UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I - CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

JOÃO VICTOR MARINHO SOUZA

MIGRAÇÃO DE MONOLITO PARA MICRO SERVIÇOS: UM ESTUDO DE CASO EM APLICAÇÃO DA SEFAZ

CAMPINA GRANDE - PB
2024

JOÃO VICTOR MARINHO SOUZA

MIGRAÇÃO DE MONOLITO PARA MICRO SERVIÇOS: UM ESTUDO DE CASO EM APLICAÇÃO DA SEFAZ

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Ciências da Computação.

Área de concentração: Arquitetura e desenvolvimento de sistemas.

Orientador: Prof. Dra. Kézia Oliveira Dantas.

CAMPINA GRANDE - PB

2024

S729m Souza, Joao Victor Marinho.
Migração de monolito para micro serviços [manuscrito] : um estudo de caso em aplicação da SEFAZ / Joao Victor Marinho Souza. - 2024.
51 f. : il. color.

Digitado.
Trabalho de Conclusão de Curso (Graduação em Ciência da computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2024.
"Orientação : Prof. Dra. Kézia de Vasconcelos Oliveira Dantas, Departamento de Computação - CCT".

1. Arquitetura de software. 2. Desenvolvimento de sistemas. 3. Refatoração. I. Título

21. ed. CDD 005.12

Elaborada por Lêda Cristina Diniz Andrade - CRB - 15/1032 BC

JOAO VICTOR MARINHO SOUZA

MIGRAÇÃO DE MONOLITO PARA MICRO SERVIÇOS: UM ESTUDO DE CASO EM APLICAÇÃO DA SEFAZ

Monografia apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação

Aprovada em: 21/11/2024.

BANCA EXAMINADORA

Documento assinado eletronicamente por:

- **Diogo Florêncio de Lima** (***.660.428-**), em **25/11/2024 14:28:18** com chave **a91a1cfeab5211ef887606adb0a3afce**.
- **Kézia de Vasconcelos Oliveira Dantas** (***.714.244-**), em **25/11/2024 14:35:41** com chave **b16a1d2cab5311ef92a61a7cc27eb1f9**.
- **Fábio Luiz Leite Júnior** (***.848.564-**), em **25/11/2024 16:19:55** com chave **4109cab4ab6211ef96af1a7cc27eb1f9**.

Documento emitido pelo SUAP. Para comprovar sua autenticidade, faça a leitura do QrCode ao lado ou acesse https://suap.uepb.edu.br/comum/autenticar_documento/ e informe os dados a seguir.
Tipo de Documento: Folha de Aprovação do Projeto Final
Data da Emissão: 31/03/2025
Código de Autenticação: 9ce5e3

RESUMO

Este trabalho analisa o processo de refatoração de um sistema monolítico da SEFAZ (Secretaria da Fazenda) para uma arquitetura baseada em micro serviços, abordando desde a análise inicial do sistema até o planejamento e a documentação da nova estrutura, além do desenvolvimento do micro serviço de usuários e das bibliotecas bi_dw_connection_lib e bi_dw_logging_lib. Com foco na escalabilidade, segurança e modularidade, o projeto visa transformar a estrutura original para melhorar a performance e atender as necessidades específicas de expansão. A metodologia aplicada inclui uma avaliação profunda da arquitetura monolítica e o planejamento de uma nova organização modular, que separa os serviços por funcionalidades, permitindo maior eficiência no agrupamento e na execução de funções similares. Os resultados mostram uma arquitetura que facilita a manutenção, promove a escalabilidade e melhora a eficiência operacional do sistema. Conclui-se que a refatoração para micro serviços não apenas aprimora a performance, mas também prepara o sistema para futuras expansões e adaptações, atendendo melhor às demandas tecnológicas e de negócios da organização.

Palavras-Chave: refatoração; arquitetura de software; arquitetura de micro serviços; escalabilidade;

ABSTRACT

This work analyzes the process of refactoring a monolithic system from SEFAZ (Secretary of Finance) to an architecture based on microservices, covering everything from the initial analysis of the system to the planning and documentation of the new structure, in addition to the development of the user microservice and the bi_dw_connection_lib and bi_dw_logging_lib libraries. Focusing on scalability, security and modularity, the project aims to transform the original structure to improve performance and meet specific expansion needs. The methodology applied includes an in-depth assessment of the monolithic architecture and the planning of a new modular organization, which separates services by functionality, allowing greater efficiency in the grouping and execution of similar functions. The results show an architecture that facilitates maintenance, promotes scalability and improves the system's operational efficiency. It is concluded that refactoring for microservices not only improves performance, but also prepares the system for future expansions and adaptations, better meeting the organization's technological and business demands.

Keywords: refactoring; software architecture; microservices architecture; scalability;

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADs ..... *Architectural Descriptions*
API ..... *Application Programming Interface*
ATF..... Administração Tributária e Financeira
HTTP ..... *Hypertext Transfer Protocol*
ICMS ..... Imposto sobre Circulação de Mercadorias e Serviços
JWT ..... JSON Web Token
SCAMF ..... Sistema de Controle e Acompanhamento da Malha Fiscal
SEFAZ ..... Secretaria da Fazenda

SUMÁRIO

1 INTRODUÇÃO

Um dos pilares para a longevidade de um software é uma boa arquitetura, que, como descreve Nick Rozanski em *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, envolve a organização estrutural do sistema em várias camadas compostas por elementos de software, suas propriedades e as relações entre eles, esse planejamento pode garantir que o sistema será escalável e que vai se adequar aos requisitos iniciais e futuros. O primeiro passo para definir como o sistema irá evoluir é a escolha de um modelo arquitetural, os principais modelos de arquitetura são: arquitetura monolítica e arquitetura de micro serviços.

No livro *Monolith to microservices*, Sam Newman (2019) descreve alguns tipos de monolitos, mas o que ele considera como principal durante todo o livro é o “Monolito de processo único”, uma aplicação onde todas as funcionalidades são integradas em um único processo, podendo ter várias instâncias, mas essencialmente possui todas as funcionalidades no mesmo pacote de código, o que torna o desenvolvimento inicial simples, mas leva a problemas de escalabilidade e dificuldades de manutenção conforme o sistema cresce, além de não podermos dirigir recursos como melhor processamento para uma parte específica da aplicação, caso uma parte precise de mais recursos, é necessário escalar a aplicação toda, impedindo que funcionalidades independentes sejam escaladas individualmente.

Em contraste, no livro *Monolith to microservices*, Sam Newman (2019), define micro serviços como uma arquitetura de micro serviços que separa as funcionalidades da aplicação em serviços menores e independentes, que comunicam-se entre si por meio da rede. Essa separação permite que cada serviço seja desenvolvido, testado e escalado de forma independente, proporcionando uma maior flexibilidade e resiliência do sistema.

A aplicação atual foi desenvolvida em uma arquitetura monolítica, onde todas as funcionalidades estão integradas em um único código-fonte. Essa estrutura centralizada faz com que operações mais complexas e de alto consumo de recursos, como a geração de relatórios extensos, acabem competindo por recursos com operações mais simples, como autenticação e consultas rápidas. Como resultado, a execução de relatórios com grandes volumes de dados pode comprometer o

desempenho de toda a aplicação, podendo até deixá-la temporariamente indisponível.

Nesse trabalho, essa mudança de arquitetura se torna relevante devido ao aumento na complexidade e volume de dados que a aplicação precisa processar tornando frequente problemas de desempenho, como o consumo excessivo de recursos para a geração de relatórios e a sobrecarga nas operações de busca em grandes volumes de dados, demonstrando a necessidade de uma estrutura que permita o escalonamento específico para esses componentes críticos.

1.1 Objetivo

O objetivo deste trabalho é realizar a refatoração de um sistema monolítico da SEFAZ para uma arquitetura baseada em micro serviços, documentar e implementar essa transição a fim de conseguir melhorias em escalabilidade e desempenho.

1.1.1 Objetivos específicos

Gerar documentos e diagramas que detalham a nova arquitetura, servindo como guia para a implementação dos micro serviços futuros.

Desenvolver o micro serviço de gerenciamento de usuários, responsável por autenticar e autorizar o acesso dos usuários de forma eficiente, utilizando recursos de maneira otimizada.

Comparar o desempenho entre o sistema monolítico e o novo micro serviço, evidenciando os ganhos em escalabilidade e modularidade promovidos pela nova arquitetura.

2 REFERENCIAL TEÓRICO

Este trabalho aborda o planejamento de uma nova arquitetura baseada em micro serviços, a migração de um sistema em funcionamento e o desenvolvimento de um dos micro serviços planejados. Neste capítulo, vemos os conceitos fundamentais para compreensão abrangente dos desafios enfrentados.

2.1 Arquitetura de Software

A arquitetura de software é um conceito fundamental no desenvolvimento de sistemas. Ela refere-se à estrutura de um sistema, incluindo seus componentes principais, as interações entre eles e os princípios que guiam seu design. Segundo Nick Rozanski (2011), a arquitetura de software é a organização estrutural do sistema em várias camadas compostas por elementos de software, suas propriedades e as relações entre eles, oferecendo uma visão de como os diversos módulos do sistema se organizam para cumprir suas funcionalidades e como se conectam uns aos outros.

A importância de uma arquitetura bem definida está no fato de que ela impacta diretamente a capacidade de evolução, manutenção e escalabilidade de um sistema. Rozanski (2011) explica que qualquer sistema computacional é composto por partes interconectadas, sejam elas poucas ou muitas, e a complexidade dessa ligação pode variar. Dessa forma, a arquitetura define como essas partes interagem e como se comportam em relação ao ambiente externo, garantindo previsibilidade e controle sobre o comportamento do sistema. Nas subseções a seguir, apresentarei as arquiteturas monolítica e de micro serviços, discutindo também algumas técnicas para documentar sistemas de forma eficaz e estratégias para migrar de uma arquitetura monolítica para uma baseada em microsserviços.

2.1.1 Arquitetura de Monolito

A arquitetura monolítica é a forma tradicional de desenvolver sistemas de software, onde todas as funcionalidades são implementadas em um único bloco de código. Um sistema monolítico tem uma base de código unificada, o que simplifica o desenvolvimento inicial e a implementação de transações que envolvem diferentes partes do sistema.

Os principais benefícios dessa arquitetura incluem:

- **Facilidade de desenvolvimento:** É mais simples desenvolver e testar todas as funcionalidades em um único código.
- **Menor sobrecarga operacional:** Como há apenas um único artefato para gerenciar, o processo de implantação e monitoramento pode ser mais simples.

Os principais desafios dessa arquitetura incluem:

- **Dificuldade de Escalabilidade**: Em uma arquitetura monolítica, a aplicação só pode ser escalada como um todo. Isso significa que, mesmo que apenas um módulo específico precise de mais recursos, é necessário escalar a aplicação inteira, o que pode ser ineficiente e aumentar os custos operacionais.
- **Baixa Flexibilidade de Desenvolvimento**: Em um monólito, todas as funcionalidades estão interconectadas no mesmo código-base. Isso limita a capacidade de adotar diferentes tecnologias para diferentes funcionalidades e dificulta a experimentação ou a evolução tecnológica de partes específicas do sistema.
- **Resiliência Limitada**: Em um sistema monolítico, uma falha pode comprometer toda a aplicação, já que todos os módulos compartilham o mesmo espaço de memória. Isso reduz a tolerância a falhas e pode resultar em indisponibilidades totais em caso de problemas.

2.1.2 Arquitetura de Micro serviços

A arquitetura de micro serviços surgiu como uma alternativa à arquitetura monolítica tradicional, oferecendo maior flexibilidade e escalabilidade para sistemas complexos. Segundo Sam Newman (2015), micro serviços são serviços pequenos e autônomos que trabalham juntos. Essa abordagem promove a divisão de um sistema em pequenos serviços independentes, cada um responsável por uma funcionalidade específica.

Os principais benefícios dessa arquitetura incluem:

- **Flexibilidade tecnológica**: Cada micro serviço pode ser desenvolvido usando uma linguagem de programação, banco de dados e tecnologias mais adequados à sua funcionalidade. Isso permite uma maior adaptação às necessidades específicas de cada serviço.
- **Evolução autônoma**: Como os micro serviços são independentes, cada um pode evoluir e ser atualizado sem impactar os demais. Isso facilita a introdução de novas tecnologias e a atualização de partes do sistema conforme as necessidades mudam.
- **Tolerância a falhas**: Se um micro serviço específico falhar, o sistema como um todo não será impactado.
- **Escalabilidade:** É possível escalar apenas os serviços que tenham maior demanda de recursos, otimizando a utilização de infraestrutura e reduzindo custos.

Os principais desafios dessa arquitetura incluem:

- **Complexidade de Comunicação**: Em micro serviços, cada serviço é independente e, frequentemente, precisa se comunicar com outros serviços para realizar suas funções. Essa comunicação geralmente acontece por meio de APIs ou filas de mensagens, o que aumenta a complexidade e exige uma infraestrutura de rede robusta e confiável.
- **Dificuldade na Depuração e Rastreamento de Erros**: Quando ocorre um problema em uma aplicação monolítica, é mais fácil rastrear a origem do erro. Em um sistema de micro serviços, um erro pode propagar-se entre serviços, tornando difícil identificar o serviço exato onde a falha ocorreu.

Essa arquitetura facilita o desenvolvimento ágil, pois permite que novas funcionalidades sejam adicionadas ou modificadas sem a necessidade de alterar todo o sistema.

2.1.3 Técnicas para Descrever uma Arquitetura de Software

Para descrever e documentar adequadamente a arquitetura de um sistema, são utilizadas técnicas como *Architectural Descriptions (ADs)*. Segundo Nick Rozanski (2011), uma descrição arquitetural consiste em um conjunto de produtos que documenta a arquitetura de forma compreensível para os *stakeholders*,

demonstrando que as preocupações deles foram consideradas. Assim, um AD deve ser capaz de comunicar a arquitetura de modo claro e compreensível para todas as partes interessadas, evidenciando como a solução atende aos requisitos de negócios e técnicos.

Para facilitar a compreensão e comunicação da arquitetura, utilizam-se viewpoints (pontos de vista). Rozanski (2011) descreve um *viewpoint* como uma coleção de padrões, templates e convenções que orientam a criação de um tipo específico de visão arquitetural. Cada viewpoint reflete as preocupações de um grupo de stakeholders e fornece diretrizes para a criação das visões arquiteturais correspondentes.

Principais tipos de viewpoints incluem:

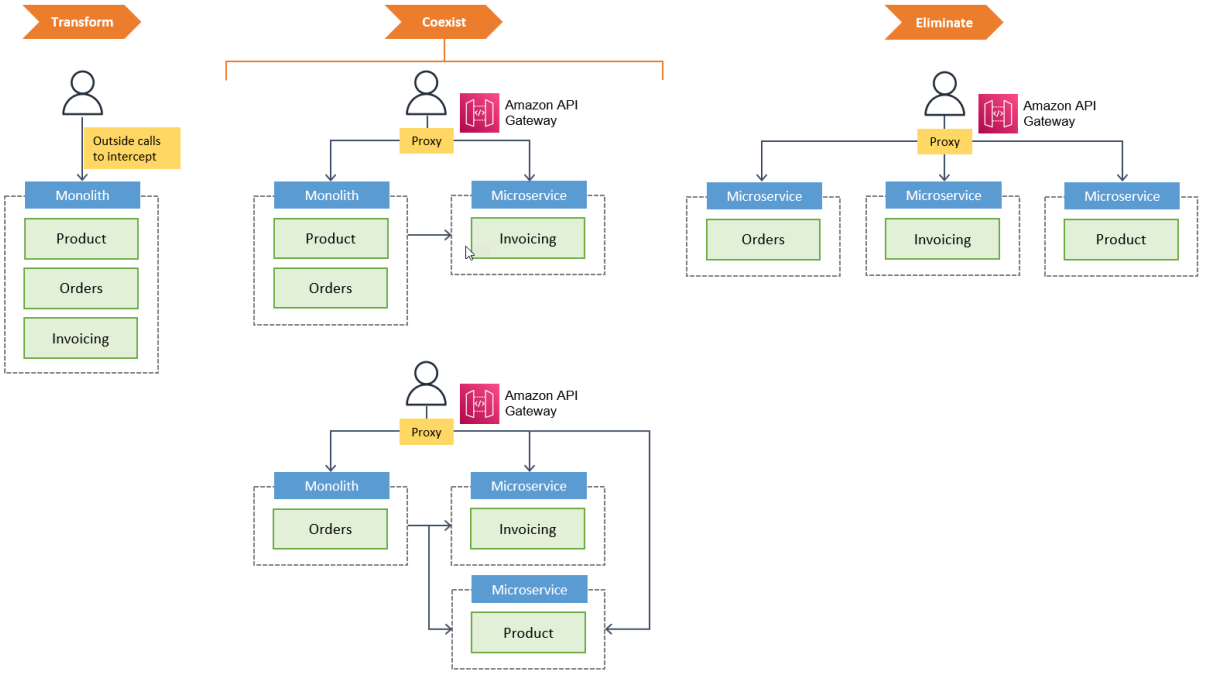
- **Context View:** Descreve as interações entre o sistema e seu ambiente, incluindo usuários, sistemas externos e outros atores. Essa visão mostra as fronteiras do sistema e como ele se conecta com o restante do ecossistema (Rozanski, 2011).
- **Functional View:** Detalha os elementos funcionais do sistema, suas responsabilidades e interações primárias. É a visão mais importante para muitos stakeholders, pois define como o sistema será estruturado para atingir suas funcionalidades (Rozanski, 2011). Além disso, essa visão influencia diretamente aspectos de qualidade como segurança e desempenho.
- **Development View:** Esta visão descreve a arquitetura de suporte ao processo de desenvolvimento, sendo relevante para desenvolvedores e engenheiros que estão envolvidos na criação e manutenção do sistema (Rozanski, 2011).

2.1.4 Técnicas de Migração de Monolito para Micro serviços

Migrar de uma arquitetura monolítica para uma arquitetura de micro serviços é um desafio que exige planejamento cuidadoso. Algumas das principais técnicas incluem o *strangler fig* que consiste em criar micro serviços em volta de funcionalidades mais desacopladas do monolito, roteando gradualmente as requisições para esse micro serviço através de um *API Gateway* até que todas as requisições sejam roteadas para o micro serviço, como exemplificado na figura 1,

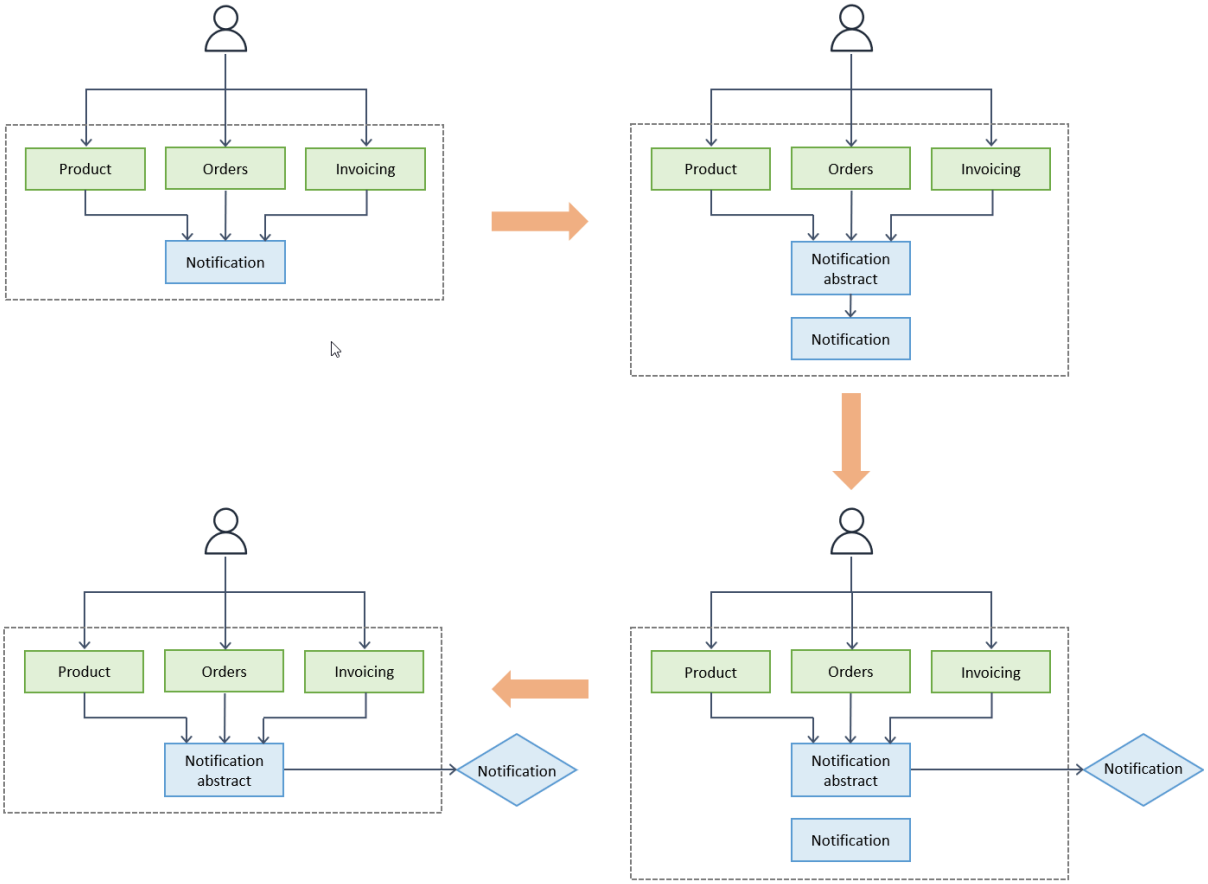
onde extraímos o serviço de “*Invoicing”* do monolito e utilizamos o Amazon APi gateway para redirecionar as requisições para o serviço, fazendo o mesmo processo com o micro serviço “*Product*”, além dessa técnica podemos usar o *Branch by abstraction,* nessa técnica fazemos uma abstração da funcionalidade que queremos extrair para o micro serviço, após isso podemos implementar mecanismos para alterar entre a funcionalidade do monolito ou fazer uma requisição para o micro serviço responsável, idealmente isso pode ser configurado com feature flags, esse processo está exemplificado na figura 2, onde fazemos uma abstração da funcionalidade de notificações, inicialmente usando a funcionalidade do próprio sistema, mas em seguida substituímos por uma requisição para o serviço.

Figura 1 - exemplificação do padrão strangler fig



Fonte: AWS Prescriptive Guidance: Decomposing monoliths into microservices, 2024. Disponível em: <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html>. Acesso em: 9 nov. 2024.

Figura 2 - Exemplificação do padrão Branch by abstraction



Fonte: AWS Prescriptive Guidance: Decomposing monoliths into microservices, 2024. Disponível em: <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html>. Acesso em: 9 nov. 2024.

Para iniciar a migração do sistema foi desenvolvido o micro serviço de gerenciamento de usuários, que desempenha um papel central na nova arquitetura, dada sua responsabilidade em controlar o acesso e as permissões de uso do sistema. Utilizando a técnica Strangler Fig, foi feita a implementação do gerenciamento de usuários de forma isolada, facilitando a transição e minimizando os riscos ao sistema monolítico.

A técnica Strangler Fig é ideal para esse processo, permitindo que funcionalidades específicas e independentes do monólito sejam gradualmente migradas para micro serviços sem impacto direto nas operações existentes. Dessa forma, o novo micro serviço de gerenciamento de usuários começa a operar em paralelo com o sistema original, oferecendo as mesmas funcionalidades de maneira independente, enquanto migramos gradualmente as requisições para esse serviço. Com o passar do tempo e

com a garantia de que o novo serviço está funcionando corretamente, o módulo original do monólito é “estrangulado” ou desativado, redirecionando todas as requisições para o novo micro serviço.

2.2 Tecnologias usadas

Neste capítulo vamos abordar as tecnologias escolhidas para desenvolvimento do sistema, mostrando suas vantagens e os motivos de terem sido escolhidas.

2.2.1 Python

Python é a principal linguagem usada no desenvolvimento do sistema. A escolha dessa linguagem se dá principalmente pela grande flexibilidade que ela propõe, tendo uma sintaxe simples, além de possuir um rico ecossistema de bibliotecas e frameworks que tornam possível o desenvolvimento dos serviços necessários e integrações com outras tecnologias como o RabbitMQ por exemplo, tudo isso com uma alta escalabilidade e um bom suporte à micro serviços. Nesse projeto, Python é aplicado em três frentes principais:

- API REST: A API REST é desenvolvida com Python utilizando o Flask, um micro framework leve e intuitivo, por ser um framework pouco opinativo e não ter muitas dependências, torna-se ideal para construir micro serviços web. O Flask permite o desenvolvimento rápido e eficiente, mantendo a simplicidade na criação de endpoints e no gerenciamento de requisições.
- Plugins: Os plugins do sistema são desenvolvidos com Python e a biblioteca setuptools, que facilita a criação de pacotes reutilizáveis e a gestão de dependências. Isso permite uma modularidade maior, pois os plugins podem ser integrados a outros projetos de maneira prática.
- Consumers: Os consumers são responsáveis pelo processamento assíncrono de mensagens, essenciais para a comunicação entre micro serviços. Utilizamos Python junto com a biblioteca Pika, que facilita a integração com o RabbitMQ para receber, processar e encaminhar mensagens, garantindo a eficiência e a robustez da comunicação assíncrona no sistema.

2.2.2 RabbitMQ

A comunicação assíncrona entre micro serviços no sistema é gerenciada pelo RabbitMQ, um message broker amplamente utilizado por sua confiabilidade e escalabilidade. Algumas das vantagens do RabbitMQ que fizeram com que ele fosse escolhida são:

- Alta Confiabilidade para Mensageria: RabbitMQ garante uma troca de mensagens segura e confiável entre serviços, com suporte a filas persistentes que evitam a perda de dados mesmo em caso de falhas de rede ou serviço.
- Escalabilidade: Com suporte a clusters, RabbitMQ permite a escalabilidade horizontal, o que possibilita o aumento do número de mensagens e conexões processadas conforme a demanda cresce.
- Fácil Integração com Python: No sistema, RabbitMQ é utilizado com a biblioteca Pika, que facilita a integração e comunicação entre o message broker e os micro serviços em Python, permitindo que as mensagens sejam trocadas de forma simples e eficiente.

2.2.3 SQL Server

SQL Server é o banco de dados relacional escolhido para o sistema, sendo responsável pelo armazenamento de dados estruturados com alta consistência e integridade. Algumas das vantagens do SQL Server são:

- Armazenamento Estruturado: SQL Server oferece um modelo de dados relacional que garante a integridade das informações e permite consultas complexas, facilitando o gerenciamento dos dados principais do sistema.
- Integração com Python: Para conectar as aplicações Python ao SQL Server, utilizamos a biblioteca SQLAlchemy, que oferece uma camada de abstração com suporte a mapeamento objeto-relacional (ORM). Isso simplifica o desenvolvimento e mantém o código mais legível, facilitando a manutenção do sistema.

2.2.4 Kong

Kong é utilizado como API Gateway no sistema, atuando como um ponto central de entrada para as requisições HTTP direcionadas aos micro serviços. Algumas das vantagens do Kong que fizeram com que ele fosse escolhido são:

- Roteamento Inteligente: Kong possui recursos avançados de roteamento que permitem redirecionar as requisições para o micro serviço apropriado, garantindo que cada requisição chegue ao destino correto de forma rápida e eficiente.
- Resiliência e Distribuição de Tráfego: Kong facilita a resiliência do sistema ao distribuir o tráfego de maneira equilibrada entre instâncias dos micro serviços, melhorando o desempenho e a disponibilidade.
- Extensibilidade com Plugins: Kong é altamente extensível, com suporte para plugins que adicionam funcionalidades extras, como controle de taxa de requisições e cache. Esses plugins podem ser criados pela comunidade ou desenvolvidos sob medida pela equipe, atendendo às necessidades específicas do projeto.

2.2.5 ELK Stack

Para centralizar os logs das aplicações, utilizamos a ELK Stack (ElasticSearch, Logstash e Kibana), que proporciona uma visão consolidada do comportamento do sistema. Algumas das vantagens da ELK Stack que fizeram com que ela fosse escolhida são:

- Centralização de Logs: A ELK Stack permite centralizar todos os logs dos micro serviços em um único local, simplificando o monitoramento e facilitando a auditoria e análise de desempenho do sistema.
- ElasticSearch para Armazenamento e Busca: ElasticSearch indexa os logs de forma eficiente, possibilitando buscas rápidas e detalhadas, além de facilitar a análise de grandes volumes de dados.
- Visualização com Kibana: Kibana oferece uma interface gráfica para visualização dos logs e criação de dashboards personalizados, ajudando na identificação de padrões e monitoramento de eventos críticos em tempo real.

3 METODOLOGIA

3.1 Estudo de caso

Para mapear o sistema atual, foi feita uma pesquisa exploratória e confirmatória com a hipótese de que a transição para uma arquitetura baseada em micro serviços resolveria os problemas da arquitetura atual, esse processo incluiu a coleta de informações, desenvolvimento e documentação de uma nova arquitetura, implementação de um dos serviços sugeridos e validação dos resultados.

A coleta de informações foi realizada por meio de entrevistas com os desenvolvedores, análises do código-fonte e análises do documento de requisitos da aplicação, possibilitando uma melhor compreensão das principais funcionalidades, da atual arquitetura e dos principais desafios enfrentados pela aplicação.

Como resposta a essas limitações, foi proposta uma nova arquitetura baseada em micro serviços. Essa proposta incluiu a definição e documentação de micro serviços independentes, além da implementação de um dos serviços propostos.

Para validação dos resultados, foi realizada uma análise comparativa entre a arquitetura atual e a proposta, principalmente o tempo de resposta das aplicações. mostrando uma redução significativa no tempo de resposta em cenários de alta carga, maior eficiência na alocação de recursos e uma organização do código que diminuiu consideravelmente a complexidade de manutenção. Essa validação confirmou a hipótese inicial, reforçando a viabilidade e os benefícios da transição para a arquitetura proposta.

3.1.1 Descrição Geral do Sistema

O sistema atual é uma solução monolítica que centraliza todas as funcionalidades em uma única aplicação. Ele foi desenvolvido para a SEFAZ com o objetivo de auxiliar os auditores e fiscais na execução dos trabalhos operacionais da auditoria de indicadores.

As principais funcionalidades da aplicação são:

- Permitir a consulta, visualização e exportação de relatórios.
- Dar suporte aos auditores para realizarem homologação dos valores levantados.
- Gerar demonstrativos com o valor do imposto homologado pela auditoria.

- Dar suporte para a finalização dos trabalhos de auditoria.

3.1.2 Requisitos do Sistema

De acordo com o documento de requisitos, o sistema foi projetado para atender às seguintes demandas:

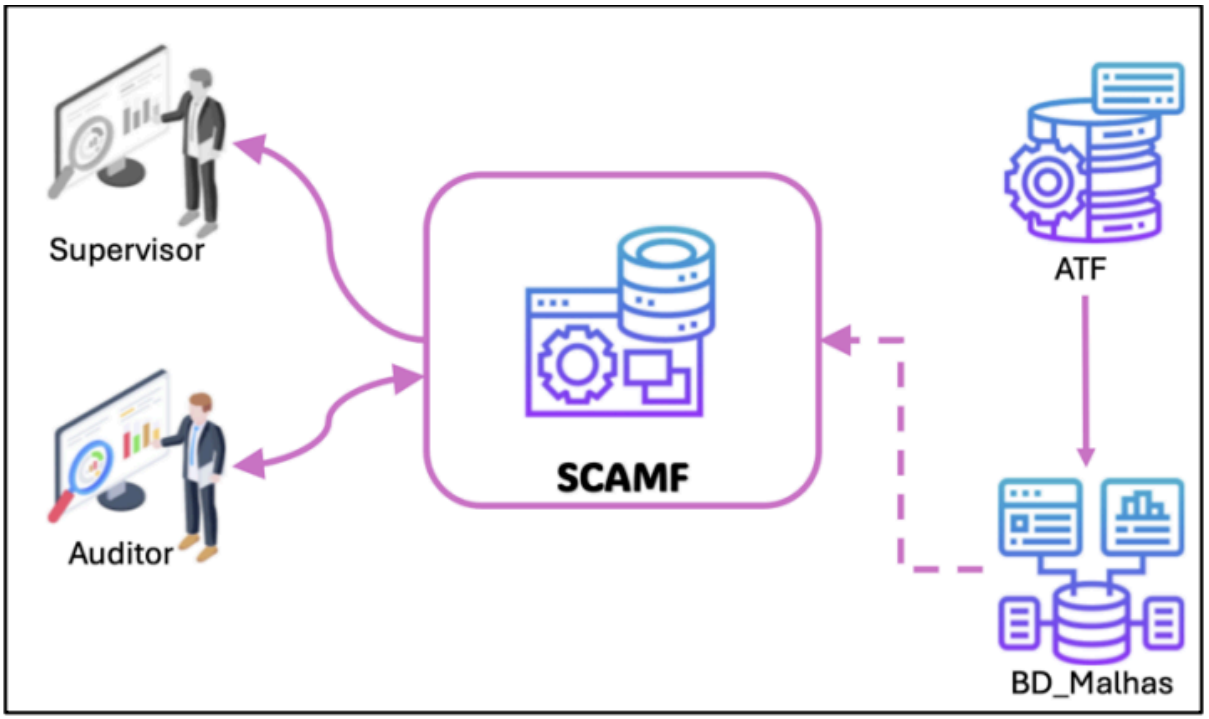
- **Funcionais:**
    - Listar bancos de dados: O usuário deve poder visualizar uma lista de bancos de dados que ele tem permissão para visualizar e homologar a depender do tipo do usuário.
    - Pesquisar bancos de dados: O usuário deve poder pesquisar pelo banco que deseja visualizar ou homologar utilizando o número da ordem de serviço, inscrição estadual ou nome do banco de dados.
    - Listar dados dos relatórios de inconsistências para encaminhamento ao contribuinte: O usuário deve poder consultar/exportar os dados dos bancos para que sejam encaminhados aos contribuintes.
    - Exportar dados dos relatórios de inconsistências para encaminhamento ao contribuinte: O sistema deve permitir que o usuário exporte os dados para uma planilha excel.
    - Destaque dos relatórios de inconsistências a serem homologados: O sistema deve identificar os relatórios com inconsistências pendentes de homologação e destacá-los com a cor vermelha.
    - Verificação dos itens homologados: O sistema deve verificar se todos os relatórios de inconsistências foram homologados e sinalizar caso algum relatório ainda não tenha sido.
    - Avaliação da justificativa do cliente: O sistema deve permitir que o auditor fiscal possa definir a justificativa do contribuinte.
    - Homologação em bloco: O sistema deve permitir que o usuário possa definir as justificativas em bloco para o campo Avaliação justificativa contribuinte, enviando a justificativa de várias linhas da tabela de uma vez.
    - Registro do cálculo do valor ICMS: O sistema deve permitir que o auditor fiscal possa definir o valor para o campo Valor ICMS homologado pelo auditor.

- Filtros específicos para homologação: O sistema deve permitir que se faça filtragem dos dados tabulares por campos definidos previamente.
- Autenticação: O sistema deve permitir apenas o acesso aos bancos de dados por usuários com perfis previamente definidos no contexto da SEFAZ.
- Autorização: Os usuários só devem poder ver as bases de dados que tenham acesso e apenas os auditores devem ter acesso à escrita. Esses auditores podem realizar alterações apenas nos bancos de dados que tiverem permissão concedida pelo sistema ATF (Administração Tributária e Financeira), responsável por controlar e preencher os relatórios de inconsistências e acesso às bases de dados.

- **Não Funcionais:**
  - Segurança: O sistema deve fornecer mecanismos de segurança e autenticação e autorização alinhados com os adotados pela SEFAZ, além disso todas as ações realizadas dentro do sistema devem ser registradas referenciando o usuário que a executou.
  - Usabilidade: O sistema deve ter uma interface simples, permitindo que os usuários aprendam rapidamente as funcionalidades, reconheçam facilmente as operações realizadas e tenha uma proteção contra ações que possam gerar erros.
  - Confiabilidade: O sistema deve ser capaz de executar as ações disparadas pelos usuários sob as condições e ambientes especificados durante todo o tempo em que estiver disponível nos servidores da SEFAZ-PB.
  - Performance: O sistema deve operar dentro de padrões estabelecidos de consumo de recursos para executar as suas funcionalidades. Além disso, o consumo de memória deve estar dentro dos limites estipulados e acertados com a equipe técnica da SEFAZ.

3.1.3 Context Viewpoint do Sistema Atual

Figura 3 - Context viewpoint do sistema atual



Fonte: Documento de requisitos SCAMF

A partir do viewpoint apresentado na figura 3, é possível identificar os principais usuários e sistemas que interagem com o sistema atual. O sistema, inicialmente, monitora o BD_Malhas com o objetivo de detectar a disponibilidade de novos bancos de dados para processamento. O BD_Malhas é um sistema externo responsável por criar bancos de dados que contêm as malhas de documentos fiscais, utilizando os dados extraídos das bases operacionais da SEFAZ.

Quando novos bancos de dados são gerados e ficam disponíveis, o auditor fiscal responsável é notificado. Esse auditor precisa ter acesso rápido e direto a esses dados, o que ocorre por meio do SCAMF. Esse fluxo garante que o auditor possa acompanhar de forma eficiente as atualizações nas bases de dados, permitindo a continuidade das atividades de fiscalização com dados atualizados.

3.1.4 Desafios e Problemas Atuais

Embora o sistema atenda às suas funcionalidades essenciais, ele enfrenta alguns desafios que afetam seu desempenho e evolução:

- **Dificuldade em Escalar Funcionalidades Específicas:** Em um sistema monolítico, não é possível escalar partes isoladas da aplicação, de modo que, se uma funcionalidade específica demandar mais recursos, será necessário replicar toda a aplicação. Por exemplo, há limitações na exportação de relatórios e na consulta de dados em bases volumosas, o que compromete o desempenho e aumenta o consumo de recursos para essas operações.
- **Manutenção Complexa:** O acoplamento forte entre as funcionalidades faz com que pequenas mudanças em um módulo exijam a recompilação e reimplantação do sistema inteiro.
- **Conexões com o banco**: No estado atual do sistema, cada chamada ao sistema abre uma conexão com o banco de dados, esse tratamento das conexões não é ideal e pode gerar um aumento no tempo de respostas da aplicação.
- **Testes unitários**: O sistema atual não possui estrutura para testes unitários na aplicação, o que pode gerar problemas de segurança ao implantar uma nova funcionalidade à aplicação, além disso, outros problemas como o forte acoplamento entre funcionalidades, dificultam a implementação de testes unitários no sistema.

3.2 Proposta inicial da nova arquitetura

Após uma análise do sistema atual, foram identificadas algumas oportunidades para aprimoramentos, especialmente em termos de manutenção, escalabilidade, integração entre os componentes e testes. Diante disso, foi elaborada uma nova proposta de arquitetura que visa a transição do monolito para micro serviços, essa abordagem permitirá a divisão do sistema em aplicações menores com responsabilidades específicas, garantindo uma melhor performance, desacoplamento, maior facilidade de manutenção e mais facilidade para escalar as aplicações.

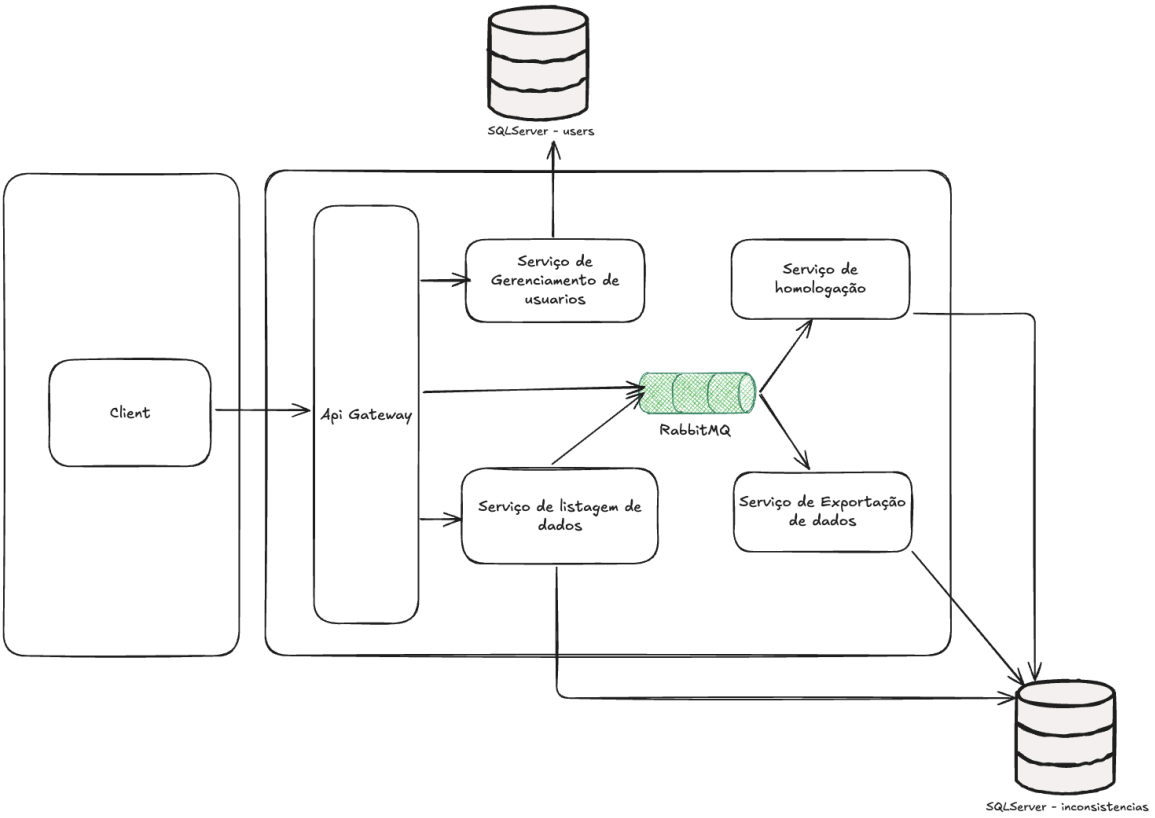
Para guiar a criação e a estruturação dessa nova proposta, foram utilizados os viewpoints de Context, Functional e Development. Esses viewpoints facilitam a compreensão de como os componentes se relacionam entre si, suas funcionalidades e como a implementação deve ser organizada para garantir uma maior compatibilidade e eficiência no desenvolvimento.

Inicialmente as funcionalidades do sistema foram analisadas e agrupadas de maneira que as funcionalidades que tiverem mais em comum serão agrupadas no mesmo serviço, como resultado disso temos os serviços abaixo:

- **Serviço de Gerenciamento de Usuários**: Esse serviço é central para o controle de acesso e segurança do sistema, sendo responsável pelo gerenciamento e pela autenticação dos usuários. Na autenticação, o serviço utiliza mecanismos de verificação, como tokens de segurança, para garantir que apenas usuários autenticados acessem os recursos. O controle de permissões permite ter diferentes níveis de acesso, possibilitando uma gestão de perfis, garantindo a integridade e a confidencialidade das informações no sistema.
- **Serviço de Listagem de Dados**: Este micro serviço fornece acesso rápido e eficiente a dados relacionados a inconsistências e homologações. Ele é estruturado para operar de forma síncrona, permitindo que os usuários obtenham as informações de forma quase imediata. Esse serviço é responsável por realizar consultas complexas e retornar os resultados de maneira organizada e filtrada, facilitando o acompanhamento e a análise dos dados. Ele também possibilita a aplicação de filtros personalizados para que os usuários visualizem exatamente as informações relevantes para suas necessidades, otimizando o tempo e a eficácia do trabalho.
- **Serviço de Homologação**: Focado na gestão do processo de homologação, este micro serviço permite o registro e o acompanhamento das homologações dentro do sistema. Ele oferece uma interface para que os usuários registrem novas homologações e acompanhem os resultados de cada uma.
- **Serviço de Exportação de Dados**: Esse micro serviço é dedicado à exportação de dados de maneira assíncrona, o que é fundamental para o tratamento eficiente de grandes volumes de informação. Ao operar de forma assíncrona, ele permite que o sistema processe solicitações de exportação em segundo plano, liberando o usuário para outras atividades enquanto os dados são preparados. Este serviço cuida de todo o fluxo de exportação, desde a coleta e estruturação dos dados até a formatação em arquivos como CSV ou Excel. Essa abordagem aumenta a escalabilidade do sistema e possibilita o atendimento a múltiplas solicitações de exportação simultâneas, garantindo agilidade e eficiência na entrega dos dados.

Para melhor compreensão da proposta feita, foi feito o diagrama da Figura 4 que mostra todos os micro serviços e como eles se comunicam entre si.

Figura 4 - Proposta inicial da nova arquitetura de micro serviços



Fonte: Elaborado pelo autor, 2024.

Tabela 1 - Lista de micro serviços da arquitetura

Serviço	**Responsabilidade**
Serviço de gerenciamento de usuários	Gerenciamento de usuários
Serviço de Listagem de dados	Listar dados de inconsistências e resultado da homologação de forma síncrona
Serviço de Homologação	Registro das homologações
Serviço de Exportação de dados	Exportar dados de maneira assíncrona

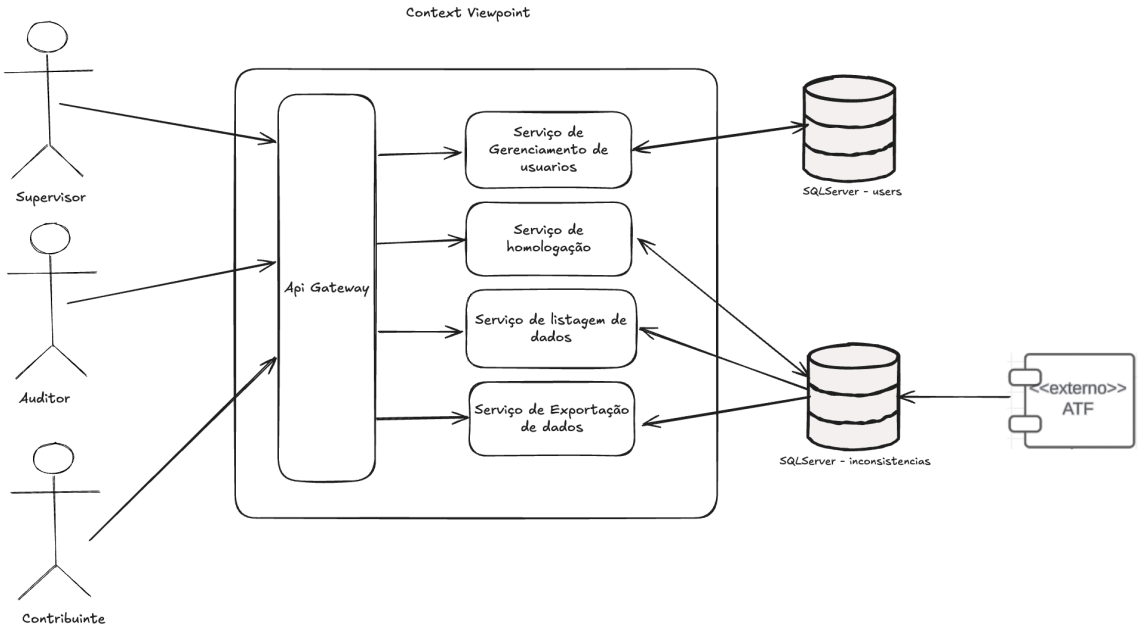
Fonte: Elaborado pelo autor, 2024.

3.2.1 Documentação da nova arquitetura

Para documentar a nova arquitetura foram escolhidos 3 viewpoints: context view, functional view, development view, nesse tópico irei mostrar os viewpoints e falar mais sobre eles.

3.2.1.1 Context view

Figura 5 - Context view da nova arquitetura de micro serviços

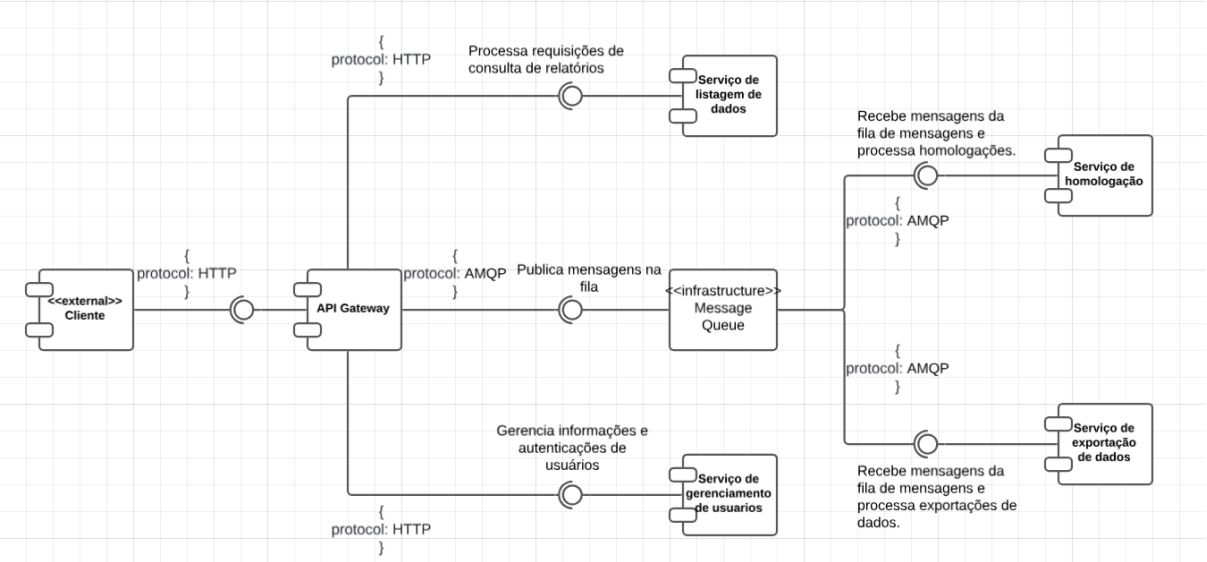


Fonte: Elaborado pelo autor, 2024.

O Context Viewpoint contido na figura 5 mapeia as interações entre os diferentes micro serviços da nova arquitetura, ilustrando como o API Gateway, os consumidores e as APIs que se conectam aos bancos de dados e aos usuários, garantindo que cada serviço tenha um escopo bem definido. Nele podemos ver algumas mudanças no sistema como a adição de um novo tipo de usuário, resultado de novos requisitos que visam a expansão do sistema permitindo que os contribuintes tenham acesso de leitura aos relatórios gerados pela aplicação, assim como a adição dos micro serviços.

3.2.1.2 Functional view

Figura 6 - Functional view da nova arquitetura de micro serviços



Fonte: Elaborado pelo autor, 2024..

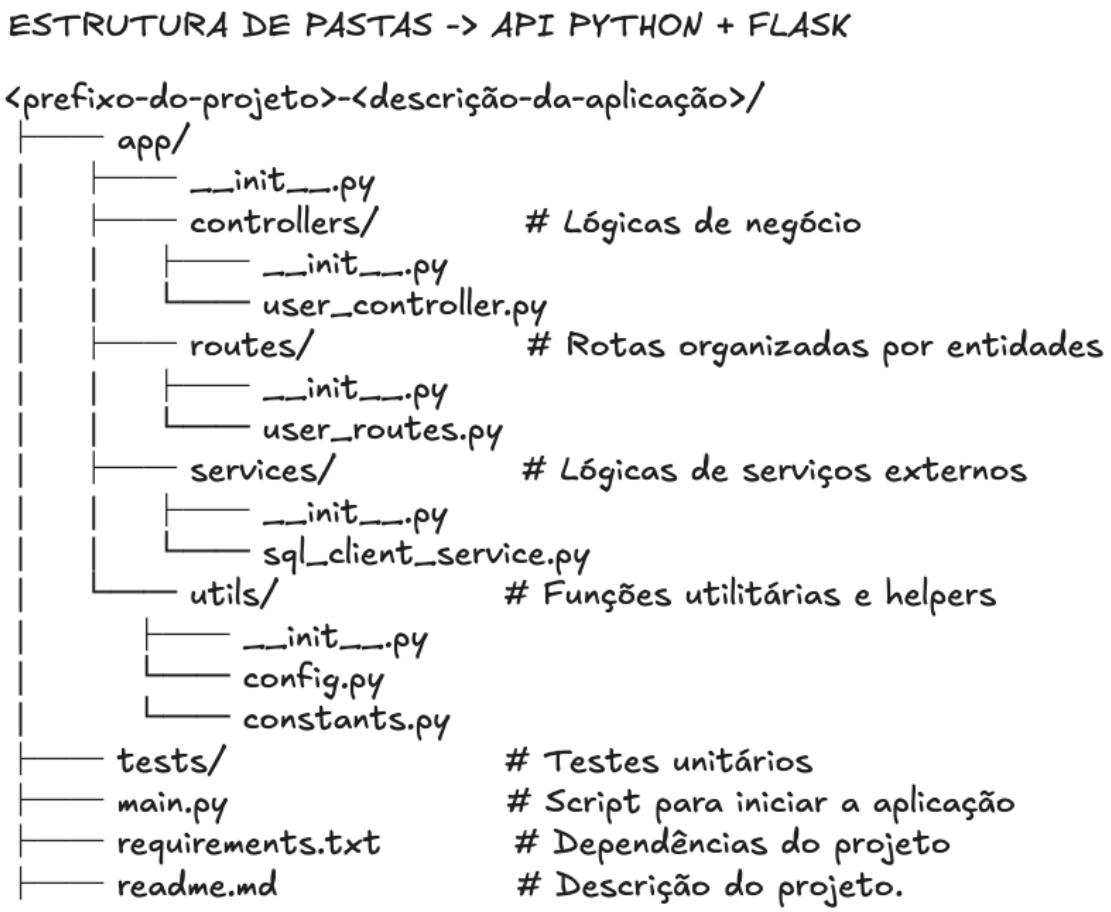
A figura 6 apresenta a interação entre os diferentes componentes do sistema SCAMF, destacando a utilização do api gateway responsável por receber todas as requisições dos usuários e distribuir para o serviço correto e o uso de uma fila de mensagens para orquestrar o processamento assíncrono de dados.

3.2.1.3 Development view

O Development Viewpoint foca nos aspectos técnicos do desenvolvimento, como a estrutura do código, ferramentas e padrões de desenvolvimento.

Para facilitar a compreensão e promover boas práticas no desenvolvimento dos sistemas, foram definidas estruturas específicas para cada tipo de projeto, sendo eles APIs, consumers ou plugins. Essas estruturas são descritas a seguir:

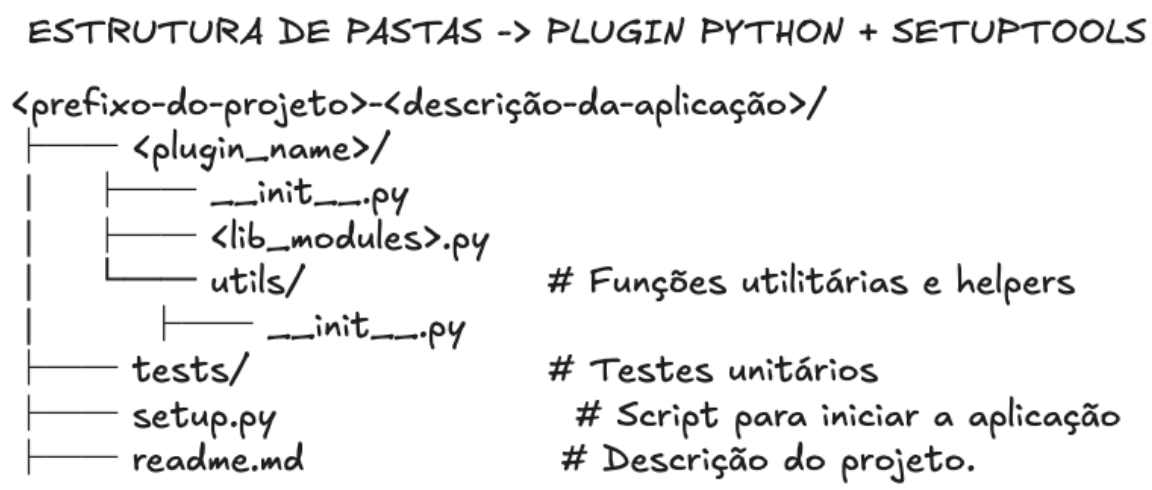
Figura 7 - Estrutura de pastas para os projetos de API



Fonte: Elaborado pelo autor, 2024.

A estrutura apresentada na Figura 7 foi projetada para uma API REST desenvolvida em Python e organizada com base nos princípios de Clean Code e na arquitetura em camadas. O projeto possui um diretório principal denominado app, que centraliza a aplicação. No interior da pasta app, o diretório controllers concentra a lógica de negócios. A pasta routes define as rotas da API, organizando as URLs por entidade, como em "user_routes.py", responsável pelas rotas relacionadas a usuários. No diretório services, encontram-se os serviços externos, incluindo, por exemplo, o "sql_client_service.py", que gerencia interações com o banco de dados ou chamadas a APIs externas. A pasta utils abriga funções utilitárias e arquivos auxiliares. Fora do diretório app, existe a pasta tests, destinada aos testes unitários dos componentes da aplicação, além de arquivos essenciais como main.py, ponto de entrada da aplicação; requirements.txt, que lista as dependências; e readme.md, que fornece a descrição do projeto.

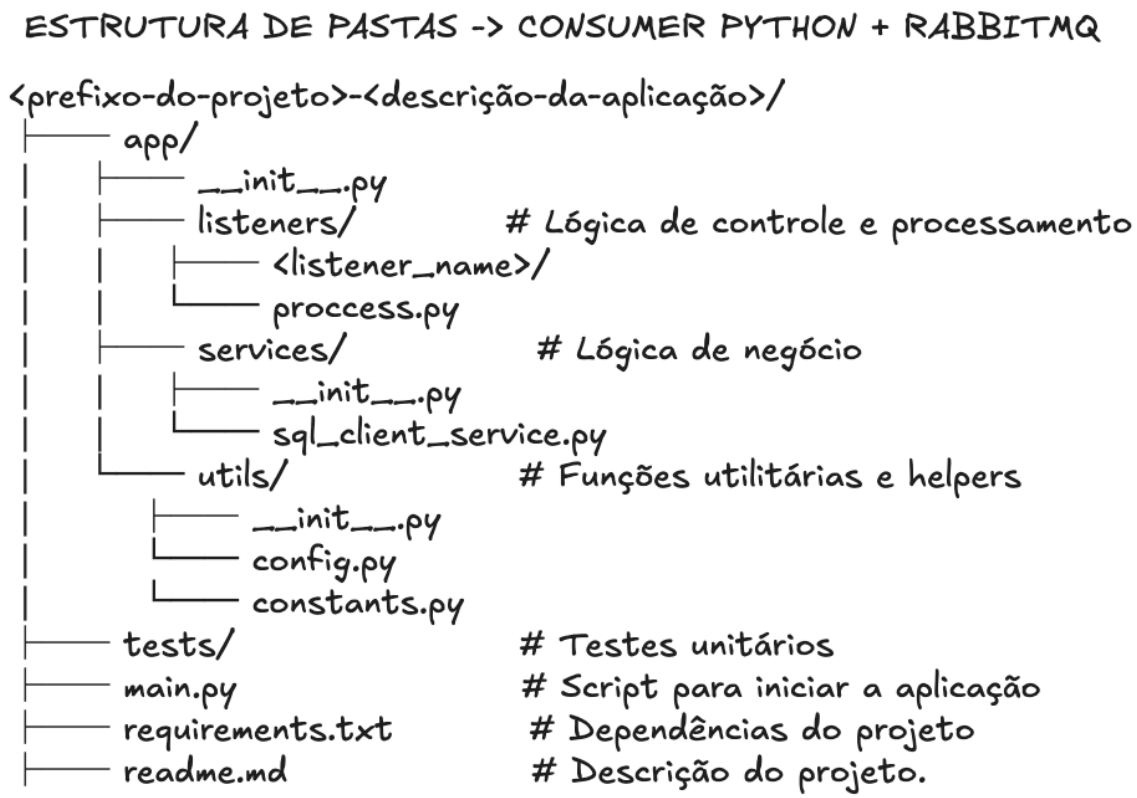
Figura 8 - Estrutura de pastas para projetos de Plugins



Fonte: Elaborado pelo autor, 2024.

A estrutura da figura 8 foi projetada para criar plugins simples em Python modulares em Python, que podem ser reutilizados em diferentes contextos, como em conexões com bancos de dados ou na padronização de logs. O nome do plugin é representado pelo diretório <plugin_name>, onde a lógica principal do plugin deve ser implementada no arquivo <lib_modules.py>. O diretório utils armazena funções auxiliares do plugin. Externamente, existe o diretório tests, onde devem ser incluídos os testes unitários para assegurar a funcionalidade do plugin. O arquivo setup.py facilita a instalação do plugin utilizando o setuptools e o readme.md incluindo um guia mais aprofundado sobre o plugin, mostrando as dependências, objetivos e exemplos de como usar o plugin.

Figura 9 - Estrutura de pastas para projetos de Consumers.



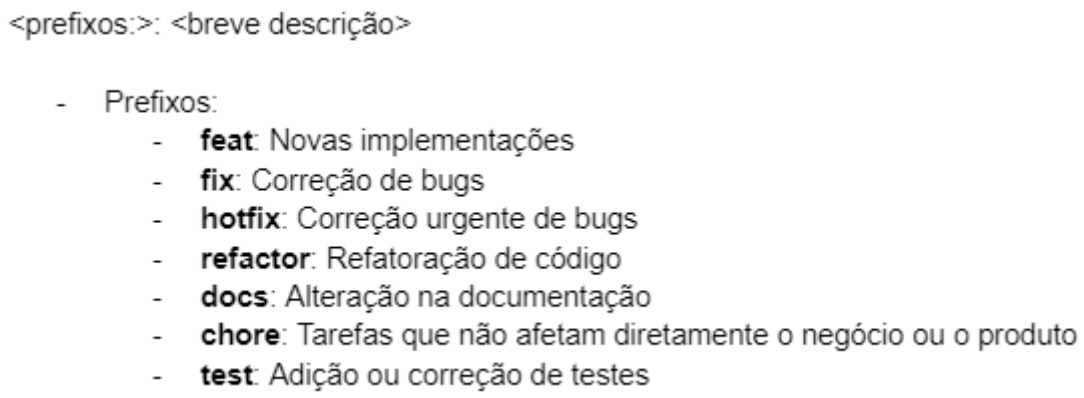
Fonte: Elaborado pelo autor, 2024.

A estrutura da figura 9 foi projetada para uma aplicação em python capaz de consumir as mensagens do RabbitMQ, adequada para processar as mensagens usando o padrão Factory Method, processando as mensagens de acordo com o tipo. A aplicação está organizada no diretório *app*, onde o subdiretório *listeners* centraliza a lógica de controle e processamento de mensagens recebidas. Cada *listener* possui um diretório específico para tratar tipos diferentes de mensagens, e o arquivo *process.py* em cada diretório contém a lógica de processamento dessas mensagens. O subdiretório *services* implementa a lógica para serviços externos, como interações com o banco de dados ou chamadas à api externas e outros micro serviços. O subdiretório *utils* armazena funções auxiliares e as configurações e constantes do consumidor. Fora do diretório *app*, encontra-se os arquivos *main.py*, que é o ponto de entrada da aplicação, o *readme.md*, com a descrição do projeto e a pasta *tests*, destinada aos testes unitários dos componentes da aplicação.

Para os logs será utilizada a stack elk para centralizar e visualizar os logs. Além disso, foi desenvolvido um plugin implementando a lógica de logs no ELK, garantindo padronização dos logs entre os micro serviços. Para fazer a padronização de logs entre projetos foi feito o plugin “bi_dw_logging_lib”, que implementa a conexão com o Logstash a fim de ter uma implementação padronizada entre os projetos.

Para manter um histórico de mudanças claro e organizado, os commits devem seguir o padrão da figura 10:

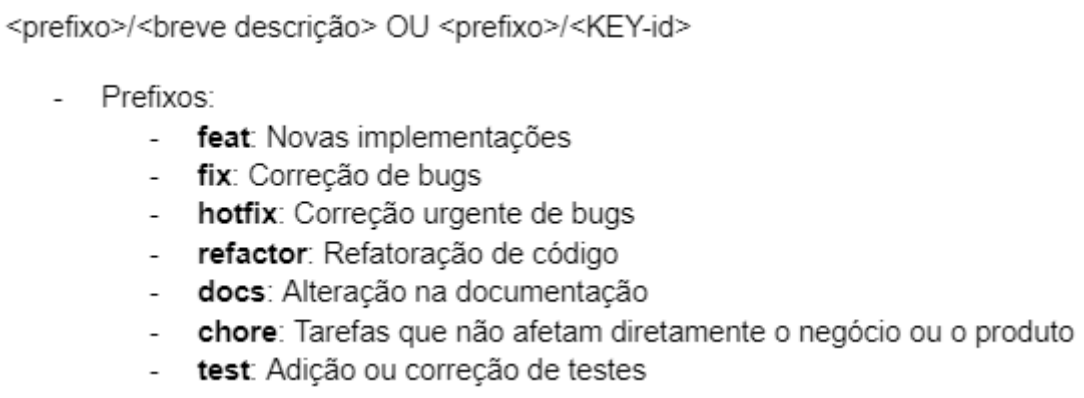
Figura 10 - Padrão de commit



Fonte: Elaborado pelo autor, 2024.

Para facilitar o gerenciamento de branches, permitindo identificar mais facilmente o conteúdo de cada branch, é aconselhável seguir o padrão da figura 11:

Figura 11 - Padrão de criação de branches



Fonte: Elaborado pelo autor, 2024.

3.2.2 Implementação do serviço de gerenciamento de usuários

O micro serviço de usuários é responsável pelo gerenciamento de todas as informações relacionadas aos usuários, incluindo cadastro, autenticação, e permissões de acesso. Esse micro serviço centraliza o controle de usuários e garante que cada micro serviço possa verificar a identidade e permissões de cada usuário. Por ser a base da segurança da aplicação, ele foi escolhido como o primeiro serviço a ser criado.

3.2.2.1 Principais Endpoints do Micro serviço

Figura 12 - rotas de health_check

```
7   health_routes = Blueprint('health_routes', __name__)
8
9   @health_routes.route('/health', methods=['GET'])
10  def health_route():
11      return APIReturns.SUCCESS({'message': "SERVER IS RUNNING"})
```

Fonte: Elaborado pelo autor, 2024.

Para assegurar o funcionamento contínuo do sistema, a implementação de health checks é uma prática recomendada. Com um endpoint de health check configurado, é possível monitorar periodicamente o status da aplicação. Caso a aplicação não responda a essas verificações, um alerta de erro pode ser emitido, permitindo uma resposta rápida, como a reinicialização automática da API, garantindo assim a continuidade dos serviços. Na figura 12 mostramos como esse endpoint foi implementado na aplicação. Essa abordagem ajuda a identificar e resolver problemas proativamente, minimizando o tempo de inatividade.

Figura 13 - rotas de autorização

```
10  auth_routes = Blueprint('auth_routes', __name__)
11
12  @auth_routes.route('/login', methods=['POST'])
13  def auth_login():
14      return method_auth_login(**request.get_json())
15
16  @auth_routes.route('/login_contrib', methods=['POST'])
17  def auth_login_contrib():
18      return method_auth_login_contrib(**request.get_json())
```

Fonte: Elaborado pelo autor, 2024.

Para garantir a segurança do sistema, foram implementados mostramos dois tipos de autenticação implementados na aplicação, estão representados na figura 13 e funcionam da seguinte maneira:

1. login - Esse endpoint autentica os usuários cadastrados em nossa base de dados principal. No entanto, a verificação da senha é realizada por meio de uma conexão com um banco específico no SQL Server. Esse método garante que as credenciais sejam validadas diretamente na base de dados segura, mantendo a integridade e segurança dos dados.
2. login_contrib - Este endpoint autentica os contribuintes. Os tokens de autenticação desses usuários são parcialmente armazenados em nossa base, mas a validação final ocorre em uma base Informix, onde é verificado se os tokens ainda são válidos.

Após a autenticação bem-sucedida em ambos os casos, uma chave privada é usada para gerar um token JWT, que acompanha as requisições subsequentes. Esse JWT é então utilizado para verificar as permissões dos usuários ao acessarem outros endpoints do sistema, garantindo uma autenticação centralizada e segura.

Figura 14 - rotas de criação de tokens dos contribuintes

```
10  create_token_routes = Blueprint('create_token_routes', __name__)
11
12  @create_token_routes.route('/create_token_batch', methods=['POST'])
13  @validate_jwt
14  @check_permissions(['create_token_batch'])
15  def create_token_batch():
16      return method_create_token_batch(**request.get_json())
17
18  @create_token_routes.route('/create_token', methods=['POST'])
19  @validate_jwt
20  @check_permissions(['create_token'])
21  def create_token():
22      return method_create_token(**request.get_json())
```

Fonte: Elaborado pelo autor, 2024.

Os contribuintes são usuários externos ao sistema e, portanto, não possuem cadastro direto na nossa base de dados. Para que eles possam acessar os endpoints, é necessário que um gestor ou fiscal gere um token de acesso específico para cada contribuinte, dando acesso a uma base específica, os tokens podem ser

gerados utilizando os endpoints da figura 14, sendo métodos para gerar de maneira individual ou em grandes quantidades. Esse processo garante que apenas usuários autorizados possam interagir com a aplicação.

Para atender a essa necessidade, foram desenvolvidas duas funções:

1. create_token - Cria tokens únicos para contribuintes individuais, permitindo acesso personalizado e seguro ao sistema.
2. create_token_batch - Permite a geração simultânea de tokens para múltiplos bancos, agilizando o processo de autorização em casos de acesso em larga escala.

Esses tokens garantem que contribuintes externos possam interagir com a aplicação de forma controlada e segura, sempre sob supervisão de um gestor autorizado.

3.2.2.2 Fluxo de Autenticação e Autorização

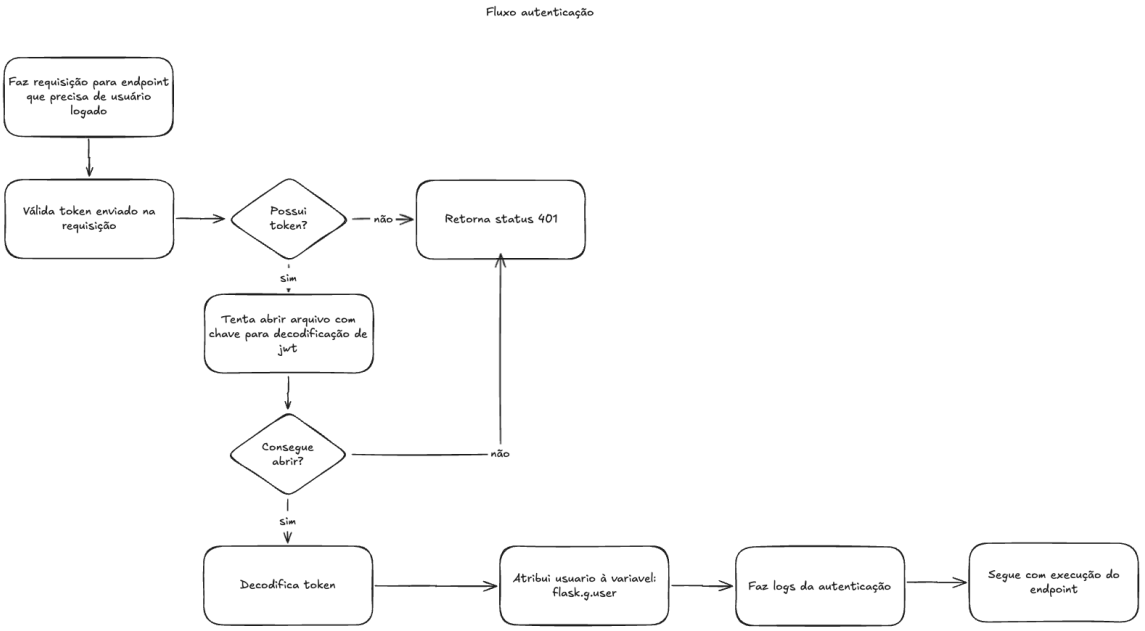
Para o processo de autenticação, há dois modos distintos:

1. Gestores/Fiscais**:**
   Primeiro, verificamos se o gestor ou fiscal está cadastrado em nossa base no SQL Server. Caso esteja, utilizamos as credenciais fornecidas para tentar conectar a uma base específica. Se a conexão for bem-sucedida, um token JWT é gerado utilizando uma chave privada .pem e, em seguida, retornado ao usuário.
2. Contribuintes**:**
   Iniciamos a verificação dos tokens consultando o SQL Server. Depois, verificamos a validade do token em uma base InformixDB. Se o token existir e estiver válido em ambas as bases, ele é retornado ao usuário, permitindo o acesso.

A autorização de acesso aos endpoints é realizada em duas etapas, cada uma com seu próprio decorador:

1. validate_jwt: Este decorador decodifica o token JWT enviado pelo usuário, validando sua autenticidade e extraindo as informações do usuário, esse processo pode ser observado na figura 15.

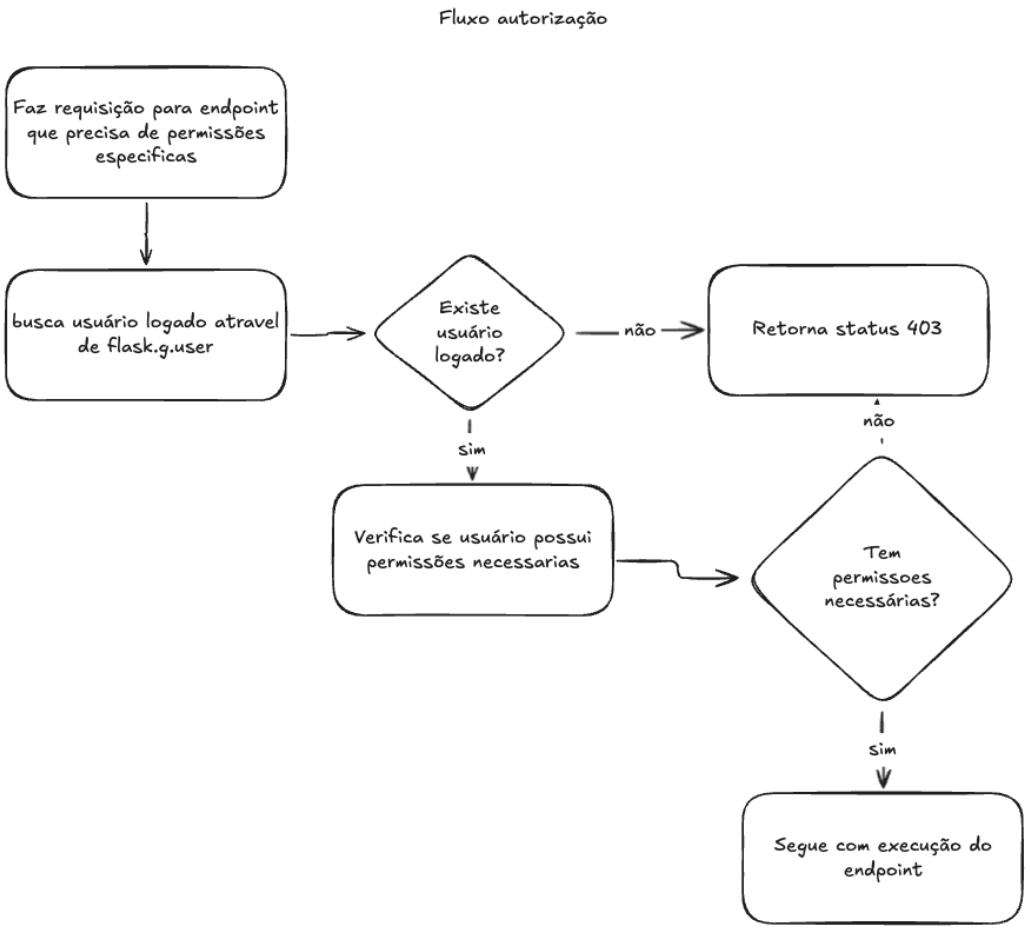
Figura 15 - Fluxo para autenticação



Fonte: Elaborado pelo autor, 2024.

2. check_permissions: Após a validação do JWT, este decorador verifica se o usuário possui as permissões necessárias para acessar o endpoint específico. Utilizando o sistema de permissões baseado no cargo do usuário, ele recebe uma lista de permissões exigidas para o endpoint e confere se o usuário autenticado possui essas permissões, utilizando o objeto ROLES_PERMISSIONS como referência, esse processo pode ser observado na figura 16.

Figura 16 - decorator de verificação de permissões do usuário logado



Fonte: Elaborado pelo autor, 2024.

Essas etapas garantem que apenas usuários com permissões apropriadas acessem as funcionalidades do sistema.

3.2.2.3 Desenvolvimento das Bibliotecas bi_dw_loggin_lib e bi_dw_connection_lib

Para padronizar modelos de tabelas e registros de logs nos micro serviços, foram desenvolvidas as bibliotecas bi_dw_loggin_lib e bi_dw_connection_lib.

A biblioteca bi_dw_loggin_lib é uma solução de logging simplificada para a stack ELK via Logstash, oferecendo reutilização em diferentes aplicações. As principais aplicações dessa biblioteca são:

- Conexão com o Logstash para envio de logs.
- Registro de logs com diferentes níveis, como info, error, warning, e debug.

A biblioteca bi_dw_connection_lib facilita a conexão com bancos de dados SQL Server via SQLAlchemy, além de gerenciar conexões com RabbitMQ para troca de mensagens assíncronas. As principais funcionalidades dessa biblioteca são:

- Conexão com bancos de dados SQL Server.
- Gerenciamento de sessões SQLAlchemy e definição de modelos.
- definição de queries utilizando padrão DAO.
- Conexão e publicação de mensagens em filas RabbitMQ.

Essas bibliotecas promovem a uniformização dos processos de log e comunicação com o banco de dados, otimizando o desenvolvimento e a manutenção dos micro serviços.

4 ANÁLISE COMPARATIVA ENTRE AS ARQUITETURAS MONOLÍTICA E DE MICRO SERVIÇOS.

Para comparar as arquiteturas, foi realizada uma análise de desempenho entre os endpoints do micro serviço de usuários e seus equivalentes na aplicação monolítica em funcionamento. Utilizando o *Collection Runner* do Postman, repetimos a execução das chamadas às APIs para garantir consistência nos resultados. Os testes foram conduzidos em duas máquinas: uma equipada com um processador Ryzen 5500U e 12 GB de RAM, responsável por rodar os serviços auxiliares necessários, como SQL Server e ELK Stack; e outra com especificações de i5 10th e 8 GB de RAM, destinada à execução das aplicações. A separação dos serviços em diferentes máquinas introduziu um gargalo na conexão com o banco de dados, um fator observado e considerado nos resultados dos testes (mais detalhes na seção 4.1).

4.1 Considerações sobre a infraestrutura de testes

Durante a execução dos testes foi necessário dividir os serviços entre duas máquinas distintas devido a uma restrição de acesso ao banco de dados InformixDB, uma das principais dependências do projeto. Apenas uma das máquinas possuía permissão para acessar este banco de dados, mas, devido às suas especificações limitadas, ela não tinha recursos suficientes para rodar todos os serviços necessários. Por essa razão, optou-se por rodar o banco de dados SQL Server e o ELK Stack em uma máquina, enquanto a aplicação principal foi executada na máquina com acesso ao InformixDB.

Entretando, essa configuração introduziu um gargalo de conexão entre a aplicação e o banco de dados, visto que a comunicação entre as máquinas precisou ocorrer via internet, aumentando a latência. Esse fator impactou os resultados dos testes, pois a dependência de uma conexão externa para operações de banco de dados adicionou um tempo considerável nas respostas da aplicação, especialmente nas consultas e operações de escrita no banco, algo que pode ser observado principalmente na seção 4.4. Essa limitação foi levada em consideração ao avaliar os resultados, porém, a execução em um ambiente de produção, com servidores

adequados e acesso direto ao banco, poderá reduzir significativamente essa latência e melhorar o desempenho geral dos serviços.

4.2 Login

Descrição: Autentica o usuário e retorna um token JWT para acesso aos serviços protegidos da API.

Método HTTP: POST

URL: /login

Parâmetros de Requisição:

- username: string – Nome de usuário para autenticação.
- password: string – Senha do usuário para autenticação.

Figura 17 - Performance micro serviço /login

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	1m 5s	0	586 ms

Fonte: Elaborado pelo autor, 2024.

Figura 18 - Performance monolito /api/login

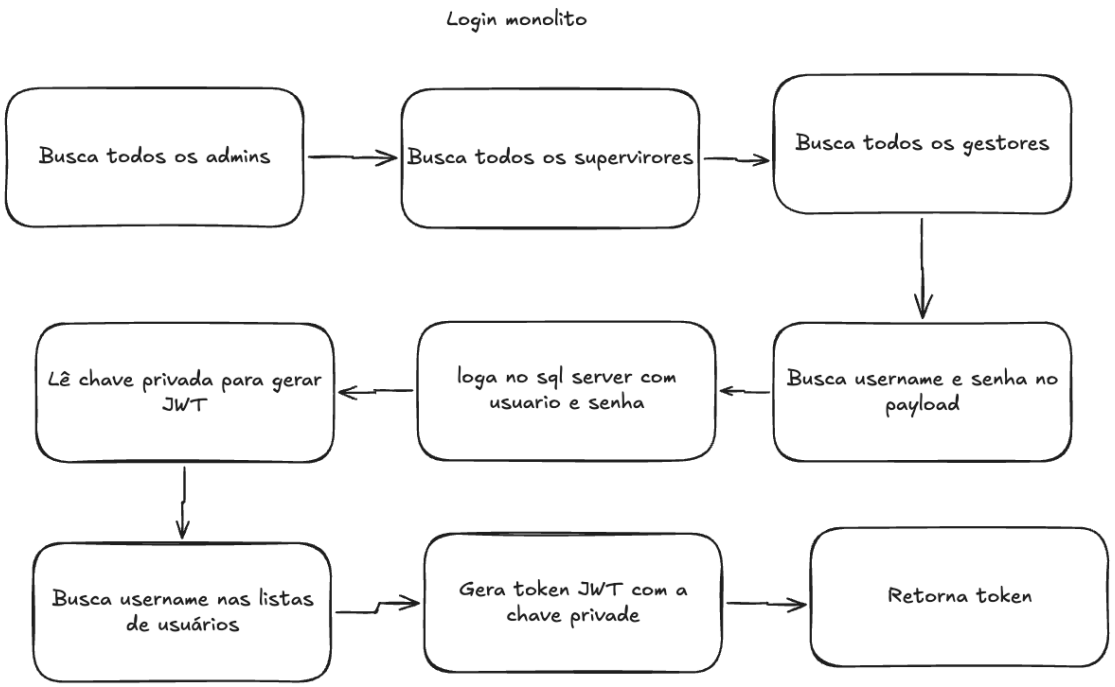
Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	1m 11s	0	644 ms

Fonte: Elaborado pelo autor, 2024.

Observou-se uma melhora de 9% na performance do endpoint de login, alcançada por meio de otimizações na busca de usuários na base de dados e no controle das conexões. No sistema monolítico, todos os usuários são carregados em memória, e a busca pelo usuário que tenta logar ocorre na aplicação em vez de na consulta SQL, conforme observado na figura 17. Essa abordagem tende a se tornar ainda mais lenta à medida que a base de usuários cresce, ampliando a diferença de desempenho em comparação à arquitetura de micro serviços, que executa a busca diretamente no banco de dados, melhorando a eficiência do login conforme observado na figura 18, nas figuras 19 e 20 podemos observar as mudanças que trouxeram essa melhora, no micro serviço temos uma query mais eficiente procurando pelo usuário que está tentando logar ao invés de carregar toda a base e procurar pelo usuário dentro da aplicação, além de ter um melhor gerenciamento de

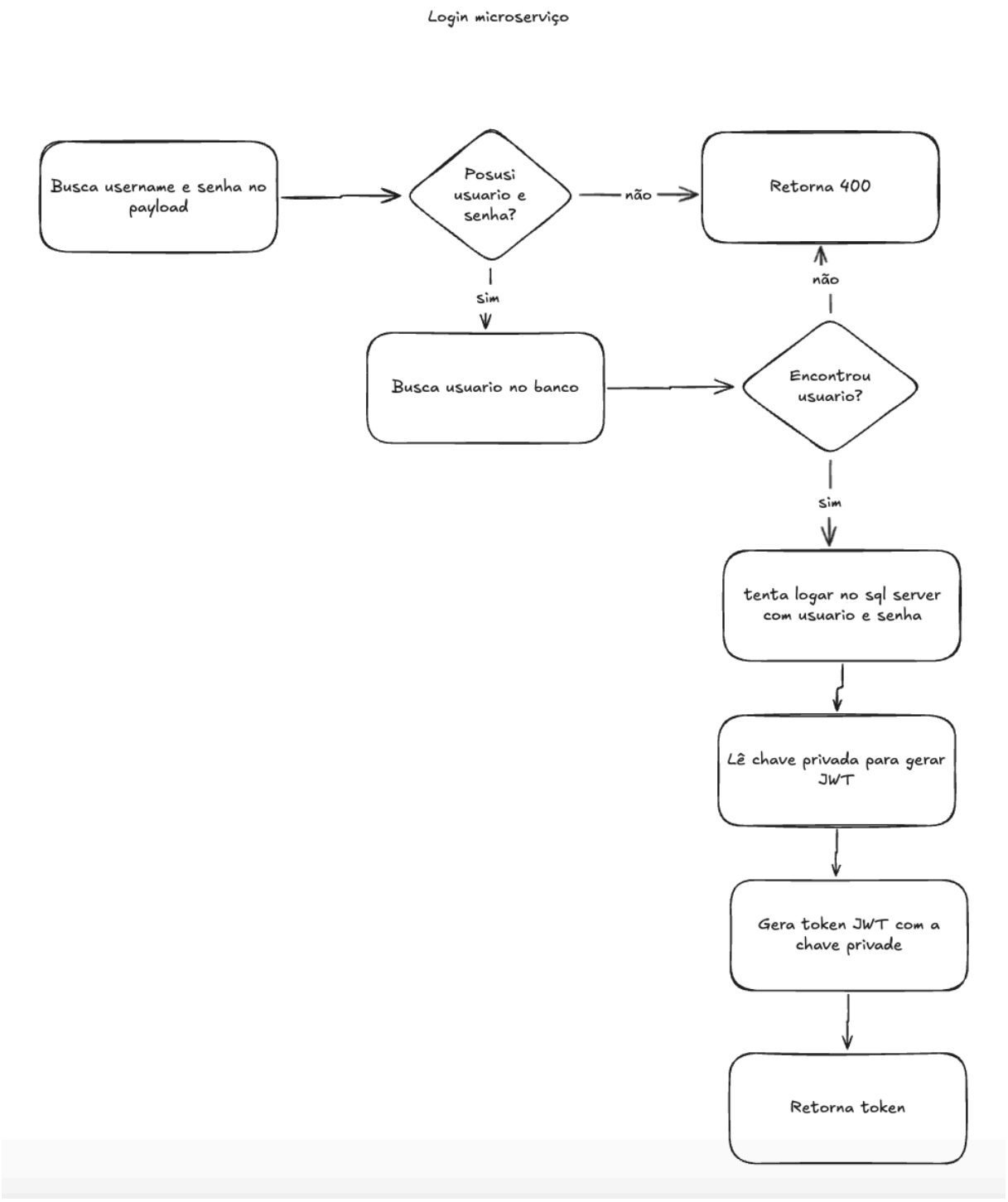
conexões, utilizando uma connection pool dentro do *SqlClientSingleton* ao invés de abrir a conexão com o banco dentro do endpoint.

Figura 19 - Auth login monolito



Fonte: Elaborado pelo autor, 2024.

Figura 20 - Auth login micro serviço de usuários



Fonte: Elaborado pelo autor, 2024.

4.3 Login Contribuintes

Descrição: Autentica o contribuinte e retorna um token JWT para acesso aos serviços protegidos da API.

Método HTTP: POST

URL: /login_contrib

Parâmetros de Requisição:

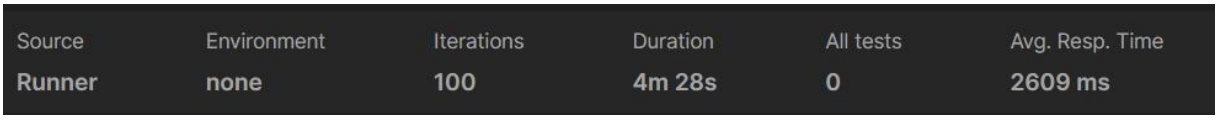
- chave_acesso: string – chave do contribuinte para autenticação.

Figura 21 - Performance micro serviço: /login_contrib

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	3m 9s	0	1822 ms

Fonte: Elaborado pelo autor, 2024.

Figura 22 - Performance micro serviço: /api/login_contrib



Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	4m 28s	0	2609 ms

Fonte: Elaborado pelo autor, 2024.

Conforme observado nas figuras 21 e 22, houve uma melhora de aproximadamente 30% na performance desse endpoint. No micro serviço, a maior parte do tempo é consumida na conexão com o ifxpy (0,89s) e na execução da consulta para verificar o token no banco Informix (0,84s), evidenciando um gargalo na biblioteca ifxpy. Como ifxpy não oferece suporte a connection pools, uma nova conexão precisa ser aberta para cada requisição, impactando a velocidade de processamento. Esse tempo, no entanto, pode ser reduzido ao rodar a aplicação em um servidor de produção com maior poder de processamento. O principal fator para a melhoria de performance nesse endpoint foi a implementação de uma conexão persistente com o SQL Server, em vez de abrir uma nova conexão a cada chamada do endpoint.

4.4 Criação de token

Descrição: Autentica o usuário e retorna um token JWT para acesso aos serviços protegidos da API.

Método HTTP: POST

URL: /create_token

Parâmetros de Requisição:

- database: string – Base de dados que o contribuinte poderá acessar com o token.
- duration: integer – Quantidade de dias que o token vai estar ativo.

Figura 23 - Performance micro serviço: /create_token

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	7m 43s	0	4562 ms

Fonte: Elaborado pelo autor, 2024.

Figura 24 - Performance monolito: /api/create_token

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	7m 41s	0	4541 ms

Fonte: Elaborado pelo autor, 2024.

Conforme observado nas figuras 23 e 24, neste endpoint, não foi observada uma diferença significativa de performance entre os projetos, mas os testes revelaram alguns gargalos importantes. Uma query simples no SQL Server, por exemplo, levou cerca de 1,5s mesmo em uma tabela com apenas um registro. Esse tempo elevado provavelmente se deve ao fato de o backend e o banco de dados estarem em máquinas distintas, ambas sem especificações ideais para operarem como servidores. Em um ambiente de produção com infraestrutura mais próxima e robusta, esses tempos tendem a diminuir. Ainda foi possível notar gargalos nas operações do ifxpy: a conexão leva 0,93s, uma query simples demora 1,07s, um update leva 1,25s, e um insert, 1,32s. Esse desempenho pode melhorar em servidores com melhores especificações, mas também deve-se considerar alternativas, como bibliotecas com suporte a connection pools e o uso de índices no banco Informix para otimizar as consultas.

4.5 Criação de tokens em batch

Descrição: Autentica o usuário e retorna um token JWT para acesso aos serviços protegidos da API.

Método HTTP: POST

URL: /create_token_batch | /create_token_gest

Parâmetros de Requisição:

- sqparamcargafisc_list: list – lista dos sqparams de cada banco que deve ter um token de contribuinte criado.

Figura 25 - Performance micro serviço: /create_token_batch

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	100	4m 18s	0	2505 ms

Fonte: Elaborado pelo autor, 2024.

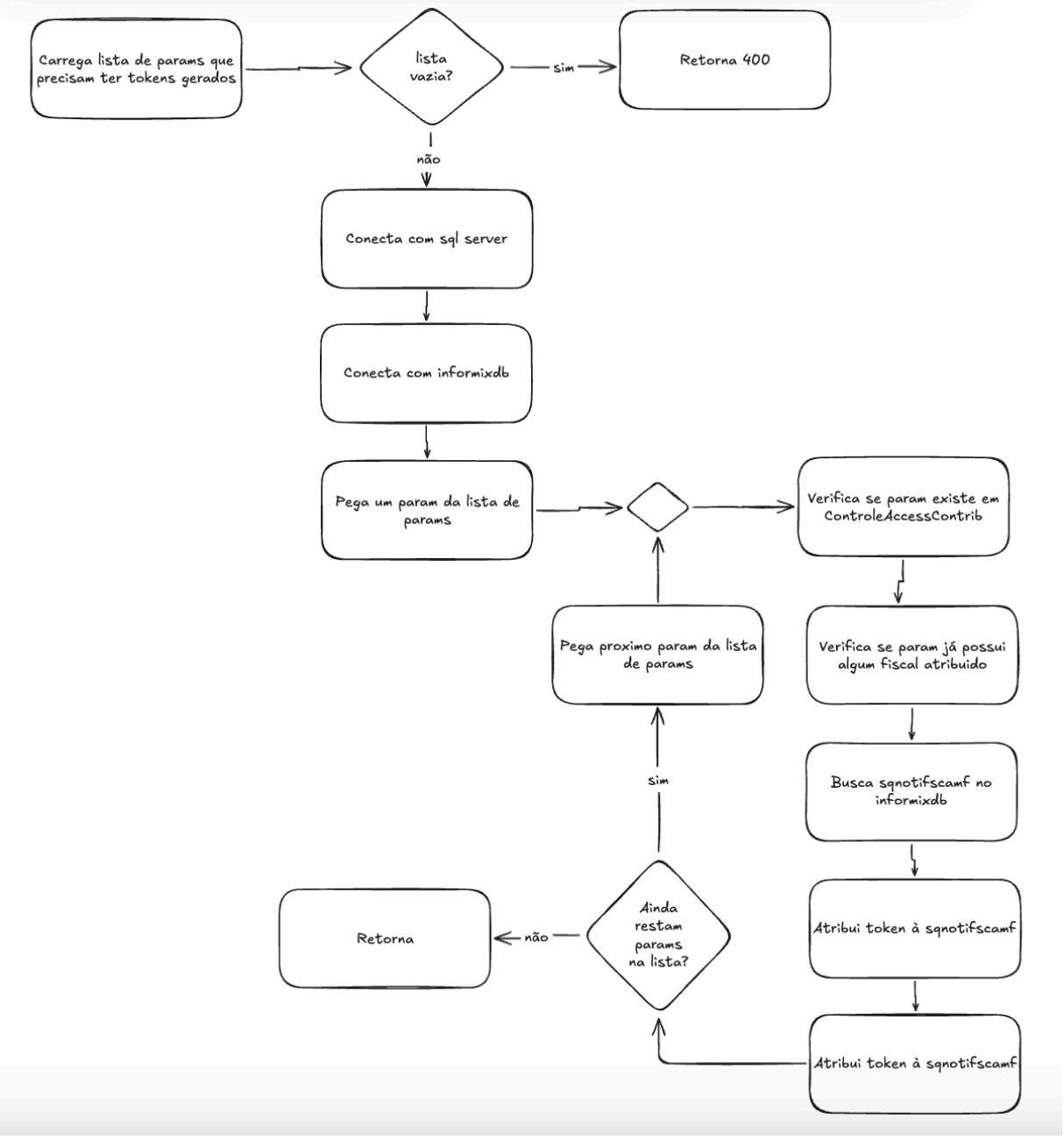
Figura 26 - Performance micro serviço: /api/create_token_gest

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	none	39	7m 35s	0	11659 ms

Fonte: Elaborado pelo autor, 2024.

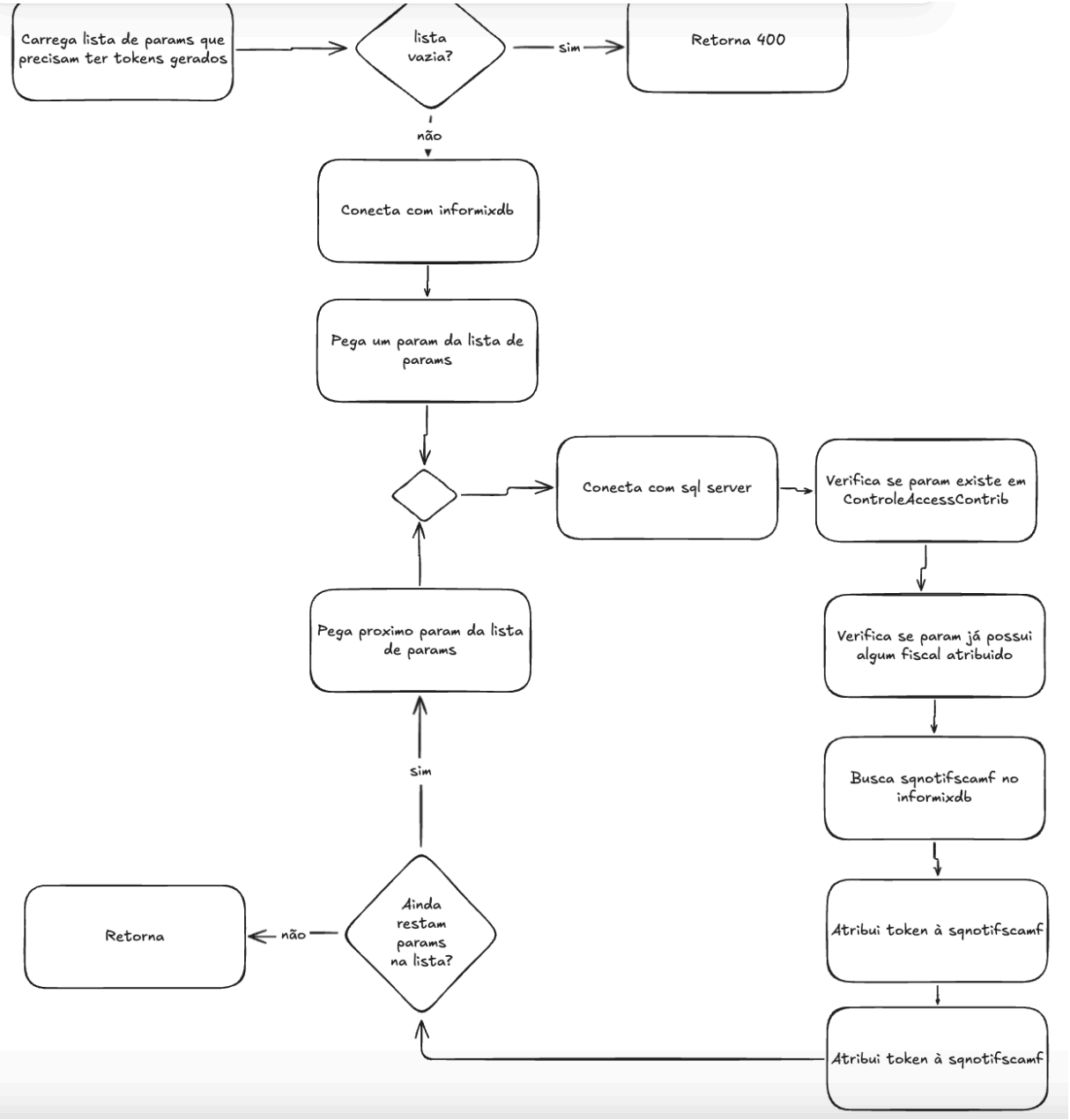
Neste endpoint, foram realizadas requisições contendo dois itens em sqparamcargafisc_list para testar a criação em batch em múltiplos bancos. Observou-se uma melhora de aproximadamente 78,5% na performance, obtida por um uso mais eficiente das conexões. Conforme a figura 25, na arquitetura antiga, uma nova conexão era aberta para cada token a ser criado, o que tornava a operação altamente custosa e aumentava significativamente o tempo de execução conforme mais itens eram adicionados a sqparamcargafisc_list, isso foi resolvido no micro serviço e pode ser observado na figura 26. Podemos observar isso na figura 28, onde o monolito abre uma conexão dentro do loop que está passando pelos bancos, enquanto na figura 27 podemos ver que no micro serviço é utilizada a mesma conexão. Além disso, não foi possível concluir todos os testes no monólito, pois o serviço frequentemente recebia timeouts do SQL Server devido ao excesso de conexões abertas e à falta de reciclagem dessas conexões após o uso.

Figura 27 - Criação de tokens em batch no micro serviço.



Fonte: Elaborado pelo autor, 2024.

Figura 28 - Criação de tokens em batch no monolito.



Fonte: Elaborado pelo autor, 2024.

Este capítulo descreve as etapas realizadas ao longo do trabalho, incluindo o estudo de caso, onde se apresenta o sistema original em sua arquitetura monolítica e os desafios que motivaram a refatoração. Em seguida, expõe a nova proposta de arquitetura, abordando a documentação da estrutura baseada em micro serviços, a implementação do micro serviço de usuários e das bibliotecas de conexão e logging. O capítulo finaliza comparando as duas arquiteturas, mostrando as melhorias obtidas em performance e escalabilidade na nova solução.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

A aplicação atual foi desenvolvida em uma arquitetura monolítica, que traz algumas desvantagens significativas para seu funcionamento. Com o compartilhamento de recursos entre tarefas sem ligação, o sistema pode enfrentar problemas ao executar tarefas que precisam de muitos recursos em paralelo. Esses problemas, agravados pelo aumento no volume de dados e pela necessidade de escalabilidade, tornaram essencial a transição para uma arquitetura baseada em micro serviços. Inicialmente, foram gerados documentos e diagramas que delineiam essa nova estrutura, seguido pela implementação de um dos serviços fundamentais.

A idealização da arquitetura de micro serviços resultou em uma divisão de responsabilidades mais eficiente entre as aplicações, tornando-as mais modulares e permitindo sua execução e escalabilidade independentes. Essa abordagem, combinada com o uso otimizado de recursos — como conexões com bancos de dados, aprimoramento de queries e refatoração das funcionalidades — tem promovido melhorias significativas nos tempos de resposta da aplicação.

O desenvolvimento e implementação do micro serviço de usuários já gerou ganhos consideráveis em desempenho, escalabilidade e facilidade na manutenção, especialmente na otimização dos processos de autenticação e autorização de usuários, cerca de 10%, mas que pode aumentar com base no número de usuários. Em comparação com o sistema monolítico anterior, a nova arquitetura de micro serviços se mostrou muito mais eficiente, principalmente em operações de alta carga, como a criação em lote de tokens, onde foi possível observar uma melhora de aproximadamente 78,5%. Além disso, a modularidade oferecida pelos micro serviços facilita a manutenção e a evolução do sistema, pois cada serviço pode ser atualizado de forma isolada, sem causar impactos no funcionamento global da aplicação.

Para os próximos passos, destaca-se uma maior quantidade e variedade de testes, é necessário a criação de testes unitários garantindo maior segurança ao implementar ou atualizar as funcionalidades, testes de carga com requisições em paralelo, já que os testes fizeram apenas execuções consecutivas, com requisições em paralelo pode mostrar uma maior diferença entre as arquiteturas, e por fim, é necessário testes em ambiente real, visto que os testes foram realizados utilizando máquinas locais, a implantação do micro serviço de usuários em ambiente de

produção. A implementação de novos micro serviços, ampliando ainda mais a modularidade e independência dos componentes da aplicação. Uma prioridade é a criação de um micro serviço de exportação de dados de forma assíncrona, que visa superar uma das principais limitações da arquitetura atual. Essa funcionalidade permitirá a exportação de grandes volumes de dados sem sobrecarregar o sistema, eliminando gargalos e proporcionando uma melhor experiência para o usuário ao reduzir o tempo de espera em operações intensivas de leitura e exportação. Além disso, a abordagem assíncrona permitirá maior eficiência, liberando recursos para outras tarefas enquanto o processo de exportação é executado em segundo plano.

Adicionalmente, como parte dos trabalhos futuros, deve ser elaborado o deployment view detalhado da aplicação, descrevendo a infraestrutura de implantação dos componentes, incluindo servidores, contêineres, balanceadores de carga e configurações de rede, além de também documentar as dependências entre os serviços e integrações externas, proporcionando uma visão clara e compreensível do ambiente de execução da aplicação.

REFERÊNCIAS

AWS PRESCRIPTIVE GUIDANCE. *Decomposing monoliths into microservices*. 2024. Disponível em: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/strangler-fig.html. Acesso em: 9 nov. 2024.

NEWMAN, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. 1. ed. [S.l.]: O'Reilly Media, 2019.

NEWMAN, Sam. *Building Microservices: Designing Fine-Grained Systems*. [S.l.]: O'Reilly Media, 2015.

ROZANSKI, Nick; WOODS, Eóin. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. 2. ed. Boston: Addison-Wesley, 2011.

MARTIN, Robert C. *Código Limpo: Habilidades Práticas do Agile Software*. Capa comum. 1. ed. Rio de Janeiro: Alta Books, 2009.

RABBITMQ DOCUMENTATION. Rabbit Technologies Ltd. Disponível em: https://www.rabbitmq.com/docs. Acesso em: 12 nov. 2024.

PYTHON DOCUMENTATION. Python Software Foundation. Disponível em: https://docs.python.org/3.8/. Acesso em: 12 nov. 2024.

PIKA DOCUMENTATION. Disponível em: https://pika.readthedocs.io/en/stable/index.html. Acesso em: 12 nov. 2024.

SQLALCHEMY DOCUMENTATION. Disponível em: https://docs.sqlalchemy.org/en/20/. Acesso em: 12 nov. 2024.

FLASK DOCUMENTATION. Pallets Projects. Disponível em: https://flask.palletsprojects.com/en/stable/. Acesso em: 12 nov. 2024.

SETUPTOOLS DOCUMENTATION. Python Packaging Authority. Disponível em: https://setuptools.pypa.io/en/latest/. Acesso em: 12 nov. 2024.

KONG DOCUMENTATION. Kong Inc. Disponível em: https://docs.konghq.com. Acesso em: 12 nov. 2024.

ELASTIC STACK DOCUMENTATION. Disponível em: https://elastic-stack.readthedocs.io/en/latest/introduction.html. Acesso em: 12 nov. 2024.