Lucas Silva Mendes

Estudo da Geração de Código via JS-Distributor para Sistema de Autenticação e Autorização

São Carlos, SP

2025

Lucas Silva Mendes

Estudo da Geração de Código via JS-Distributor para Sistema de Autenticação e Autorização

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia de Computação da Universidade Federal de São Carlos, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Universidade Federal de São Carlos - UFSCar

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação

Orientador: Daniel Lucrédio

São Carlos, SP

2025

Lucas Silva Mendes

Estudo da Geração de Código via JS-Distributor para Sistema de Autenticação e Autorização

Trabalho de Conclusão de Curso apresentado ao curso de Engenharia de Computação da Universidade Federal de São Carlos, como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Daniel Lucrédio
Orientador

Delano M. Beder
Professor Convidado

Helio C. Guardia
Professor Convidado

São Carlos, SP
2025

Este trabalho é dedicado a todas as mãos que me ajudaram a levantar nas dificuldades, aos ombros onde pude descansar e aos ouvidos que pacientemente souberam me escutar. Família e amigos, eu não seria nada sem vocês.

Agradecimentos

Os agradecimentos principais são aos meus pais. Durante toda a minha vida, desde o momento que decidi sair de casa para correr atrás dos meus sonhos, ainda no ensino médio, vocês sempre foram o lugar seguro que podia retornar. Sem o apoio de vocês eu não teria conquistado metade do que fiz, e por isso vocês sempre terão minha gratidão.

Aos meus amigos, sejam aqueles que conheci nessa jornada pela universidade, ou aqueles que carrego comigo desde a adolescência, obrigado pelas conversas na madrugada quando a vida estava complicada, as ajudas nas tarefas acadêmicas, as colaborações que foram essenciais para minhas conquistas. Sem sua parceria, nada disso seria possível.

Ao Prof. Dr. Daniel Lucrédio, pela orientação e paciência com minha inexperiência para a realização deste trabalho. Sem sua ajuda, não teria conseguido lapidar a minha ideia bruta em um projeto acadêmico viável.

A todos os professores e colaboradores que tive contato na Universidade Federal de São Carlos, obrigado por todo conhecimento e infraestrutura que me foi fornecida ao longo desses anos. Só fui capaz de aprender e me desenvolver tanto graças a seus esforços.

Por fim, agradeço a todos que me ajudaram de qualquer forma ao longo dessa jornada. Sejam com palavras, carinho ou orientações, todas as pessoas que trouxeram algo de positivo para minha jornada estarão sempre em meu coração. Esse é apenas o começo da jornada, e como ela vai ser trilhada e todos aqueles que passarem por ela, são muito mais importantes que o destino.

“O passo mais importante que um homem pode dar. Não é o primeiro, certo? É o próximo.
Sempre o próximo passo..."
(Brandon Sanderson, 2017)

Resumo

A arquitetura de microsserviços consolidou-se como um paradigma dominante no desenvolvimento de software, oferecendo flexibilidade e escalabilidade ao permitir que componentes sejam desenvolvidos e implantados de forma autônoma. Por isso, é comum muitas empresas migrarem seus monólitos para essa arquitetura, e a ferramenta `js-distributor` foi criada para facilitar essa transição. No entanto, essa distribuição inerente gera desafios para a implementação de uma comunicação segura e um controle de acesso robusto entre os serviços. Este trabalho tem como objetivo sugerir aprimoramentos à ferramenta `js-distributor`, propondo uma abordagem para acrescentar mais segurança aos serviços gerados utilizando o modelo de Controle de Acesso Baseado em Funções (RBAC). A implementação foi realizada com o Keycloak, um provedor de identidade e acesso *self-hosted*, que possibilita a geração da lógica de segurança a partir de uma configuração inicial. Com isso, busca-se abstrair a complexidade da configuração manual de um sistema de autorização, oferecendo aos desenvolvedores um mecanismo eficiente para proteger funcionalidades e recursos em projetos baseados em microsserviços. Como resultado, foi elaborada uma estratégia para a geração de código, aproveitando-se do funcionamento existente da ferramenta de distribuição, a qual foi validada em sistemas exemplo.

Palavras-chave: Geração de código, microsserviços, aplicações distribuídas, desenvolvimento web, segurança, RBAC, Keycloak

Abstract

The microservices architecture has established itself as a dominant paradigm in software development, offering flexibility and scalability by allowing components to be developed and deployed autonomously. Consequently, it is common for many companies to migrate their monoliths to this architecture, and the `js-distributor` tool was created to facilitate this transition. However, this inherent distribution poses challenges for implementing secure communication and robust access control between services. This work aims to suggest enhancements to the `js-distributor` tool, proposing an approach to increase the security of the generated services using the Role-Based Access Control (RBAC) model. The implementation was carried out using Keycloak, a self-hosted identity and access provider, which allows for the generation of security logic from an initial configuration. Thereby, we seek to abstract the complexity of manually configuring an authorization system, offering developers an efficient mechanism to protect functionalities and resources in microservice-based projects. As a result, a code generation strategy was devised, leveraging the tool's existing mechanics, which was validated on example systems.

Keywords: Code Generation, Microservices, Distributed Applications, Web Development, Security, RBAC, Keycloak.

Lista de ilustrações

Figura 1 – Exemplo de transformação de código monolítico em microsserviços pelo js-distributor 17
Figura 2 – Arquitetura distribuída inicial (sem segurança) 29
Figura 3 – Monólito simples após distribuição e proteção 30
Figura 4 – Fluxo de propagação transparente do Token JWT entre serviços 39
Figura 5 – Interface da aplicação Acme Air em funcionamento após a decomposição em serviços Alpha, Beta, Gamma e Delta. 47
Figura 6 – Arquitetura de microsserviços do Acme Air protegida pela solução proposta. 48
Figura 7 – Fluxo consolidado da estratégia de segurança no js-distributor 51

Lista de tabelas

Tabela 1 – Parâmetros Globais de Segurança (Seção `security`) 49
Tabela 2 – Parâmetros de Segurança por Serviço 50

Lista de abreviaturas e siglas

API	*Application Programming Interface* (Interface de Programação de Aplicação)
HTTP	*Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto)
IAM	*Identity and Access Management* (Gestão de Identidade e Acesso)
IdP	*Identity Provider* (Provedor de Identidade)
IETF	*Internet Engineering Task Force* (Força-Tarefa de Engenharia da Internet)
JSON	*JavaScript Object Notation* (Notação de Objetos JavaScript)
JWT	*JSON Web Token*
OIDC	*OpenID Connect*
RBAC	*Role-Based Access Control* (Controle de Acesso Baseado em Papéis)
RFC	*Request for Comments*
TCC	Trabalho de Conclusão de Curso

Sumário

1 INTRODUÇÃO . . . 11

2 FUNDAMENTAÇÃO TEÓRICA . . . 14
2.1 Principais conceitos envolvidos . . . 14
2.1.1 Arquitetura de microsserviços e a decomposição de monólitos . . . 14
2.1.2 A ferramenta `js-distributor` . . . 15
2.1.3 Autenticação e Autorização . . . 17
2.1.4 Controle de Acesso Baseado em Funções (RBAC) . . . 18
2.1.5 *JSON Web Token* (JWT) . . . 18
2.1.6 Protocolos de Autorização e Identidade: OAuth 2.0 e OpenID Connect . . . 19
2.1.7 Keycloak . . . 20
2.2 Principais Trabalhos Relacionados . . . 22

3 METODOLOGIA . . . 24
3.1 Levantamento Bibliográfico . . . 24
3.2 Estudo de Caso Exploratório . . . 24
3.3 Estudo do Caso de Validação . . . 25
3.4 Criação do roteiro de implementação . . . 26

4 RESULTADOS . . . 27
4.1 Cenário de Referência . . . 27
4.1.1 Definição da Arquitetura de Segurança . . . 29
4.2 Configuração do *realm* do Keycloak via Terraform . . . 31
4.3 Implementação da biblioteca de contexto compartilhado . . . 39
4.4 Refatoração do serviço de entrada . . . 41
4.5 Refatoração nos serviços de recursos . . . 43
4.6 Cenário de validação: Acme Air . . . 45
4.6.1 Adaptação da Arquitetura . . . 46
4.6.2 Resultados da Integração . . . 47
4.7 Estratégia para futura implementação no `js-distributor` . . . 48

5 CONCLUSÃO . . . 52
5.1 Limitações e Trabalhos Futuros . . . 52

REFERÊNCIAS . . . 54

1 Introdução

A crescente complexidade dos sistemas de software impõe desafios significativos ao desenvolvimento de aplicações modernas. Requisitos como escalabilidade, agilidade na entrega de novas funcionalidades e tolerância a falhas tornaram-se fatores essenciais para a competitividade das organizações. Durante muitos anos, o estilo arquitetural monolítico, no qual uma aplicação é desenvolvida e implantada como uma única unidade, foi a abordagem predominante.

No entanto, esse modelo apresenta limitações importantes. Alterações realizadas em uma pequena parte do sistema exigem a reconstrução e a redistribuição de toda a aplicação, o que reduz a eficiência do processo de desenvolvimento. Além disso, com o passar do tempo, a manutenção de uma estrutura modular bem definida torna-se cada vez mais complexa, dificultando o isolamento de mudanças que deveriam afetar apenas um módulo específico. A escalabilidade também se torna um problema, pois é necessário escalar todo o sistema, mesmo quando apenas determinados componentes demandam mais recursos (FOWLER; LEWIS, 2014).

Visando corrigir alguns desses problemas, a arquitetura de microsserviços foi proposta. Em resumo, é uma abordagem para desenvolver uma única aplicação como uma suíte de pequenos serviços, cada um executando em seu próprio processo e comunicando-se com mecanismos leves, frequentemente uma API de recurso HTTP (FOWLER; LEWIS, 2014), ou tecnologias de mensageria como RabbitMQ[1] ou Apache Kafka[2].

Apesar dos benefícios, a adoção da arquitetura de microsserviços introduz pontos negativos intrínsecos à segurança. A própria descentralização dos serviços resulta em uma superfície de ataque consideravelmente maior quando comparada a uma arquitetura monolítica. Essa ampliação dos pontos de entrada para potenciais ataques exige uma maior e mais complexa coordenação entre as equipes de desenvolvimento e os componentes do sistema para garantir a proteção adequada de toda a aplicação (MATEUS-COELHO; CRUZ-CUNHA; FERREIRA, 2021).

Para garantir essa segurança, é preciso se atentar a dois principais fatores: autenticação e autorização. Segundo Pereira-Vale et al. (2019), esses mecanismos intimamente relacionados podem ser definidos como:

- **Autorização:** sistemas que indicam quem é autorizado a acessar determinados

[1] RabbitMQ é um software de mensageria (message-broker) de código aberto amplamente utilizado que implementa o protocolo AMQP. Disponível em: <https://www.rabbitmq.com/>.

[2] Apache Kafka é uma plataforma de streaming de eventos distribuída de código aberto utilizada para pipelines de dados de alto desempenho. Disponível em: <https://kafka.apache.org/>.

recursos do sistema e de que forma;

- **Autenticação:** sistemas que verificam se o indivíduo que está acessando o sistema é quem ele diz ser.

Diante desse contexto, a ferramenta `js-distributor` foi desenvolvida para auxiliar desenvolvedores na migração de códigos monolíticos para microsserviços. A ferramenta automatiza a geração de código necessário para a configuração de servidores e a comunicação, suportando requisições HTTP API ou mensagens assíncronas baseadas em filas com RabbitMQ. Isso permite que as funções sejam inicialmente implementadas em um único código monolítico e, posteriormente, migradas para uma arquitetura distribuída sem a necessidade de adicionar ou modificar o código original (ESCHER et al., 2025a).

No entanto, ela não possui opções nativas para a criação automatizada de um sistema de segurança de acesso baseado em autenticação e autorização. Caso o desenvolvedor deseje que os serviços gerados sejam protegidos, ele necessitará desenvolvê-los manualmente, antes ou depois da distribuição do monólito. Isso pode ser feito criando sistemas locais para cada serviço ou conectando todos a um serviço dedicado à segurança.

Visando simplificar essa implementação, a proposta desse trabalho foi estudar como um sistema de *Role Based Access Control* (Controle de Acesso baseado em Papéis ou RBAC) poderia ser gerado pela `js-distributor`, visando automatizar esse processo. Em pesquisa sobre tecnologias compatíveis no mercado, foi escolhido o Keycloak[3], uma solução *open-source*. Ele é uma solução de Gerenciamento de Identidade e Acesso (em inglês, *Identity and Access Management* ou IAM) que fornece autenticação e autorização centralizadas para aplicações.

Ele atua como um servidor de autorização central, emitindo tokens no formato *JSON Web Token* (JWT) quando um usuário se autentica com sucesso. Estes tokens contêm informações sobre o usuário (*claims*) e suas permissões, codificados e compactados para a transmissão de informações, e que podem ser verificados pelos microsserviços sem necessidade de consulta contínua ao servidor Keycloak.

Desta forma, este trabalho se dedica a sugerir uma estratégia para implementar mecanismos de geração de segurança na ferramenta `js-distributor`. O objetivo central foi definir e validar uma arquitetura que integre, de forma automática, o controle de quem acessa (autenticação) e do que cada usuário pode fazer nos sistemas gerados (autorização). Para isso, utilizamos padrões de mercado para gerenciamento de identidade (Keycloak) e técnicas para configurar toda essa proteção através de código, garantindo que o processo seja replicável e livre de erros manuais. O desenvolvimento foi testado em dois sistemas de microsserviços distintos, gerados pelo `js-distributor` a partir de sistemas monolíticos.

[3] Disponível em: https://www.keycloak.org/.

A solução proposta organiza a forma como os serviços conversam entre si, assegurando que a identificação do usuário seja transportada e validada em cada etapa do processo, seja na porta de entrada ou nos serviços internos. Dessa forma, todo o código criado para garantir o funcionamento do sistema foi sintetizado em uma estratégia para a futura implementação dessas funcionalidades no `js-distributor`. A intenção final é sugerir a evolução da ferramenta para que ela entregue sistemas distribuídos não apenas funcionais, mas também robustos, e que já nasçam com um maior grau de proteção contra acessos indevidos.

2 Fundamentação teórica

Neste capítulo, são apresentados os principais conceitos necessários para o entendimento deste trabalho (Seção 2.1) e os principais trabalhos relacionados, que serviram de referência e embasamento para o desenvolvimento (Seção 2.2).

2.1 Principais conceitos envolvidos

2.1.1 Arquitetura de microsserviços e a decomposição de monólitos

A arquitetura monolítica é uma abordagem que foi muito utilizada para a construção de software por um longo tempo. Nesse modelo, toda a aplicação é desenvolvida como um único bloco, autossuficiente e independente de outros sistemas. Porém, devido ao fato de todas as suas funcionalidades ficarem concentradas em um mesmo conjunto de componentes, à medida que o sistema evolui e novas funções são adicionadas, elas podem se tornar fortemente acopladas e pouco coesas (ABGAZ et al., 2023).

Embora essa arquitetura ainda seja muito utilizada, ela apresenta limitações ao escalar o sistema em tamanho e complexidade. Os principais desafios são a dificuldade de manutenção, a baixa eficiência ao escalar (pois é necessário replicar toda a aplicação, e não apenas componentes isolados) e os ciclos de desenvolvimento mais lentos. Pequenas alterações em uma parte do código, por exemplo, acabam exigindo a recompilação, os testes e a implantação de toda a aplicação, o que aumenta o tempo e o esforço de desenvolvimento (ABGAZ et al., 2023; FOWLER; LEWIS, 2014).

Em resposta a esses desafios, surgiu o estilo arquitetural de microsserviços. A definição popularizada por Fowler e Lewis (2014) descreve essa abordagem como "o desenvolvimento de uma única aplicação como uma suíte de pequenos serviços, cada um executando em seu próprio processo e comunicando-se com mecanismos leves, frequentemente uma API de recurso HTTP". Esses serviços são construídos em torno de capacidades de negócio específicas, podem ser implantados de forma independente, inclusive em linguagens de programação distintas, e possuem um gerenciamento centralizado mínimo. O resultado é um sistema mais ágil, robusto, escalável e resiliente a falhas.

No entanto, realizar a transição de um sistema monolítico para um baseado em microsserviços pode ser um processo custoso e trabalhoso, sendo o principal desafio a decomposição do monólito. A decomposição manual é uma tarefa complexa, que exige um profundo conhecimento das complexidades do código e do domínio de negócio para identificar corretamente os limites de cada serviço (ABGAZ et al., 2023). A falha em definir essas fronteiras de maneira adequada pode resultar em antipadrões como o "monólito

distribuído”, que combina a complexidade operacional de um sistema distribuído com o forte acoplamento de um monólito, anulando os benefícios da migração.

2.1.2 A ferramenta `js-distributor`

Nesse contexto, foi desenvolvida a ferramenta `js-distributor`, projetada para a automação do processo de migração de arquiteturas monolíticas em JavaScript para um sistema de microsserviços. Sua função principal é a criação de código, permitindo ao desenvolvedor automatizar tarefas repetitivas e propensas a erros dentro do processo de decomposição de monólitos para arquiteturas de microsserviços.

Segundo Escher et al. (2025a), a ferramenta gera grande parte do código *boilerplate* necessário para Node.js, incluindo a configuração do servidor e o código de comunicação entre serviços, suportando tanto APIs HTTP quanto sistemas de mensagens assíncronas. Essa capacidade de gerar toda a infraestrutura de comunicação e configurações de servidor acelera significativamente o processo de decomposição.

Através dessas funcionalidades, o `js-distributor` atua como ponte entre a fase de análise e implementação da arquitetura de microsserviços, cobrindo uma lacuna na literatura nesse processo. A maioria das abordagens existentes foca na análise arquitetônica ou identificação de serviços, oferecendo pouco suporte para implantação real destes. Ao “preencher a lacuna entre decomposição e implantação, a ferramenta permite a experimentação rápida com diferentes arquiteturas de microsserviços” (ESCHER et al., 2025b), viabilizando a validação prática de diferentes estratégias de decomposição.

A ferramenta utiliza uma estratégia baseada na análise do código-fonte monolítico e gera os serviços por meio de *templates*, processo orquestrado pelos componentes:

- **Analisador Sintático:** é o núcleo da ferramenta, sendo um analisador sintático (*lexer* e *parser*) construído com ANTLR e gramática JavaScript de código aberto. Ele é responsável por ler o monólito e convertê-lo em uma Árvore de Análise Sintática (Parse Tree), que representa a estrutura do código;
- **Visitante Base (*Base Visitor*):** também gerado pelo ANTLR, fornece uma classe base com métodos que podem ser herdados para percorrer os nós da árvore e executar ações semânticas sobre ela;
- ***PrepareTreeVisitor*:** visita a árvore, estabelecendo relações pai-filho, para facilitar as transformações posteriores;
- ***ReplaceRemoteFunctionsVisitor*:** percorre cada função de servidor. Caso aquela função pertença àquele servidor, não faz nada; caso contrário, substitui por uma chamada HTTP ou mensagem RabbitMQ para o servidor que abriga a função;

- ***FixAsyncFunctionsVisitor*:** confere se as funções transformadas possuem o prefixo *async*, necessário para chamadas e mensagens, e corrige caso necessário;

- ***JavaScriptGeneratorVisitor*:** produz o código JavaScript de cada microsserviço.

Além desses componentes centrais, alguns de suporte também são necessários:

- **Templates de Código:** utilizados para gerar o código repetitivo, como configuração de servidores Express.js, definição de endpoints HTTP e código cliente para realizar chamadas remotas;

- ***Helpers*:** funções auxiliares que apoiam os visitantes, com lógicas para manipulação da árvore e geração de código.

O fluxo da geração de código começa pela entrada dos dados, tanto do código-fonte monolítico quanto do arquivo de configuração (config.yml). Esse arquivo de configuração é essencial, pois nele o desenvolvedor descreve a distribuição das funções entre serviços, portas HTTP que eles irão ocupar e métodos de exposição para cada função.

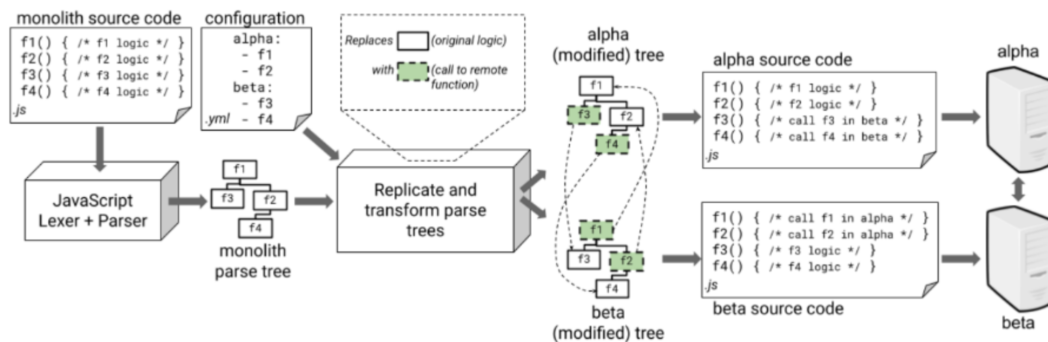
A seguir, o Lexer e o Parser do ANTLR analisam o monólito e geram uma árvore sintática que representa a aplicação. Essa árvore é então replicada para cada servidor que deve ser gerado, e em seguida as transformações são aplicadas por cada um dos visitantes. Após a transformação das árvores, o *JavaScriptGeneratorVisitor* é executado sobre cada uma, produzindo o código final para cada microsserviço, assim como arquivos de inicialização com configuração do Express.js, criação de *endpoints* da API e tratamento de requisições e respostas. Assim são criados os servidores, e resta ao desenvolvedor apenas o processo de teste.

O produto final desse processamento é a geração automática de estruturas de diretórios independentes para cada microsserviço especificado. Diferente de uma simples cópia de arquivos, a ferramenta reescreve a lógica interna: cada serviço gerado contém apenas as funções pertinentes ao seu escopo, encapsuladas em um servidor Express.js pré-configurado. O código original do monólito é transformado de modo que as chamadas de função locais sejam substituídas por requisições de rede (HTTP ou AMQP) quando a função destino reside em outro serviço, abstraindo a complexidade da comunicação distribuída.

A Figura 1 ilustra um exemplo prático dessa transformação. Nela, observa-se o código monolítico original à esquerda, onde as funções interagem diretamente em memória. À direita, apresenta-se o resultado pós-processamento pelo `js-distributor`: a criação de serviços distintos (Service A e Service B), onde a interação original foi substituída por um *proxy* de comunicação. Além do código funcional, a ferramenta entrega também os arquivos de infraestrutura necessários, como o `package.json` com as dependências

atualizadas e scripts de inicialização, entregando ao desenvolvedor um ambiente pronto para execução e testes.

Figura 1 – Exemplo de transformação de código monolítico em microsserviços pelo js-distributor



Adaptado de Escher et al. (2025b)

2.1.3 Autenticação e Autorização

Após análise do código gerado pelo `js-distributor`, é possível perceber que os *endpoints* e requisições gerados são abertos, cabendo ao usuário uma configuração manual posterior de segurança. Essa é uma das principais preocupações ao implementar um sistema baseado em microsserviços. Ao descentralizar a aplicação, ampliam-se os pontos de entrada para ataques em comparação a arquiteturas monolíticas, exigindo maior coordenação entre equipes e componentes (MATEUS-COELHO; CRUZ-CUNHA; FERREIRA, 2021).

Em seu mapeamento sistemático dos mecanismos de segurança mais abordados na literatura sobre microsserviços, Pereira-Vale et al. (2019) chegaram às seguintes estatísticas:

- 46% dos estudos abordaram Autorização;
- 42% abordaram Autenticação;
- 39% abordaram Credenciais.

Outros mecanismos variaram entre 4% e 19% de ocorrência, o que indica a predominância desses métodos na literatura. Segundo esse mesmo artigo, é possível defini-los como:

- Autorização: define quem é autorizado a acessar determinados recursos do sistema e de que maneira;
- Autenticação: define como verificar se um usuário ou recurso que tenta acessar o sistema é quem diz ser;

- Credenciais: meios de identificação e comprovação de autenticidade, essenciais na autenticação (verificação da identidade) e na autorização (definição de permissões), garantindo que apenas entidades legítimas acessem os recursos de forma segura.

2.1.4 Controle de Acesso Baseado em Funções (RBAC)

O Controle de Acesso Baseado em Funções (RBAC) é um modelo de segurança predominante para gestão de autorizações em larga escala. Sua abordagem não atribui permissões diretamente a usuários individuais, e sim a funções que cada usuário pode executar no sistema. Os usuários então são designados a papéis apropriados, adquirindo as permissões associadas a eles (SANDHU et al., 1996).

A principal vantagem do RBAC é a simplicidade na gestão de segurança. Em grandes sistemas, propagar mudanças de autorização por toda a cadeia de usuários, um a um, pode ser muito trabalhoso. Nesse caso, atribuir a esses indivíduos papéis que possuem um grupo de permissões definidas — e que, ao atualizar esse grupo, as mudanças são propagadas para todos os usuários que possuem o papel — torna a gestão muito mais simples e eficiente.

Ele segue alguns princípios, que visam diminuir os riscos de danos, intencionais ou não, ao sistema:

- **Privilégio mínimo:** garantir que cada usuário receba um conjunto mínimo de permissões necessárias para realizar suas tarefas;
- **Separação de deveres:** tarefas sensíveis devem ser distribuídas entre diferentes indivíduos, para que ninguém tenha controle total do processo;
- **Administração Centralizada:** permite aos administradores de segurança gerenciar centralmente as políticas de acesso;
- **Neutralidade de políticas:** o modelo RBAC não dita quais políticas o sistema deve seguir, apenas oferece orientações flexíveis para que cada organização estabeleça suas próprias.

2.1.5 *JSON Web Token* (JWT)

JSON Web Token (JWT), como padronizado pela RFC (*Request for Comments*) 7519, define-se fundamentalmente como um meio compacto, seguro e autocontido para a transmissão de informações entre partes no formato de um objeto JSON. Sua característica 'autocontida' (*self-contained*) implica que o próprio token carrega todas as informações necessárias sobre o usuário e os metadados de expiração, eliminando a necessidade de o servidor consultar o banco de dados para validar a sessão a cada requisição.

Essa arquitetura torna o JWT um padrão ideal para ambientes com restrição de largura de banda e, principalmente, para sistemas de autenticação em arquiteturas distribuídas e *stateless*, onde o servidor não armazena nenhuma informação sobre a sessão do usuário após enviar a resposta para o cliente. Exemplos são a arquitetura de microsserviços e APIs RESTful.

Formalmente, a estrutura do JWT é constituída por uma string de caracteres dividida em três segmentos distintos, codificados em Base64Url e separados por pontos: o cabeçalho (*Header*), a carga útil (*Payload*) e a assinatura (*Signature*). O cabeçalho especifica o tipo do token e o algoritmo de criptografia utilizado, enquanto a carga útil contém as declarações (*claims*) sobre a entidade e outros dados pertinentes. A segurança do mecanismo é garantida pelo terceiro segmento, a assinatura digital, que é matematicamente calculada a partir da concatenação dos dois primeiros segmentos codificados, utilizando o algoritmo definido e uma chave secreta, assegurando assim a integridade e a autenticidade dos dados transmitidos.

2.1.6 Protocolos de Autorização e Identidade: OAuth 2.0 e OpenID Connect

A segurança em aplicações modernas é baseada na separação clara entre autenticação e autorização. Como discutido anteriormente, a primeira se refere à identificação do usuário, e a segunda, ao controle de acesso a recursos. Um dos principais protocolos da indústria para autorização é o `OAuth 2.0`, definido pela IETF (*Internet Engineering Task Force*) na RFC 6789.

Segundo Auth0 (2025), ele foi projetado como um meio de conceder acesso a um conjunto de recursos, por exemplo, APIs remotas ou dados de usuários. Isso é feito com o uso de tokens de acesso, um dado que representa a autorização de acessar recursos em nome do usuário final. O formato JWT é frequentemente utilizado, mesmo que não seja definido algum formato específico para esses tokens.

De forma simplificada, o processo envolve a aplicação cliente (o sistema que deseja acessar os dados), que deve previamente se cadastrar para obter credenciais. Quando essa aplicação necessita acessar um recurso protegido, ela envia uma solicitação ao servidor de autorização. Este servidor verifica a confiabilidade da aplicação e solicita o consentimento do usuário (dono dos dados). Mediante a aprovação do usuário, o servidor emite um token de acesso para a aplicação, permitindo que esta consulte as informações desejadas sem jamais ter acesso à senha pessoal do usuário.

Já na etapa da autenticação, o OpenID Connect (OIDC) é um protocolo que opera como uma camada de identidade construída em cima do OAuth 2.0 (OpenID Foundation, 2025). OIDC estende essa capacidade para fornecer autenticação, permitindo que aplicações clientes verifiquem a identidade do usuário final com base na autenticação realizada por

um Servidor de Autorização, conhecido como OpenID Provider (OP). Essa arquitetura utiliza estruturas padrão de mercado, como REST e JSON, garantindo interoperabilidade e facilidade de integração em ambientes web e móveis.

O fluxo de funcionamento inicia-se com a aplicação cliente enviando uma solicitação ao Provedor OpenID para autenticar o usuário. Após a verificação das credenciais e a confirmação do consentimento pelo usuário, o provedor retorna tokens de segurança à aplicação. O diferencial principal do protocolo é a emissão do ID Token, um JWT que contém declarações de identidade assinadas e verificáveis sobre o usuário, além do Access Token tradicional do OAuth. Além disso, o protocolo especifica um *endpoint* de UserInfo, que pode ser consultado pela aplicação cliente utilizando o token de acesso. Sua função é recuperar atributos adicionais do perfil do usuário.

2.1.7 Keycloak

Ao implementar uma arquitetura de microsserviços, deixar a responsabilidade para cada um lidar com a segurança isoladamente pode ser trabalhoso e inseguro, caso não seja bem executado. Nesses cenários, serviços centralizados de segurança se destacam. Uma opção relevante é o Keycloak, uma solução *open-source* de Gerenciamento de Identidade e Acesso (IAM) desenvolvida para oferecer autenticação e autorização como um servidor autônomo e *self-hosted*. Ela centraliza a segurança e a desacopla da lógica de negócio das aplicações. Além disso, permite a implementação de padrões como *Single Sign-On* (SSO), *OAuth 2.0* e *OpenID Connect* (OIDC).

A arquitetura do Keycloak é baseada em alguns níveis:

- ***Realms* (Reinos):** nível mais alto de administração do sistema, gerencia um conjunto de usuários, credenciais, papéis e clientes de forma isolada. Isso permite que uma única instância do Keycloak gerencie diferentes sistemas e projetos;
- ***Clients* (Clientes):** entidades que podem solicitar autenticação de um usuário ao Keycloak. Na prática, em uma arquitetura de microsserviços, cada um seria configurado como um cliente dentro de um reino. Tokens de autorização gerados em um cliente podem permitir autorização a outros, se assim configurados, permitindo um controle fino de acesso;
- **Usuários:** são entidades que podem fazer login no sistema. Possuem atributos como e-mail, nome e senha armazenados no Keycloak, e é possível gerenciá-los pela interface administrativa do sistema. Podem receber papéis específicos, tornando-se parte do sistema de RBAC;
- **Roles (Papéis):** são a base para a implementação do RBAC. Um papel associa uma série de permissões ao usuário que os possui, em dois níveis: papéis de reino

(disponíveis em qualquer cliente do reino) e papéis de cliente (específicos para cada cliente);

- **Grupos:** usados para gerenciar conjuntos de usuários, atribuindo papéis e atributos a todos os membros e simplificando a administração.

No contexto desta implementação com microsserviços, o fluxo começa na autenticação, quando o cliente (o *front-end* ou serviço *entrypoint*) redireciona o usuário não autenticado para a página de login gerenciada pelo Keycloak. Nesta etapa, ele delega a verificação das credenciais. O usuário pode optar por utilizar credenciais locais (armazenadas no banco de dados do próprio Keycloak) ou utilizar a validação de identidade com provedores externos (Identity Providers - IdPs), como Google ou Facebook. Após a validação bem-sucedida das credenciais, o Keycloak estabelece uma sessão de usuário e emite o ID Token (conforme o padrão OpenID Connect), que atesta ao cliente que a autenticação ocorreu com sucesso e fornece os dados básicos do perfil do usuário.

Uma vez autenticado, o foco é a autorização, ou seja, o que o usuário tem permissão para fazer nos microsserviços. Para isso, o Keycloak emite o *Access Token* (no formato JWT). Diferente do ID Token, este token é destinado aos serviços de *back-end*. Para o controle de acesso, foi adotado o modelo RBAC. No painel administrativo do Keycloak, é necessário que tenham sido configurados *Roles* (papéis) que representam as funções de negócio. Esses papéis são injetados automaticamente nas *claims* do Access Token no momento de sua emissão.

Dessa forma, quando o cliente realiza uma requisição HTTP para um microsserviço protegido, um *Access Token* é enviado no cabeçalho *Authorization*. O microsserviço intercepta a requisição, valida a assinatura digital do JWT e inspeciona os papéis contidos no *payload*. O acesso ao recurso é então permitido ou negado com base na presença do papel exigido para aquela rota específica, garantindo que a lógica de permissões esteja desacoplada da lógica de negócio.

Para configurar o Keycloak, é possível utilizar a interface própria do sistema, acessando a URL apropriada no servidor, comandos de terminal ou através de arquivos de configuração do Terraform. A primeira estratégia é funcional para um gerenciamento diário e por membros não desenvolvedores do time, mas para uma configuração inicial a última abordagem foi escolhida para este trabalho.

Por sua compatibilidade com arquivos de configuração, utilizar o Terraform para configurar o Keycloak se torna a solução natural. Essa ferramenta permite criar arquivos de configuração descritivos para cada elemento do sistema (reino, clientes, papéis, usuários, gerenciar permissões e mapear relações entre clientes), que serão aplicados posteriormente através de comandos simples de terminal, e por meio dos quais é possível fazer alterações

na arquitetura conforme necessidade. Isso permite também a replicação facilitada de estruturas, o que é essencial para o funcionamento do `js-distributor`.

2.2 Principais Trabalhos Relacionados

Em sua revisão sistemática, Pereira-Vale et al. (2019) analisaram quais eram os principais métodos utilizados de acordo com a literatura para segurança de microsserviços. Nela, chegaram à conclusão de que a maioria das soluções propostas visa impedir ou mitigar ataques. Além disso, as técnicas mais comuns envolvem autorização (46%) e autenticação (42%). Os resultados deste estudo comprovaram a relevância dessas duas estratégias, solidificando o embasamento para a sua aplicação na ferramenta `js-distributor`. Esses dados levaram à pesquisa de formas eficientes de implantar esses elementos na arquitetura atual do projeto, como o Keycloak.

O estudo de Chatterjee e Prinz (2022) apresenta uma solução de segurança integrada usando Spring Security e Keycloak (SSK) para proteger APIs na arquitetura de microsserviços. Seu trabalho comprovou, em um caso prático, a eficácia dessas tecnologias para proteger o intercâmbio de dados. Essa combinação de serviços conta com funcionalidades como OAuth 2.0 e gerenciamento de usuários. A comprovação da utilidade do Keycloak nesse sistema ajudou na escolha do mesmo para uso no `js-distributor`.

A pesquisa de Haindl, Kochberger e Sveggen (2024) aponta uma lacuna na literatura focada especificamente na autenticação e autorização entre serviços. A revisão não só cataloga as ameaças, mas também as estratégias de mitigação mais eficazes. Ao optar pelo uso do Keycloak, que implementa os padrões OAuth2 e OpenID Connect, e um modelo RBAC, citados como técnicas robustas, o presente trabalho alinha-se diretamente às melhores práticas de "Defesa em Profundidade" (*Defense-in-Depth*) e "Confiança Zero" (*Zero-Trust*) citadas no artigo.

A Defesa em Profundidade consiste na aplicação de múltiplas camadas de segurança redundantes, de modo que a falha de um mecanismo não comprometa a integridade de todo o sistema. Já o paradigma de Confiança Zero parte da premissa de que nenhuma entidade, seja interna ou externa à rede, deve ser considerada confiável por padrão, exigindo verificação explícita a cada tentativa de acesso. A arquitetura proposta concretiza esses conceitos ao não confiar implicitamente na rede interna dos microsserviços, exigindo a validação de tokens criptografados (JWT) e a verificação de permissões (RBAC) em cada requisição individual, garantindo assim camadas robustas de verificação contínua.

O estudo de Araújo e Marinho (2023) traz outro mapeamento da literatura, e enfatiza em uma de suas proposições a importância de combinar soluções de segurança para um projeto mais robusto. Na pesquisa, OAuth 2.0, Tokens JWT e RBAC foram as 3 tecnologias mais citadas, justificando a escolha do Keycloak como provedor de segurança,

por ser compatível com todas elas.

O artigo de Shethiya (2025) reforça a ideia de abstrair a segurança, centralizando-a através de Provedores de Identidade (como o Keycloak) e API Gateways, um componente que pode ser adicionado na arquitetura como um ponto de entrada único, que roteia requisições aos microsserviços adequados e pode centralizar a autenticação, segurança e controle de tráfego. Essa abordagem é essencial para o Modelo de Segurança de Confiança Zero (*Zero Trust*), também defendido pelo autor. O protótipo desenvolvido neste TCC atua em alinhamento com esse princípio, abstraindo a complexidade de desenvolvimento desse modelo em uma arquitetura de microsserviços, utilizando o Keycloak como uma fonte central de identidade e o RBAC como mecanismo de verificação.

Kalubowila et al. (2021) propõe usar um modelo de validação externa, fora da lógica do microsserviço, colocando-a no nível de proxy (na entrada do serviço) para melhorar o desempenho. O modelo proposto conseguiu reduzir a latência de resposta para requisições inválidas de 30% a 45%. Esse dado fundamenta a decisão arquitetural adotada no presente trabalho de utilizar os mecanismos de aplicação e verificação de políticas de acesso do Keycloak diretamente na camada de entrada dos microsserviços. Ao configurar a função de proteção nos *endpoints*, o sistema intercepta e valida o token JWT antes que a requisição atinja o processamento da regra de negócio (js-distributor). Dessa forma, a arquitetura assegura que recursos computacionais não sejam desperdiçados com solicitações não autorizadas, alinhando-se à estratégia de otimização de desempenho preconizada pelo estudo citado.

Através do panorama traçado pelos estudos apresentados, fica evidente que o gerenciamento de identidade e acesso é uma etapa crítica e em constante evolução no ecossistema de microsserviços. Embora existam múltiplas visões sobre como executar essa tarefa, há um consenso sobre a necessidade de ferramentas que reduzam o esforço manual e o risco de falhas humanas. Nesse sentido, a proposta deste trabalho não é apenas adotar tecnologias de ponta, mas orquestrá-las de forma a oferecer uma estrutura sólida para a geração automática de código, garantindo a segurança da aplicação da maneira mais simples e efetiva.

3 Metodologia

O objetivo deste trabalho foi o estudo de melhorias para o `js-distributor`, focado na configuração automatizada de provedores de segurança para gerenciar a autenticação e autorização em sistemas distribuídos. A ferramenta já possui capacidade de criar microsserviços baseados em um código monolítico e toda configuração de comunicação.Esse projeto visou estudar como adicionar essa nova funcionalidade na arquitetura atual. Para atingir os objetivos propostos neste trabalho, a metodologia adotada foi de natureza aplicada e exploratória, dividida em quatro etapas sequenciais. O processo partiu da fundamentação teórica, avançou para a experimentação prática em um cenário controlado, seguiu para a validação em um cenário mais complexo e culminou na definição de um roteiro para a automação na ferramenta `js-distributor`. As etapas serão detalhadas a seguir.

3.1 Levantamento Bibliográfico

A primeira etapa consistiu no estudo aprofundado dos conceitos de autenticação e autorização em arquiteturas distribuídas, com foco nos protocolos OAuth 2.0 e OpenID Connect, além de modelos de controle de acesso como o RBAC (*Role-Based Access Control*). Paralelamente, foram analisadas as ferramentas necessárias para a implementação dessa segurança: Keycloak, como servidor de identidade e Terraform, como ferramenta de infraestrutura como código (IaC) para automatizar a configuração do ambiente. O resultado desta etapa dá origem à fundamentação teórica apresentada no Capítulo 2.

3.2 Estudo de Caso Exploratório

O objetivo desta etapa foi entender como o `js-distributor` gera os servidores distribuídos, e as adições necessárias para injeção da segurança via Keycloak no sistema. Para isso, foi utilizado o exemplo contido na documentação do projeto, um monólito simples (cujo código completo encontra-se na listagem 4.1) que é particionado em três serviços distintos: `Alpha`, `Beta` e `Gamma`.

A partir dele, foram estudados os arquivos de configuração do `js-distributor` (config.yml, Dockerfile, package.json, etc.), para entender como cada configuração definida gera um resultado no sistema distribuído.

Com este entendimento básico do funcionamento, foi configurada manualmente uma instância do Keycloak, com um cliente dedicado a cada serviço e permissões de comunicação entre eles configuradas. Isso permitiu compreender todas as configurações

necessárias para o funcionamento do gerenciamento de identidade, e a partir disso, criar arquivos de configuração com Terraform.

Além disso, todas as mudanças necessárias nos serviços e suas comunicações foram analisadas. Os pontos principais foram a configuração dos clientes do Keycloak, adição da autenticação no servidor de entrada (Alpha), e autorização nos servidores que recebem as requisições (Beta e Gamma). Além disso foi criado um quarto servidor, Delta, para testar a verificação de papéis, permitindo acesso apenas a usuários autorizados com um papel exemplo *premium*. Tudo isso foi feito de forma padronizada, visando uma implementação automatizada em futuras versões do `js-distributor`.

3.3 Estudo do Caso de Validação

Para verificar se os padrões identificados na etapa anterior eram aplicáveis a sistemas de maior complexidade, realizou-se um segundo estudo de caso utilizando o sistema **Acme Air**. Trata-se de uma aplicação de referência para testes, de código aberto que simula o sistema de gerenciamento de uma companhia aérea fictícia. Originalmente desenvolvida pela IBM para demonstrar capacidades de escalabilidade em nuvem, é amplamente adotada na academia para validar arquiteturas de microsserviços e estratégias de decomposição.

O sistema é implementado em JavaScript (Node.js) e utiliza um banco de dados NoSQL (MongoDB) para persistência. Sua arquitetura oferece um conjunto rico de regras de negócio distribuídas em múltiplas rotas de API, incluindo:

- **Autenticação e Perfil:** Gerenciamento de sessões de usuários (*login*/*logout*), cadastro de clientes e atualização de perfis;
- **Gestão de Voos:** Consulta de trechos disponíveis, cadastro de aeroportos e rotas de voo;
- **Reservas:** Lógica transacional para reserva e cancelamento de passagens, interagindo com serviços de dados e voos.

Devido a essa complexidade inerente, que simula um cenário real de alta concorrência e interdependência de serviços, o Acme Air é frequentemente utilizado como objeto de estudo em pesquisas sobre engenharia de software orientada a serviços.

É importante ressaltar que o repositório original do projeto[1] encontra-se sem atualizações significativas há diversos anos, apresentando incompatibilidades críticas com as versões mais recentes do ambiente de execução Node.js e dependências de bibliotecas depreciadas.

[1] Projeto original disponível em: <https://github.com/acmeair/acmeair-nodejs>.

Para viabilizar este estudo, foi utilizada uma versão atualizada (*fork*) do código[2], mantida para fins de pesquisa. Nesta versão, foram realizadas correções de compatibilidade, atualização de *drivers* de banco de dados e modernização da sintaxe JavaScript, garantindo a estabilidade necessária para os testes de segurança.

Dessa forma, o processo de configuração manual para o *realm* do Keycloak e a refatoração das requisições foram replicados neste ambiente, validando a generalização da proposta em um cenário realista e robusto.

3.4 Criação do roteiro de implementação

Esta etapa definiu quais parâmetros devem ser adicionados aos arquivos de configuração do distribuidor, para definir as regras de segurança. Além disso, validou os modelos de código que o `js-distributor` deve gerar para incluir configurações dos clientes, função de comunicação (*fetch*) personalizada e *middlewares* de proteção como o `keycloak.protect()`. Com essas alterações mapeadas, foi possível criar um roteiro de implementação.

Com as melhorias propostas, o `js-distributor` será capaz de gerar microsserviços com maior facilidade. Estes serviços terão suporte total a autenticação e autorização, que serão controladas pelo Keycloak. Além disso, a configuração será feita de maneira simplificada, aumentando significativamente a segurança de todos os microsserviços gerados.

[2] Versão utilizada (*fork*) disponível em: <https://github.com/dlucredio/acmeair-nodejs>.

4 Resultados

A fim de desenvolver um método eficiente e de fácil recriação pelo `js-distributor`, foram realizadas as etapas descritas anteriormente na metodologia. Esse capítulo tem como objetivo descrever as alterações propostas, e como isso pode ser implementado na ferramenta. Os testes foram realizados em dois cenários distintos.

O primeiro consistiu em uma aplicação de prova de conceito (PoC), baseada no exemplo de referência da documentação do `js-distributor`. Trata-se de um sistema minimalista composto por serviços interdependentes (`Alpha`, `Beta` e `Gamma`), cuja função principal é validar o fluxo básico de comunicação e a concatenação de dados entre microsserviços. O segundo cenário utilizou o Acme Air, um sistema de referência (*benchmark*) que simula as operações de uma companhia aérea, abrangendo funcionalidades complexas como autenticação, reservas, cancelamentos e gestão de voos.

Após a implementação das mudanças propostas, as rotas ficaram mais protegidas, permitindo apenas usuários cadastrados no Keycloak acessar as funcionalidades restritas dos sistemas, e apenas indivíduos com o papel adequado foram capazes de realizar requisições a rotas protegidas por RBAC. Isso foi validado através de testes manuais, enviando requisições tanto pela interface do Acme Air, quanto direto aos servidores. As seções a seguir descrevem as mudanças necessárias.

4.1 Cenário de Referência

Antes de abordar a implementação da infraestrutura, é fundamental compreender o objeto de estudo inicial. Para o primeiro cenário de validação, foi utilizada uma prova de conceito minimalista. Trata-se de um sistema exemplo, presente na documentação oficial da ferramenta para demonstrar suas funcionalidades[1]. Originalmente um monólito simples composto por três funções principais que trocavam mensagens entre si localmente, ele foi decomposto seguindo as instruções contidas na referida documentação.

Código 4.1 – Código do Monólito Exemplo

```
1 // Transforma-se no serviço Beta
2 function getMessage(greeting, person) {
3     console.log("Getting message");
4     return greeting + ", " + person + "!";
5 }
```

[1] O tutorial contendo o código deste exemplo está disponível em: <https://github.com/dlucredio/js-distributor/blob/main/GettingStarted.md>.

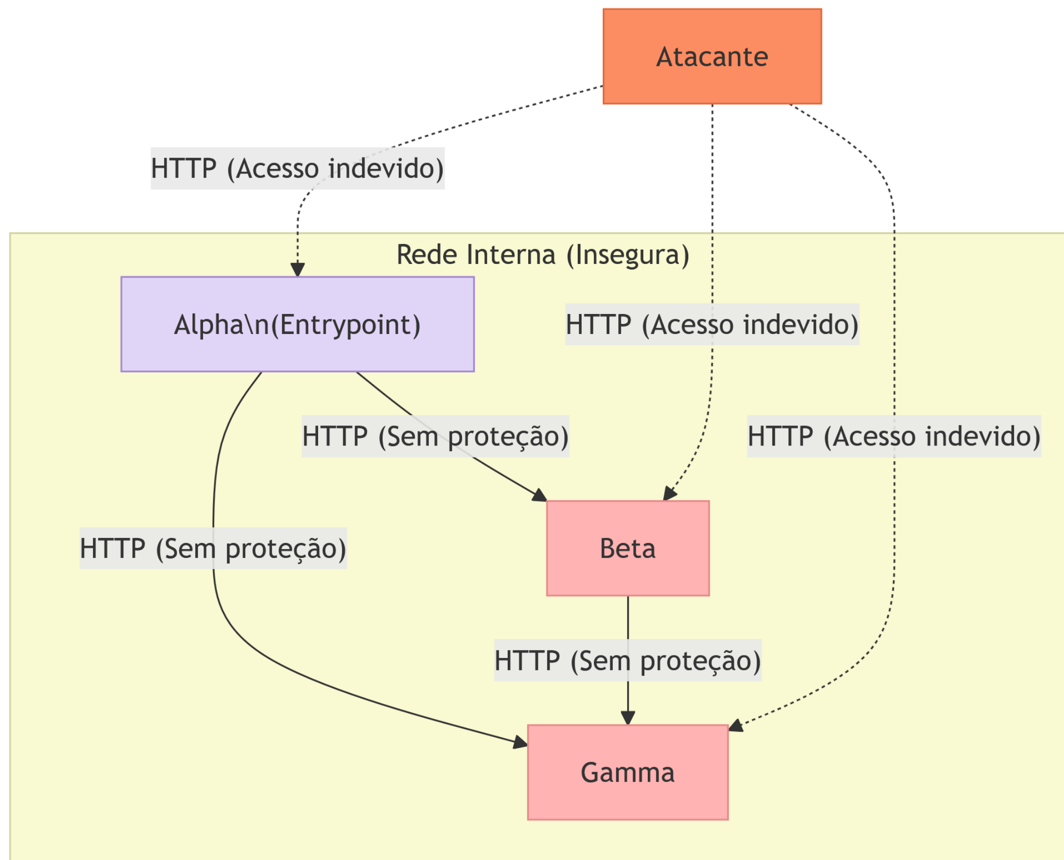
```
6
7 // Transforma-se no serviço Gamma
8 function getFullName(firstName, lastName) {
9     console.log("Getting full name");
10     return firstName + " " + lastName;
11 }
12
13 //Transforma-se no Alpha (ponto de entrada)
14 function main() {
15     console.log("Running application");
16     const fullName = getFullName("Fulano", "Silva");
17     const greeting = "Hello";
18     const message = getMessage(greeting, fullName);
19     console.log(message);
20 }
21
22 export default main;
```

Após o processamento pela ferramenta `js-distributor`, esta estrutura foi decomposta em três microsserviços independentes:

- **Alpha (Gateway):** Serviço de entrada, responsável por receber a requisição externa. Executa a função `main()` do trecho acima, orquestrando os demais serviços;
- **Beta e Gamma (Recursos):** Serviços internos que processam a lógica de negócio, cada um recebendo as funções `getMessage()` e `getFullName()`, respectivamente.

Esta configuração inicial não possui nenhuma segurança, e a comunicação ocorre via requisições HTTP diretas. Os serviços `Beta` e `Gamma` expõem seus *endpoints* publicamente, aceitando qualquer requisição recebida, o que representa uma vulnerabilidade crítica.

Figura 2 – Arquitetura distribuída inicial (sem segurança)



Fonte: Elaborado pelo autor

4.1.1 Definição da Arquitetura de Segurança

Para mitigar a vulnerabilidade exposta acima, foi definida uma estratégia de segurança centralizada baseada no `OpenID Connect` (OIDC). O objetivo é garantir que apenas requisições autenticadas e autorizadas alcancem os serviços internos.

A arquitetura proposta delega a responsabilidade de autenticação para o Keycloak, retirando essa complexidade dos microsserviços. O fluxo seguro foi desenhado da seguinte forma:

- O serviço `Alpha` é configurado como um *Confidential Client*. Nessa modalidade, a aplicação é capaz de manter credenciais secretas (*client secrets*) armazenadas de forma segura no lado do servidor. Isso permite que o serviço atue como um *Secure Gateway*, um padrão arquitetural no qual um ponto único de entrada centraliza o tráfego externo e impõe a autenticação do usuário via Keycloak antes da execução de qualquer regra de negócio.
- Os serviços `Beta` e `Gamma` são configurados como *Bearer-only Clients*. Esta configuração específica instrui o adaptador a desabilitar qualquer tentativa de redirecionamento

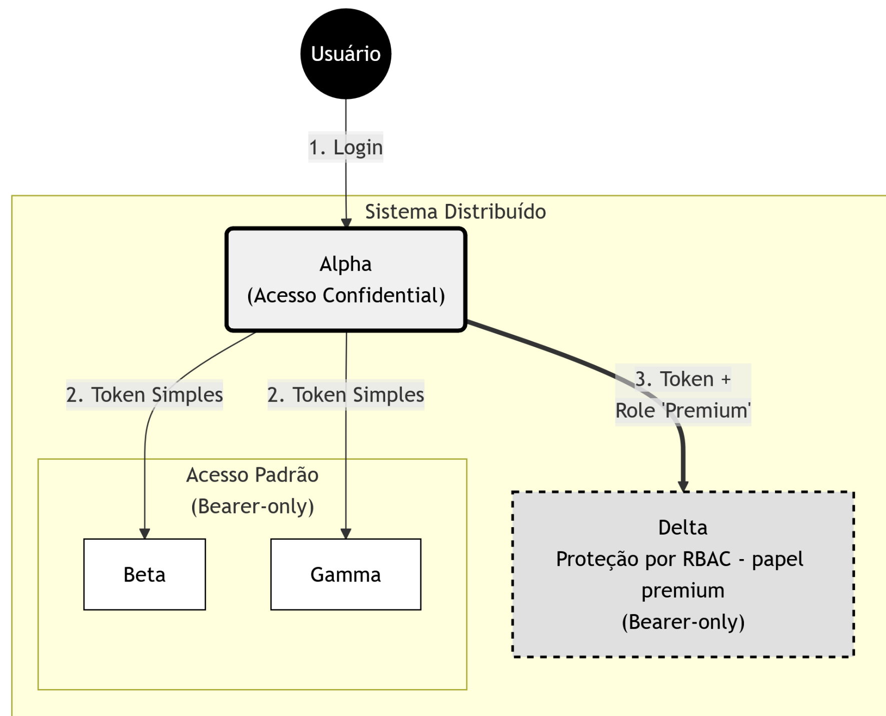
para login no navegador, condicionando o acesso exclusivamente a requisições que apresentem um *token* de acesso (JWT) válido no cabeçalho *Authorization*. Dessa forma, eles atuam estritamente como validadores de permissão, delegando a gestão da identidade para o `Alpha`.

Dessa forma, cria-se um perímetro de segurança onde a identidade é estabelecida na borda (Alpha) e propagada de forma confiável para o interior da malha (Beta e Gamma).

Um ponto essencial para a validação do RBAC foi a definição de um papel de nível de domínio (*realm role*), denominado `premium`, cuja posse seria obrigatória para acessar determinados recursos. Nas aplicações de estudo de caso, foram delimitadas áreas que exigiriam este papel.

No cenário do monólito simples, originalmente composto por `Alpha`, `Beta` e `Gamma`, foi introduzido um quarto serviço chamado `Delta`, especificamente para este teste de segurança. Embora a funcionalidade do serviço `Delta` limite-se ao retorno de uma mensagem de confirmação, seu acesso foi configurado para exigir explicitamente o papel `premium`. Essa restrição estabelece um cenário de privilégio elevado, diferenciando-o dos demais serviços que demandam apenas a autenticação padrão. A Figura 3 ilustra a arquitetura de segurança resultante em ambos os cenários, destacando a classificação dos clientes e as zonas protegidas por papel.

Figura 3 – Monólito simples após distribuição e proteção



Fonte: Elaborado pelo autor

4.2 Configuração do *realm* do Keycloak via Terraform

A implementação realizada poderia ser realizada sem o auxílio de uma ferramenta de infraestrutura. O Keycloak possui a possibilidade de ser configurado diretamente via uma API REST administrativa ou pela sua interface de comando. No entanto, a primeira alternativa necessitaria que o `js-distributor` possuísse um script complexo para fazer todas as requisições, e validar se foram corretamente realizadas.

Já a segunda necessita de um *input* manual do desenvolvedor, por ser uma interface WEB. Isso pode ser trabalhoso, e causar erros devido a descuidos durante o processo. Esquecer uma única configuração pode quebrar o acesso a um serviço, tornando o sistema instável. Além disso, essa solução vai contra o princípio do `js-distributor` de automatizar essa transição de arquiteturas.

É neste contexto que a adoção do Terraform como intermediário se justifica tecnicamente. Ao utilizar Infraestrutura como Código (IaC), a responsabilidade de gerenciar o estado e as chamadas de API é delegada ao Terraform. Para o `js-distributor`, o processo de automação torna-se estritamente declarativo: a ferramenta precisa apenas gerar arquivos de texto (`.tf`) descrevendo o estado desejado, muito similar aos arquivos de configuração que ela já é capaz de interpretar.

O Terraform assume a responsabilidade de calcular a diferença entre o estado atual e o desejado e aplicar as mudanças necessárias. Essa separação de responsabilidades viabiliza a geração automática de ambientes de segurança mais complexos com menor esforço de implementação na ferramenta de distribuição.

Para ilustrar a redução de complexidade proporcionada por essa abordagem, é possível comparar os três métodos distintos para realizar uma mesma tarefa simples: configurar um novo cliente (*Client*) chamado `alpha-service` no Keycloak.

Na abordagem tradicional, o administrador precisa acessar o painel, navegar entre menus, preencher formulários e clicar em botões de ação. Embora amigável para humanos, é inviável para automação, pois exigiria ferramentas de *scrapping* ou simulação de navegador, que são frágeis e lentas.

Para automatizar sem Terraform, o `js-distributor` precisaria gerar e executar requisições HTTP complexas. O exemplo abaixo demonstra uma chamada para criar o cliente. Note que, antes disso, a ferramenta precisaria ter realizado outra requisição para obter o *token* de autenticação e, posteriormente, trataria erros caso o cliente já existisse.

Código 4.2 – Exemplo de complexidade: Criação via API REST

```
1 POST /admin/realms/my-realm/clients
2 Host: localhost:8080
3 Authorization: Bearer <TOKEN_AUTENTICACAO_DO_ADMINISTRADOR>
```

```
4 Content-Type: application/json
5
6 {
7     "clientId": "alpha-service",
8     "enabled": true,
9     "protocol": "openid-connect",
10    "publicClient": false,
11    "bearerOnly": false,
12    "redirectUris": ["http://localhost:3000/*"]
13 }
```

Já adotando a estratégia proposta, toda a complexidade de autenticação e verificação de estado é abstraída. O `js-distributor` encarrega-se de gerar o código de configuração abaixo, o qual será posteriormente consumido pelo Terraform para o provisionamento do adaptador:

Código 4.3 – Exemplo de simplicidade: Declaração via Terraform

```
1 resource "keycloak_openid_client" "alpha_service" {
2   realm_id  = keycloak_realm.realm.id
3   client_id = "alpha-service"
4   enabled   = true
5
6   access_type = "CONFIDENTIAL"
7   valid_redirect_uris = ["http://localhost:3000/*"]
8 }
```

A comparação evidencia que a abordagem via Terraform remove a necessidade de o gerador de código lidar com tokens, *endpoints* e requisições HTTP, focando apenas na definição do resultado final desejado.

Assim, através dessa ferramenta, implementou-se um *Realm* dedicado para o ecossistema de microsserviços. Esta decisão arquitetural isola os usuários e políticas de segurança da aplicação (tempo de vida de tokens, requisitos de senha) das configurações administrativas do provedor de identidade.

Código 4.4 – Arquivo exemplo de configuração realm.tf

```
1 resource "keycloak_realm" "realm" {
2   realm   = "my-realm"
3   enabled = true
4   ssl_required    = "external"
```

```
5   access_code_lifespan = "1h"
6   default_signature_algorithm = "RS256"
7 }
```

O código define o recurso `keycloak_realm` com o identificador 'my-realm' (linha 2), e o ativa. A configuração `ssl_required` = 'external' (linha 6) é utilizada para permitir tráfego HTTP não criptografado dentro da rede interna do Docker, facilitando testes. Já a diretiva `default_signature_algorithm` = 'RS256' (linha 10) impõe o uso de criptografia assimétrica, essencial para que os microsserviços possam validar a integridade dos *tokens offline* utilizando apenas a chave pública.

Os testes foram realizados utilizando uma credencial única para todo o sistema. Dessa forma, foi estabelecida uma topologia de clientes baseada na responsabilidade de cada serviço:

- **Cliente de Acesso (`access_type = "CONFIDENTIAL"`):** Atribuído ao serviço `Alpha`. No Terraform, essa definição instrui o Keycloak a gerar um segredo de cliente (*client secret*), habilitando o fluxo de autorização (*standard flow*) necessário para que o *gateway* realize a troca de credenciais de login por *tokens* de acesso válidos.
- **Clientes de Recurso (`access_type = "BEARER-ONLY"`):** Atribuído aos serviços `Beta`, `Gamma` e `Delta`. Ao definir este parâmetro no código, o Keycloak configura esses clientes sem URLs de redirecionamento ou fluxo de login. Isso força a aplicação a rejeitar qualquer requisição que não possua um token *Bearer* no cabeçalho, consolidando a política de que serviços internos jamais devem iniciar sessões de usuário.

Código 4.5 – Arquivo de configuração de clientes clients.tf

```
1  # Exemplo de cliente de acesso
2  resource "keycloak_openid_client" "alpha_client" {
3    realm_id                     = keycloak_realm.realm.id
4    client_id                    = "alpha-client"
5    name                         = "Alpha Service (Entrypoint)"
6    enabled                      = true
7    access_type                  = "CONFIDENTIAL"
8    standard_flow_enabled        = true
9    direct_access_grants_enabled = true
10   service_accounts_enabled     = true
11   valid_redirect_uris = [
```

```
12      "http://localhost:3000/*",
13      "http://localhost:9080/*"
14    ]
15  }
16
17  # Exemplo de cliente de Recurso
18  resource "keycloak_openid_client" "beta_client" {
19    realm_id    = keycloak_realm.realm.id
20    client_id   = "beta-client"
21    name        = "Beta Service"
22    enabled     = true
23    access_type = "BEARER-ONLY"
24  }
```

A Listagem 4.5 demonstra a materialização da topologia de segurança via código. No primeiro bloco, o recurso `alpha_client` é configurado com `access_type` = "CONFIDENTIAL"(linha 7). Essa configuração instrui o Keycloak a gerar um segredo (client secret), habilitando o fluxo de autorização (`standard_flow_enabled` = true na linha 8) e permitindo que este serviço negocie tokens em nome do usuário.

A configuração `standard_flow_enabled` = true habilita o *Authorization Code Flow*, que é o fluxo padrão do OpenID Connect baseado em redirecionamento de navegador. É este mecanismo que permite ao usuário ser enviado para a tela de login do Keycloak e retornar com um código de autorização. Já a linha `direct_access_grants_enabled` = true ativa o *Resource Owner Password Credentials Grant*, um fluxo que permite a troca direta de usuário e senha por um token através de uma chamada REST, sem interface gráfica. Ambos foram habilitados para testes.

As linhas 11 a 14 restringem as URLs de redirecionamento (`valid_redirect_uris`) para evitar ataques, limitando o retorno do login apenas às portas locais da aplicação.

Já no segundo bloco, define-se o `beta_client` com uma estratégia distinta. A linha 23 define seu `access_type` como "BEARER-ONLY". Diferente do Alpha, essa configuração remove a capacidade de iniciar *logins* ou redirecionar o navegador, transformando o serviço em um validador passivo que aceita estritamente requisições REST contendo um *token Bearer* válido no cabeçalho.

Como dito anteriormente, o acesso ao serviço `Delta` no sistema exemplo é limitado a usuários que possuam o papel *premium*. A criação desse papel é possível diretamente em um arquivo de configuração do Terraform, como pode ser visto a seguir.

Código 4.6 – Arquivo de configuração de papéis roles.tf

```
resource "keycloak_role" "premium_role" {
  realm_id    = keycloak_realm.realm.id
  name        = "premium"
  description = "Acesso VIP no sistema"
}
```

Na Listagem 4.6 o recurso `keycloak_role` cria a função lógica *premium* (linha 3). Ao associá-la diretamente ao realm (linha 2), define-se um papel de escopo global. Isso permite que a permissão seja transportada no *token* do usuário e verificada uniformemente por qualquer microsserviço da arquitetura (Alpha, Beta ou Delta) para restringir o acesso a funcionalidades sensíveis.

Um desafio crítico em sistemas distribuídos é a validação de confiança entre serviços. No cenário estudado, o serviço `Alpha` posiciona-se na fronteira do sistema, recebendo as requisições provenientes do ambiente externo. Enquanto isso, `Beta` e `Gamma` operam como serviços que apenas recebem chamadas internas ao sistema.

Por padrão, quando `Alpha` autentica um usuário, o Keycloak emite um token destinado exclusivamente à esse serviço (audiência `aud: "alpha-client"`). Se este mesmo *token* for enviado ao `Beta`, ele será rejeitado, pois `Beta` não se reconhece como o destinatário daquela credencial.

Para solucionar isso via infraestrutura, foi desenvolvido um padrão de Mapeamento de Audiência. No contexto do protocolo OpenID Connect, a **Audiência** (`aud`) é um campo crítico do token JWT que lista os destinatários finais autorizados a processar aquele *token*; por segurança, qualquer serviço que não encontre seu próprio identificador nesta lista deve rejeitar a requisição imediatamente.

Para gerenciar essa lista de forma eficiente, utilizou-se o conceito de **Escopo de Cliente** (*Client Scope*). No Keycloak, um escopo atua como um modelo reutilizável de configurações e reivindicações (*claims*) que pode ser compartilhado entre múltiplos clientes, eliminando a necessidade de configurar mapeamentos individualmente. Através do Terraform, configurou-se um Escopo compartilhado que injeta automaticamente os identificadores de todos os microsserviços internos na audiência do JWT no momento da emissão.

Isso permite que o token seja aceito por clientes que não o emitiram originalmente. Na prática, isso garante que um token gerado no login do `Alpha` contenha a instrução `aud: ["alpha-client", "beta-client", "gamma-client"]`, permitindo que a requisição transite livremente por todos os níveis de requisições.

Código 4.7 – Arquivo de configuração de mapeamentos - mappers.tf

```
# Escopo compartilhado para audiência
resource "keycloak_openid_client_scope" "audience_scope" {
  realm_id = keycloak_realm.realm.id
  name     = "audience-scope"
}

# Mapper: Adiciona o cliente exemplo Beta à audiência
resource "keycloak_openid_audience_protocol_mapper" "audience_beta" {
  realm_id        = keycloak_realm.realm.id
  client_scope_id = keycloak_openid_client_scope.audience_scope.id
  name            = "audience-beta-mapper"

  included_client_audience = keycloak_openid_client.beta_client.client_id
  add_to_access_token      = true
  add_to_id_token          = true
}

# ... adição dos demais clientes

# Mapper: adição do papel exemplo 'premium' ao escopo
resource "keycloak_openid_user_realm_role_protocol_mapper" "roles_mapper" {
  realm_id        = keycloak_realm.realm.id
  client_scope_id = keycloak_openid_client_scope.audience_scope.id
  name            = "realm-roles-mapper"
  claim_name      = "realm_access.roles"
  multivalued     = true
  claim_value_type = "String"
  add_to_access_token = true
}

# Associa tudo isso ao cliente exemplo Alpha, responsável pela autenticação e geração do token
resource "keycloak_openid_client_default_scopes" "alpha_scopes" {
  realm_id  = keycloak_realm.realm.id
  client_id = keycloak_openid_client.alpha_client.id

```

```
36    default_scopes = [
37      "profile",
38      "email",
39      "roles",
40      "web-origins",
41      keycloak_openid_client_scope.audience_scope.name,
42    ]
43  }
```

A listagem 4.7 possui as seguintes sessões:

- **Escopo de Audiência (Linhas 1-5):** Define o recurso `audience_scope`, que atua como um contêiner lógico reutilizável para agrupar as regras de mapeamento abaixo.
- **Mapeamento de Audiência (Linhas 8-16):** O recurso `audience_beta` injeta o identificador do cliente Beta no campo `aud` do token (`included_client_audience`). As flags `add_to_access_token` e `add_to_id_token` garantem que essa informação persista em todos os tipos de tokens gerados.
- **Mapeamento de Papéis (Linhas 21-29):** O recurso `roles_mapper` insere os papéis globais do usuário no *token* sob a chave `realm_access.roles` (`claim_name`). A propriedade `multivalued = true` assegura que o campo seja tratado como uma lista, permitindo múltiplos papéis.
- **Associação ao Cliente Alpha (Linhas 32-43):** O recurso `alpha_scopes` vincula o escopo criado ao serviço Alpha. A lista `default_scopes` assegura que o mapeamento de audiência e papéis seja aplicado automaticamente a cada login realizado pelo Gateway, sem necessidade de requisição extra.

É necessária também a configuração geral do Keycloak, no arquivo `main.tf`, e variáveis globais no `variables.tf`:

Código 4.8 – Arquivos de configurações gerais

```
1  # main.tf
2  terraform {
3    required_providers {
4      keycloak = {
5        source  = "mrparkers/keycloak"
6        version = ">= 4.0.0"
7      }
8    }
```

```
9  }
10
11 provider "keycloak" {
12   client_id     = "admin-cli"
13   username      = var.keycloak_user
14   password      = var.keycloak_password
15   url           = var.keycloak_url
16 }
17
18 # variables.tf
19 variable "keycloak_url" {
20   default = "http://localhost:8080"
21 }
22
23 variable "keycloak_user" {
24   default = "admin"
25 }
26
27 variable "keycloak_password" {
28   default = "admin"
29 }
```

O código inicia definindo a dependência do provedor `mrparkers/keycloak` na versão 4.0.0 ou superior (linhas 2-9), assegurando a estabilidade da automação. Na sequência, o bloco `provider` (linhas 11-16) configura a conexão administrativa utilizando o cliente `admin-cli`, mas delega as credenciais e a URL para variáveis dinâmicas (`var.`), evitando a exposição de segredos no código principal. Essas variáveis são detalhadas no arquivo auxiliar `variables.tf` (linhas 19-29), onde recebem valores padrão (como `localhost` e `admin`) para facilitar a execução imediata em ambiente de desenvolvimento.

Após a definição dos arquivos, a infraestrutura foi provisionada através do comando `terraform apply`, que interpretou o estado desejado e realizou as chamadas de API necessárias ao Keycloak. Essa configuração foi capaz de replicar o processo manual de configuração que foi feito a princípio, e gerar o seguinte sistema:

Como resultado final temos o Keycloak agindo como um serviço à parte, e responsável pela autenticação do sistema. O cliente `Alpha` é o único serviço que mantém contato constante, fazendo a autenticação do usuário, e recebendo o token, com os mapeamentos necessários para acessar os demais serviços. Eles por sua vez apenas realizam uma requisição inicial ao serem iniciados, para receber o *client secret*, uma chave necessária para a validação do token. Depois disso, apenas analisam os *tokens*, verificando a audiência para autorizar o acesso.

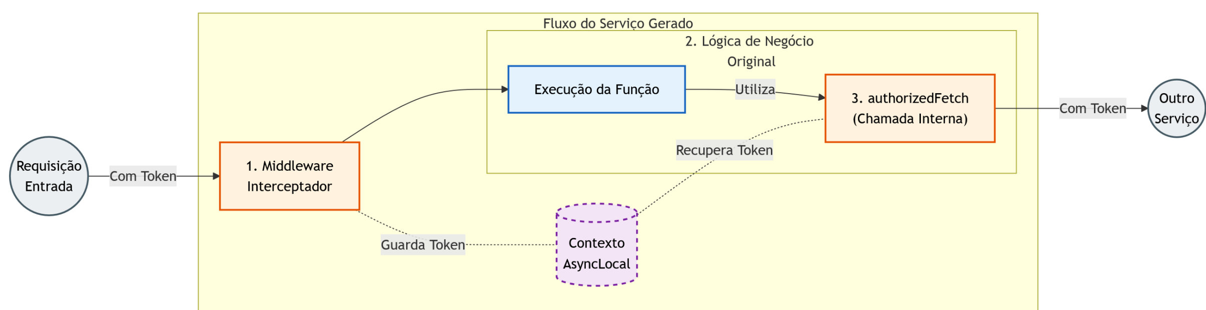
4.3 Implementação da biblioteca de contexto compartilhado

Um dos desafios fundamentais na migração de um monólito para microsserviços é a manutenção do estado da autenticação. Em uma aplicação monolítica, a identidade do usuário reside na memória compartilhada e está acessível a todas as funções. No entanto, ao distribuir o sistema, essa memória é fragmentada.

O desafio específico no contexto do `js-distributor` é garantir a homogeneidade do código gerado. Se um serviço `Alpha` recebe um *token* e precisa chamar o serviço `Beta`, a abordagem ingênua seria alterar a assinatura de todas as funções de negócio para trafegar esse dado. Essa estratégia divergiria da filosofia de design da ferramenta, que prioriza a realização de intervenções mínimas na lógica de negócio original.

Para solucionar isso de forma transparente, foi implementado um mecanismo de propagação de contexto. Essa estratégia cria uma área de memória temporária e isolada para cada requisição, que acompanha todo o seu ciclo de vida dentro do serviço. Conforme ilustrado na Figura 4, o mecanismo intercepta o *token* na entrada e o armazena nesse contexto, tornando-o acessível durante todo o fluxo de execução. Isso permite que qualquer chamada HTTP que saia daquele serviço em direção a outro possa recuperar a credencial automaticamente no momento de ser realizada, sem que a lógica de negócio precise ter conhecimento dessa operação.

Figura 4 – Fluxo de propagação transparente do Token JWT entre serviços



Fonte: Elaborado pelo autor

O mecanismo é composto por dois componentes principais que abstraem essa complexidade:

- **Middleware de Interceptação (Entrada):** Posicionado no início do pipeline do Express.js, ele captura o cabeçalho `Authorization` da requisição recebida e o armazena em um contexto isolado (*AsyncLocalStorage*), vinculado exclusivamente àquele ciclo de vida da requisição.

- **Cliente HTTP Personalizado (Saída):** Uma função encapsulada (`authorizedFetch`) que substitui o `fetch` padrão. Antes de realizar qualquer chamada externa, ela consulta o contexto isolado, recupera o *token* armazenado (se houver) e o injeta automaticamente no cabeçalho da nova requisição.

O código a seguir mostra como isso foi implementado em JavaScript:

Código 4.9 – Biblioteca de contexto

```
import { AsyncLocalStorage } from 'async_hooks';

export const requestContext = new AsyncLocalStorage();

export function contextMiddleware(req, res, next) {
    const store = new Map();
    const authHeader = req.headers.authorization || req.headers.
       Authorization;

    if (authHeader) {
        store.set('authorizationHeader', authHeader);
    }

    requestContext.run(store, () => {
        next();
    });
}

export async function authorizedFetch(url, options = {}) {
    const store = requestContext.getStore();
    const authorizationHeader = store ? store.get('
       authorizationHeader') : null;

    const defaultHeaders = { 'Content-Type': 'application/json'
       };

    if (authorizationHeader) {
        defaultHeaders['Authorization'] = authorizationHeader;
    }

    const newOptions = {
        ...options,
        headers: { ...defaultHeaders, ...options.headers }
```

```
31     };
32
33     return fetch(url, newOptions);
34 }
```

A implementação da biblioteca, apresentada na Listagem 4.7, é dividida em três componentes lógicos que garantem o isolamento e a propagação dos dados:

O código utiliza a classe nativa `AsyncLocalStorage` do Node.js. Na linha 3, uma instância global chamada `requestContext` é criada. Ela atua como um contêiner de armazenamento que, diferente de variáveis globais comuns, mantém os dados isolados para cada fluxo de execução assíncrona.

A função `contextMiddleware` é responsável por capturar o contexto na entrada do serviço. Ela extrai o token do cabeçalho `Authorization` (linha 7) e o armazena em um mapa de dados (linha 10). O ponto crítico ocorre na linha 13: o método `run` executa a função `next()`, o que dá continuidade ao processamento da requisição dentro de um escopo isolado, garantindo que o *token* armazenado esteja acessível apenas durante o ciclo de vida daquela requisição específica.

A função `authorizedFetch` atua como um *wrapper* (envoltório) para a função `fetch` padrão. Antes de realizar a chamada externa, ela consulta o armazenamento isolado através do método `getStore()` (linha 19) para recuperar o *token* salvo anteriormente. Caso o token exista, ele é injetado automaticamente nos cabeçalhos da nova requisição (linhas 24-26), mesclando-se com as opções originais antes de disparar o envio real na linha 33.

Essa abstração eliminou a necessidade de alterar o código das requisições HTTP nos serviços, apenas importando a função *fetch* personalizada, e adicionando o *middleware* que captura os tokens no início do código. Essa abstração permitiu que a comunicação segura fosse implementada sem alterar a assinatura das funções originais do monólito. Isso garante a homogeneidade do código gerado, facilitando a automação pelo `js-distributor`.

4.4 Refatoração do serviço de entrada

Nos estudos de caso realizados, o serviço `Alpha` desempenha o papel de *gateway* (ponto de entrada). Devido a essa posição arquitetural, ele centraliza a responsabilidade de autenticar os usuários com o Keycloak. Após a autenticação bem-sucedida, o *token* JWT é capturado e armazenado no contexto da requisição utilizando a biblioteca detalhada na seção 4.3, o que permite sua propagação automática para os serviços dependentes. Para viabilizar esse fluxo, a lógica original de *login* foi refatorada para delegar a validação de credenciais ao provedor de identidade, seguindo o padrão do protocolo OpenID Connect.

Código 4.10 – Arquivo app.js - Função de Login

```
app.use((req, res, next) => {
    if (req.cookies['access_token'] && !req.headers.authorization
        ) {
        req.headers.authorization = `Bearer ${req.cookies['
            access_token']}`;
    }
    next();
});

app.use(contextMiddleware);

async function login(req, res) {
    // ... params ...
    const kcResponse = await fetch(tokenUrl, { method: 'POST',
        body: params, ... });
    const data = await kcResponse.json();
    // ... define cookie e retorna 200 ...
}
```

- **Middleware Adaptador:** Transfere o *token* dos cookies para o cabeçalho `Authorization`, garantindo a compatibilidade necessária entre o navegador do usuário e as APIs REST.
- **Ativação de Contexto:** Inicializa o mecanismo de armazenamento, capturando o *token* do cabeçalho e guardando-o na memória isolada para ser propagado automaticamente nas requisições seguintes.
- **Função Login:** Substitui a validação local, delegando a autenticação ao Keycloak via requisição HTTP e retornando o *token* JWT gerado diretamente para o cliente.

Na definição das funções que realizam requisições aos demais serviços, o `fetch` refatorado descrito na seção anterior deve ser utilizado, para a transmissão do *token*. Qualquer microsserviço que atue como cliente na malha deve configurar o middleware de segurança para processar *tokens* de entrada e, obrigatoriamente, utilizar a função `authorizedFetch` para realizar requisições a outros serviços.

Código 4.11 – Importação e uso da fetch personalizada

```
import { authorizedFetch } from '../requestContext.js';

```

```
3 async function createSession(customerId) {
4     const response = await authorizedFetch('http://gamma:3000/
        createSession', {
5         method: 'POST',
6         body: JSON.stringify({ customerId: customerId })
7     });
8
9     if (!response.ok) throw new Error("Erro no Gamma");
10
11    const { executionResult } = await response.json();
12    return executionResult;
13 }
```

A Listagem 4.11 demonstra a aplicação prática da biblioteca no código de negócio. O ponto de maior relevância encontra-se na **linha 4**, onde a função nativa `fetch` é substituída pela `authorizedFetch`.

Esta alteração constitui a principal adaptação na camada de comunicação entre serviços. Substitui-se a extração manual de *tokens* e a composição de cabeçalhos pela utilização da função `authorizedFetch`, que abstrai essa complexidade. A função gerencia a injeção do contexto de segurança de forma transparente, mantendo a compatibilidade sintática com a API *Fetch* nativa.

4.5 Refatoração nos serviços de recursos

A estratégia para os serviços de recursos, nos dois exemplos representados por `Beta`, `Gamma` e `Delta`, consistiu em preservar a lógica de negócios dos sistemas, intervindo apenas na camada de transporte para injetar os mecanismos de segurança do Keycloak. Como a função oficial `keycloak.protect()` foi projetada para operar sobre o protocolo HTTP, todas as interfaces de comunicação foram geradas pelo `js-distributor` de acordo.

Nesses serviços, existem dois diferentes casos: serviços protegidos simplesmente com o token de acesso do Keycloak ou protegidos com RBAC. Serviços sem autorização de papéis simplesmente adicionam a validação do `keycloak.protect()` em seus endpoints, como pode ser observado no exemplo a seguir:

Código 4.12 – Serviços com autorização simples, sem RBAC

```
1 import express from 'express';
2 import session from 'express-session';
3 import Keycloak from 'keycloak-connect';
4 import { contextMiddleware } from './requestContext.js';
```

```
5
6 const app = express();
7
8 // 1. Configuração do Adaptador Keycloak
9 const memoryStore = new session.MemoryStore();
10 app.use(session({ secret: 'service-secret', resave: false,
     saveUninitialized: true, store: memoryStore }));
11 const keycloak = new Keycloak({ store: memoryStore });
12
13 // 2. Aplicação dos Middlewares de Segurança
14 app.use(keycloak.middleware()); // Interceptação e Validação JWT
15 app.use(contextMiddleware);     // Propagação de Contexto (caso
     este serviço chame outro)
16
17 // 3. Definição de Rotas Protegidas
18 // O método keycloak.protect() garante que apenas requisições com
      token válido cheguem à função
19 app.post('/recurso-protegido', keycloak.protect(), async (req,
     res) => {
20     // Lógica de negócio...
21 });
```

Para proteger rotas sensíveis, utiliza-se o método `keycloak.protect('realm:premium')`. O prefixo `realm:` indica que estamos verificando um papel global. Já o nome `premium` faz a ponte direta com a infraestrutura: ele referencia exatamente o mesmo papel que foi criado via Terraform (Listagem 4.4), garantindo que a regra definida na nuvem seja aplicada no código.

Código 4.13 – Serviços com autorização e RBAC

```
1 app.post('/recurso-protegido-RBAC', keycloak.protect(realm:role),
      async (req, res) => {
2     // Lógica de negócio...
3 });
```

A última etapa de configuração é a criação de um arquivo *keycloak.json*, que será lido pelo *middleware* mostrando no trecho de código 4.12. Ele deve conter as configurações do cliente mapeado para aquele serviço, para o correto funcionamento da validação via `keycloak.protect()`.

Código 4.14 – Arquivo keycloak.json, com as configurações do cliente

```
{
  "realm": "my-realm",
  "auth-server-url": "http://host.docker.internal:8080",
  "ssl-required": "external",
  "resource": "delta-client",
  "bearer-only": true,
  "confidential-port": 0
}
```

Vale ressaltar que a URL de autenticação configurada nos microsserviços (`host.docker.internal`), no trecho anterior, foi utilizada para permitir que os contêineres Docker se comuniquem com o serviço do Keycloak hospedado na máquina hospedeira (host), superando as limitações de rede do ambiente de virtualização durante os testes. Em ambientes não executados com o Docker, essa URL seria diferente.

O design de segurança da arquitetura de microsserviços priorizou a resiliência e a performance através da implementação da validação de token *off-line* nos servidores de recurso. Na inicialização de cada serviço, o adaptador Keycloak realiza uma única requisição ao Provedor de Identidade (IdP) para buscar a chave pública do *realm*. Esta chave é então armazenada localmente, permitindo que as validações subsequentes de Tokens JWT sejam realizadas de forma autônoma e criptograficamente segura (verificando a assinatura digital do token). Este mecanismo elimina a dependência de rede em tempo de execução, garantindo que os serviços continuem a processar e autorizar requisições rapidamente, mesmo que o servidor Keycloak esteja temporariamente indisponível, mitigando o risco de um Ponto Único de Falha (SPOF).

4.6 Cenário de validação: Acme Air

Para validar a robustez e a escalabilidade das alterações propostas nas seções anteriores, replicou-se a estratégia de segurança em uma arquitetura de referência de mercado: o `Acme Air`[2]. Ele é uma aplicação de *benchmark* desenvolvida para simular o funcionamento de uma companhia aérea. Diferente do cenário anterior, este sistema apresenta uma complexidade inerente a aplicações reais, contendo fluxos de negócio interdependentes, persistência de dados e requisitos de interface de usuário.

[2] Repositório atualizado utilizado como referência: <https://github.com/dlucredio/acmeair-nodejs>

4.6.1 Adaptação da Arquitetura

Para este experimento, a versão monolítica original em Node.js foi refatorada através do `js-distributor`, decomposta em 4 serviços distintos. Eles são *Alpha*, *Beta*, *Gamma* e *Delta*, comunicando-se inteiramente por requisições HTTP síncronas para se adequar a estratégia seguida neste trabalho. A distribuição das responsabilidades ficou definida da seguinte forma:

- **Serviço Alpha:** Atua como o ponto de entrada principal e cliente dos demais serviços. Ele retém a lógica não distribuída e orquestra as chamadas para os microsserviços especializados.
- **Serviço Beta:** Responsável exclusivamente pelas consultas de rotas e voos (funções como `getFlightByAirportsAndDepartureDate`), acessível via HTTP POST na porta 3000.
- **Serviço Gamma:** Centraliza a lógica de validação e manipulação de sessões de usuário (funções com padrão `*Session`), também acessível via HTTP POST.
- **Serviço Delta:** Um serviço utilitário dedicado à inicialização e carga de dados no banco (função `startLoadDatabase`), operando via HTTP GET. Esse serviço será protegido com o *role* `premium`, servindo como a validação do RBAC.

O sistema resultante opera de forma distribuída, com cada serviço isolado em seu próprio container. A Figura 5 ilustra a interface visual da aplicação, que permanece inalterada para o usuário final, apesar da refatoração no *backend*.

Figura 5 – Interface da aplicação Acme Air em funcionamento após a decomposição em serviços Alpha, Beta, Gamma e Delta.

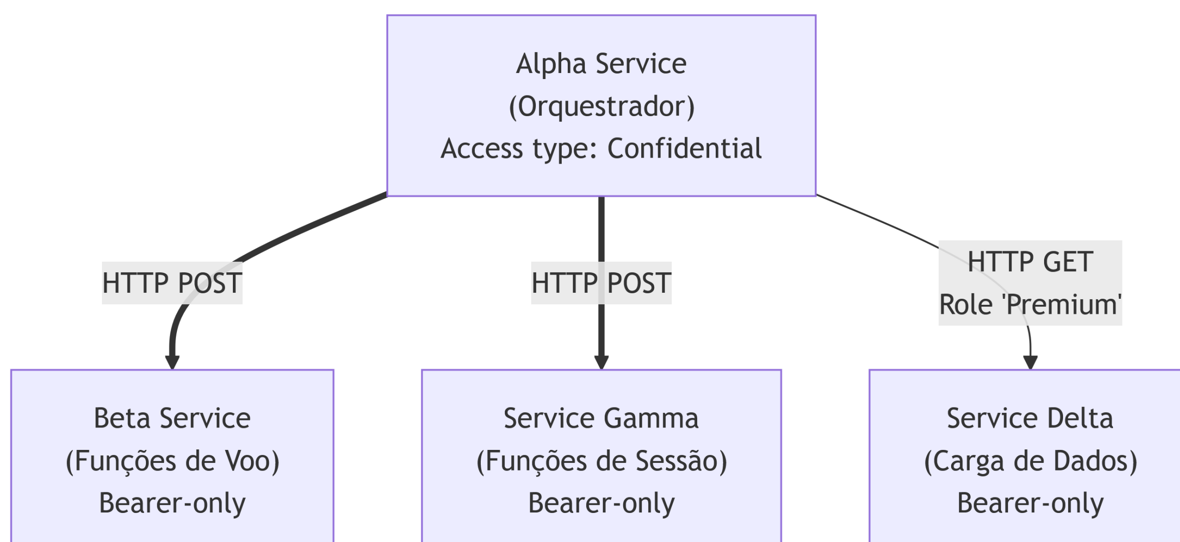
Fonte: Elaborado pelo autor

A imagem destaca as funcionalidades de busca e login, que agora disparam requisições HTTP autenticadas entre os serviços Alpha, Beta e Gamma.

4.6.2 Resultados da Integração

O objetivo central desta etapa foi validar a robustez da biblioteca de contexto (`requestContext`) em um cenário de maior densidade, caracterizado por um volume superior de *endpoints* e funcionalidades agrupadas por serviço. A validação focou em garantir o funcionamento da propagação de identidade em uma topologia onde o serviço de entrada (`Alpha`) hospeda o *frontend* e orquestra chamadas HTTP para os serviços de apoio (`Beta`, `Gamma` e `Delta`). Foi utilizada a mesma infraestrutura de código desenvolvida para o cenário simples. O Terraform provisionou os clientes necessários no Keycloak para cada um dos novos serviços identificados na configuração, além de papéis e *mappers* para correto funcionamento. A Figura 6 apresenta essa visão lógica dos microsserviços comunicando-se com propagação de identidade via HTTP.

Figura 6 – Arquitetura de microsserviços do Acme Air protegida pela solução proposta.



Fonte: Elaborado pelo autor

4.7 Estratégia para futura implementação no `js-distributor`

O resultado final deste trabalho é apresentar uma proposta de extensão da geração de código do `js-distributor`, focada na segurança dos microsserviços. O primeiro passo é permitir que o desenvolvedor declare os requisitos de segurança diretamente no arquivo de entrada (`config.yaml`). O `js-distributor` deve ser capaz de ler as configurações de segurança desse arquivo, e gerar os arquivos de configuração do Terraform baseados nisso. A seguir é sugerida uma forma de implementar essas adições:

Código 4.15 – Arquivo config.yaml

```
1  # Adição de uma seção global para definir o provedor de segurança
2  security:
3    provider: keycloak
4    realm: acme-air
5    discoveryUrl: http://host.docker.internal:8080
6
7  servers:
8      - id: alpha
9          # ... configurações já existentes
10         # Adição de uma seção interna ao microsserviço para a
               segurança
```

```
11        security:
12          type: confidential
13          mappers: 'beta-client', 'gamma-client', 'delta-client'
14          role: none
15    - id: gamma
16        # ... configurações já existentes
17        security:
18          type: bearer-only
19          role: none
20    - id: delta
21        # ... configurações já existentes
22        security:
23          type: bearer-only
24          role: 'realm:premium'
```

A estrutura proposta para o arquivo de configuração foi desenhada para ser intuitiva, abstraindo a complexidade dos arquivos do Terraform. A Tabela 1 e a Tabela 2 detalham a semântica de cada parâmetro introduzido.

Tabela 1 – Parâmetros Globais de Segurança (Seção `security`)

Parâmetro	**Valores Aceitos**	**Descrição e Uso**
`provider`	`keycloak` (atualmente)	Define qual adaptador de tecnologia será utilizado para gerar a infraestrutura. No conceito atual, suporta apenas o Keycloak, mas o conceito pode ser expandido para outros provedores como o Auth0 e Okta.
`realm`	*String* (ex: `acme-air`)	Nome do *realm* a ser criado. Este valor é utilizado tanto pelo Terraform para criar o recurso lógico quanto pelos microsserviços para validar a emissão dos tokens.
`discoveryUrl`	URL válida	Endereço base do servidor de identidade. Essencial para que os microsserviços baixem as chaves públicas e para que o Terraform consiga aplicar as configurações.

Além das configurações globais, cada microsserviço descrito no arquivo recebe uma seção `security` própria, que determina seu comportamento dentro da malha de segurança:

Tabela 2 – Parâmetros de Segurança por Serviço

Parâmetro	**Valores Aceitos**	**Descrição e Uso**
`type`	`confidential`, `bearer-only`	Define o perfil de acesso do cliente. `confidential` deve ser usado apenas no serviço de entrada (gera *Client Secret* e habilita login). Já `bearer-only` deve ser usado em serviços internos (apenas valida tokens).
`mappers`	Lista de *Strings* (IDs dos clientes relativos aos serviços)	Define a relação de confiança. Lista quais outros serviços o token gerado por esse componente pode acessar. O gerador utiliza essa lista para configurar a *Audiência* no token, permitindo a propagação de contexto.
`role`	*String* ou `none`	Especifica se o acesso a este serviço requer um papel específico (ex: `realm:admin`). Se definido, o gerador configura automaticamente a verificação de RBAC no *middleware* do serviço.

Atualmente, o `js-distributor` já dispõe de um mecanismo de manipulação de arquivos que permite replicar recursos do monólito original para os microsserviços gerados. Essa funcionalidade é controlada através de regras configuráveis no arquivo de entrada, que utilizam expressões regulares simples para selecionar quais arquivos devem ser copiados.

Propõe-se aqui evoluir esse mecanismo existente para incluir automaticamente as bibliotecas de suporte à segurança, sem exigir configuração manual do usuário. Isso consiste na inserção mandatória do módulo de propagação de contexto (`requestContext.js`) e na geração dinâmica de arquivos de configuração do adaptador (`keycloak.json`) para cada microsserviço. Dessa forma, todos os nós da rede possuirão as dependências necessárias para operar como servidores de recurso seguros.

Além disso, a lógica de geração do `js-distributor` deve ser aprimorada para incorporar o gerenciamento de contextos de segurança. Ao detectar chamadas entre microsserviços, a ferramenta deverá instrumentar o código resultante para utilizar a função `authorizedFetch()` — conforme detalhado na Figura 4 — em substituição às requisições HTTP padrão. Essa abordagem aprimora a propagação automática das credenciais, abstraindo essa responsabilidade do desenvolvedor.

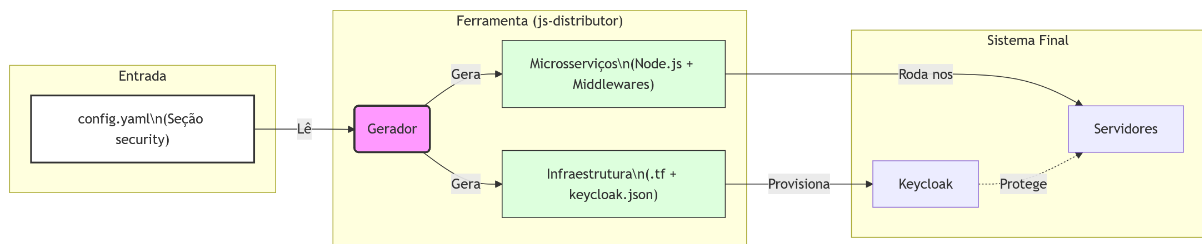
Essa alteração garante que o *token* de acesso seja propagado automaticamente entre os serviços, abstraindo a complexidade da autenticação da lógica de negócio reescrita. Também é necessário cuidar para que as importações realizadas apontem para a versão correta da função utilizada, como a do `authorizedFetch()`, caso necessário.

A geração do código de inicialização (`start.js`) dos microsserviços deve incorporar nativamente o *pipeline* de segurança. Os modelos utilizados para gerar os servidores

`Express.js` deverão ser modificados para instanciar automaticamente os *middlewares* de autenticação e contexto. Além disso, a criação de rotas API passará a aplicar condicionalmente o método `keycloak.connect()` baseado nos metadados definidos na configuração, blindando os *endpoints* sem intervenção manual. Essa abordagem sistêmica preenche a lacuna entre a análise arquitetural e a implantação segura, permitindo que o `js-distributor` entregue microsserviços que são seguros por design.

Para consolidar a proposta, a Figura 7 ilustra o fluxo completo da estratégia de implementação. O diagrama relaciona o arquivo de configuração de entrada com as ações de transformação realizadas pela ferramenta, demonstrando como os artefatos de infraestrutura e código são gerados e orquestrados para compor o sistema final seguro.

Figura 7 – Fluxo consolidado da estratégia de segurança no js-distributor



Fonte: Elaborado pelo autor

5 Conclusão

Decompor sistemas monolíticos em microsserviços é um processo complexo, que vai além da simples separação de código. O `js-distributor` é uma ferramenta excelente nesse processo, mas que possui uma lacuna quanto à automatização da geração de microsserviços com maior segurança. Este Trabalho de Conclusão de Curso teve como objetivo propor uma estratégia de implementação para essa nova funcionalidade, que foi corretamente validada e apresentada no Capítulo 4.

A validação realizada através de dois sistemas distintos demonstrou a viabilidade técnica da proposta. A implementação do Keycloak via Terraform mostrou-se eficaz para garantir a reprodutibilidade do ambiente de identidade, solucionando problemas comuns de configuração manual, como a definição de audiências e escopos de tokens.

No nível da aplicação, a padronização dos serviços sobre o protocolo HTTP e a implementação do padrão de propagação de contexto (via AsyncLocalStorage) permitiram que a identidade do usuário trafegasse de forma transparente entre o serviço de acesso (Alpha) e os serviços de recursos (Beta, Gamma e Delta). Além disso, os testes de aceitação evidenciaram a eficácia do mecanismo de Controle de Acesso Baseado em Papel (RBAC) na proteção de operações críticas no serviço administrativo, corroborando a pertinência do modelo de segurança em profundidade.

5.1 Limitações e Trabalhos Futuros

Apesar dos resultados positivos, a solução atual apresenta limitações. A implementação da segurança foi realizada através de refatoração manual pós-geração, o que ainda demanda esforço do desenvolvedor. Além disso, a arquitetura proposta padronizou a comunicação em HTTP síncrono, não abrangendo o suporte a mensageria assíncrona (RabbitMQ) para simplificar a validação de tokens, o que pode não ser ideal para todos os cenários de alto desempenho. No entanto, isso pode ser explorado futuramente, mantendo os mesmos princípios de autenticação e autorização, apenas mudando o mecanismo de comunicação.

Como trabalhos futuros, sugere-se a implementação da estratégia de automação descrita no Capítulo 4 diretamente no código fonte do `js-distributor`. Isso envolve a atualização dos *Visitors* da ferramenta para injetar os *middlewares* de segurança e a geração automática dos arquivos de configuração do Terraform. Recomenda-se também investigar padrões para validação de tokens em protocolos de mensageria (AMQP/Kafka) para introduzir o suporte a comunicações assíncronas seguras.

Além disso, o presente trabalho focou-se na utilização do provedor de identidade Keycloak. Em futuros trabalhos, é possível estender o suporte da ferramenta a outros IAMs como o Auth0 ou Okta, também muito relevantes na indústria.

Em suma, este trabalho estabelece as fundações para que o `js-distributor` evolua de uma ferramenta de geração de código para uma plataforma de modernização de legado robusta e segura.

Referências

ABGAZ, Y. et al. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering*, IEEE, v. 49, n. 8, p. 4213–4242, 2023. Citado na página 14.

ARAÚJO, L.; MARINHO, E. Desafios da autenticação e autorização na comunicação entre serviços em arquiteturas de microsserviços. *CESAR School*, 2023. Citado na página 22.

Auth0. *O que é OAuth 2.0?* 2025. Auth0 Identity Platform. Acesso em: 29 nov. 2025. Disponível em: <https://auth0.com/pt/intro-to-iam/what-is-oauth-2>. Citado na página 19.

CHATTERJEE, A.; PRINZ, A. Applying spring security framework with KeyCloak-based OAuth2 to protect microservice architecture APIs: A case study. *Sensors*, MDPI, v. 22, n. 5, p. 1703, 2022. Citado na página 22.

ESCHER, G. et al. Js-distributor: decomposing monolith applications into microservices. In: *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2025. p. 879–885. ISSN 2833-0633. Disponível em: <https://sol.sbc.org.br/index.php/sbes/article/view/37073>. Citado 2 vezes nas páginas 12 e 15.

ESCHER, G. S. et al. Js-distributor: decomposing monolith applications into microservices. In: SBC. *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. [S.l.], 2025. Citado 2 vezes nas páginas 15 e 17.

FOWLER, M.; LEWIS, J. *Microservices: a definition of this new architectural term*. 2014. Martinfowler.com. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado 2 vezes nas páginas 11 e 14.

HAINDL, P.; KOCHBERGER, P.; SVEGGEN, M. A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures. *IEEE Access*, v. 11, p. 1–35, 2024. Disponível em: <https://doi.org/10.1109/ACCESS.2024.3406500>. Citado na página 22.

KALUBOWILA, D. et al. Optimization of microservices security. In: *2021 3rd International Conference on Advancements in Computing (ICAC)*. IEEE, 2021. p. 49–54. Disponível em: <https://doi.org/10.1109/ICAC54203.2021.9671131>. Citado na página 23.

MATEUS-COELHO, N.; CRUZ-CUNHA, M.; FERREIRA, L. G. Security in microservices architectures. *Procedia Computer Science*, Elsevier, v. 181, p. 1225–1236, 2021. Citado 2 vezes nas páginas 11 e 17.

OpenID Foundation. *How OpenID Connect Works*. 2025. OpenID Foundation Documentation. Acesso em: 29 nov. 2025. Disponível em: <https://openid.net/developers/how-connect-works/>. Citado na página 19.

PEREIRA-VALE, A. et al. Security mechanisms used in microservices-based systems: A systematic mapping. In: *2019 XLV Latin American Computing Conference (CLEI)*. [S.l.]: IEEE, 2019. p. 1–10. Citado 3 vezes nas páginas 11, 17 e 22.

SANDHU, R. S. et al. Role-based access control models. *IEEE Computer*, IEEE, v. 29, n. 2, p. 38–47, 1996. Citado na página 18.

SHETHIYA, A. S. Building scalable and secure web applications using .net and microservices. *Academia Nexus Journal*, v. 4, n. 1, p. 1–7, Apr. 2025. Disponível em: <https://academianexusjournal.com>. Citado na página 23.