subjacentes. Isso significa

que a decisão de acesso para duas solicitações iguais pode ser diferente em momentos diferentes. O ABAC também é escalável. Com suporte de seu mecanismo flexível, o ABAC consegue lidar com inúmeras políticas e solicitações, desobrigando a modificação de regras e atributos de objetos existentes quando um novo sujeito é adicionado, oferecendo grande flexibilidade na gestão.

Dentre as desvantagens da utilização do modelo ABAC, estão:

- **Complexidade de configuração:** A complexidade de configuração no ABAC se deve ao fato de que as políticas de acesso são definidas com base em atributos específicos de sujeitos, objetos e condições de ambiente. Configurar corretamente essas políticas requer uma compreensão profunda dos atributos envolvidos, bem como das relações entre eles. Isso pode ser especialmente desafiador em ambientes com uma variedade de atributos complexos e diversos. Erros na configuração das políticas podem resultar em decisões de acesso incorretas e violações de segurança.
- **Gerenciamento de políticas:** À medida que o número de políticas ABAC aumenta, o gerenciamento e a manutenção dessas políticas se tornam uma tarefa trabalhosa. Garantir que as políticas sejam consistentes, coesas e não entrem em conflito é fundamental para o funcionamento eficaz do sistema. Manter um grande conjunto de políticas requer monitoramento constante, atualizações regulares e garantir que as políticas permaneçam alinhadas com os requisitos organizacionais. Conflitos entre políticas podem levar a decisões de acesso inconsistentes ou imprevistas.
- **Desempenho:** A avaliação de políticas ABAC pode ser intensiva em recursos, especialmente em ambientes com muitas políticas e atributos. Isso ocorre porque o sistema precisa analisar os atributos de sujeitos, objetos e condições de ambiente para tomar decisões de acesso. Em cenários com cargas de trabalho pesadas, essa avaliação constante de políticas pode afetar o desempenho do sistema, causando atrasos e gargalos. Garantir que o sistema seja escalável e eficiente é um desafio importante.
- **Compatibilidade:** Integrar o ABAC em sistemas existentes pode ser complicado, especialmente quando se trata de sistemas legados que não foram projetados para suportar o ABAC. Isso pode exigir adaptações substanciais nos sistemas legados ou até mesmo substituições. Além disso, a compatibilidade entre diferentes implementações de ABAC também pode ser um problema, já que as definições e formatos de atributos podem variar. Isso pode dificultar a interoperabilidade entre sistemas que usam ABAC.

2.5 MICROSSERVIÇOS

Os microsserviços tem se tornado uma opção cada vez mais popular entre os desenvolvedores tornando-se o padrão de escolha no planejamento das arquiteturas de aplicações Segundo a FORBES [41], o mercado de microsserviços está projetado para triplicar entre os anos de 2020 a 2026. Uma das vantagens inerentes a utilização dos microsserviços é que eles podem ser introduzidos de forma independente e modelados baseados nas regras de negócio, fazendo com que um serviço encapsule uma funcionalidade e a torne acessível a outros serviços através de uma conexão de rede.

Do ponto de vista externo, um microsserviço pode ser tratado como uma caixa-preta, hospedando uma funcionalidade de negócios em um ou mais *endpoints*, que podem ser acessados através de qualquer protocolo que seja apropriado. Os detalhes internos de implementação da tecnologia permanecem totalmente ocultos ao ambiente externo, fazendo com que no processo de desenvolvimento de sistemas seja possível a divisão em equipes menores, separando-as de acordo com a *expertise* de cada uma e, posteriormente, juntando cada pedaço (microsserviço) que constitui uma aplicação completa.

Outras vantagens em adotar um design baseado em microsserviços são que [42] cada serviço de processamento de dados individual pode ser implantado próximo ao sistema, aplicação ou usuário monitorado [43], podendo ser dimensionado em termos da quantidade de informações que precisa processar; independentemente dos outros microsserviços. Além disso, [44] o design descentralizado espalha o risco de violações de dados e reduz a capacidade de um intermediário de dados capturar informações sensíveis, uma vez que um único microsserviço fornece acesso a um subconjunto dos dados. De fato, as aplicações monolíticas preferem um único banco de dados para dados persistentes, enquanto os microsserviços gerenciam seu próprio banco de dados em cima de, possivelmente, diferentes tecnologias de banco de dados. No entanto, uma consequência da governança descentralizada e do gerenciamento de dados é que a imposição de controle de acesso nos pontos de extremidade de dados para evitar vazamentos de informações ou preservar a privacidade torna-se consideravelmente mais desafiadora. Se por um lado, as políticas de autorização devem restringir o acesso aos pontos de extremidade de dados dos próprios microsserviços, por outro, eles também dependem dos mesmos pontos de extremidade de dados para avaliar suas decisões de controle de acesso.

2.5.1 *Containers Docker*

Uma das formas de implementação de microsserviços é através de *containers* que, por sua vez é uma unidade padrão de *software* que empacota o código e todas as suas dependências para que a aplicação funcione de maneira rápida e confiável em diferentes ambientes de computação, retirando, em alguns casos, a necessidade de virtualização a nível de sistema operacional [45]. Uma imagem de contêiner *Docker* é um pacote leve, independente e executável de software que inclui tudo o que é necessário para executar uma aplicação: código, tempo de execução, ferramentas do sistema, bibliotecas do sistema e configurações. A diferença de aplicações levantadas através de máquinas virtuais e contêineres pode ser observada na figura 2.5.

As imagens de contêiner se tornam contêineres durante a execução e, no caso dos contêineres *Docker*, as imagens se tornam contêineres quando são executadas no *Docker Engine*. Disponíveis tanto para aplicativos baseados em Linux quanto para *Windows*, o software em contêiner sempre será executado da mesma forma, independentemente da infraestrutura. Os contêineres isolam o software de seu ambiente e garantem que ele funcione de maneira uniforme, independentemente das diferenças, como as que podem ocorrer entre ambientes de desenvolvimento e produção.

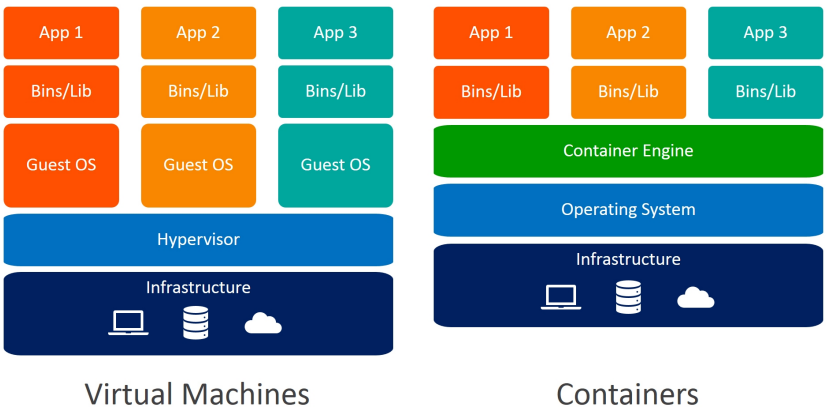

Figura 2.5: Comparativo de camadas computacionais entre máquinas virtuais e contêineres [5]

2.6 *REST APIS*

O estilo arquitetônico *Representational State Transfer (REST)* estabelece um conjunto de regras para o design de sistemas distribuídos que têm orientado a concepção e desenvolvimento da Web conforme a conhecemos. Serviços web que seguem o estilo arquitetônico REST são chamados de serviços web *RESTful*, enquanto as interfaces programáticas desses serviços são denominadas APIs REST. Os princípios que orientam o design de APIs REST, em grande parte, resultam das escolhas arquiteturais da Web com o objetivo de fomentar a escalabilidade e a robustez de sistemas orientados a recursos em rede, baseados no protocolo HTTP [46]. Os princípios centrais incluem [47]:

- **Identificação de recursos:** as APIs gerenciam e disponibilizam recursos que representam conceitos do domínio; cada recurso é unicamente identificado e acessível por meio de um Identificador Uniforme de Recursos (URI) adequado.
- **Representação de recursos:** os clientes não têm conhecimento direto do formato interno e do estado dos recursos; eles interagem com representações dos recursos (por exemplo, JSON ou XML) que refletem o estado atual ou desejado do recurso. A definição de tipos de conteúdo nos cabeçalhos das mensagens HTTP permite que clientes e servidores processem adequadamente essas representações.
- **Interface uniforme:** os recursos são acessados e manipulados por meio dos métodos padrão definidos pelo protocolo HTTP (GET, POST, PUT, etc.). Cada método possui seu próprio comportamento esperado e códigos de status padrão.
- **Ausência de estado:** as interações entre um cliente e uma API são sem estado, o que significa que cada solicitação contém todas as informações necessárias para ser processada pela API; nenhum estado de interação é mantido no servidor.

No conjunto, esses princípios explicam o termo "transferência de estado representacional": o estado da interação não é armazenado no lado do servidor; ele é transportado (transferido) em cada solicitação do cliente para o servidor e codificado na representação do recurso à qual a solicitação se refere.

2.7 *GATEWAYS* DE APIS

Os *gateways* de API é um dos modos de *setup* na arquitetura de microsserviços. O uso do *gateway* de API pode resolver o problema de como os clientes acessam os microsserviços independentes. Em termos simples, o *gateway* de API é um servidor especial que serve como a única entrada para todos os microsserviços, encapsulando os aspectos internos do sistema e a implementação específica da interface; por outro lado, ele também possui funcionalidades como verificação de permissão, balanceamento de carga, armazenamento em cache e monitoramento [48], podendo sua topologia ser observada na figura 2.6.

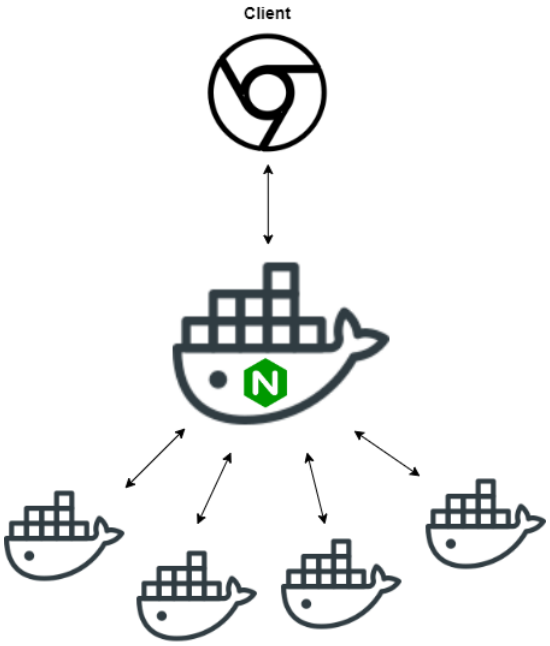

Figura 2.6: Topologia com *gateway* de API

Com mais foco no tráfego norte-sul, a principal preocupação de um *gateway* de API em um ambiente de microsserviços é mapear as requisições de componentes externos para os microsserviços internos. Essa responsabilidade é semelhante ao que poderia se ter com um simples *proxy* HTTP e, com efeitos, os gateways de API em geral disponibilizam outras funcionalidades com base em *proxies* HTTP já existentes.

Dentro das vantagens da utilização de *gateways* de API estão:

- **Gerenciamento de tráfego:** Os *gateways* de API permitem o roteamento inteligente do tráfego para diferentes versões ou instâncias de uma API, bem como para servidores ou microsserviços subjacentes. Isso facilita a implantação gradual e o gerenciamento de atualizações. Além disso, os *gateways* de API podem realizar balanceamento de carga entre servidores para distribuir o tráfego de maneira eficaz e garantir alta disponibilidade.
- **Segurança aprimorada:** Os *gateways* de API permitem a implementação de autenticação robusta e mecanismos de autorização para controlar o acesso às APIs. Podem também incluir recursos de segurança de *WAF (Web Application Firewall)*, como proteção contra ataques de injeção SQL, ataques

de negação de serviço (DDoS) e outras ameaças comuns à segurança das APIs, podendo filtrar e bloquear tráfego malicioso antes que ele alcance suas APIs. Os *gateways* de API também simplificam a gestão de certificados SSL/TLS, garantindo que as comunicações entre clientes e APIs sejam criptografadas e seguras, retirando a responsabilidade dos microsserviços subjacentes ao gerenciamento de certificado.

- **Monitoramento e análise:** Os *gateways* de API registram detalhes sobre as solicitações e respostas das APIs, permitindo uma visibilidade completa do tráfego da API. Isso facilita a solução de problemas e a análise de desempenho. Na análise de tráfego, também podem fornecer um ensaio detalhado sobre o uso da API, incluindo informações sobre taxas de sucesso, tempos de resposta e padrões de tráfego. Esses dados podem então ser utilizado para a otimização de performance e tomadas de decisão.

- **Transformação e personalização de dados:** Os *gateways* de API podem executar transformações de dados em tempo real, como mapeamento de campos, agregação e conversão de formatos. Isso facilita a integração de sistemas com diferentes estruturas de dados. Além disso, também podem incluir funcionalidades de personalização de respostas, permitindo alterar as respostas das APIs com base nas necessidades dos clientes, o que é útil para fornecer dados sob medida para diferentes aplicativos ou dispositivos.

2.8 AUTENTICAÇÃO BASEADA EM *TOKEN*

Dentro do processo comum para o gerenciamento de sessões de usuário em uma aplicação web, o cliente realiza uma requisição HTTP com o método POST onde o mesmo fornece suas credenciais para a aplicação. A aplicação então verifica com o seu banco de dados para confirmar que as informações de login estão corretas. Caso positivo, a aplicação gera um ID de Sessão, o associa ao usuário e anexa esse dado na resposta ao cliente através de um *web cookie*. A partir disso sempre que o cliente realiza uma solicitação à aplicação, ele inclui o *cookie* com sua solicitação. A aplicação então verifica o ID de Sessão com o banco de dados para garantir que o usuário está autenticado. Embora esse processo funcione, ele pode ser problemático para aplicações com alto tráfego, pois pode diminuir o desempenho na validação com o banco de dados por requisição. Além disso, essa abordagem não é adequada para sistemas distribuídos baseados em microsserviços, que requerem autenticação e autorização contínuas. Além da baixa performance, o gerenciamento de sessão com *cookies*, quando mal configurado, pode ser vulnerável a ataques como *Cross-site Scripting* [49]. Uma solução mais adequada é usar sistemas de autorização baseados em tokens, que permitem a autorização sem a necessidade de verificar o banco de dados a cada nova solicitação. Com essa abordagem, credenciais de curto prazo (tokens) são trocadas, em vez de credenciais de longo prazo como senhas, e os métodos de verificação de autenticidade e integridade gerados pelo HMAC são utilizados para garantir a segurança [50].

A autenticação baseada em tokens é um processo que gera tokens de segurança criptografados, permitindo que os usuários verifiquem sua identidade. Esse token concede aos usuários acesso a páginas e recursos protegidos por um período limitado de tempo, sem a necessidade de reintroduzir seu nome de

usuário e senha.

Os processos da utilização de tokens de autenticação se resumem com o usuário iniciando o procedimento de login em um serviço utilizando suas credenciais, o que desencadeia a uma solicitação de acesso a um servidor ou recurso protegido. O servidor, então, realiza um processo de verificação, validando as informações de login fornecidas e assegurando que a senha inserida corresponda ao nome de usuário correspondente. Após a verificação bem-sucedida, o servidor gera um token de autenticação seguro e assinado, válido por um período específico. Esse token é transmitido de volta para o navegador do usuário, onde é armazenado para visitas futuras ao site. Conforme o usuário acessa novos sites, o token de autenticação armazenado é decodificado e verificado. Se as informações estiverem corretas, o usuário obtém acesso aos recursos solicitados. Importante destacar que o token permanece ativo até que o usuário faça logout ou encerre a sessão, proporcionando um método de autenticação simplificado, porém seguro, com limite de tempo. Dentre as características dos tokens de autenticação, estão [51]:

- **Tokens são stateless:** Tokens de autenticação são criados por um serviço de autenticação e contêm informações que permitem a um usuário verificar sua identidade sem precisar inserir credenciais de login.
- **Tokens expiram:** Quando um usuário encerra sua sessão de navegação e faz logout do serviço, o token que lhe foi concedido é destruído. Isso garante que as contas dos usuários estejam protegidas e não estejam em risco de ciberataques.
- **Tokens utilizam criptografia:** A autenticação baseada em tokens utiliza códigos criptografados e gerados por máquina para verificar a identidade de um usuário. Cada token é único para a sessão de um usuário e é protegido por um algoritmo, o que garante que os servidores possam identificar um token que foi adulterado e bloqueá-lo. A criptografia oferece uma opção muito mais segura do que depender de senhas.
- **Tokens simplificam o processo de login:** Tokens de autenticação garantem que os usuários não precisem reentrar com suas credenciais de login toda vez que visitam um site. Isso torna o processo mais rápido e amigável ao usuário, mantendo as pessoas nos sites por mais tempo e as incentivando a visitar novamente no futuro.

2.8.1 *JSON Web Token (JWT)*

O *JSON Web Token*, ou JWT, é um padrão aberto descrito na RFC 7519 que define um formato compacto de representação de declarações destinado a ambientes com restrição de espaço, como cabeçalhos de autorização HTTP e parâmetros de consulta de *URI*. *JWTs* codificam declarações a serem transmitidas como um objeto JSON que é usado como o *payload* de uma estrutura *JSON Web Signature (JWS)* ou como o texto simples de uma estrutura *JSON Web Encryption (JWE)*, possibilitando que as declarações sejam assinadas digitalmente ou protegidas quanto à integridade com um Código de Autenticação de Mensagem (MAC) e/ou criptografadas. *JWTs* são sempre representados usando a Serialização Compacta *JWS* ou a Serialização Compacta JWE [52].

É possível observar na figura 2.7, o objeto *JSON* codificado em Base64 que contém uma carga útil de informações (*payload*) em roxo, como dados de usuário, bem como uma assinatura digital (em azul) para garantir que as informações não tenham sido alteradas durante a transmissão. O *JWT* é amplamente utilizado na autenticação e autorização de usuários em aplicativos web e serviços de API, pois é portátil, escalável e seguro. Além disso, o *JWT* pode ser facilmente integrado com outras tecnologias de segurança, como *HTTPS, SSL* e *OAuth*.

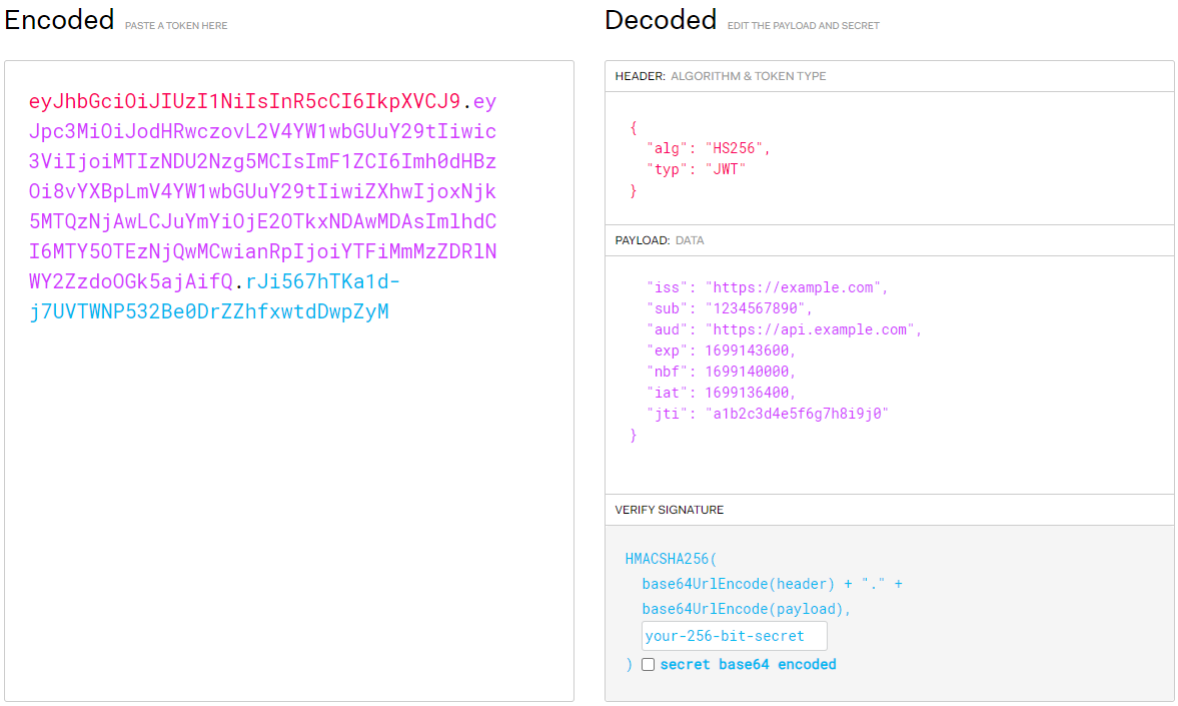

Figura 2.7: Exemplo de JSON Web Token

Além disso, o *JWT* vem em duas formas: simétrica e assimétrica. No *JWT* simétrico, o *hash* da concatenação do cabeçalho e *payload* é assinado usando uma chave secreta, que pode ser gerada usando algoritmos *SHA-256, SHA-384 e SHA-512*. A mesma chave é usada para assinar e validar o *token*. Por outro lado, o *JWT* assimétrico usa um par de chaves pública/privada. A chave privada assina o *token*, enquanto a chave pública o valida, podendo ser utilizado os algoritmos *RSA*, *ECDSA* e *RSASSA-PSS* para gerar o par de chaves. A chave privada assina digitalmente a mensagem, enquanto a chave pública verifica sua integridade.

Os algoritmos de criptografia mais utilizados nas implementações de *JWTs* são o HS256 e o RS256. O *HS256* (*HMAC* com *SHA-256*) é um algoritmo de hash com chave simétrica que usa uma única chave secreta, sendo a chave usada tanto para gerar a assinatura quanto para validá-la. Já o *RS256* (Assinatura *RSA* com *SHA-256*) é um algoritmo assimétrico que utiliza um par de chaves pública/privada. O provedor de identidade possui uma chave privada para gerar a assinatura, enquanto o receptor do *JWT* usa uma chave pública para valida-la

As informações contidas no JWT são chamadas de *claims* e, alguns deles, são padrões registrados na RFC como:

- **"iss"(*Issuer*):** identifica o emissor do token.
- **"sub"(*Subject*):** identifica o sujeito do JWT, geralmente o ID exclusivo do usuário ou da entidade.
- **"aud"(*Audience*):** especifica a audiência pretendida do JWT, ou seja, para quem o token se destina.
- **"exp"(*Expiration Time*):** indica a data e hora de expiração do JWT. Após esse momento, o token não é mais válido.
- **"nbf"(*Not Before*):** especifica a data e hora a partir da qual o JWT pode ser considerado válido.
- **"iat"(*Issued At*):** indica a data e hora em que o JWT foi emitido.
- **"jti"(*JWT ID*):** fornece um ID exclusivo para o JWT, que pode ser útil para evitar a reutilização do token.

2.8.2 *JSON Web Key Set (JWKS)*

JSON Web Key Set (JWKS) é um ponto de extremidade somente de leitura que fornece o conjunto de chaves públicas do servidor de autorização no formato *JWKS*. Este conjunto contém a(s) chave(s) de assinatura que a Parte Confiante utiliza para validar as assinaturas do servidor. O principal benefício de permitir a configuração do ponto de extremidade *JWKS* é a sua capacidade de lidar com a rotação de chaves por provedores de identidade externos. Configurar este ponto de extremidade permite descobrir programaticamente as chaves da *web JSON*, permitindo que os provedores de identidade de terceiros publiquem novas chaves sem a sobrecarga de notificar cada aplicação cliente individualmente. Na figura 2.8 pode ser observado o padrão do conteúdo do arquivo *JSON JWKS*, sendo seus parâmetros descritos na tabela 2.1


```
{
   "keys":[
      {
         "kty":"RSA",
         "e":"AQAB",
         "use":"sig",
         "kid":"ZjRmYTMwNTJjOWU5MmIzMjgzNDI3Y2IyMmIyY2EzMjdhZjViMjc0Zg",
         "alg":"RS256",
         "n":"j3mYMT8N2SR8cpimdMOTpk_M8fOxPF1BHQAiCtld4nbksILgJKsA34GSP5Oh4gLI
      },
      {
         "kty":"RSA",
         "e":"AQAB",
         "use":"sig",
         "kid":"ZjRmYTMwNTJjOWU5MmIzMjgzNDI3Y2IyMmIyY2EzMjdhZjViMjc0Zg_RS256"
         "alg":"RS256",
         "n":"j3mYMT8N2SR8cpimdMOTpk_M8fOxPF1BHQAiCtld4nbksILgJKsA34GSP5Oh4gLI
      }
   ]
}
```


Figura 2.8: Arquivo JSON inserido no *JWKS endpoint*

Tabela 2.1: Descrição dos valores contidos no arquivo JWKS

kty	Tipo de chave pública
e	O valor do expoente da chave pública
use	Indica como a chave está sendo usada. O valor "sig"representa uma assinatura.
kid	Este valor é usado para identificar a chave que deve ser usada para verificar a assinatura.
alg	O algoritmo usado para garantir a Assinatura Web JSON.
n	O valor do módulo da chave pública

2.8.3 PASETO

O Platform Agnostic Security Token (PASETO) compartilha semelhanças com o JWT, mas impõe um conjunto rigoroso de diretrizes sobre a seleção de algoritmos criptográficos disponíveis para os desenvolvedores. Essa característica é consideravelmente vantajosa porque o JWT fornece uma abundância de opções, algumas das quais, inseguras. O PASETO é composto por três elementos obrigatórios (versão, propósito e carga útil) e um elemento opcional (rodapé). Assim como os JWTs, Os tokens do tipo PASETO são codificados em base64 antes de serem transmitidos [53]. Desde a publicação inicial do artigo, o PASETO passou por quatro versões; no entanto, apenas as versões 3 e 4 são atualmente empregadas e recomendadas.

A carga útil do token PASETO é o conteúdo que autoriza a identidade do cliente para a aplicação. Além disso, a estrutura do PASETO inclui um rodapé opcional que permite a transferência de metadados não criptografados. Esse recurso pode ser útil no gerenciamento de funções de rotação de chaves.

O elemento *purpose* do PASETO pode ser configurado como local ou público. A opção local é recomendada em cenários onde uma chave secreta compartilhada pode ser armazenada com segurança entre os endpoints que validarão o token. O token é criado e assinado com a mesma chave secreta compartilhada entre os sistemas e sempre transmitido de forma criptografada. Por outro lado, quando o *purpose* do PASETO é público, pares de chaves são gerados com a API *libsodium* e transferidos em texto claro, mas sempre assinados. Portanto, a aplicação deve evitar enviar informações sensíveis do usuário por tokens.

Quando o PASETO é usado para fins locais, uma função segura gera um número aleatório R que é criptografado com o algoritmo *Blake2b* e gera um *nonce*. O cabeçalho do PASETO *Ph* é combinado com o *nonce* e o rodapé (se presente) para fazer uma *string* de pré-autenticação PA. A carga útil do token *Tp* é criptografada usando *xChaCha20-Poly1305*, que é uma variante do *ChaCha20* com um *nonce* estendido, permitindo que *nonces* aleatórios sejam seguros, usando a chave secreta Ks com a *string* de pré-autenticação [54].

$$nonce = Hb(R);$$
$$P\,A = Ph + nonce;$$
$$Tp = Ecc[(Ks + P\,As);\ JSON\ ];$$

O PASETO também pode ser usado com propósito público. O processo na geração de tokens acontece usando a biblioteca *libsodium* para criar um par de chaves pública/privada usando o algoritmo *Ed25519* (Assinatura Digital de Curva *Edwards*). O cabeçalho do PASETO (*v4.public*) é combinado com a carga útil e o rodapé em uma *string* de pré-autenticação (PA). O PA é assinado digitalmente pela chave privada usando *Ed25519* e isso é embalado na carga útil do PASETO no formato *v4.public.signed payload.footer*

do PASETO [54].

$$crypto\ box\ keypair() \rightarrow Ks, Kp;$$
$$P\ A = Header + Payload + Footer;$$
$$HMACe(Ks, P\ A);$$

2.8.4 Macaroon

Macaroons são credenciais de autorização que fornecem suporte flexível para compartilhamento controlado em sistemas descentralizados e distribuídos [55]. *Macaroons* têm um recurso chave de delegação, que permite ao proprietário de um *macaroon* gerar um sub-macaroon com autoridade reduzida. Um *macaroon* é composto por quatro componentes: localização, ID público, ressalvas (*caveats*) e assinatura. O componente de localização é uma *string* de caracteres que especifica a origem do *macaroon* e geralmente é representado por uma URL. O ID público é usado para identificar o *macaroon* e vincula-lo à sua chave privada. As ressalvas podem ser vistas como a carga útil do *macaroon* e afirmam um predicado que deve ser verdadeiro para qualquer solicitação que o *macaroon* derivado autorizará no serviço pretendido. Eles são usados para restringir as capacidades do *macaroon*, indicando quando, onde, por quem e para que propósito o serviço alvo deve conceder acesso. O componente final é a assinatura, que é calculada usando o HMAC com a chave privada salva no servidor. Ao contrário dos JWTs e dos PASETOs, os *Macaroons* não têm uma estrutura definida para transmissão, mas seu conteúdo é seguro para ser comunicado através de cabeçalhos HTTP, *cookies* ou URLs.

Macaroons são um tipo de token de autorização que usa HMACs encadeados. Este design permite que o titular de um *macaroon* adicione ressalvas, ou restrições, ao token sem poder removê-las. Além disso, os *macaroons* permitem a delegação de autorização a terceiros, aderindo ao princípio do menor privilégio e fornecendo autorização granular. Cada *macaroon* é criado começando com um identificador de chave *nonce* aleatório e uma chave secreta *Kts* mantida pelo serviço alvo. *Macaroons* sucessivos são então derivados adicionando predicados de ressalva e encadeando as assinaturas HMAC. Cada *macaroon* autoriza solicitações a um serviço alvo que armazena dados em formato chave-valor, com ressalvas no *macaroon* restringindo o acesso a pedaços específicos de dados e operações. Para verificar a integridade de um *macaroon*, o serviço alvo só precisa verificar sua assinatura contra os valores HMAC que são derivados usando a chave secreta Kts do serviço.

Macaroons podem ser implementados de duas maneiras: com ressalvas do emissor original e com ressalvas de terceiros. Ressalvas do emissor original são usadas quando apenas o emissor original adiciona ressalvas a um *macaroon*. Por sua vez, ressalvas de terceiros permitem que qualquer serviço adicione ressalvas a um *macaroon* garantindo que cada solicitação autorizada por um *macaroon* esteja sujeita a qualquer número de etapas extras para autenticação, autorização ou auditoria.

3 MÉTODO DE PESQUISA E TECNOLOGIAS UTILIZADAS

No contexto da autorização em microsserviços, a pesquisa adotou uma abordagem que considerou a diversidade de implementações de arquiteturas possíveis. Reconhecendo a impossibilidade de resolver todos os desafios de segurança com uma única técnica ou recurso em infraestruturas de TI, foram explorados mecanismos adaptáveis às necessidades dos sistemas de comunicação. Três cenários arquiteturais foram identificados: o primeiro, enfocando a comunicação direta entre microsserviços, onde cada contêiner validava requisições em sua API exposta; o segundo, introduzindo um *gateway* de API que intermediava toda comunicação, sendo um *proxy* reverso responsável pela autorização e roteamento das requisições; e o terceiro, integrando um serviço de nuvem *authorization as a service*, delegando a gestão de tokens e permissões. Esses cenários foram baseados nas melhores práticas da indústria, extraídas de fontes confiáveis, e serviram como base para os testes de implementação.

Para alcançar os objetivos da pesquisa, foram selecionadas tecnologias amplamente reconhecidas e eficazes. A *engine Docker* foi escolhido para a implementação da arquitetura de microsserviços, devido à sua difusão, documentação abundante e facilidade de gerenciamento de dependências, garantindo a escalabilidade ágil, além de ser o padrão utilizado pela industria e rodar em diversas distribuições *linux*. As tecnologias específicas incluíram o *Keycloak* para gestão de identidade e autorização, o *NGINX* como servidor *web* e *proxy* reverso, o serviço de nuvem *Permit.io* para controle de acesso, o *FastAPI* para desenvolvimento rápido de APIs, o *Node.js* para funcionalidades de *front-end* tendo inteiração com o usuário, e o *PostgreSQL* como banco de dados. A escolha do protocolo OIDC (OpenID Connect) para autenticação reforçou a identidade dos usuários nos testes de autorização nos microsserviços, estabelecendo uma base confiável para análises qualitativas posteriormente realizadas.

3.1 MÉTODO DE PESQUISA

Quando se trata sobre autorização em microsserviços, é necessário inserir sobre este contexto os diferentes tipos de implementações de arquiteturas possíveis. Assim como não é possível resolver todos os problemas de segurança empregando uma técnica ou recurso sobre uma infraestrutura de TI, levando em consideração o fato de que todos os ambientes possuem suas particularidades, é fundamental estudar mecanismos que se adaptem às necessidades dos sistemas de comunicação de forma customizada. Portanto, para esta pesquisa de autorização em microsserviços, foi levado em consideração três cenários arquiteturais possíveis: o primeiro visando a comunicação direta entre os microsserviços, sendo cada contêiner responsável por realizar a validação das requisições que alcançam sua API exposta, neste caso é necessário que o código que roda no microsserviço seja capaz de realizar uma análise das requisições, decodificar o *token* recebido e determinar se a requisição possui privilégio suficiente para ter como resposta os dados solicitados. No segundo cenário foi introduzido à arquitetura um *gateway* de API que intermedeia toda a

comunicação que visa tráfego norte-sul, sendo o *gateway* um *proxy* reverso responsável pela autorização e roteamento das requisições para os microsserviços subsequentes, neste caso, não há acoplamento entre código responsável pela autorização e a aplicação em si. Na terceira arquitetura a ser estudada foi introduzido um serviço de nuvem que realiza *authorization as a service*, onde os contêineres enviam para um serviço gerenciado em nuvem o token de autorização e o tipo de ação que a requisição faz referência, autorizando de volta para o microsserviço de acordo com o nível de permissão identificado no token. No serviço gerenciado em nuvem é possível que uma equipe especializada configure políticas de acesso utilizando uma interface gráfica.

Os cenários escolhidos para os testes de implementação foram retirados do livro *Building Microservices, 2021* [16] e reforçados também na pesquisa de Barabanov, 2020 [21] onde é extraído as melhores práticas da indústria em padrões de arquitetura de autenticação e autorização, suas vantagens e desvantagens, bem como sua aplicabilidade, dependendo das características do ambiente. Para cada padrão descrito, foi revisado suas particularidades, as quais podem ser utilizadas como critérios de tomada de decisão para arquitetos de segurança, considerando a implementação de autenticação e autorização em um ambiente orientado a serviços. Vale ressaltar que os microsserviços introduzem novos desafios de segurança, tais como o aumento da superfície de ataque, a redução da eficácia de sistemas de registro tradicionais baseados em arquiteturas centralizadas de agregação de logs, o desarranjo do ciclo de vida de desenvolvimento entre múltiplos componentes da aplicação em vez de uma aplicação monolítica, e o aumento do nível de tráfego devido ao crescente número de comunicações entre microsserviços. Essas considerações destacam a importância de estratégias de segurança sólidas na mitigação dos desafios inerentes à adoção de microsserviços.

Dentro dos três cenários implementados foi realizado o mesmo fluxo de autenticação, utilizando-se do protocolo OIDC (*Open ID Connect*) para garantir a identidade dos usuários utilizados para os testes de autorização nos microsserviços. Após a autenticação por meio das credenciais de usuário e senha, o provedor de identidade (ou servidor de autorização dentro do contexto de *OAuth 2.0*) é responsável por identificar o registro de usuário e associa-lo ao seu devido papel, entregando-o para o seu solicitante um código de autenticação que será utilizado posteriormente para a retirada do *token* de acesso que, por sua vez, irá garantir a obtenção dos dados de acordo com o seu nível de permissão. Todo o fluxo de autenticação e autorização (OIDC/OAuth2.0) foi realizado com o provedor de identidade *Keycloak*, simplificando a implementação de IAM de forma segura e que cumpra com as melhores práticas na gestão de identidades e concessão de privilégios, fazendo com que os usuários se autentiquem no *Keycloak* em vez de em aplicações individuais. Isso significa que as aplicações não precisam lidar com formulários de login, autenticação dos usuários ou armazenamento de informações sensíveis. Uma vez autenticados no *Keycloak*, os usuários não precisam fazer login novamente para acessar uma aplicação diferente.

Na implementação de *JSON Web Tokens* (JWTs), a preocupação com possíveis vulnerabilidades foi abordada de maneira cuidadosa, priorizando as melhores práticas de segurança. Reconhecendo que o JWT é uma peça crítica na autenticação e autorização, foram adotadas, através das bibliotecas *jwt* e *PyJWKClient* medidas preventivas para mitigar possíveis ameaças. Quanto à escolha de algoritmos, a melhor prática é manter uma lista branca *(whitelist)* de algoritmos aceitáveis, minimizando riscos de manipulação do token que, no caso do experimento, foi utilizado RSA-256 gerando uma assinatura assimétrica, o que significa que uma chave privada deve ser utilizada para assinatura do JWT e uma chave pública deve ser utilizada

para verificar a assinatura. Isso garante que a fonte esperada assinou o JWT. Ademais, é crucial verificar a identidade do emissor (*iss claim*) e a audiência (*aud claim*) do token, garantindo que os *tokens* sejam usados conforme pretendido e não confiando de forma precipitada em todas as reivindicações. Quando se trata de reivindicações baseadas em tempo, a prática recomendada é definir tempos de expiração curtos e validar reivindicações como *nbf (not-before)* e *iat (issued at)* para rejeitar *tokens* antigos. Além disso, cuidado especial deve ser tomado ao lidar com reivindicações relacionadas à assinatura, evitando confiar cegamente em valores do cabeçalho do token que podem ser manipulados. Também foi seguido a recomendação de rotação de chaves públicas realizadas pelo provedor de identidade (*Keycloak*), que disponibiliza um *JWKS endpoint* para expor suas chaves, fazendo possível a verificação das chaves de forma dinâmica para cada validação de requisição.

A parte do *backend*, desenvolvido com FAST API, representa o sistema a ser protegido e, assim como os outros componentes, sua implementação foi realizada em cima de um contêiner *Docker*. A estrutura do *backend* incorpora um PATH específico chamado "*/secured*", que desempenha o caminho dos dados a serem protegidos. Para requisições com destino a este PATH é necessário a inclusão de um *JSON Web Token* (JWT) no cabeçalho HTTP *authorization* das requisições para garantir a sua autenticação e autorização. O *endpoint* "*/secured*"é responsável por fornecer dados personalizados de acordo com as permissões e identidade contidas no JWT. A decisão de condicionar o acesso aos dados com base no token JWT contribui significativamente para a segurança do sistema, garantindo que apenas usuários autenticados e autorizados tenham acesso a informações sensíveis.

Além disso, o *backend* implementado oferece um *endpoint* público no caminho raiz ("/"), que não exige validação de token, tornando-o acessível para qualquer requisição que alcance sua API. Essa abordagem diferencia claramente os recursos públicos dos protegidos, que visa proporcionar uma experiência flexível e segura para os usuários, equilibrando a acessibilidade pública com a necessidade de proteção de dados sensíveis por meio da validação rigorosa de tokens JWT para o PATH "*/secured*". Essa estratégia reflete a preocupação com a segurança da API e demonstra a capacidade do sistema em lidar de forma eficaz com diferentes níveis de acesso, proporcionando uma arquitetura flexível e orientada para a segurança.

Para alcançar os objetivos propostos nesta pesquisa, foi empregado inicialmente uma abordagem metodológica que consiste na inicialização de microsserviços que serão responsáveis pela estabilização do fluxo de autenticação e autorização projetado. Para implementação da arquitetura de microsserviços foi utilizado contêineres *Docker* pela sua ampla difusão de uso e abundante documentação, além de sua utilização permitir a escalabilidade ágil e a gestão simplificada de dependências, enquanto mantém a reprodutibilidade do ambiente e otimiza o uso de recursos de *hardware*. Após a realização dos cenários, foi realizado uma análise qualitativa que aborda as principais vantagens e limitações da implementação de autorização em microsserviços, contrastando cada arquitetura com os fatores de segurança, performance e escalabilidade.

O resultado da análise qualitativa deste trabalho foi obtido por meio da comparação de cenários, possibilitando julgar propriedades, como segurança, performance e escalabilidade. Dessa forma, alcançou-se uma relação de hierarquia entre tais cenários, o que nos guia para a solução de um problema.

Neste estudo, a plataforma *Google Scholar* foi empregada como uma ferramenta fundamental para a busca e recuperação de trabalhos científicos relevantes no âmbito da autorização em microsserviços. O *Google Scholar* oferece uma vasta base de dados que abrange periódicos acadêmicos, conferências, teses

e outros tipos de publicações. A estratégia de pesquisa envolveu a utilização de termos-chave específicos relacionados à autorização em microsserviços, permitindo a identificação de estudos pertinentes a esse tema, sendo escolhido o seguinte filtro para a busca de artigos: *"security in microservices"OR "microservices authorization"OR "microservices IAM"OR "Zero Trust microservices"OR "microservices security best practices"*. Além disso, foram exploradas as funcionalidades avançadas de pesquisa, incluindo filtros por data e relevância, para garantir a obtenção de informações atualizadas e significativas que foram publicadas após o ano de 2019. A revisão sistemática de cerca de 30 artigos científicos obtidos por meio do *Google Scholar*, proporcionou uma base sólida para a análise crítica e a síntese das abordagens existentes na literatura sobre a autorização em microsserviços, contribuindo para a fundamentação teórica deste estudo.

3.2 TECNOLOGIAS UTILIZADAS

Esta seção também é dedicada para o resumo das tecnologias implementadas dentro da arquitetura de microsserviços. O ambiente foi inicializado através da ferramenta *docker-compose* que é utilizado para rodar aplicações *multi-container* baseadas em *docker*. O arquivo pode ser encontrado no apêndice deste trabalho.

3.2.1 Keycloak

O *Keycloak* é uma solução de código aberto que oferece recursos abrangentes de gerenciamento de identidade e acesso, projetada para simplificar a autenticação e a autorização em aplicativos e serviços. Atuando como um servidor de autenticação e autorização, o *Keycloak* oferece suporte a uma variedade de protocolos de autorização e autenticação, como *OAuth 2.0* e *OpenID Connect*, permitindo que os desenvolvedores integrem facilmente autenticação única (*SSO*) e autorização em suas aplicações. Além disso, o *Keycloak* fornece recursos avançados de gerenciamento de usuários, como o registro de contas, a recuperação de senhas e a gestão de permissões granulares por meio do modelo RBAC (*Role-Based Access Control*). Com sua arquitetura flexível e extensível, o *Keycloak* se destaca como uma ferramenta poderosa para garantir a segurança e o controle de acesso em ambientes de desenvolvimento ágil e microsserviços, tornando mais eficiente a implementação de funcionalidades de autenticação e autorização em uma variedade de cenários de aplicativos web e APIs.

Keycloak vem com seu próprio banco de dados de usuários, o que torna uma ferramenta mais fácil de se utilizar. Também podendo integrar-se à infraestruturas de identidades existentes. Por meio de seus recursos de identidade, é possível conectar-se a bases de usuários existentes de redes sociais ou de outros provedores de identidade empresarial, também podendo ser integrado a diretórios de usuários existentes, como *Active Directory* e servidores LDAP [20].

3.2.2 Nginx

O *Nginx*, um servidor web de código aberto amplamente utilizado, desempenha um papel centralizador de sistemas distribuídos e na gestão de tráfego de rede. Uma das suas aplicações mais importantes é na atuação como um *gateway* de API, sendo um componente crucial em arquiteturas de microsserviços e na exposição segura e eficiente de APIs para clientes externos e internos. O *Nginx* oferece recursos poderosos de roteamento, balanceamento de carga, autenticação e autorização, tornando-se uma solução atraente para controlar o acesso e a distribuição de tráfego entre diferentes microsserviços que compõem um sistema complexo.

No contexto de um *Gateway* de API, o *Nginx* desempenha um papel crucial na orquestração e gerenciamento de solicitações de API, garantindo o direcionamento correto das requisições para os serviços apropriados, aplicando políticas de segurança, limitando a exposição de *endpoints* sensíveis e permitindo o monitoramento do tráfego em tempo real. A sua capacidade de lidar com grande volume de conexões simultâneas, a eficiência na manipulação de solicitações *HTTP* e a configuração flexível por meio de arquivos de configuração tornam-o uma escolha preferencial para a implementação de *gateways* de API em ambientes de alta demanda. O *Nginx*, portanto, emerge como uma ferramenta versátil e robusta no contexto de arquiteturas de microsserviços, contribuindo para a escalabilidade, segurança e eficiência das aplicações que dependem de uma infraestrutura de serviços distribuídos.

3.2.3 Permit.io

O *permit.io* é uma solução de autorização que permite aos desenvolvedores incorporar o controle de acesso em seus produtos em tempo reduzido e tê-los prontos para atender às futuras demandas de clientes e regulamentações. O SDK para desenvolvedores se integra ao aplicativo e permite que se adicione verificações de permissão declarativas que são tão fáceis de usar quanto *flags* de recursos. A solução é construída sobre sólidos alicerces de código aberto, habilita o *Git-Ops* prontamente e vai além da aplicação - proporcionando experiências de controle de acesso destinadas a serem usadas por desenvolvedores prontas para uso [56].

O *Permit.io* fornece uma autorização de nível de aplicação *Plug & Play*, permitindo criar regras de execução em seu *Frontend* ou *Backend*, controlando tudo por meio de uma interface gráfica sem necessidade de codificação. Com a utilização do *Permit.io* é possível separar equipes especializadas para a construção de políticas de acesso através de interfaces gráficas intuitivas, realizando o desacoplamento entre times especializados de segurança e equipe de desenvolvimento de software.

3.2.4 PostgreSQL

O *PostgreSQL* é um sistema de gerenciamento de banco de dados relacional (RDBMS) de código aberto amplamente reconhecido pela sua confiabilidade e recursos avançados. Com uma arquitetura sólida, o *PostgreSQL* é capaz de lidar com tarefas complexas de armazenamento e recuperação de dados, incluindo consultas sofisticadas, manipulação de índices e suporte a transações ACID. Sua flexibilidade e extensibilidade permitem que os desenvolvedores implementem soluções personalizadas, incluindo tipos

de dados, funções e extensões. Com foco na integridade dos dados, segurança e desempenho, o *PostgreSQL* é uma escolha popular tanto para aplicações de pequena escala quanto para sistemas empresariais exigentes, onde a confiabilidade e a capacidade de gerenciar grandes volumes de informações são cruciais.

Para o ambiente desenvolvido no experimento que este trabalho faz referência, o *PostgreSQL* desempenha a função de base de dados do microsserviço *Keycloak*, onde será armazenado todas as informações de configuração do *Keycloak*, dos usuários, seus papéis, credenciais e demais propriedades.

3.2.5 Node.js

Node.js é um ambiente de tempo de execução de código aberto que permite a execução de *JavaScript* do lado do servidor. Sua arquitetura baseada em eventos assíncronos possibilita a construção de aplicações escaláveis e de alta performance. O *Node.js* utiliza o mecanismo V8 de *JavaScript* da Google para executar código de maneira eficiente. Sua maior característica é a manipulação assíncrona de I/O, que permite que várias operações sejam realizadas sem bloquear a execução do programa, adequando-se bem a aplicações em tempo real e baseadas em rede. Além disso, o gerenciamento de pacotes é facilitado pelo npm (*Node Package Manager*), que oferece uma ampla variedade de módulos e bibliotecas prontas para uso. Sua crescente comunidade e adoção generalizada fazem do *Node.js* uma opção atrativa para desenvolvedores interessados em construir aplicações web modernas e eficientes.

3.2.6 FastAPI

O *FastAPI* é um *framework* de desenvolvimento web de alto desempenho, de código aberto baseado em *Python*, projetado para a criação rápida de APIs RESTful. Sua abordagem inovadora combina tipagem estática do *Python* (usando a ferramenta *Pydantic*) e geração automática de documentação interativa (usando o padrão OpenAPI), simplificando o processo de desenvolvimento, validação e documentação das *APIs*. Através da exploração das funcionalidades assíncronas do *Python*, o *FastAPI* oferece uma execução assíncrona eficiente, resultando em alta velocidade e capacidade de lidar com várias solicitações concorrentes. Sua estrutura intuitiva permite a criação de *endpoints* personalizados, enquanto os recursos de validação automática, serialização de dados e autenticação simplificam consideravelmente o processo de construção de *APIs* seguras e confiáveis. A combinação de desempenho, tipagem estática, documentação automática e facilidade de uso faz do *FastAPI* uma excelente escolha para desenvolvedores que buscam criar *APIs* modernas, eficientes e bem documentadas usando a linguagem *Python*.

3.3 ESCOLHA DO *TOKEN*

A utilização de JSON Web Tokens (JWT) como um token de autenticação e autorização é justificada por algumas razões substanciais no contexto de testes de implementação. Em primeiro lugar, JWT é um padrão amplamente aceito para a representação de informações de segurança em um formato estruturado e autocontido, o que facilita a troca de informações entre sistemas e serviços de forma segura e eficiente, oferecendo flexibilidade na inclusão de informações necessárias para avaliar com precisão os controles de

acesso e a identificação de usuários.

Além disso, a escalabilidade e a independência de estado do JWT são fatores significativos que justificam sua escolha em testes de implementação. Como os JWTs são autossuficientes, não é necessário armazenar informações adicionais no servidor, simplificando a implementação e minimizando a sobrecarga de armazenamento. Isso é particularmente vantajoso em ambientes distribuídos e sistemas altamente escaláveis, onde a autenticação e a autorização precisam ser eficientes e ágeis. Portanto, a escolha do JWT como token para testes de implementação de autenticação e autorização oferece uma solução robusta, flexível e escalável que se alinha com as melhores práticas da indústria e fornece uma estrutura sólida para avaliação e validação de sistemas de segurança.

3.4 AMBIENTE DE TESTES

O ambiente de testes em que foram levantados os microsserviços consiste em um servidor físico com as características de *hardware* descritas na tabela 3.1

Tabela 3.1: Tabela do *host* físico que hospeda os microsserviços

Sistema Operacional	Ubuntu 20.04.2 LTS
Tipo de Sistema Operacional	64-bit
Memória	7,6 GiB
Processador	Intel Core i5-3317U CPU @ 1.70GHz x 4
Disco	502.6 GB

Os testes foram conduzidos com o objetivo específico de proporcionar uma análise e comparação abrangente dos cenários implementados. Ao realizar esses testes, foi possível não apenas observar, mas também vivenciar na prática os desafios intrínsecos às arquiteturas relacionadas, especialmente no que diz respeito às complexidades envolvidas na autenticação e autorização em ambientes de microsserviços. Essa abordagem prática permitiu uma compreensão mais profunda e contextualizada dos obstáculos enfrentados, destacando nuances cruciais que podem impactar significativamente a implementação bem-sucedida dessas arquiteturas distribuídas. Dessa forma, os resultados obtidos contribuem para o corpo de conhecimento relacionado a microsserviços, enriquecendo a compreensão dos profissionais envolvidos e fornecendo visões valiosas para a melhoria contínua dessas arquiteturas.

3.5 CENÁRIOS DE TESTES

Os cenários reproduzidos tem como objetivo restringir o acesso ao serviço *backend* de acordo com a tabela 3.2. Pela tabela pode-se observar que apenas o administrador tem permissão para realizar requisições utilizando-se de todos os métodos *HTTP* que a API suporta, enquanto os papéis de Operador, Desenvolvedor e Visitante possuem restrições na consumação da API. Sendo o operador apenas capaz de realizar os métodos GET, POST e PUT, o Desenvolvedor, GET e POST e o Visitante apenas GET.

Tabela 3.2: Restrição de permissões ao *backend* de acordo com os papéis dos usuários

	Administrador	**Operador**	**Desenvolvedor**	**Visitante**
GET	YES	YES	YES	YES
POST	YES	YES	YES	NO
PUT	YES	YES	NO	NO
DELETE	YES	NO	NO	NO

3.5.1 Cenário 1 - Autorização descentralizada (*mesh*)

No primeiro cenário foi abordado um ambiente de autenticação e autorização com o *framework OAuth 2.0* utilizando *authorization code flow* em uma infraestura de microsserviços descentralizada, isto é, cada microsserviço torna-se responsável em fazer as validações de autorização em cada requisição que chega até ele. No cenário evidenciado na figura 3.1 temos os seguintes elementos:

- **Resource Owner:** Representado pelo símbolo de navegador, o *Resource Owner* é o detentor das informações contidas no *backend*. Ele é o responsável por realizar a submissão de credenciais ao *Identity Provider* e, possivelmente, consentir permissão ao *frontend* para acessar informações em seu nome.
- **Identity Provider**: O provedor de identidade é o responsável pela administração de todas as identidades dos usuários, validando as credenciais submetidas pelo *user-agent* do RO. É nele que o *token* é gerado de acordo com as permissões dos usuários, além disso, o *IdP* disponibiliza a chave pública em seu *JWKS endpoint* que será utilizada para a validação de JWTs realizadas pelos elementos que precisam validar os *tokens* recebidos em suas requisições.
- **Identity Provider Database**: Pela própria característica efêmera de contêineres, é necessário um microsserviço separado para que haja persistência nos dados caso esses microsserviços por qualquer motivos tenham a necessidade de serem reiniciados. Para isso é necessário que se realize um mapeamento de diretórios entre o contêiner de banco de dados e o servidor *host* que ele esteja rodando.
- **Frontend**: O *frontend* terá o papel de *client* no *framework OAuth2*. Esse é o microsserviço que realiza as requisições em nome do RO sem precisar gerenciar suas credenciais.
- **Backend**: O *backend* possui os dados que devem ser assegurados. Para o cenário de autorização descentralizada, ele é o responsável por realizar a validação de todas as requisições recebidas.

As etapas do processo foram evidenciadas na figura 3.1 com as seguintes etapas:

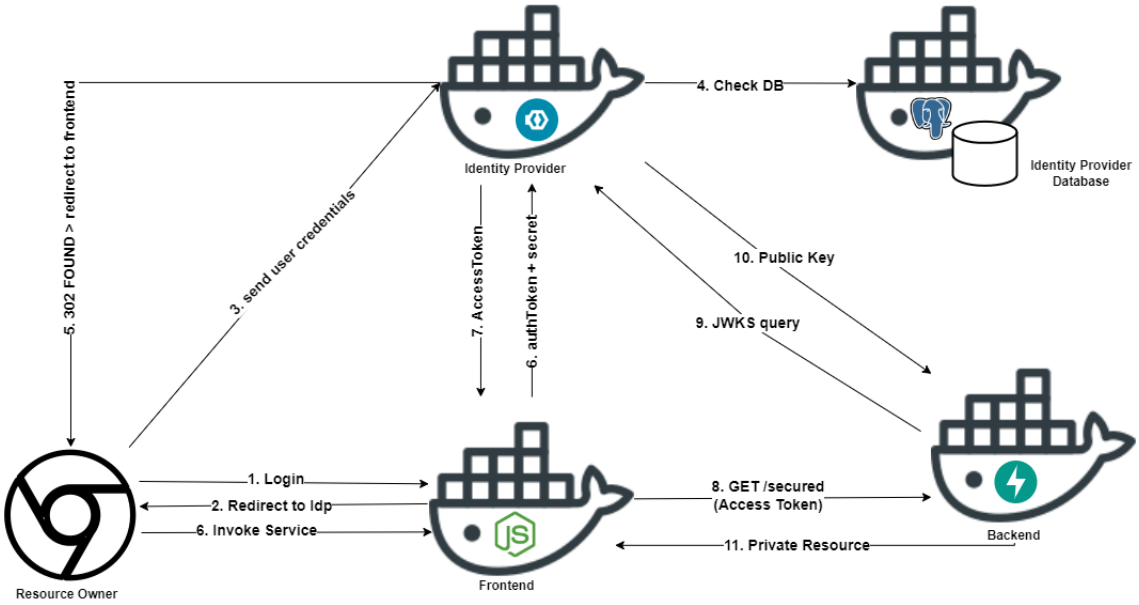

Figura 3.1: Etapas de autenticação e autorização utilizando arquitetura de microsserviços

1. *Resource Owner (RO) browser* acessa a área de *login* fornecida pelo *front-end Node.js*.

2. *Front-end* redireciona RO para o *IdP Keycloak*. Pode-se observar as seguintes informações das figuras 3.2 e 3.3 que é a requisição realizada pelo RO para o servidor de autorização (*Identity Provider*) que ela possui os seguintes parâmetros destacáveis: o *client_id* que identifica para o *keycloak* o cliente registrado que receberá o token de autorização. Também é possível verificar que o RO envia como parâmetro o *redirect_uri* que orienta o servidor de autorização a redireciona-lo para o *frontend* após o processo de autenticação e *scope* que significa que o *user-agent* realizará uma requisição de autenticação utilizando o protocolo OIDC.

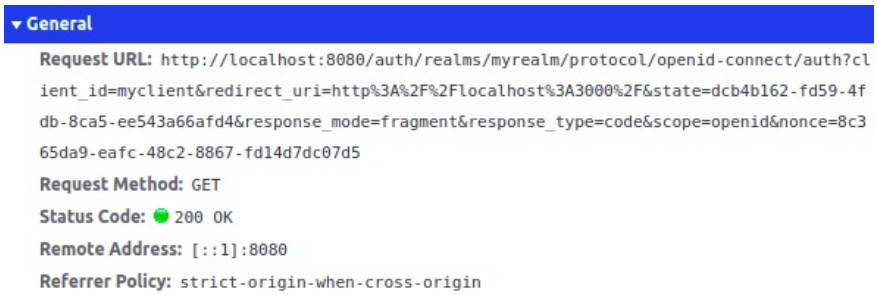

Figura 3.2: Requisição para o Keycloak

Figura 3.3: Payload da requisição para o Keycloak

3. Cliente submete a credencial para o *IdP*. Observe que essa requisição deve ser sempre realizada utilizando o protocolo HTTPS para evitar vazamento de credencial.

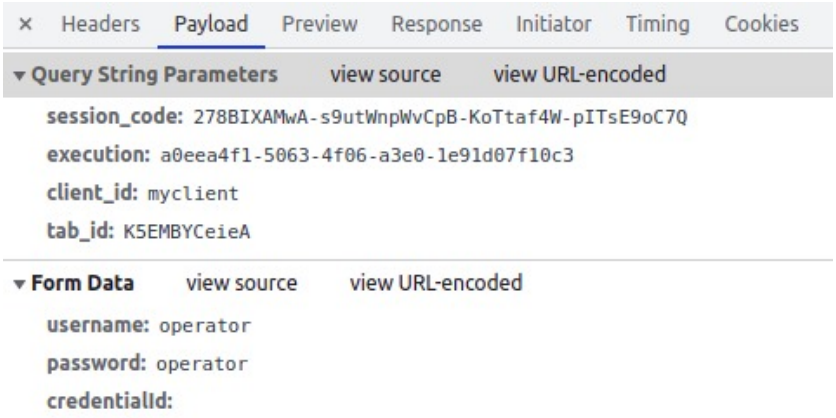

Figura 3.4: RO enviando as credenciais para o servidor de autorização

4. *IdP* verifica a credencial submetidas pelo cliente com o seu banco de dados *PostgreSQL*.

5. Após credencial validada, o *IdP* redireciona o cliente para o *front-end*, inserindo em sua resposta um *authorization token*.

6. *Front-end* consulta o *IdP* com o *authorization token* recebido pelo cliente que o retorna, finalmente, o *access token* que será utilizado para realizar requisições ao *back-end* em nome do *Resource Owner*. O tipo de concessão de código de autorização é usado para obter tanto tokens de acesso quanto tokens de atualização *(refresh)* e é otimizado para clientes confidenciais. Como este é um fluxo baseado em redirecionamento, o cliente deve ser capaz de interagir com o agente do proprietário dos recursos (tipicamente um navegador da web) e ser capaz de receber solicitações de entrada (via redirecionamento) do servidor de autorização.

7. *Front-end* realiza a chamada ao *back-end* com o *access token* encorporado ao cabeçalho *Authorization* da requisição.

8. *Back-end* consulta o *JWKS endpoint* do *IdP*, onde é possível capturar a chave pública utilizada para validar o *access token*.

9. IdP responde à requisição com a chave pública para a validação do *token*
10. *Back-end* realiza a validação de assinatura do *access token*, assim como as validações necessárias de acesso do *JSON Web Token*.
11. *Back-end* retorna a resposta ao *front-end*.

No fluxo evidenciado, é possível observar que o *back-end* se torna responsável por realizar a validação de tokens e, de acordo com as *claims* incorporadas ao *JWT* e retornar informações ao seu solicitante. Essa abordagem pode se tornar inadequada para cenários onde deve-se avaliar muitas condições de atributos para a concessão de acesso, tornando o manejamento de controle de acesso inescalável e fora de controle pelo desenvolvedor responsável.

Na figura 3.5 é possível observar que para cada *endpoint* declarada na API *FAST API*, é necessário realizar a validação de autorização do token recebido no cabeçalho *authorization* de cada requisição HTTP. O que se torna problemático se tivermos um número excessivo de *endpoints*, de serviços e de *roles*, tornando a implementação ineficiente e não escalável.

```
38  @app.get("/")
39  def read_root():
40      return {"Hello": "World"}
41
42  @app.get("/secured")
43  async def get_secured(authorization:str = Header(default=None)):
44      if verifyJWT(authorization.split()[1])['assignedRole'] == "administrator" or "operator" or "developer" or "guest":
45          return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ", this is a secured area"
46
47  @app.post("/secured")
48  async def get_secured(authorization:str = Header(default=None)):
49      if verifyJWT(authorization.split()[1])['assignedRole'] == "administrator" or "operator" or "developer":
50          return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ", this is a secured area"
51
52  @app.put("/secured")
53  async def get_secured(authorization:str = Header(default=None)):
54      if verifyJWT(authorization.split()[1])['assignedRole'] == "administrator" or "operator":
55          return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ", this is a secured area"
56
57  @app.delete("/secured")
58  async def get_secured(authorization:str = Header(default=None)):
59      if verifyJWT(authorization.split()[1])['assignedRole'] == "administrator":
60          return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ", this is a secured area"
61      else:
62          raise HTTPException(
63              status_code=401,
64              detail= "Not authorized!"
65          )
```

Figura 3.5: Validação do JSON Web Token por *endpoint*

3.5.2 Cenário 2 - Utilizando API Gateway

Outra forma de arquitetura que centraliza políticas de controle de acesso em microsserviços é utilizando *gateways de API* que são capazes de realizar a validação de *tokens* contidos nas requisições às APIs. Pode-se observar na arquitetura da figura 3.6 que até o passo número 8 do *flow* de autenticação e autorização com *oAuth2* permanece o mesmo, no entanto, ao invés de realizar a requisição diretamente para o serviço

de backend, o *gateway* de API que lida com a requisição e decide baseado no *token* de acesso se vai encaminhar a solicitação para o microsserviço de *backend*.

- **Resource Owner:** Representado pelo símbolo de navegador, o *Resource Owner* é o detentor das informações contidas no *backend*. Ele é o responsável por realizar a submissão de credenciais ao *Identity Provider* e, possivelmente, consentir permissão ao *frontend* para acessar informações em seu nome.
- **Identity Provider**: O provedor de identidade é o responsável pela administração de todas as identidades dos usuários, validando as credenciais submetidas pelo *user-agent* do RO. É nele que o *token* é gerado de acordo com as permissões dos usuários, além disso, o *IdP* disponibiliza a chave pública em seu *JWKS endpoint* que será utilizada para a validação de JWTs realizadas pelos elementos que precisam validar os *tokens* recebidos em suas requisições.
- **Identity Provider Database**: Pela própria característica efêmera de contêineres, é necessário um microsserviço separado para que haja persistência nos dados caso esses microsserviços por qualquer motivos tenham a necessidade de serem reiniciados. Para isso é necessário que se realize um mapeamento de diretórios entre o contêiner de banco de dados e o servidor *host* que ele esteja rodando.
- **Frontend**: O *frontend* terá o papel de *client* no *framework OAuth2*. Esse é o microsserviço que realiza as requisições em nome do RO sem precisar gerenciar suas credenciais. Observa-se que, para este cenário, o *frontend* não se conecta diretamento ao *backend* e sim ao *gateway* de API.
- **Backend**: Responsável apenas pela parte lógica da aplicação. Não é realizado nenhuma configuração de autenticação ou autorização.
- **API Gateway**: É o *proxy* reverso que antecede às requisições realizadas para o *backend*, sendo o responsável pela centralização das políticas de controle de acesso dos serviços que se encontram sobre a sua cobertura.

Durante a verificação do JWT, o *NGINX* Plus valida automaticamente apenas as reivindicações "nbf"("*not before*") e "exp"("*expires*"). No entanto, para o caso implementado, é necessário definir condições adicionais para uma validação bem-sucedida do JWT.

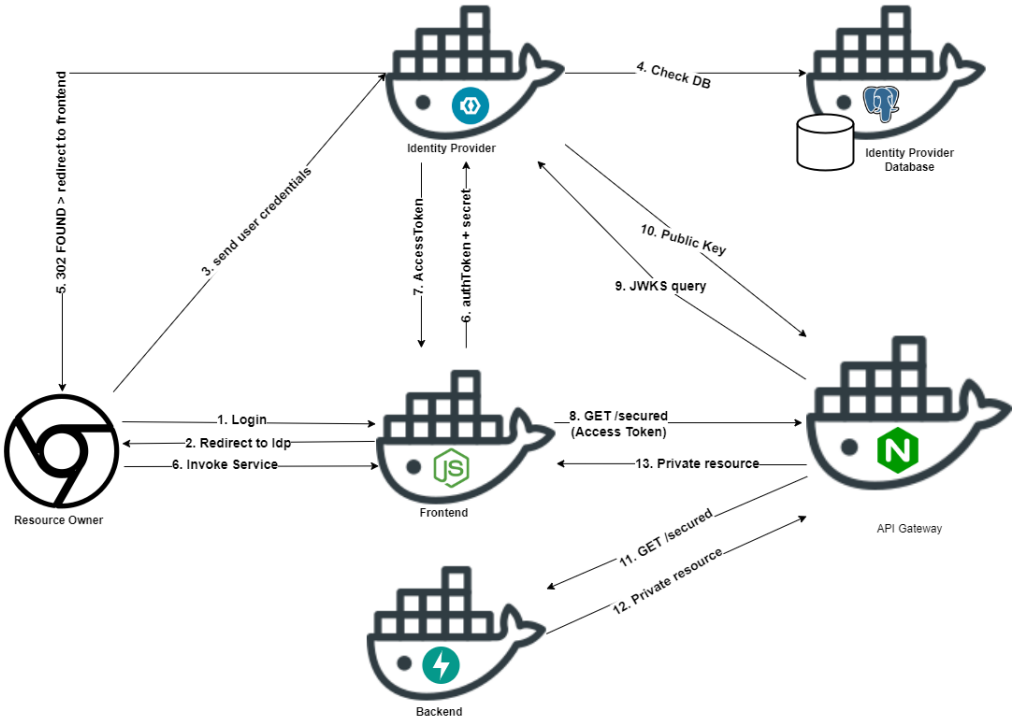

Figura 3.6: Topologia de microsserviços utilizando *gateway* de API para a validação de token e autorização.

Na figura 3.6 é possível observar o seguinte fluxo:

1. *Resource Owner (RO) browser* acessa a área de *login* fornecida pelo *front-end Node.js*.
2. *front-end* redireciona cliente para o *IdP Keycloak*.
3. Cliente submete a credencial para o *IdP*.
4. *IdP* verifica a credencial submetidas pelo cliente com o seu banco de dados *PostgreSQL*.
5. Após credencial validada, o *IdP* redireciona o cliente para o *front-end*, inserindo em sua resposta um *authorization token*.
6. *Front-end* consulta o *IdP* com o *authorization token* recebido pelo cliente que o retorna, finalmente, o *access token* que será utilizado para realizar requisições ao *gateway* de API em nome do *Resource Owner*.
7. *Front-end* realiza a chamada ao *gateway* de API com o *access token* encorporado ao cabeçalho *Authorization* da requisição.
8. *Gateway* de API consulta o *JWKS endpoint* do *IdP*, onde é possível capturar a chave pública utilizada para gerar o *access token*.
9. IdP responde à requisição com a chave pública para a validação de integridade do *token* que, continua sendo realizada pelo *gateway* de API.

10. O *gateway* de API realiza as validações necessárias de integridade de token e autorização segundo as *claims* contidas no *payload* do JWT. Após a validação de integridade e *claim*, o *gateway* de API realiza a requisição sem o cabeçalho de autorização ao *backend*

11. O *backend* retorna ao *gateway* de API o recurso confidencial sem a necessidade de avaliar nenhum aspecto de autenticação ou autorização da requisição.

12. O *gateway* de API retorna o recurso confidencial ao *frontend*

Pode-se observar pela figura 3.7 a configuração necessária para a validação do token JWT no *gateway* de API foi utilizado uma política de controle de acesso de roteamento (*access-control-routing*) evidenciada na linha de número 2. Entre a linha 7 e 18 do arquivo de configuração indicam que para a URL /secured utilizando o método POST é necessário que na *claim* do JWT contenha a chave *assignedRole* que possua os valores *administrator, operator* ou *developer*, caso contrário, o *gateway* retornar a resposta HTTP 403 que indica que o servidor entende a requisição, mas se recusa a autoriza-la.

nginx.conf

```
1   "policies": {
2     "access-control-routing": [
3       {
4         "action": {
5           "conditions": [
6             {
7               "allowAccess": {
8                 "httpMethods":["POST"]
9                 "uri": "/secured"
10              },
11              "when": [
12                {
13                  "key": "token.assignedRole",
14                    "matchOneOf": {
15                      "values": [
16                        "administrator",
17                        "operator",
18                        "developer"
19                      ]
20                    }
21                }
22              ]
23            }
24          ],
25          "returnCode": 403
26        }
27      }
28    ]
```

Figura 3.7: Validação de token realizado pelo Nginx.

3.5.3 Cenário 3 - *Authorization as a Service*

No cenário utilizando *authorization as a service* o microsserviço *backend* ainda possui a responsabilidade de realizar a validação de integridade do *access token*, no entanto, a lógica de concessão de privilégio

fica encarregado ao *permit.io* que irá validar as políticas configuradas de forma centralizada, retirando a responsabilidade da implementação de autorização pelos desenvolvedores e tornando a aplicação escalável no número de *roles*, serviços e *endpoints*. A figura 3.8 tem representada os seguintes elementos:

- **Resource Owner:** Representado pelo símbolo de navegador, o *Resource Owner* é o detentor das informações contidas no *backend*. Ele é o responsável por realizar a submissão de credenciais ao *Identity Provider* e, possivelmente, consentir permissão ao *frontend* para acessar informações em seu nome.
- **Identity Provider**: O provedor de identidade é o responsável pela administração de todas as identidades dos usuários, validando as credenciais submetidas pelo *user-agent* do RO. É nele que o *token* é gerado de acordo com as permissões dos usuários, além disso, o *IdP* disponibiliza a chave pública em seu *JWKS endpoint* que será utilizada para a validação de JWTs realizadas pelos elementos que precisam validar os *tokens* recebidos em suas requisições.
- **Identity Provider Database**: Pela própria característica efêmera de contêineres, é necessário um microsserviço separado para que haja persistência nos dados caso esses microsserviços por qualquer motivos tenham a necessidade de serem reiniciados. Para isso é necessário que se realize um mapeamento de diretórios entre o contêiner de banco de dados e o servidor *host* que ele esteja rodando.
- **Frontend**: O *frontend* terá o papel de *client* no *framework OAuth2*. Esse é o microsserviço que realiza as requisições em nome do RO sem precisar gerenciar suas credenciais.
- **Backend**: Neste cenário, o *backend* não é mais o responsável pela validação do token, utilizando *Authorization as a Service*, o *backend* apenas realiza a validação de integridade do *token*, para isso ainda é necessário que ele realize a consulta ao *JWKS endpoint* do *IdP*. A parte lógica de concessão de autorização é integralmente realizada pelo *permit.io*
- **Permit.io**: É o serviço de *authorization as a service* onde é implementada as políticas de controle de acesso que realizarão a conferências das *claims* contidas nos JWTs e retorna a resposta ao *backend* a autorização ou não para a requisição.

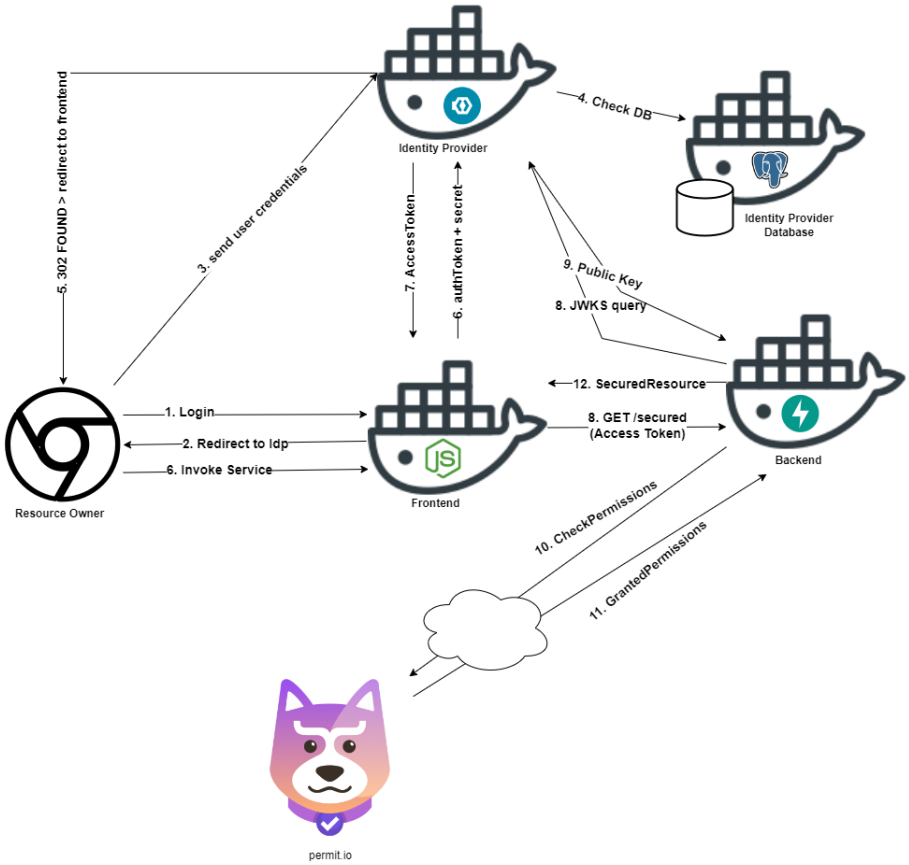

Figura 3.8: Fluxo de informação utilizando *authorization as a service*

Na figura 3.8 é possível observar o seguinte fluxo:

1. *Resource Owner (RO) browser* acessa a área de *login* fornecida pelo *front-end Node.js*.
2. *front-end* redireciona cliente para o *IdP Keycloak*.
3. Cliente submete a credencial para o *IdP*.
4. *IdP* verifica a credencial submetidas pelo cliente com o seu banco de dados *PostgreSQL*.
5. Após credencial validada, o *IdP* redireciona o cliente para o *front-end*, inserindo em sua resposta um *authorization token*.
6. *Front-end* consulta o *IdP* com o *authorization token* recebido pelo cliente que o retorna, finalmente, o *access token* que será utilizado para realizar requisições ao *back-end* em nome do *Resource Owner*.
7. *Front-end* realiza a chamada ao *back-end* com o *access token* encorporado ao cabeçalho *Authorization* da requisição.
8. *Back-end* consulta o *JWKS endpoint* do *IdP*, onde é possível capturar a chave pública utilizada para gerar o *access token*.

9. IdP responde à requisição com a chave pública para a validação de integridade do *token* que, continua sendo realizada pelo *backend*.

10. Após a validação de integridade do *token*, o *backend*, envia o JWT para o serviço de nuvem *permit.io*, onde será realizado a validação de autorização pela análise de *claims*, contidos no *payload* do *token*.

11. O serviço de nuvem autoriza a requisição ao *backend*.

12. *Backend* retorna a informação protegida ao *frontend*.

A parte lógica contida no código para validação de *claims* contidas no JWT é realizada pelo *permit.io*. Observe que para a comunicação do microsserviço e o *permit.io* deve ser realizada de forma segura, para isso é necessário que as chamadas realizadas pelo microsserviço contenha uma chave privada fornecida pela console de administração do *permit.io*. É possível observar pela figura 3.9 (linhas 50 e 66) que para os dois endpoints em questão (GET /secured e POST /secured), o desenvolvedor não precisa tomar conhecimento das regras de autorização, apenas enviar para a plataforma de *authorization as a service* através do método *permit.check()* o usuário, a ação e o recurso a ser acessado como atributos.

```
44      @app.get("/secured")
45      async def get_secured(authorization:str = Header(default=None)):
46          # After we created this user in the previous step, we also synced the user's identifier
47          # to permit.io servers with permit.write(permit.api.syncUser(user)). The user identifier
48          # can be anything (email, db id, etc) but must be unique for each user. Now that the
49          # user is synced, we can use its identifier to check permissions with `permit.check()`.
50          permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "get", "myservice")
51          if not permitted:
52              raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
53                  "result": "NOT PERMITTED to GET myservice!"
54              })
55
56          return JSONResponse(status_code=status.HTTP_200_OK, content={
57              "result": "PERMITTED to GET myservice!"
58          })
59
60      @app.post("/secured")
61      async def post_secured(authorization:str = Header(default=None)):
62          # After we created this user in the previous step, we also synced the user's identifier
63          # to permit.io servers with permit.write(permit.api.syncUser(user)). The user identifier
64          # can be anything (email, db id, etc) but must be unique for each user. Now that the
65          # user is synced, we can use its identifier to check permissions with `permit.check()`.
66          permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "post", "myservice")
67          if not permitted:
68              raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
69                  "result": "NOT PERMITTED to POST myservice!"
70              })
71
72          return JSONResponse(status_code=status.HTTP_200_OK, content={
73              "result": "PERMITTED to POST myservice!"
74          })
```

Figura 3.9: Validação em código com *permit.io*

Dentro do *permit.io* foi necessário realizar as seguintes configurações para o correto funcionamento da autorização centralizada:

- **Recurso:** A primeira configuração necessária é configurar o recurso (ou API) que o *permit.io* vai realizar a autorização. Como evidenciado na figura 3.10, é possível identificar que foi utilizado o recurso *myservice* e dentro dele configurado 4 *actions* que, por sua vez, são os métodos GET, POST, PUT e DELETE.

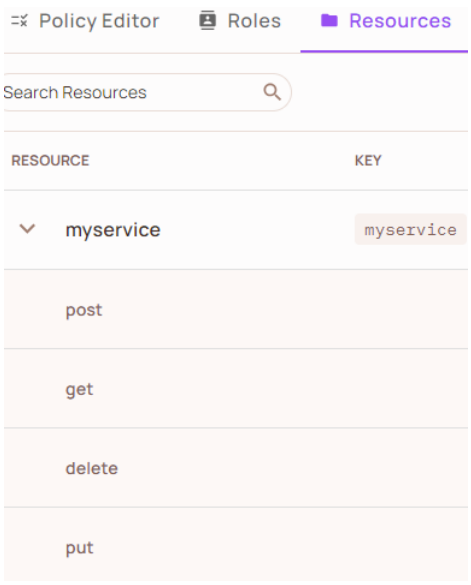

Figura 3.10: Configuração de recursos no *permit.io*

- **Roles:** Também deve-se realizar a configuração de papéis que, no caso, são: *administrator*, *operator*, *developer* e *guest*.
- **Usuários:** Após a configuração dos papéis, deve-se associar os papéis aos usuários cadastrados no *permit.io*
- **Política:** Finalmente, na política é necessário indicar o nível de permissão para cada papel associado, vide figura 3.11.

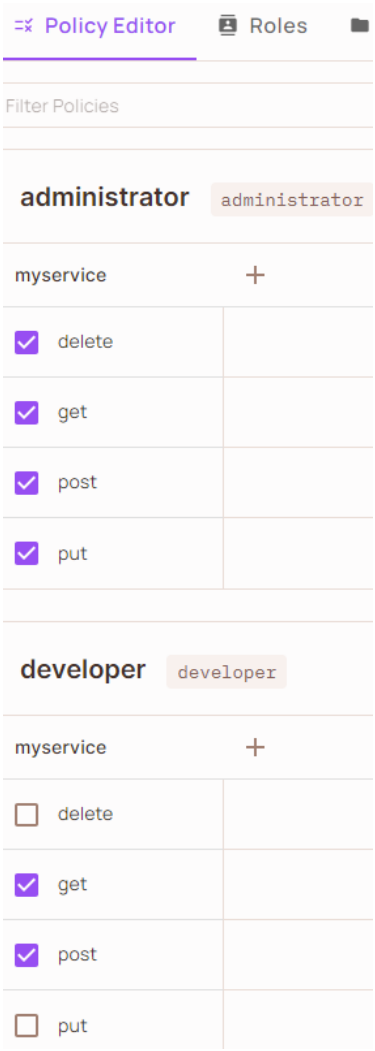

Figura 3.11: *Policy Editor* no *permit.io*

4 RESULTADOS

4.1 RESULTADOS

Este capítulo tem como objetivo evidenciar por meio dos cenários implementados, as vantagens e limitações da validação de *token* realizada por microsserviços de acordo com o tipo de arquitetura levada em consideração. Para isso, foi implantado 3 arquiteturas que alteram-se quanto modo de validação dos *tokens* que são submetidos junto às requisições, que buscam pelas respostas das APIs REST disponíveis.

As arquiteturas implementadas pelos contêineres foram escolhidas de acordo com as melhores práticas utilizadas no mercado, sendo a arquitetura *mesh* e utilização de *gateway* de API formas bem conhecidas de comunicação entre microsserviços. No primeiro cenário, foi implementado uma arquitetura de autorização descentralizada, isto é, cada microsserviço que recebe a requisição através da sua API em modo REST é o responsável por realizar a validação de integridade do JWT recebido, assim como a validação das *claims* contidas no *token*. No segundo cenário utilizando *gateway* de API, é colocado na frente do microsserviço um *proxy* reverso que, além de outras funcionalidades de segurança, também é o responsável por validar cada JWT contida nas requisições antes de realizar o encaminhamento para os microsserviço que ele próprio antecede. Por ultimo, foi implementado um cenário que utiliza a validação dos tokens de autorização em um serviço de nuvem *authorization as a service*, onde as *claims* do token são validadas de acordo com as políticas configuradas na nuvem que, por sua vez, retornam para o microsserviço se a requisição é válida ou não.

4.1.1 Cenário 1

A validação descentralizada de tokens de autorização pode ser inadequada por diversas razões de natureza prática e de segurança. Primeiramente, tal abordagem pode aumentar significativamente a complexidade da infraestrutura de autenticação e autorização. Isso ocorre porque a validação descentralizada requer que cada serviço ou microsserviço dentro de uma arquitetura de microsserviços implemente e gerencie sua própria lógica de validação de tokens. Isso não apenas aumenta a sobrecarga operacional, mas também torna mais difícil manter a consistência nas políticas de segurança e nas atualizações de validação em toda a arquitetura. Outra questão que deve ser levada em consideração é a configuração de *rate limit*, a qual caracteriza o número de requisições por segundo que pode ser realizada para o *backend*, devendo ser replicada para todos os microsserviços como medida protetiva para o processo de autorização e autenticação.

Além disso, a validação descentralizada pode levar a uma maior exposição a riscos de segurança. A gestão descentralizada de tokens abre a porta para inconsistências nas políticas de autorização e potenciais vulnerabilidades de segurança, uma vez que cada serviço pode interpretar as informações do token de forma diferente. Isso pode resultar em brechas de segurança, autorizações incorretas e maior dificuldade em controlar e auditar o acesso de usuários e serviços. Portanto, centralizar a validação de tokens em um serviço ou componente especializado, como um serviço de autenticação único (Single Sign-On - SSO) ou um servidor de autorização, é frequentemente uma abordagem mais segura e gerenciável.

4.1.2 Cenário 2

O uso do *gateway* de API como um intermediário para todas as chamadas leste-oeste entre microsserviços é problemático tendo em vista a adição de um *hop* de rede em cada chamada entre as APIs, devendo-se considerar o impacto da latência com a chamada de rede adicional e o *overhead* de qualquer que seja o trabalho que o *proxy* fizer. Dentre as desvantagens da utilização do *gateway* de API para a validação de *tokens*, estão:

- **Falta de granularidade na autorização:** A validação de tokens em gateways de API tende a ser baseada em decisões de autorização de alto nível, como o acesso a uma API como um todo, em vez de autorizações granulares no nível do recurso ou da funcionalidade específica. Isso ocorre porque os gateways de API geralmente não têm o contexto interno dos microsserviços subjacentes para realizar autorizações detalhadas. Portanto, essa abordagem pode não ser adequada para sistemas que requerem um controle fino sobre quem pode acessar recursos específicos.
- **Complexidade e sobrecarga operacional:** Implementar a lógica de validação de tokens em gateways de API pode aumentar a complexidade do gateway e exigir a manutenção de uma lógica de autorização que pode se tornar complicada com o tempo, especialmente em arquiteturas de microsserviços em constante evolução. Isso resulta em uma sobrecarga operacional considerável, uma vez que a manutenção e a atualização contínua do gateway se tornam mais desafiadoras.
- **Fragilidade da segurança:** Colocar a lógica de validação de tokens em gateways de API significa que qualquer comprometimento do gateway pode levar a uma falha na segurança, uma vez que os tokens são validados centralmente. Isso pode tornar o sistema mais vulnerável a ataques direcionados ao gateway, pois um único ponto de falha pode comprometer todo o sistema.
- **Restrições de escalabilidade:** Dependendo da arquitetura e das demandas de escalabilidade do sistema, os gateways de API podem se tornar gargalos de desempenho se forem responsáveis por tarefas de validação de tokens em escala. Isso pode limitar a capacidade do sistema de lidar com solicitações de API em grande escala e pode levar a problemas de desempenho.
- **Dependência de fornecedores de gateways de API:** A implementação da validação de tokens em gateways de API pode resultar em uma dependência significativa de fornecedores específicos e suas tecnologias, o que pode ser restritivo em termos de flexibilidade e portabilidade do sistema.

4.1.3 Cenário 3

A validação de tokens em um serviço de autorização na nuvem, como *Authorization as a Service*, pode ser inadequada por uma série de razões, incluindo preocupações com segurança, privacidade e dependência de terceiros. Entre as desvantagens estão:

- **Segurança e privacidade dos dados:** Ao optar por validar tokens em um serviço de autorização na nuvem, as organizações podem estar transferindo informações sensíveis, como tokens de autenticação e dados de autorização, para terceiros. Isso levanta preocupações sobre a segurança e a privaci-

dade desses dados. Terceiros podem ter acesso a informações confidenciais, o que pode representar um risco significativo, especialmente em setores regulamentados, como saúde e finanças.

- **Dependência de terceiros:** Ao utilizar um serviço de autorização na nuvem, as organizações se tornam dependentes desse serviço, sendo, por vezes, equivalente ao risco de aprisionamento forçado na nuvem de um provedor, tornando o cenário irreplicável em outros ambientes, o que é chamado de *cloud lock-in* . Isso cria uma única ponto de falha significativa, uma vez que a interrupção ou falha do serviço de terceiros pode paralisar as operações da organização. Além disso, a dependência de terceiros pode resultar em falta de controle sobre as políticas de segurança e autorização, uma vez que as decisões de autorização são tomadas fora do controle direto da organização.

- **Latência e disponibilidade:** A validação de tokens na nuvem pode introduzir latência nas operações, uma vez que as solicitações de autorização precisam ser enviadas e processadas remotamente. A disponibilidade do serviço de autorização na nuvem também é uma preocupação, uma vez que a incapacidade de acessar o serviço pode interromper as operações e causar indisponibilidades inaceitáveis.

- **Conformidade regulatória:** Em muitos setores, as organizações estão sujeitas a regulamentações rígidas em relação ao controle e à proteção de dados confidenciais. A transferência de informações de autenticação e autorização para terceiros pode criar desafios significativos de conformidade regulatória, uma vez que a organização deve garantir que os padrões de segurança e privacidade sejam mantidos em conformidade com as regulamentações relevantes.

4.2 DISCUSSÕES E ANÁLISES

Esta seção ira se limitar a realização da discussão e análise levando em consideração os critérios de segurança, desempenho e escalabilidade nos cenários realizados.

1. **Segurança:** A segurança é fundamental na validação de tokens de autorização. Foi avaliado a arquitetura em relação à proteção dos tokens, a capacidade de detectar e prevenir ameaças, como ataques de falsificação e autenticação incorreta, e a capacidade de garantir a confidencialidade e a integridade dos dados sensíveis.

2. **Desempenho:** Este critério avalia o desempenho da arquitetura, considerando a velocidade e a eficiência da validação de tokens, especialmente em cenários de alto tráfego. Isso inclui a capacidade de lidar com solicitações de validação em tempo real de maneira eficaz e com baixa latência. Implementações que exigem saltos de rede adicionais para a validação de tokens foram classificados como arquiteturas de menos performance, assim como implementações que exige a espera de validação em infraestruturas terceirizadas de nuvens públicas.

3. **Escalabilidade:** Este critério avalia a escalabilidade, sendo esta, uma característica fundamental em microsserviços, verificando se a arquitetura pode se redimensionar horizontalmente para atender ao aumento da demanda. Isso envolve a capacidade de adicionar ou remover servidores de validação de

tokens conforme necessário, sem interrupção do serviço. Além de avaliar a dependência do sistema em sua totalidade no caso de interrupção de serviços de um de seus componentes.

4.2.1 Segurança

A segurança da comunicação entre microsserviços, particularmente no contexto de autenticação e autorização de requisições por meio de REST APIs, é uma consideração crítica em arquiteturas distribuídas. Para garantir a integridade e confidencialidade das transmissões de dados entre os microsserviços, a autenticação desempenha um papel fundamental. A implementação de mecanismos robustos, como tokens de acesso com OAuth 2.0, certificados SSL/TLS ou esquemas de autenticação baseados em API keys, permite que os microsserviços validem a identidade de quem está fazendo a requisição. Além disso, a autorização adequada é essencial para garantir que apenas os microsserviços autorizados possam acessar determinados recursos. O uso de sistemas de controle de acesso, como JWT (JSON Web Tokens) para transmitir informações sobre as permissões do usuário, proporciona uma camada adicional de segurança. A implementação cuidadosa dessas práticas de autenticação e autorização não apenas fortalece a segurança da arquitetura de microsserviços, mas também contribui para a construção de um ambiente confiável e resistente a potenciais ameaças à integridade dos dados e à privacidade das informações transmitidas.

Para arquiteturas descentralizadas, torna-se possível uma maior flexibilidade na implementação de controles de acesso mais granular baseado em atributos. Ao mesmo tempo que aumenta-se a complexidade de implementação dos controles de segurança para a validação de integridade e condições contidas nas *claims* dos tokens. Em caso de indisponibilidade de um dos microsserviços, só serão afetados os sistemas cujo o fluxo de informação dependa do microsserviço indisponível, tornando o sistema como um todo mais tolerante a falhas. Garantir a consistência na validação de tokens em um ambiente descentralizado pode ser mais desafiador, tendo em vista que os tokens podem ser emitidos e validados em diferentes partes do sistema, o que pode levar a inconsistências na maneira como a autenticação e a autorização são aplicadas, levando em consideração o princípio de divisão em equipes menores que são responsáveis por cada microsserviço de acordo com a cultura *DevOps*.

Foi observado que para arquiteturas que utilizam *gateways* de API para a validação de tokens JWT o *gateway* de API pode representar para a arquitetura um *single point of failure*, isto é, no caso do *gateway* por algum motivo estar fora do ar ou sobrecarregado, todos os microsserviços atrás dele serão afetados, trazendo indisponibilidade ao sistema como um todo. Pelo princípio de centralização das requisições aos *gateways* de API, eles também são visados para a realização de ataques do tipo *DDoS*, tornando-os mais vulneráveis a deixar todo um sistema indisponível.

No caso da utilzação de serviços de nuvem com *Authorization as a Service* pode-se observar uma maior facilidade na implementação de políticas de acesso que consomem os microsserviços, pois além da possibilidade de gerar granularidade de permissões de forma descentralizada, é possível que uma equipe de segurança separada seja responsável pela delegação de permissões de todos os microsserviços como um todo, possibilitando os desenvolvedores responsáveis pelo microsserviço se concentrarem apenas no desenvolvimento das aplicações e não na delegação de permissão que podem ser muitas. Outra vantagem da utilização de um serviço especializado para a validação de tokens é a redução de exposição a riscos de segurança, evitando erros comuns de implementação de autenticação e autorização. Isso inclui a preven-

ção de vulnerabilidades, vazamento de informações sensíveis e mitigação de ameaças de segurança, além de oferecerem recursos avançados de monitoramento e auditoria, permitindo que as equipes de segurança rastreiem atividades de autenticação e autorização, identifiquem comportamentos anômalos e detectem potenciais ameaças de segurança de maneira mais eficaz, melhorando a visibilidade e a resposta a incidentes de segurança.

4.2.2 Desempenho

A importância do desempenho em arquiteturas de microsserviços, especialmente quando implementadas em uma arquitetura *multi-cloud*, é crucial para garantir a eficiência operacional e a satisfação do usuário. Em ambientes *multi-cloud*, onde diferentes microsserviços podem estar distribuídos em nuvens públicas ou privadas, o desempenho torna-se um fator determinante para a resposta ágil às demandas variáveis e a manutenção da integridade do serviço. A natureza modular dos microsserviços permite que componentes individuais sejam escalados independentemente, otimizando recursos conforme necessário. No entanto, a coordenação eficaz entre esses serviços, a minimização da latência de comunicação e a garantia de tempos de resposta rápidos são aspectos fundamentais para assegurar a eficácia da arquitetura. O desafio reside não apenas na busca pelo desempenho máximo de cada microsserviço, mas também na orquestração harmoniosa de todo o ecossistema, promovendo uma experiência contínua e eficiente para os usuários finais, independentemente da complexidade e distribuição geográfica da arquitetura de microsserviços.

Na arquitetura de validação de tokens em microsserviços descentralizados, temos como uma das maiores características a baixa latência, tendo em vista que o fluxo de informação é direto entre os microsserviços. O gerenciamento de recursos computacionais é realizado de forma individualizada, fazendo com que haja a necessidade de se replicar um padrão entre os microsserviços.

Com a arquitetura de validação de tokens com *gateways* de API, invariavelmente é adicionado um nó adicional para a validação de *tokens* que é validada pelo próprio *gateway* antes da requisição ser entregue ao microsserviço. Além disso, os *gateways* de API são responsáveis pelo gerenciamento de conexão de todos os clientes o que pode sobrecarrega-los, tornando o sistema mais lento como um todo. No entanto, também é possível realizar implementações de *rate limit*, o que limita o número de requisições por segundo que um cliente pode realizar, fazendo com que os recursos computacionais dos microsserviços atrás do *gateway* de API sejam preservados e respondam às suas requisições com mais eficiência.

Na arquitetura de validação de tokens utilizando serviços de *Authorization as a Service*, é necessário a adição de um nó no fluxo de informação para as validações de token. Observe que, para este caso, não é necessário para o *backend* encaminhar toda a requisição para o serviço de autorização, ele apenas precisa enviar o token junto a ação que a requisição exige, fazendo com que o serviço de autorização retorne uma resposta de acordo com as suas políticas configuradas. Por este fato, a arquitetura não requer um nó adicional no fluxo de informação, no entanto é necessário considerar uma latência adicional tendo em vista o fluxo exclusivo do token.

4.2.3 Escalabilidade

A escalabilidade em arquiteturas de microsserviços é um princípio fundamental que impulsiona a capacidade de lidar com o aumento da carga e demanda de serviços em um ambiente dinâmico. Ao adotar a abordagem de microsserviços, as aplicações são decompostas em componentes independentes, cada um representando um serviço específico. Essa modularidade permite que cada microsserviço seja escalado individualmente, respondendo de forma flexível às variações no volume de tráfego ou demanda específica de um serviço. Dessa forma, é possível alocar recursos de maneira eficiente, direcionando-os para os microsserviços que necessitam de mais capacidade, sem impactar outros serviços que podem requerer menos recursos. A escalabilidade granular proporcionada pelos microsserviços contribui para uma arquitetura mais ágil e resiliente, capaz de se adaptar rapidamente às exigências do ambiente operacional.

Contudo, é importante destacar que a escalabilidade com microsserviços também apresenta desafios, como a necessidade de implementar mecanismos eficientes de comunicação entre os serviços, gerenciamento de dados distribuídos e coordenação de transações. A complexidade aumenta à medida que o número de microsserviços cresce, exigindo uma estratégia robusta de orquestração e descoberta de serviços para garantir que a escalabilidade seja alcançada de maneira coordenada e coesa.

Uma das vantagens gerais do microsserviço em *container* é que, a medida que a demanda por esses microsserviços aumentam, eles podem ser migrados para diversos ambientes de forma ágil que possuem maior capacidade computacional para lidar com o aumento de chamadas que ele recebe. Na arquitetura de validação de tokens descentralizada, temos que o gerenciamento de recursos do *container* deve ser realizado de forma individual. Em um contexto de utilização de orquestrador de containers como *Kubernetes*, a escalabilidade é dada de forma vertical, o que significa que os contêineres deveriam ser replicados de acordo com o volume de carga recebido.

Observando a arquiteturas com *gateways* de API, deve-se focar maior esforço de gerenciamento de recursos no próprio gateway, tendo em vista que o mesmo receberá todas as requisições que serão encaminhadas para os microsserviços subsequentes. Além disso, com essa arquitetura, também é possível a realização de configuração de balanceamento de carga inteligente que realiza o encaminhamento das requisições para os microsserviços de acordo com o recurso computacional disponível nos contêineres posteriores. No caso da utilização de um orquestrador de contêineres como *Kubernetes*, a escalabilidade deve ser realizado de forma vertical no *Ingress Controller*, isto é, aumentar a locação computacional para o elemento, disponibilizando assim mais *hardware* para o componente.

Para a arquitetura que conta com o serviço de *Authorization as a Service* a parte de escalabilidade dos elementos que realizarão a validação de *token* fica por conta do serviço de nuvem, não sendo assim uma responsabilidade da equipe que encaminha os *tokens* para o CSP.

5 CONCLUSÕES E TRABALHOS FUTUROS

Torna-se evidente que a adoção de microsserviços apresenta novos desafios de segurança. Em primeiro lugar, há um aumento na superfície de ataque das aplicações modernas, decorrente da distribuição dos serviços. Em segundo lugar, a eficácia dos sistemas de registro tradicionais, que dependem de uma arquitetura de agregação de logs centralizada, é reduzida. Além disso, há uma fusão do ciclo de vida de desenvolvimento mediante vários componentes da aplicação, em oposição ao modelo monolítico e também pode-se dizer que o nível de tráfego é ampliado devido ao crescente número de comunicações entre os microsserviços, demandando uma abordagem cautelosa para garantir a segurança em todo o ecossistema.

Neste trabalho foi possível realizar o estudo de autorização em microsserviços considerando 3 arquiteturas: microsserviços em arquitetura *mesh* com autorização descentralizada, microsserviços utilizando *gateways* de API e, por fim, *authorization as a service* como um serviço de nuvem que desacopla a autorização do desenvolvimento. Cada uma dessas abordagens apresentou características e desafios específicos no contexto da autorização em microsserviços. A arquitetura *mesh* proporciona uma descentralização da autorização, promovendo maior flexibilidade, mas ao custo de complexidade na gestão e coordenação. Por outro lado, o uso de *gateways* de API centraliza a autorização, simplificando a gestão, mas pode gerar gargalos em ambientes distribuídos por se tornarem um ponto único de falha (*SPOF*). Já a perspectiva de *authorization as a service* destaca-se pela capacidade de desacoplar a lógica de autorização, promovendo uma abordagem mais modular e escalável, colocando a responsabilidade de controle das políticas de acesso a um time especializado de segurança. Diante dessas análises, fica evidente que a escolha da arquitetura de autorização em microsserviços deve ser cuidadosamente ponderada, considerando os requisitos específicos do sistema e as metas almejadas em termos de segurança, flexibilidade e eficiência operacional.

Foram também analisadas as arquiteturas supracitadas considerando aspectos como segurança, desempenho e escalabilidade, onde se pode observar que a arquitetura *mesh* pode ser considerada a mais performática, tendo em vista não haver saltos adicionais que intermedeiam a comunicação entre microsserviços, no entanto, ela pode não ser a mais segura considerando que ela não centraliza as políticas de acesso e não é nativamente capaz de implementar uma padronização de gerenciamento de acesso, o que a torna por si só igualmente não escalável.

Também deve-se considerar como limitação na arquitetura *mesh* a necessidade de cada equipe de desenvolvimento compreender claramente os recursos de segurança ao utilizar uma linguagem de programação e framework, implementando-os corretamente em seus microsserviços. Além disso, é crucial que os times compreendam de maneira clara a política de controle de acesso e as permissões esperadas para um determinado papel ou grupo, uma tarefa que pode ser desafiadora devido às decisões potencialmente dispersas em um ou mais códigos. Este padrão depende da configuração manual cuidadosa pela equipe de desenvolvimento, o que é propenso a erros. Em relação à escalabilidade das aplicações modernas de microsserviços, torna-se distante da realidade para a equipe de desenvolvimento configurar e manter políticas de controle de acesso para cada microsserviço. Mudanças no código-fonte exigem testes de regressão sólidos para detectar possíveis *bugs* de autorização.

Olhando para a arquitetura que introduz ao ambiente o *gateway* de API, em primeiro lugar observa-se a adição de um componente que intermedeia a comunicação entre microsserviços, adicionando latência para a validação de controle de acesso. No entanto, do ponto de vista de segurança, ela possui vantagem sobre a arquitetura *mesh* considerando que é nativamente possível centralizar as políticas de acesso, migrando a responsabilidade do time de desenvolvedores do microsserviço na implementação do controle de acesso, para um time de segurança que pode padronizar as políticas, priorizando assim os princípios de menor privilégio. No entanto, quando considerado as premissas da arquitetura *Zero Trust*, a topologia com *gateway* de API não é a mais recomendada, tendo em vista que ela apenas prioriza a validação de *tokens* das requisições que possuem sentido norte-sul, tornando as comunicações laterais desprotegidas. Quanto o critério de escalabilidade, ao mesmo tempo que o *gateway* de API introduz ao sistema um ponto único de falha (SPOF), o mesmo pode possuir recursos nativos para balanceamento de carga e controles de taxa de transmissão, tornando a entrega de dados mais estável com o aumento de requisições recebidas.

Para o caso da arquitetura com o serviço de nuvem *authorization as a service*, apesar da comunicação das chamadas de API serem realizadas diretamente entre os microsserviços, deve ser considerado a latência da validação de autorização dos tokens realizados na nuvem. A arquitetura em questão pode ser considerada convenientemente mais segura do que a arquitetura *mesh*, ao centralizar a operação das políticas de acesso, padroniza a autorização segundo o princípio de menor privilégio. A equipe de segurança/desenvolvimento pode atualizar regras de controle de acesso sem a necessidade de modificar o código-fonte. A utilização do serviço de *authorization as a service* também fornece ganhos no monitoramento das requisições realizadas aos microsserviços para detectar anomalias de segurança com base na distribuição de chamadas das APIs observadas. Na detecção de ameaças, é possível recriar dinamicamente e aplicar novas regras de controle de acesso para mitigar riscos de segurança. Por se aproveitar de um serviço de nuvem, pode-se dizer que ele é escalável, tendo em vista que a operacionalização dos servidores internos não é de responsabilidade do sistema que se certifica da autorização, limitando-se apenas ao plano de subscrição escolhido.

5.1 TRABALHOS FUTUROS

Trabalhos futuros na área de autorização de microsserviços podem se concentrar na integração de tecnologias emergentes, como *Kubernetes* e computação *serverless*. A orquestração avançada oferecida pelo *Kubernetes* pode ser explorada para otimizar a distribuição e escalabilidade de microsserviços, ao passo que a computação *serverless* proporciona uma abordagem mais granular, executando funções específicas em resposta a eventos. Investigar como essas tecnologias podem ser sinergicamente aplicadas na autorização de microsserviços pode resultar em estratégias mais eficientes, seguras e adaptáveis.

Outra linha de pesquisa promissora é a exploração de modelos de autorização contextual e adaptativa para microsserviços. Isso envolve o desenvolvimento de sistemas que possam ajustar dinamicamente as políticas de autorização com base no contexto operacional e nas características específicas dos usuários (*Attributed Based Access Control*). A implementação de mecanismos que reconheçam a volatilidade inerente aos ambientes de microsserviços pode proporcionar uma autorização mais granular e sensível ao contexto, contribuindo para uma segurança mais eficaz.

A avaliação do desempenho e eficiência operacional de diferentes estratégias de autorização em ambientes de microsserviços é uma área que merece uma atenção aprofundada em trabalhos futuros, podendo ser explorado a implementação de outros *tokens* de autorização como PASETO e Macaroon, realizando assim a comparação de métricas como latência, consumo de recursos e escalabilidade entre abordagens diversas pode fornecer *insights* cruciais para a seleção da melhor estratégia de autorização em um dado contexto. Considerar como essas métricas são afetadas por fatores como o número de microsserviços, volume de tráfego e complexidade da política de autorização é essencial para desenvolver soluções que atendam tanto aos requisitos de segurança quanto às necessidades operacionais dos sistemas baseados em microsserviços.

REFERÊNCIAS BIBLIOGRÁFICAS

1 WHAT is identity and access management (IAM)? [S.l.], 2023. Acessado: 2023-11-11. Disponível em: <https://www.ibm.com/topics/identity-access-management>.

2 SHOKAR URL= https://medium.com/@darinder.shokar/script-for-executing-the-oauth2-authorization-code-flow-in-forgerock-access-management-am-ddd8728586a5, n. . A. D. *Script for Executing the OAuth2 Authorization Code Flow with PKCE in ForgeRock Access Management (AM)*. [S.l.], 2019.

3 SANDHU, R. S.; SAMARATI, P. Access control: principle and practice. *IEEE communications magazine*, IEEE, v. 32, n. 9, p. 40–48, 1994.

4 TOLONE, W.; AHN, G.-J.; PAI, T.; HONG, S.-P. Access control in collaborative systems. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 37, n. 1, p. 29–41, 2005.

5 WEAVEWORKS. *Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs*. [S.l.], 2020. Acessado: 2023-11-25. Disponível em: <https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms>.

6 STAFFORD, V. Zero trust architecture. *NIST special publication*, v. 800, p. 207, 2020.

7 THE layered defense approach to security. [S.l.], 2023. Acessado: 2023-09-09. Disponível em: <https://www.ibm.com/docs/en/i/7.3?topic=security-layered-defense-approach>.

8 WIJAYARATHNA, C.; ARACHCHILAGE, N. A. An empirical usability analysis of the google authentication api. In: *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*. [S.l.: s.n.], 2019. p. 268–274.

9 COST of a Data Breach Report 2023. [S.l.], 2023. Acessado: 2023-09-09. Disponível em: <https://www.ibm.com/reports/data-breach>.

10 GREENLEAF, G. Now 157 countries: Twelve data privacy laws in 2021/22. 2022.

11 TIAN, Z.; SHI, W.; WANG, Y.; ZHU, C.; DU, X.; SU, S.; SUN, Y.; GUIZANI, N. Real-time lateral movement detection based on evidence reasoning network for edge computing environment. *IEEE Transactions on Industrial Informatics*, IEEE, v. 15, n. 7, p. 4285–4294, 2019.

12 SINGH, S.; JEONG, Y.-S.; PARK, J. H. A survey on cloud computing security: Issues, threats, and solutions. *Journal of Network and Computer Applications*, Elsevier, v. 75, p. 200–222, 2016.

13 RAJAPAKSE, R. N.; ZAHEDI, M.; BABAR, M. A.; SHEN, H. Challenges and solutions when adopting devsecops: A systematic review. *Information and software technology*, Elsevier, v. 141, p. 106700, 2022.

14 ANDRADE, A. L. L. de. Estudo comparativo de tokens de autorizaÇÃo em apis rest. In: *CIAWI*. [S.l.: s.n.], 2023.

15 ALMEIDA, M. G. de; CANEDO, E. D. Authentication and authorization in microservices architecture: A systematic literature review. *Applied Sciences*, MDPI, v. 12, n. 6, p. 3023, 2022.

16 NEWMAN, S. *Building microservices*. [S.l.]: "O'Reilly Media, Inc.", 2021.

17 PEREIRA-VALE, A.; MÁRQUEZ, G.; ASTUDILLO, H.; FERNANDEZ, E. B. Security mechanisms used in microservices-based systems: A systematic mapping. In: IEEE. *2019 XLV Latin American Computing Conference (CLEI)*. [S.l.], 2019. p. 01–10.

18 HANNOUSSE, A.; YAHIOUCHE, S. Securing microservices and microservice architectures: A systematic mapping study. *Computer Science Review*, Elsevier, v. 41, p. 100415, 2021.

19 YU, D.; JIN, Y.; ZHANG, Y.; ZHENG, X. A survey on security issues in services communication of microservices-enabled fog applications. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 31, n. 22, p. e4436, 2019.

20 THORGERSEN, S.; SILVA, P. I. *Keycloak-identity and access management for modern applications: harness the power of Keycloak, OpenID Connect, and OAuth 2.0 protocols to secure applications*. [S.l.]: Packt Publishing Ltd, 2021.

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

22 HE, X.; YANG, X. Authentication and authorization of end user in microservice architecture. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2017. v. 910, n. 1, p. 012060.

23 JANDER, K.; BRAUBACH, L.; POKAHR, A. Defense-in-depth and role authentication for microservice systems. *Procedia computer science*, Elsevier, v. 130, p. 456–463, 2018.

24 TORKURA, K. A.; SUKMANA, M. I.; MEINEL, C. Integrating continuous security assessments in microservices and cloud native applications. In: *Proceedings of the10th International Conference on Utility and Cloud Computing*. [S.l.: s.n.], 2017. p. 171–180.

25 O que é o gerenciamento de identidade e acesso (IAM)? [S.l.], 2023. Acessado: 2023-16-09. Disponível em: <https://learn.microsoft.com/pt-br/azure/active-directory/fundamentals/introduction-identity-access-management>.

26 WHITMAN, M. E.; MATTORD, H. J. *Principles of information security*. [S.l.]: Cengage learning, 2021.

27 INDU, I.; ANAND, P. R.; SHAJI, S. P. Secure file sharing mechanism and key management for mobile cloud computing environment. *Indian Journal of Science and Technology*, v. 9, n. 48, p. 1–8, 2016.

28 THE OAuth 2.0 Authorization Framework. [S.l.], 2012. Https://www.rfc-editor.org/rfc/rfc6749. Disponível em: <https://auth0.com/docs/authenticate/protocols/openid-connect-protocol>.

29 SIMMONS, G. J. A survey of information authentication. *Proceedings of the IEEE*, IEEE, v. 76, n. 5, p. 603–620, 1988.

30 CREDENTIALS Definition. [S.l.], 2023. Acessado: 2023-16-09. Disponível em: <https://nordvpn.com/pt/cybersecurity/glossary/credentials/>.

31 LAL, N. A.; PRASAD, S.; FARIK, M. A review of authentication methods. *International journal of scientific & technology research*, v. 5, n. 11, p. 246–249, 2016.

32 APPLE Vision Pro. [S.l.], 2023. Acessado: 2023-16-09. Disponível em: <https://www.apple.com/apple-vision-pro/>.

33 FORCE, J. T. Assessing security and privacy controls in information systems and organizations. *NIST Special Publication*, v. 800, p. 53A, 2022.

34 SHAH, J. *FIDO2 Authentication vs. FIDO: What's the Difference?* [S.l.], 2022. Acessado: 2023-11-25. Disponível em: <https://www.1kosmos.com/authentication/fido2-authentication/>.

35 LASSAK, L.; PAN, E.; UR, B.; GOLLA, M. Why aren't we using passkeys? obstacles companies face deploying fido2 passwordless authentication (extended version).

36 OPENID Connect Protocol. [S.l.], 2023. Acessado: 2023-27-09. Disponível em: <https://auth0.com/docs/authenticate/protocols/openid-connect-protocol>.

37 LAMPSON, B. W. Protection. *ACM SIGOPS Operating Systems Review*, ACM New York, NY, USA, v. 8, n. 1, p. 18–24, 1974.

38 SANDHU, R.; MUNAWER, Q. How to do discretionary access control using roles. In: *Proceedings of the third ACM workshop on Role-based access control*. [S.l.: s.n.], 1998. p. 47–54.

39 SANDHU, R. S. Role-based access control. In: *Advances in computers*. [S.l.]: Elsevier, 1998. v. 46, p. 237–286.

40 HU, V. C.; FERRAIOLO, D.; KUHN, R.; FRIEDMAN, A. R.; LANG, A. J.; COGDELL, M. M.; SCHNITZER, A.; SANDLIN, K.; MILLER, R.; SCARFONE, K. et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST special publication*, Citeseer, v. 800, n. 162, p. 1–54, 2013.

41 IDC Top 10 Predictions For Worldwide IT, 2019. [S.l.], 2019. Acessado: 2023-27-09. Disponível em: <https://www.forbes.com/sites/louiscolumbus/2018/11/04/idc-top-10-predictions-for-worldwide-it-2019/?sh=285748a47b96>.

42 ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. *Computer networks*, Elsevier, v. 54, n. 15, p. 2787–2805, 2010.

43 KNAPPMEYER, M.; KIANI, S. L.; REETZ, E. S.; BAKER, N.; TONJES, R. Survey of context provisioning middleware. *IEEE Communications Surveys & Tutorials*, IEEE, v. 15, n. 3, p. 1492–1519, 2013.

44 YAN, L.; ZHANG, Y.; YANG, L. T.; NING, H. *The Internet of things: from RFID to the next-generation pervasive networked systems*. [S.l.]: Crc Press, 2008.

45 SE containers to Build, Share and Run your applications. [S.l.], 2023. Acessado: 2023-28-09. Disponível em: <https://www.docker.com/resources/what-container/#:~:text=A%20Docker%20container%20image%20is,tools%2C%20system%20libraries%20and%20settings.>

46 RODRÍGUEZ, C.; BAEZ, M.; DANIEL, F.; CASATI, F.; TRABUCCO, J. C.; CANALI, L.; PERCANNELLA, G. Rest apis: A large-scale analysis of compliance with principles and best practices. In: SPRINGER. *Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16*. [S.l.], 2016. p. 21–39.

47 PAUTASSO, C.; WILDE, E. Restful web services: principles, patterns, emerging technologies. In: *Proceedings of the 19th international conference on World wide web*. [S.l.: s.n.], 2010. p. 1359–1360.

48 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.

49 GUPTA, S.; GUPTA, B. B. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, Springer, v. 8, p. 512–530, 2017.

50 ZHENG, K.; JIANG, W. A token authentication solution for hadoop based on kerberos pre-authentication. In: IEEE. *2014 International Conference on Data Science and Advanced Analytics (DSAA)*. [S.l.], 2014. p. 354–360.

51 SHAH, J. *Authentication Token*. [S.l.], 2022. Acessado: 2023-11-25. Disponível em: <https://www.fortinet.com/resources/cyberglossary/authentication-token>.

52 JONES, M.; BRADLEY, J.; SAKIMURA, N. *Rfc 7519: Json web token (jwt)*. [S.l.]: RFC Editor, 2015.

53 PASETO (Platform-Agnostic SEcurity TOkens) draft-paragon-paseto-rfc-01. [S.l.], 2023. Acessado: 2023-25-11. Disponível em: <https://www.ietf.org/archive/id/draft-paragon-paseto-rfc-01.txt>.

54 DEGGES, R. *A Thorough Introduction to PASETO*. [S.l.], 2019. Acessado: 2023-11-11. Disponível em: <https://developer.okta.com/blog/2019/10/17/a-thorough-introduction-to-paseto>.

55 BIRGISSON, A.; POLITZ, J. G.; ERLINGSSON, U.; TALY, A.; VRABLE, M.; LENTCZNER, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. 2014.

56 WHAT is permit.io. [S.l.], 2012. Introduction to Permit.io, acessado em 2023-10-09. Disponível em: <https://docs.permit.io/>.

I.1 ARQUIVO DOCKER COMPOSE

Abaixo, segue o arquivo docker compose para o deploy dos microsserviços utilizados no capítulo 4.

```
version: '3'

volumes:
  postgres_data:
    driver: local

networks:
  keycloak_network:
    driver: bridge

services:
  postgres:
    image: postgres:11
    container_name: postgresdb
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: keycloak
      POSTGRES_USER: keycloak
      POSTGRES_PASSWORD: password
    networks:
      - keycloak_network
    ports:
      - 5433:5432

  keycloak:
    image: jboss/keycloak:16.1.0
    container_name: keycloak
    environment:
      DB_VENDOR: POSTGRES
      DB_ADDR: postgres
      DB_DATABASE: keycloak
      DB_USER: keycloak
      DB_SCHEMA: public
      DB_PASSWORD: password
      KEYCLOAK_USER: admin
      KEYCLOAK_PASSWORD: password
      KEYCLOAK_LOGLEVEL: DEBUG
      ROOT_LOGLEVEL: DEBUG
    networks:
      - keycloak_network
```

```
42      ports:
43        - 8080:8080
44        - 8443:8443
45      depends_on:
46        - postgres
47
48    frontend:
49      build:
50        context: /home/alla/Documents/code/Keycloak-Identity-and-Access-Management-
              for-Modern-Applications/ch2/frontend
51        dockerfile: Dockerfile
52      image: frontend
53      container_name: frontend
54      networks:
55        - keycloak_network
56      ports:
57        - 3000:8000
58
59    myservice:
60      build:
61        context: /home/alla/Documents/code/Keycloak-Identity-and-Access-Management-
              for-Modern-Applications/MyService
62        dockerfile: Dockerfile
63      image: myservice
64      container_name: myservice
65      networks:
66        - keycloak_network
67      ports:
68        - 8085:8000
69
70  #  oidc_playground:
71  #    image: oidc-playground
72  #    ports:
73  #      - 8000:8000
74
75  #  backend:
76  #    build:
77  #      context: /home/alla/Documents/code/Keycloak-Identity-and-Access-Management-
       for-Modern-Applications/ch2/backend
78  #      dockerfile: Dockerfile
79  #    image: result/latest
80  #    ports:
81  #      - 3000:3000
```

I.2 CÓDIGO FRONTEND

```
1     var express = require('express');
```

```
var app = express();
var stringReplace = require('string-replace-middleware');

var KC_URL = process.env.KC_URL || "http://localhost:8080/auth";
var SERVICE_URL = process.env.SERVICE_URL || "http://localhost:8085/secured";

app.use(stringReplace({
    'SERVICE_URL': SERVICE_URL,
    'KC_URL': KC_URL
}));
app.use(express.static('.'))

app.get('/', function(req, res) {
    res.render('index.html');
});

app.get('/client.js', function(req, res) {
    res.render('client.js');
});

app.listen(8000);
```

I.3 CÓDIGO BACKEND COM VALIDAÇÃO DE TOKEN EM ARQUITETURA *MESH*

```
    from fastapi import FastAPI, Header, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from typing import Union
import jwt
from jwt import PyJWKClient

url = "http://keycloak:8080/auth/realms/myrealm/protocol/openid-connect/certs"
ALGORITHM = 'RS256'


def verifyJWT(token):
    jwks_client =  PyJWKClient(url)
    signing_key = jwks_client.get_signing_key_from_jwt(token)
    try:
        tokenPayload = jwt.decode(token, signing_key.key, algorithms=[ALGORITHM, ]
            , audience='account', issuer='http://localhost:8080/auth/realms/
            myrealm')
        return tokenPayload
    except:
        raise HTTPException(
            status_code=498,
            detail= "Not authorized. This token is not valid: " + token
        )
```

```
22
23 app = FastAPI()
24
25 origins = [
26     "http://localhost",
27     "http://localhost:8000",
28     "http://localhost:3000"
29 ]
30
31 app.add_middleware(
32     CORSMiddleware,
33     allow_origins=origins,
34     allow_credentials=True,
35     allow_methods=["*"],
36     allow_headers=["*"],
37 )
38
39 @app.get("/")
40 def read_root():
41     return {"Hello": "World"}
42
43 @app.get("/secured")
44 async def get_secured(authorization:str = Header(default=None)):
45     if verifyJWT(authorization.split()[1])['assignedRole'] in ["administrator", "
          operator", "developer", "guest"]:
46         return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ",
               you GET in a secured area"
47     else:
48         raise HTTPException(
49             status_code=401,
50             detail= "Not authorized!"
51         )
52
53
54 @app.post("/secured")
55 async def post_secured(authorization:str = Header(default=None)):
56     if verifyJWT(authorization.split()[1])['assignedRole'] in ["administrator", "
          operator", "developer"]:
57         return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ",
               you POST in a secured area"
58     else:
59         raise HTTPException(
60             status_code=401,
61             detail= "Not authorized!"
62         )
63
64 @app.put("/secured")
65 async def put_secured(authorization:str = Header(default=None)):
66     if verifyJWT(authorization.split()[1])['assignedRole']in ["administrator", "
          operator"]:
67         return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ",
               you PUT in a secured area"
```

```
68      else:
69          raise HTTPException(
70              status_code=401,
71              detail= "Not authorized!"
72          )
73
74
75  @app.delete("/secured")
76  async def delete_secured(authorization:str = Header(default=None)):
77      if verifyJWT(authorization.split()[1])['assignedRole'] == "administrator":
78          return "Hello " + verifyJWT(authorization.split()[1])['assignedRole'] + ",
                 you DELETE in a secured area"
79      else:
80          raise HTTPException(
81              status_code=401,
82              detail= "Not authorized!"
83          )
84
85
86
87  @app.get("/items/{item_id}")
88  def read_item(item_id: int, q: Union[str, None] = None):
89      return {"item_id": item_id, "q": q}
```

I.4 CÓDIGO BACKEND COM VALIDAÇÃO DE TOKEN *AUTHORIZATION AS A SERVICE*

```
1       import asyncio, jwt
2   from jwt import PyJWKClient
3   from permit import Permit
4   from fastapi import FastAPI, status, HTTPException, Header
5   from fastapi.responses import JSONResponse
6   import permit
7
8
9   url = "http://localhost:8080/auth/realms/myrealm/protocol/openid-connect/certs"
10  ALGORITHM = 'RS256'
11
12  app = FastAPI()
13
14  # This line initializes the SDK and connects your python app
15  # to the Permit.io PDP container you've set up in the previous step.
16  permit = Permit(
17      # in production, you might need to change this url to fit your deployment
18      pdp="https://cloudpdp.api.permit.io",
19      # your api key
```

```
    token="
        permit_key_r30WSyfPypsGVC8mrK6vUuCix3ql9dgelFwYZUvhj2KD1Jom1MY7TWQ3oevItNYPVRwnwsvea1DDe
        ",
)

# This user was defined by you in the previous step and
# is already assigned with a role in the permission system.
user = {
    "id": "guest@gmail.com",
    "firstName": "gu",
    "lastName": "est",
    "email": "guest@gmail.com",
} # in a real app, you would typically decode the user id from a JWT token

def verifyJWT(token):
    jwks_client =  PyJWKClient(url)
    signing_key = jwks_client.get_signing_key_from_jwt(token)
    try:
        tokenPayload = jwt.decode(token, signing_key.key, algorithms=[ALGORITHM,
            ], audience='account')
        return tokenPayload
    except:
        raise HTTPException(
            status_code=498,
            detail= "Not authorized. This token is not valid: " + token
        )

@app.get("/secured")
async def get_secured(authorization:str = Header(default=None)):
    # After we created this user in the previous step, we also synced the user's
        identifier
    # to permit.io servers with permit.write(permit.api.syncUser(user)). The user
        identifier
    # can be anything (email, db id, etc) but must be unique for each user. Now
        that the
    # user is synced, we can use its identifier to check permissions with `
        permit.check()`.
    permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "
        get", "myservice")
    if not permitted:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
            "result": "NOT PERMITTED to GET myservice!"
        })

    return JSONResponse(status_code=status.HTTP_200_OK, content={
        "result": "PERMITTED to GET myservice!"
    })

@app.post("/secured")
async def post_secured(authorization:str = Header(default=None)):
    # After we created this user in the previous step, we also synced the user's
        identifier
```

```
63      # to permit.io servers with permit.write(permit.api.syncUser(user)). The user
            identifier
64      # can be anything (email, db id, etc) but must be unique for each user. Now
            that the
65      # user is synced, we can use its identifier to check permissions with `
            permit.check()`.
66      permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "
            post", "myservice")
67      if not permitted:
68          raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
69              "result": "NOT PERMITTED to POST myservice!"
70          })
71
72      return JSONResponse(status_code=status.HTTP_200_OK, content={
73          "result": "PERMITTED to POST myservice!"
74      })
75
76  @app.put("/secured")
77  async def post_secured(authorization:str = Header(default=None)):
78      # After we created this user in the previous step, we also synced the user's
            identifier
79      # to permit.io servers with permit.write(permit.api.syncUser(user)). The user
            identifier
80      # can be anything (email, db id, etc) but must be unique for each user. Now
            that the
81      # user is synced, we can use its identifier to check permissions with `
            permit.check()`.
82      permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "
            put", "myservice")
83      if not permitted:
84          raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
85              "result": "NOT PERMITTED to PUT myservice!"
86          })
87
88      return JSONResponse(status_code=status.HTTP_200_OK, content={
89          "result": "PERMITTED to PUT myservice!"
90      })
91
92
93  @app.delete("/secured")
94  async def post_secured(authorization:str = Header(default=None)):
95      # After we created this user in the previous step, we also synced the user's
            identifier
96      # to permit.io servers with permit.write(permit.api.syncUser(user)). The user
            identifier
97      # can be anything (email, db id, etc) but must be unique for each user. Now
            that the
98      # user is synced, we can use its identifier to check permissions with `
            permit.check()`.
99      permitted = await permit.check(verifyJWT(authorization.split()[1])['email'], "
            delete", "myservice")
100     if not permitted:
```

```
101             raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={
102                 "result": "NOT PERMITTED to DELETE myservice!"
103             })
104 
105         return JSONResponse(status_code=status.HTTP_200_OK, content={
106             "result": "PERMITTED to DELETE myservice!"
107         })
```