Micro Frontends: Transformando a Abordagem de Desenvolvimento Frontend

Guilherme Eduardo O. Lorentz[1], **Dionísio Machado Leite Filho**[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brasil

guilherme.lorentz@ufms.br, dionisio.leite@ufms.br

Abstract. *This study arises from direct practical application of a contemporary micro frontends approach in a monolithic frontend application. The practical implementation of micro frontends was conducted following a review of advantages and challenges discussed in specialized literature. This study not only highlights the main contributions of this architecture but also examines how these benefits and challenges manifested in practical application during the implementation process.*
The comparison between real-world experiences in transitioning to micro frontends and the theoretical perspectives presented in the literature provides conclusions on the effectiveness of this approach in addressing specific problems faced by monolithic applications. Thus, this work not only contributes to theoretical understanding but also offers a practical perspective on the implementation of micro frontends as a solution to challenges in web application development.

Resumo. *Este estudo surge da prática direta na aplicação de uma abordagem contemporânea de micro frontends em uma aplicação monolítica frontend. A implementação prática de micro frontends foi conduzida após uma revisão das vantagens e desafios discutidos na literatura especializada. Este estudo não apenas destaca as principais contribuições dessa arquitetura, mas também analisa como esses benefícios e desafios se manifestaram na aplicação prática durante o processo de implementação. A comparação entre as experiências reais na transição para micro frontends e as perspectivas teóricas apresentadas na literatura oferece conclusões sobre a eficácia dessa abordagem na resolução de problemas específicos enfrentados por aplicações monolíticas. Dessa forma, este trabalho não só contribui para a compreensão teórica, mas também oferece uma perspectiva prática sobre a implementação de micro frontends como solução para os desafios no desenvolvimento de aplicações web.*

1. Introdução

No panorama contemporâneo das aplicações web, a interação com os usuários é vital, e a exigência por aplicativos web ágeis, adaptáveis a diversos dispositivos e altamente responsivos tem se tornado primordial para muitos negócios [Geers 2020]. Nesse sentido, o desenvolvimento do frontend web tem recebido atenção considerável, levando à ascensão da arquitetura emergente dos micro frontends, fortemente inspirada nos microsserviços [Peltonen et al. 2021].

No contexto das arquiteturas de frontend, de acordo com Peltonen *et al.* (2021), algumas opções comuns incluem a JAMstack, que se concentra na rapidez e segurança

de páginas web, as Single-Page Applications (SPAs), que carregam um único arquivo HTML no navegador e possibilitam mudanças sem atualizar a página, e as aplicações isomórficas, que compartilham o código entre cliente e servidor para gerar HTML. Contudo, o crescimento dessas arquiteturas pode levar a sistemas monolíticos, resultando em complexidade e dependências excessivas.

Como explicado por Nascimento e Sotto (2020), a arquitetura monolítica, caracterizada por uma estrutura de software em que todos os serviços e componentes estão integrados em um único programa, gera complexidade e um alto grau de acoplamento entre os módulos do sistema. Essa abordagem dificulta não apenas a escalabilidade, mas também gera conflitos quando diferentes equipes compartilham a mesma base de código.

Estendendo os princípios dos microsserviços para o lado do frontend das aplicações web, os micro frontends surgem como uma solução disruptiva. Essa metodologia, conforme descrito por Peltonen *et al*. (2021), tem como objetivo superar desafios encontrados nas arquiteturas monolíticas e no desenvolvimento de aplicações de página única. A arquitetura de micro frontends consiste em dividir aplicações monolíticas ou frontends únicos em unidades menores e independentes, permitindo que as equipes trabalhem de maneira autônoma e ágil.

Essa abordagem se fundamenta em princípios essenciais que orientam o desenvolvimento dos micro frontends, incluindo a independência do conjunto de tecnologias adotado por cada equipe, a autonomia na gestão do código, a preferência por APIs nativas do navegador para a comunicação entre aplicações e a ênfase na resiliência, garantindo que a funcionalidade persista mesmo em situações de falha [Pavlenko et al. 2020].

Este trabalho busca investigar e avaliar a prática de aplicar uma abordagem moderna de micro frontends em uma aplicação frontend monolítica. A execução dos micro frontends foi feita após uma avaliação dos benefícios e desafios mencionados na literatura especializada.

O estudo ressalta não apenas as contribuições essenciais dessa arquitetura, mas também realiza uma análise de como esses benefícios e desafios se apresentaram na prática durante a execução. Ao contrastar a experiência real na mudança para micro frontends com as visões teóricas da literatura, o trabalho tem como objetivo fornecer conclusões sobre a eficiência dessa abordagem na solução de desafios específicos encontrados por aplicações monolíticas. Assim, o estudo não apenas aprofunda o entendimento teórico, mas também oferece uma perspectiva prática sobre a implementação de micro frontends como solução para os desafios no desenvolvimento de aplicações web.

2. Referencial Teórico

Neste referencial teórico, examinaremos as principais características, benefícios e desafios da arquitetura monolítica, a evolução para arquitetura de micro frontends e algumas formas de implementação para micro frontends. Cada seção visa fornecer uma compreensão dessas abordagens arquiteturais, permitindo uma visão abrangente das opções disponíveis no desenvolvimento de aplicações de software na era atual.

2.1. Arquitetura Monolítica

Para compor uma explicação mais detalhada sobre a arquitetura monolítica, é essencial contextualizar sua definição, desafios e a evolução ao longo do tempo. Uma arquitetura

monolítica descreve uma aplicação de software na qual os módulos não podem operar de forma independente. Nesse contexto, a aplicação é composta por diversos módulos, cada um com responsabilidades específicas. Tipicamente, uma arquitetura monolítica utiliza um único banco de dados. As interdependências entre os módulos implicam que a aplicação é compilada e montada como uma entidade coesa. Assim, qualquer modificação em um módulo não apenas requer a recompilação desse módulo específico, mas também a remontagem completa da aplicação [Milic e Makajic-Nikolic 2022].

A complexidade e o alto acoplamento existentes em aplicações monolíticas frequentemente resultam em dificuldades para equipes de desenvolvimento trabalharem em conjunto. Alterações ou adições de novos recursos podem ser problemáticas devido à estrutura monolítica, o que pode impactar o funcionamento global do sistema. A inclusão de novas tecnologias ou linguagens de programação é limitada, e qualquer modificação pode afetar todo o sistema, devido à natureza integrada da arquitetura [Nascimento e Sotto 2020]. Zateishchikov (2023), explica em sua pesquisa que, como consequência do crescimento do código e adição contínua de recursos, a manutenção e desenvolvimento dos aplicativos monolíticos tornaram-se desafiadores e consumidores de tempo.

A arquitetura monolítica é também desafiada pela crescente complexidade e dificuldade de escalar as equipes de desenvolvimento. As arquiteturas monolíticas impedem uma escalabilidade eficaz, já que adicionar mais desenvolvedores nas equipes de frontend não necessariamente resulta em um aumento proporcional na produção, devido às limitações impostas pela estrutura escolhida [Peltonen et al. 2021].

Com a evolução do desenvolvimento web e a popularização da internet, a prática de abordagens separadas para as camadas frontend e backend se tornou mais adequada na evolução do desenvolvimento de aplicativos, como apontado por Ferracaku (2021). Isso possibilitou a separação das equipes de desenvolvimento do backend e do frontend, facilitando o trabalho de profissionais com habilidades diferentes em cada camada. No entanto, essa evolução também provocou uma transferência de muita complexidade para a camada de backend. Para lidar com isso, surgiu a necessidade de subdividir essa camada em unidades menores e autônomas, as quais são conhecidas como microsserviços [Ferracaku 2021].

Em resposta a esses desafios, a arquitetura de microsserviços, como apresentada na Figura 1, ganhou destaque como uma alternativa viável para lidar com a complexidade crescente das aplicações. No entanto, essa abordagem, também, trouxe à tona desafios no frontend, uma vez que a evolução para microsserviços geralmente começava pelo backend, deixando a interface do usuário como uma grande entidade monolítica. Isso levou ao surgimento dos “micro frontends”, uma abordagem que busca decompor a interface do usuário em unidades menores e independentes, permitindo uma evolução mais flexível e escalável do frontend [Manisto et al. 2023].

Embora as arquiteturas monolíticas ofereçam simplicidade no gerenciamento e desenvolvimento de aplicativos, essa estrutura mais rígida e integrada tornou-se limitante em alguns casos, diante da evolução das necessidades do desenvolvimento de software. A transição para abordagens mais modulares e escaláveis, como os microserviços e micro frontends, é uma resposta direta aos desafios enfrentados pelas arquiteturas monolíticas,

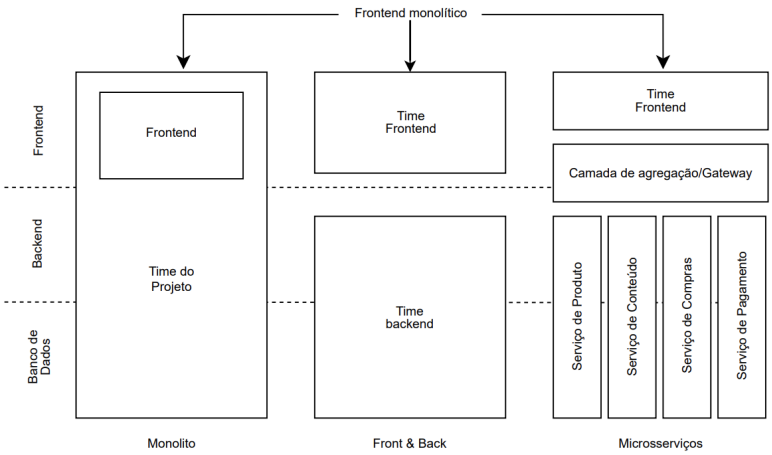

Figura 1. Arquitetura de microsserviços no backend com uma camada de apresentação frontend monolítica (adaptado de Peltonen *et al.*, 2021)

permitindo uma evolução mais ágil, flexível e adaptável das aplicações em resposta às demandas modernas [Mezzalira 2021].

2.2. Micro Frontends

A abordagem de micro frontends amplia os princípios dos microsserviços para a esfera do frontend de uma aplicação, transformando a estrutura monolítica de uma aplicação web, originalmente baseada em um único código, em um sistema que une múltiplos micro aplicativos frontend. Cada um desses aplicativos independentes é desenvolvido, executado e implantado de maneira autônoma [Peltonen et al. 2021].

Micro frontends tratam uma aplicação web como uma reunião de funcionalidades ou subdomínios de negócios, o que implica que cada equipe seja responsável por um domínio específico, como descrito por Peltonen *et al.* (2021). A ideia por trás desse conceito é dividir a aplicação em unidades menores, permitindo que cada uma lide com uma função particular, em contraste com o desenvolvimento de uma aplicação monolítica [Nascimento e Sotto 2020]. Cada equipe, assim, gerencia e aprimora uma parte da aplicação web, atendendo a um propósito ou negócio específico.

Ao considerar a adoção de uma arquitetura de micro frontend, é crucial tomar decisões estratégicas durante as fases iniciais do projeto. A fragmentação da aplicação em aplicativos menores, autocontidos e independentemente desenvolvíveis é a primeira etapa para garantir que as equipes possam trabalhar de forma desacoplada e independente. Isso assegura que um aplicativo menor não interfira no funcionamento de outros aplicativos menores. A construção de micro frontends especializados em domínios específicos com responsabilidades individuais é a chave para alcançar esse objetivo [Poloskei e Bub 2021].

Para compreender os micro frontends, a decisão essencial reside na consideração técnica sobre sua estrutura. Segundo Mezzalira (2021) , existem duas abordagens distintas: a divisão horizontal e a vertical.

Na divisão horizontal, múltiplos micro frontends são carregados em uma única página, demandando a coordenação de várias equipes, cada uma responsável por uma

seção da visualização. Isso implica em maior flexibilidade, permitindo a reutilização de alguns desses elementos em diferentes interfaces. Entretanto, requer uma disciplina e governança robustas para evitar a proliferação descontrolada de micro frontends em um mesmo projeto [Mezzalira 2021].

Por outro lado, na divisão vertical, cada equipe assume a responsabilidade por um domínio de negócios específico, como autenticação ou experiência de pagamento. Aqui, o Design Orientado por Domínio (DDD) desempenha um papel crucial [Peltonen et al. 2021]. Embora não seja comum aplicar os princípios do DDD em arquiteturas de frontend, neste cenário há uma forte justificativa para explorá-lo.

Em seu livro, Mezzalira (2021) define: o DDD é uma metodologia de desenvolvimento de software que concentra o processo de programação em torno de um modelo de domínio, compreendendo profundamente os processos e regras inerentes a esse domínio.

2.2.1. Vantagens dos Micro Frontends

Implantações em aplicações monolíticas tradicionalmente enfrentam desafios relacionados aos tempos de implantação e versionamento, como afirmado por Ferracaku (2021). Por outro lado, os micro frontends possibilitam implantações mais rápidas e apoiam a entrega contínua, uma vez que podem ser desenvolvidos, testados e implantados de forma independente [Poloskei e Bub 2021]. Essa abordagem reduz o risco associado ao lançamento de novas funcionalidades, pois, em caso de problemas, apenas a parte correspondente da aplicação é afetada, mantendo o restante funcional [Linkola 2021].

As arquiteturas monolíticas frequentemente restringem as equipes a uma única escolha de tecnologia, uma decisão que precisa ser tomada precocemente e permanece fixa [Ferracaku 2021]. Poloskei e Bub, (2021), estabelece em seu artigo que, os micro frontends, por outro lado, permitem o uso de diversas tecnologias, frameworks e bibliotecas em diferentes componentes da aplicação, possibilitando que equipes individuais escolham as ferramentas mais adequadas às suas tarefas específicas. Isso facilita atualizações incrementais e a adoção de novas tecnologias em partes específicas da aplicação, evitando a necessidade de uma reescrita em larga escala. Isso também divide os custos de desenvolvimento em um cronograma mais extenso, de acordo com Linkola, (2021).

Em comparação com as aplicações monolíticas, as aplicações de página única tradicionais enfrentam desafios na divisão de equipes e problemas de manutenção que aumentam com o tamanho da equipe [Ferracaku 2021]. Os micro frontends agilizam o processo de desenvolvimento ao dividir o frontend em partes menores e mais gerenciáveis, conferindo maior autonomia às equipes segundo Zateishchikov (2023). Além disso, permitem atualizações ou substituições de componentes individuais sem afetar toda a aplicação, tornando a migração ou modernização incremental de uma aplicação frontend monolítica mais gerenciável, se necessário, como explicado por Poloskei e Bub (2021).

Os micro frontends, sendo componentes menores em relação às aplicações monolíticas, são mais fáceis de desenvolver e projetar, desencorajando o acoplamento não intencional entre componentes [Linkola 2021]. Eles facilitam a escalabilidade, uma vez que cada parte pode ser escalada individualmente, adaptando-se aos requisitos e padrões de uso [Zateishchikov 2023]. Além disso, promovem um código modular e reutilizável,

incentivando a separação de preocupações e tornando a base de código mais fácil de manter, testar e compreender, ainda, de acordo com Zateishchikov (2023), em seu artigo.

2.2.2. Desafios dos Micro Frontends

A interface do usuário deve permanecer consistente, mesmo quando dividida em vários micro frontends [Poloskei e Bub 2021]. Manter a consistência na experiência do usuário (UX) se torna um desafio quando equipes autônomas seguem direções distintas, e é crucial estabelecer um meio comum para preservar a UX. A proliferação de novos frameworks e bibliotecas de desenvolvimento web torna mais difícil alcançar a consistência e interfaces ricas, enquanto a utilização de tecnologias diversas e isolamento aumenta o risco de inconsistências [Peltonen et al. 2021].

A duplicação de código e dependências é uma preocupação, uma vez que cada micro frontend pode usar seu próprio conjunto de bibliotecas e frameworks, como relacionado por Zateishchikov (2023). Conforme Zateishchikov (2023), a independência das equipes leva a desafios na colaboração, resultando em compartimentalização e falta de comunicação entre elas. Além disso, a adoção de uma arquitetura de micro frontends pode exigir que a equipe adquira novos conhecimentos, inicialmente atrasando o desenvolvimento [Zateishchikov 2023].

Os testes também se tornam mais complexos, dada a presença de vários componentes independentes [Zateishchikov 2023]. Integrar os diferentes micro frontends é um desafio, especialmente quando equipes separadas os desenvolvem usando tecnologias diferentes, também especificado por Zateishchikov (2023). Para que os micro frontends funcionem como uma única aplicação web, é necessário estabelecer uma forma consistente de comunicação entre eles. Uma abordagem comum é a utilização de um aplicativo como uma camada de integração, que atua como contêiner para renderizar os micro frontends e facilitar a comunicação entre eles [Poloskei e Bub 2021].

Para aplicações menores ou projetos com uma única equipe pequena, a complexidade de gerenciar os micro frontends pode superar seus benefícios. Portanto, é crucial avaliar as necessidades específicas do projeto antes de adotar essa arquitetura [Zateishchikov 2023].

2.3. Definições de Implementação

Taibi e Mezzalira (2022) propõem em seu artigo, que existem quatro decisões chave que precisam ser tomadas no início do desenvolvimento de um projeto de micro frontends. A primeira delas está relacionada à divisão horizontal ou vertical, como já apresentado anteriormente pela perspectiva de Mezzalira (2021). Taibi e Mezzalira (2022) detalham ainda, que a estrutura horizontal é mais apropriada para páginas estáticas como catálogos e e-commerces, e a vertical serve melhor aplicações mais interativas e com times que possuem experiências em desenvolvimento client-side mais tradicional.

A segunda decisão é onde a aplicação será composta, podendo ser no lado cliente (client-side composition), no lado servidor (server-side composition), ou de “borda”, no limite entre os dois (edge-side composition) [Mezzalira 2021] como apresentado na Figura 2. A definição apresentada por Peltonen *et al*. (2021), diz que no lado do cliente, um

'shell' de aplicativo carrega partes menores, os micro frontends, dentro dele. Essas partes menores devem ter um arquivo JavaScript ou HTML como ponto de entrada, permitindo que o 'shell' do aplicativo adicione dinamicamente elementos visuais no caso de um arquivo HTML, ou inicie a aplicação JavaScript quando o ponto de entrada for um arquivo JavaScript.

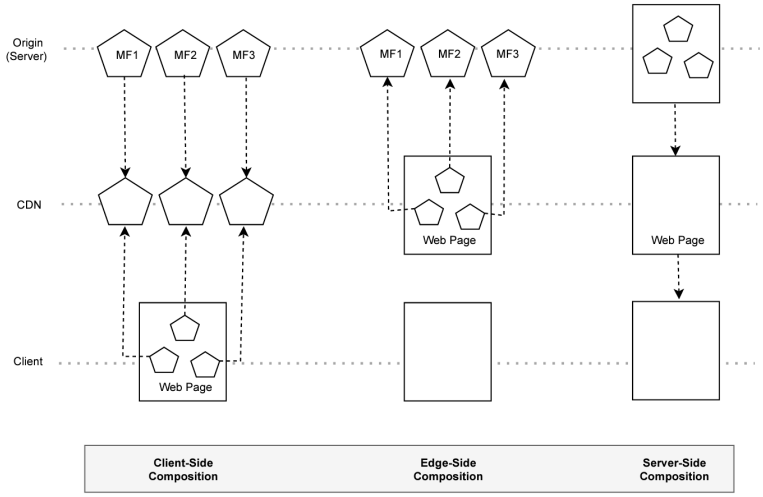

Figura 2. Composição de micro frontends client-side, edge-side e server-side (Taibi e Mezzalira, 2022)

A opção de composição no lado do servidor, pode ser realizada em tempo de execução ou durante a compilação. Nesta abordagem, o servidor de origem compõe a visualização ao reunir todos os vários micro frontends e criando a página final [Mezzalira 2021]. Para Geers (2020), a maior vantagem da integração do lado do servidor é a página chegar completamente montada ao navegador do usuário, permitindo alcançar velocidades de carregamento da primeira página extremamente rápidas, que são difíceis de igualar usando apenas técnicas de integração no lado do cliente.

Na composição no lado do servidor de borda (edge-side composition), é montada a visualização no nível do CDN (Content Delivery Network). Muitos provedores de CDN oferecem a opção de utilizar uma linguagem de marcação baseada em XML chamada Edge Side Includes (ESI). O ESI possibilita a expansão da infraestrutura web para explorar o grande número de pontos de presença ao redor do mundo fornecidos por uma rede de CDN, em comparação com a capacidade limitada dos data centers no qual a maioria do software normalmente é hospedada [Mezzalira 2021].

Segundo Taibi e Mezzalira (2022), as terceira e quarta decisões estão relacionadas ao roteamento e comunicação. Eles exploram que, a decisão sobre o roteamento, seja no lado do cliente, utilizando tecnologias como Lambda@Edge, ou no servidor através do CDN, tem um impacto significativo na forma como as visualizações são direcionadas e compostas. Além disso, a comunicação entre os micro frontends em ambas as estratégias de divisão - vertical e horizontal - requer considerações específicas. Enquanto a divisão horizontal pode adotar um modelo de eventos independentes em cada micro frontend, a divisão vertical demanda uma compreensão minuciosa sobre o compartilhamento de informações. Variáveis, strings de consulta ou o uso de URL para transferir dados entre

micro frontends são considerados, assim como o armazenamento temporário ou permanente por meio do armazenamento web [Taibi e Mezzalira 2022].

Levando em consideração todos esses pontos levantados, existem algumas formas de implementação disponíveis para micro frontends, nenhuma mais correta ou errada que a outra, mas sim a mais apropriada para cada contexto [Taibi e Mezzalira 2022].

A implementação de um Application Shell ocorre no lado cliente da aplicação. É um contêiner para aplicações SPA (Single Page Applications) ou um arquivo HTML que é responsável por carregar arquivos JavaScript ou HTML que representam as entradas dos micro frontends. Essa abordagem é próxima do desenvolvimento frontend tradicional e combina com a solução de divisão vertical da aplicação [Taibi e Mezzalira 2022].

Os iframes permitem o carregamento de diferentes micro frontends criando um ambiente isolado para cada um [Taibi e Mezzalira 2022]. Mezzalira (2021) explica, que um iframe é uma espécie de "janela" incorporada dentro de uma página da web que carrega outro documento HTML e se quisermos mostrar um micro frontend como uma parte independente, separada do resto do aplicativo, os iframes são uma das formas mais fortes de isolamento que podemos ter dentro de um navegador. São mais utilizados na divisão horizontal.

Web Components são um padrão para criar componentes independentes de frameworks. Eles permitem encapsular micro frontends em Custom Elements para compor páginas, escondendo detalhes de implementação [Taibi e Mezzalira 2022]. Quando usamos Web Components para envolver os micro frontends, é crucial evitar expor a lógica fundamental do sistema. Se permitirmos que o contêiner dos micro frontends, encapsulado nos Web Components, personalize seus comportamentos, isso resulta na exposição da lógica central para o mundo exterior. Isso significa que o contêiner do micro frontend acaba conhecendo e dependendo de detalhes específicos de interação com uma API, tudo através de atributos [Mezzalira 2021].

O Module Federation (MF), destaca-se como uma inovação crucial no desenvolvimento frontend, oferecendo uma solução eficaz para os desafios de compartilhamento de código em ambientes distribuídos [Schio 2023]. Este modelo conceitual reconfigura a dinâmica de organização e compartilhamento de código em tempo de execução, introduzindo os protagonistas conhecidos como remotes e hosts. Os remotes exportam módulos identificados, enquanto os hosts importam e integram esses remotes.

Desenvolvido no contexto do Webpack, o principal instrumento de compilação de código, o plugin ModuleFederationPlugin foi concebido por Zackary Jackson e Marais Rossouw, com orientação de Tobias Koppers, criador do Webpack, e foi integrado à versão 5 da ferramenta [Schio 2023]. Este plugin simplifica a importação de código externo e automatiza a gestão de dependências entre hosts e remotes. Atuando na camada mais fundamental do desenvolvimento, o plugin encapsula e limita o escopo das dependências, otimizando o carregamento e evitando redundâncias entre módulos. Proporciona também compartilhamento bidirecional, permitindo que um módulo atue simultaneamente como host e remote. Ao eliminar a necessidade de complexidades adicionais no código-fonte, o plugin Module Federation simplifica consideravelmente a configuração e administração de Micro Frontends, transferindo as responsabilidades de otimização, quebra de código e compilação para o Webpack [Schio 2023]. Essa abordagem simplificada

torna a implementação de projetos com micro frontends mais acessível e descomplicada, seja ao criar novos projetos ou adaptar projetos existentes.

A grande vantagem do Module Federation está na simplicidade de expor diferentes micro frontends, ou mesmo bibliotecas compartilhadas, como um design sytem, permitindo uma integração assíncrona sem complicações. A experiência do desenvolvedor é extremamente fluida, é possível importar micro frontends remotos e compor uma visualização da maneira desejada [Mezzalira 2021].

3. Trabalhos Relacionados

As informações teóricas sobre implementação dos micro frontends tem como principal referência o livro "Building Micro-Frontends" de Luca Mezzalira [Mezzalira 2021], um dos principais entusiastas dessa arquitetura atualmente. A parte prática tem influência da obra de Michael Geers, "Micro Frontends in Action" [Geers 2020], que traz vários exemplos e instruções de como se construir um software utilizando-se dessa nova abordagem.

Os artigos "Scaling a Software Platform Using Micro Frontends" de Kirill Zateishchikov, (2023) e "Design and Implementation of Modular Frontend Architecture on Existing Application" de Lasse Linkola (2021) também podem ser relacionados a este estudo. A metodologia desses artigos consiste na avaliação da variedade de cenários possíveis atualmente para a implementação da arquitetura de micro frontends, e a identificação da qual se adequa mais à aplicação a ser remodelada, se observando os ganhos com essa nova estrutura e os desafios até atingir este objetivo. Zateishchikov utiliza a implementação de Web Components e Linkola utiliza o framework Single-SPA, uma abordagem de application "shell" para a construção de seus micro frontends.

Outro trabalho que serviu de referência foi "Integração de Módulos Utilizando Micro Frontends na Plataforma +Precoce", de Alan Balen Schio (2023). Em sua dissertação, o autor realiza a integração de módulos de um sistema implementando a arquitetura de micro frontends com Module Federation.

Neste trabalho, optou-se pela abordagem de Module Federation devido às vantagens evidenciadas nas pesquisas. Mezzalira (2021), por exemplo, ressaltou a experiência de desenvolvimento (DX) concebida pela equipe criadora do ModuleFederationPlugin, que conseguiu abstrair a complexidade, proporcionando uma experiência contínua. Além disso, Schio (2023) destacou a facilidade de implementação tanto para a criação de novos projetos com essa tecnologia quanto para a adaptação de projetos existentes. A escolha dessa implementação neste projeto visa contribuir também, para o estudo dessa tecnologia, apresentando um caso prático de implementação de micro-frontends neste contexto.

4. Materiais e Métodos

Antes de abordar o processo de desenvolvimento da aplicação monolítica e a subsequente transição para a abordagem de micro frontends, é importante contextualizar o trabalho realizado por meio de um levantamento detalhado da literatura. Este levantamento visou identificar as maiores vantagens e desafios relacionados à implementação de micro frontends em comparação com abordagens tradicionais.

A revisão bibliográfica abrangeu fontes como artigos acadêmicos, livros e dissertações de mestrado. A análise focou-se em compreender as experiências e visões

compartilhadas por especialistas no campo, destacando as principais contribuições dessa nova arquitetura. A identificação das vantagens buscou oferecer uma visão abrangente das melhorias potenciais em termos de modularidade, manutenibilidade e escalabilidade.

Simultaneamente, o levantamento dedicou-se a mapear os desafios associados à implementação de micro frontends, incluindo questões relacionadas à integração, comunicação entre os módulos, e possíveis impactos na experiência do usuário. Este embasamento teórico não apenas fundamentou a escolha pela abordagem de micro frontends, mas também forneceu uma base sólida para avaliar de que maneira as vantagens e desafios identificados se manifestaram durante a implementação prática do projeto.

Após esse levantamento, o processo de desenvolvimento teve início com a criação de uma aplicação do tipo *Single-Page-Application* (SPA) utilizando o framework React em conjunto com JavaScript. A fase inicial envolveu a definição da arquitetura básica da SPA, incluindo a estruturação de arquivos principais e a criação de um diretório dedicado para os componentes. Durante essa etapa, foram implementadas as funcionalidades essenciais da aplicação para atender aos requisitos iniciais que consistiam em simular uma aplicação com módulos representativos de diversas partes de um negócio.

Considerando que cada módulo criado, referente a uma parte do negócio pudesse crescer em tamanho e a aplicação pudesse estar preparada para um cenário onde diferentes equipes pudessem trabalhar simultaneamente no projeto sem problemas, o próximo passo consistiu em ajustar o código monolítico para incorporar a abordagem de micro frontends. Esse processo envolveu uma reestruturação, visando promover maior modularidade e flexibilidade ao sistema. A transição para micro frontends permitiu a criação de unidades mais autônomas e facilmente gerenciáveis, contribuindo para uma arquitetura mais escalável e adaptável às demandas futuras do projeto.

4.1. A aplicação monolítica

O aplicativo[1] monolítico criado para este estudo, simula uma plataforma de cursos online na área de tecnologia, abrangendo desde a apresentação do catálogo de cursos, opções de planos de assinatura, autenticação até uma área reservada para membros.

O arquivo “App.js” desempenha o papel de ponto de entrada na aplicação, gerenciando a navegação e a estrutura principal. Os componentes do diretório “components“ têm funções específicas na interface. Por exemplo, o “Header.js” exibe o cabeçalho da página, enquanto “Home.js”, “MemberArea.js”, “Pricing.js”, “Signin.js” e “Signup.js” representam diferentes seções da aplicação.

Cada componente possui uma função distintiva na interface. O “Header.js” oferece navegação para o início e um botão para autenticação. “Home.js” exibe a página inicial com a lista de cursos disponíveis. “MemberArea.js” permite acesso à área restrita para membros, “Pricing.js” apresenta informações sobre planos e preços, e os componentes “Signin.js” e “Signup.js” cuidam do processo de autenticação do usuário.

Há uma dependência entre o “App.js” e os demais componentes, pois o “App.js” integra esses componentes para criar a interface completa da aplicação. Além disso, alguns componentes, como “Signin.js” e “Signup.js”, compartilham funcionalidades de autenticação.

[1] https://github.com/guilorentz/learning-platform-spa

Ao analisar a estrutura dos componentes, percebe-se que a divisão em micro frontends se apresenta como uma estratégia viável. A separação desses elementos pode contribuir significativamente para a modularização e manutenção do sistema, permitindo que cada microfrontend seja gerenciado por equipes independentes, agilizando o desenvolvimento e facilitando a manutenção.

Após a análise realizada, identificou-se que o "Header.js" pode ser um candidato ideal para ser isolado como um micro frontend independente, principalmente devido à sua presença consistente e natureza estática em todas as páginas. Isso poderia ser integrado em um contêiner que orquestraria as sub-aplicações.

Os componentes foram construídos de forma a manter um nível moderado de acoplamento, simulando uma aplicação com domínios bem definidos. A forte ligação entre "Signin.js" e "Signup.js" sugere a viabilidade de integrá-los em um único micro frontend.

O componente "Home.js" possui um botão que redireciona para a página exibida pelo componente "Pricing.js". Esses dois componentes têm similaridades em termos de domínio, sugerindo a possibilidade de serem agrupados em uma sub-aplicação. Além disso, a área de membros pode ser a base para outro grande domínio da aplicação, tornando-se mais um micro frontend.

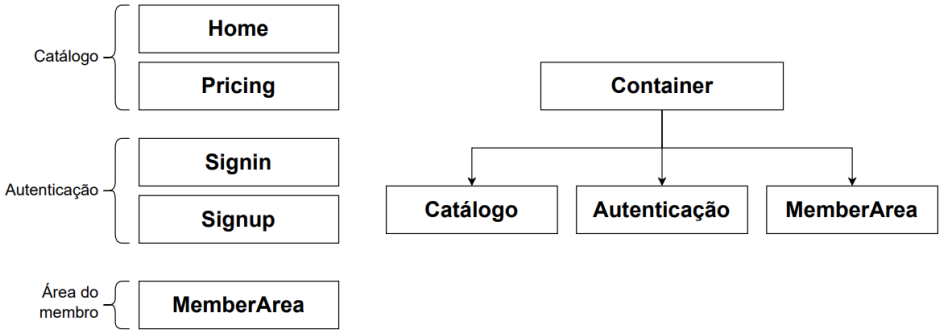

Figura 3. Diagrama de divisão da aplicação em micro frontends

4.2. Implementação de micro frontends com Module Federation

Após a divisão da aplicação monolítica em módulos de negócio como delineada no diagrama da Figura 3, foi criada uma estrutura de monorepo[2], que consiste em um único repositório que abriga diversos projetos. Essa estrutura é composta por um contêiner, atuando como hospedeiro, e três subaplicações que funcionam como módulos remotos para integrar-se em um projeto de micro frontends.

[2]https://github.com/guilorentz/learning-platform-mfe

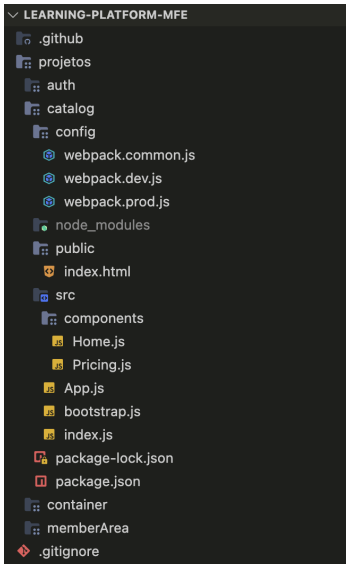

Figura 4. Estrutura do projeto de micro frontends com destaque para o projeto "catalog" (remote)

Dentro do contexto do projeto "catalog" e dos outros micro frontends que atuam como remotes, foi necessário inicialmente realizar a configuração dos arquivos webpack para se poder gerar a saída da aplicação. Foi criado um diretório "config" com as configurações para o ambiente de desenvolvimento (webpack.dev.js) e o ambiente de produção (webpack.prod.js), além de um arquivo "webpack.common.js", que serve para concentrar as configurações comuns aos dois ambientes, como visto na estrutura de diretórios da Figura 4.

No arquivo "webpack.common.js" , definem-se várias regras para o processamento de módulos. Notavelmente, estabelece-se a restrição para processar apenas arquivos JavaScript e exclui aqueles localizados no diretório "node_modules". Uma regra crucial é a configuração do Babel Loader, que desempenha a função de transpilar código JavaScript moderno (ES6+) para uma versão compatível com a maioria dos navegadores. Este loader opera em conjunto com presets específicos, como "@babel/preset-react", destinado a abordar a sintaxe JSX do React, e "@babel/preset-env", utilizado para compilar o código para uma versão do ECMAScript amplamente suportada pelos navegadores.

Além disso, incorpora-se o plugin Babel denominado "@babel/plugin-transform-runtime". Este plugin desempenha um papel crucial na redução de duplicação de código durante as compilações, movendo funções auxiliares para um módulo compartilhado. Essa estratégia contribui para a eficiência e otimização do código gerado, proporcionando um desempenho mais eficaz da aplicação resultante.

No código 1, "webpack.dev.js" tem-se as especificações para o ambiente de desenvolvimento (mode: 'development'), esse código é parte integrante de um sistema de construção (build system) para uma aplicação web.

```
1 const { merge } = require('webpack-merge');
2 const HtmlWebpackPlugin = require('html-webpack-plugin');
3 const ModuleFederationPlugin = require('webpack/lib/container/
    ModuleFederationPlugin');
```

```
const commonConfig = require('./webpack.common');
const packageJson = require('../package.json');

const devConfig = {
  mode: 'development',
  output: {
    publicPath: 'http://localhost:8081/',
  },
  devServer: {
    port: 8081,
    historyApiFallback: {
      index: '/index.html',
    },
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'catalog',
      filename: 'remoteEntry.js',
      exposes: {
        './CatalogApp': './src/bootstrap',
      },
      shared: packageJson.dependencies,
    }),
    new HtmlWebpackPlugin({
      template: './public/index.html',
    }),
  ],
};

module.exports = merge(commonConfig, devConfig);
```

Código 1. “webpack.dev.js” do módulo “catalog”

A configuração começa com a importação de diferentes módulos e plugins necessários para a construção do projeto. Notavelmente, o webpack-merge é utilizado para mesclar a configuração comum (commonConfig) com as especificidades do ambiente de desenvolvimento (devConfig). Isso permite compartilhar configurações entre diferentes ambientes de construção.

A definição do output especifica o caminho público para os arquivos no servidor de desenvolvimento, facilitando o carregamento correto dos ativos no navegador. O servidor de desenvolvimento (devServer) é configurado para executar na porta 8081, e uma estratégia de fallback é implementada para redirecionar solicitações desconhecidas para o arquivo index.html, garantindo que a aplicação funcione corretamente mesmo com URLs complexas.

O ModuleFederationPlugin expõe um módulo chamado CatalogApp a partir do arquivo ./src/bootstrap e compartilha as dependências do projeto, definidas no arquivo package.json. Isso significa que outros módulos na aplicação podem consumir o módulo CatalogApp e terão acesso às suas dependências compartilhadas.

O HtmlWebpackPlugin é outro plugin essencial que simplifica a geração de arquivos HTML para a aplicação, usando um modelo localizado em ./public/index.html.

Por fim, a exportação do módulo combina a configuração comum com as especifi-

cidades do ambiente de desenvolvimento, proporcionando uma configuração abrangente para o webpack.

```
const { merge } = require('webpack-merge');
const ModuleFederationPlugin = require('webpack/lib/container/
   ModuleFederationPlugin');
const commonConfig = require('./webpack.common');
const packageJson = require('../package.json');

const prodConfig = {
  mode: 'production',
  output: {
    filename: '[name].[contenthash].js',
    publicPath: '/catalog/latest/',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'catalog',
      filename: 'remoteEntry.js',
      exposes: {
        './CatalogApp': './src/bootstrap',
      },
      shared: packageJson.dependencies,
    }),
  ],
};

module.exports = merge(commonConfig, prodConfig);
```

Código 2. “webpack.prod.js” do módulo “catalog”

No código 2, o foco está na configuração específica para o ambiente de produção da aplicação. Assim como anteriormente, o código é parte integrante do processo de construção (bundling) do projeto, utilizando o webpack.

Primeiramente, são importados diferentes módulos e plugins essenciais para a configuração do webpack. O webpack-merge continua desempenhando um papel crucial, permitindo mesclar a configuração comum (commonConfig) com a configuração específica para o ambiente de produção (prodConfig). Esta abordagem é valiosa, pois facilita a consistência entre diferentes ambientes de construção, mantendo um único ponto de definição para configurações compartilhadas.

A definição do output neste contexto específico é ajustada para atender às necessidades do ambiente de produção. O nome do arquivo de saída agora incorpora um hash de conteúdo ([contenthash]), uma prática comum para lidar com o cache do navegador e garantir que os usuários recebam sempre a versão mais recente do código. Além disso, o publicPath é configurado para /catalog/latest/, indicando o caminho público para os arquivos no servidor de produção.

Dentro da seção de plugins, o ModuleFederationPlugin mantém sua presença, desempenhando o mesmo papel fundamental que desempenhou no ambiente de desenvolvimento. Ele continua a expor o módulo CatalogApp a partir do arquivo ./src/bootstrap e compartilhar as dependências do projeto. Essa abordagem assegura uma integração consistente entre diferentes partes da aplicação, permitindo que módulos consumam o

CatalogApp e acessem suas dependências compartilhadas.

No diretório “src”, estão os arquivos “index.js”, “bootstrap.js” e “App.js” e a pasta “components”, com os componentes “Home.js” e “Pricing.js” que foram extraídos da aplicação monolítica para formar o micro frontend “catalog”.

No arquivo index.js, a operação principal consiste na importação do módulo bootstrap de forma dinâmica. Essa importação assíncrona ocorre por meio do comando import(’./bootstrap’), indicando que a execução do código contido em bootstrap ocorrerá apenas quando necessário.

Por sua vez, o arquivo bootstrap.js (código 3) é fundamental para a inicialização da aplicação. Ele inicia importando as dependências necessárias, como React, ReactDOM, history e o componente App do próprio diretório. Em seguida, a função mount é definida, responsável por montar a aplicação no elemento do DOM especificado, além de configurar e manipular a navegação.

A função mount aceita diversos parâmetros, incluindo o elemento DOM no qual a aplicação será montada (el), uma função de callback para lidar com eventos de navegação (onNavigate), o histórico de navegação padrão (defaultHistory), e o caminho inicial (initialPath). Internamente, ela cria um histórico de navegação utilizando a biblioteca history, renderiza o componente App na raiz da aplicação e retorna um objeto contendo uma função onParentNavigate para sincronizar a navegação com o contêiner pai.

A última parte do arquivo contém uma verificação condicional para ambiente de desenvolvimento. Se o código estiver sendo executado nesse contexto, procura-se um elemento do DOM com o id “_catalog-dev-root” e, se encontrado, monta a aplicação no mesmo utilizando o histórico de navegação do navegador.

Por fim, a função mount é exportada, possibilitando que outros módulos ou arquivos possam utilizar essa função para montar a aplicação em diferentes contextos. Essa estrutura modular e a capacidade de montar a aplicação de maneira independente são elementos essenciais para a construção de micro frontends, proporcionando flexibilidade e reusabilidade no desenvolvimento da aplicação. Código do arquivo “bootstrap.js”:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createMemoryHistory, createBrowserHistory } from 'history';
import App from './App';

const mount = (el, { onNavigate, defaultHistory, initialPath }) => {
  const history =
    defaultHistory ||
    createMemoryHistory({
      initialEntries: [initialPath],
    });

  if (onNavigate) {
    history.listen(onNavigate);
  }

  ReactDOM.render(<App history={history} />, el);

  return {
```

```
    onParentNavigate({ pathname: nextPathname }) {
      const { pathname } = history.location;

      if (pathname !== nextPathname) {
        history.push(nextPathname);
      }
    },
  };
};

if (process.env.NODE_ENV === 'development') {
  const devRoot = document.querySelector('#_catalog-dev-root');

  if (devRoot) {
    mount(devRoot, { defaultHistory: createBrowserHistory() });
  }
}

export { mount };
```

Código 3. “bootstrap.js” do módulo “catalog”

O código de “App.js” apresenta um componente React que desempenha um papel central na configuração do roteamento e navegação da aplicação, utilizando a biblioteca react-router-dom. Além disso, integra estilos com Material-UI por meio do `<StylesProvider>` e da função createGenerateClassName. O componente renderiza dois caminhos distintos com base nas rotas definidas: “/pricing”, que exibe o componente Pricing, e a rota padrão (“/”), que exibe o componente Home. Essa configuração é essencial para garantir a correta apresentação dos componentes associados a cada rota na aplicação.

Os projetos “auth” e “memberArea”, que também desempenham papéis como “remotes” neste sistema, seguem uma configuração semelhante à do projeto “catalog”. As particularidades das rotas expostas nos ambientes de desenvolvimento e produção, bem como as configurações específicas do output e do ModuleFederationPlugin, são levadas em consideração.

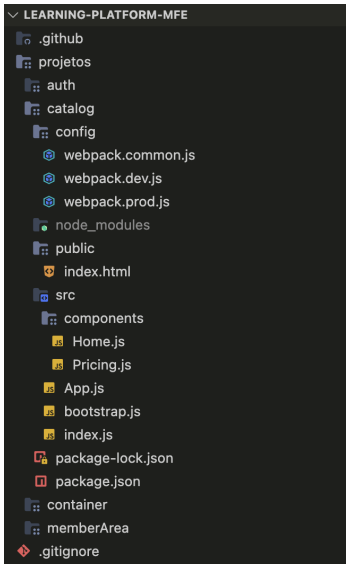

Figura 5. Estrutura do projeto de micro frontends com destaque para o projeto "container" (host)

As definições do "container" (host) possuem uma estrutura semelhante ao do "catalog" (Figura 5) com algumas diferenças importantes a nível de código (código 4). A primeira diferença reside na configuração do servidor de desenvolvimento (devServer), onde a porta foi ajustada para 8080, além disso, o caminho público (publicPath) de saída foi modificado para 'http://localhost:8080/'. A maior diferença está na configuração do Plugin ModuleFederation, onde este arquivo atua como o container principal para carregar módulos remotos dos demais aplicativos, que estão sendo expostos isoladamente em seus respectivos projetos. Neste plugin, os nomes dos remotos a serem carregados devem coincidir com os nomes gerados em cada subaplicação.

```
const { merge } = require('webpack-merge');
const ModuleFederationPlugin = require('webpack/lib/container/
   ModuleFederationPlugin');
const commonConfig = require('./webpack.common');
const packageJson = require('../package.json');

const devConfig = {
  mode: 'development',
  output: {
    publicPath: 'http://localhost:8080/',
  },
  devServer: {
    port: 8080,
    historyApiFallback: {
      index: '/index.html',
    },
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'container',
      remotes: {
        catalog: 'catalog@http://localhost:8081/remoteEntry.js',
        auth: 'auth@http://localhost:8082/remoteEntry.js',
```

```
        memberArea: 'memberArea@http://localhost:8083/remoteEntry.js',
      },
      shared: packageJson.dependencies,
    }),
  ],
};

module.exports = merge(commonConfig, devConfig);
```

Código 4. “webpack.dev.js” do módulo “container”

No ambiente de produção para o host, como mostrado no código 5, uma característica distintiva é a introdução da variável “domain”, destinada a extrair o domínio de produção a partir da variável de ambiente “PRODUCTION_DOMAIN”. No que diz respeito à configuração de saída (output), assim como na definição do aplciativo “catalog” apresentado, destaca-se a definição do nome do arquivo como ’[name].[contenthash].js’. Essa escolha não apenas garante a unicidade dos nomes de arquivo, mas também incorpora hashes de conteúdo para evitar problemas de cache, contribuindo assim para a eficiente gestão de versões.

No contexto do plugin ModuleFederationPlugin, observa-se a configuração do nome do módulo de federação como ’container’. A seção ’remotes’ desempenha um papel crucial ao especificar os módulos remotos - catalog, auth e memberArea - juntamente com os caminhos dos pontos de entrada remotos. Esses caminhos são dinamicamente formados com base no domínio de produção e os caminhos específicos para cada módulo remoto.

```
const { merge } = require('webpack-merge');
const ModuleFederationPlugin = require('webpack/lib/container/
    ModuleFederationPlugin');
const commonConfig = require('./webpack.common');
const packageJson = require('../package.json');

const domain = process.env.PRODUCTION_DOMAIN;

const prodConfig = {
  mode: 'production',
  output: {
    filename: '[name].[contenthash].js',
    publicPath: '/container/latest/',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'container',
      remotes: {
        catalog: `catalog@${domain}/catalog/latest/remoteEntry.js`,
        auth: `auth@${domain}/auth/latest/remoteEntry.js`,
        memberArea: `memberArea@${domain}/memberArea/latest/remoteEntry
    .js`,
      },
      shared: packageJson.dependencies,
    }),
  ],
};

```

```
27 module.exports = merge(commonConfig, prodConfig);
```

Código 5. “webpack.prod.js” do módulo “container”

O arquivo “index.js” atua como ponto de entrada para a renderização e inicialização do aplicativo, a função import(’./bootstrap’) é utilizada para carregar dinamicamente o módulo de inicialização “bootstrap.js”. O “bootstrap.js” importa as bibliotecas essenciais React e ReactDOM, além do componente principal do aplicativo (App.js). Em seguida, utiliza o ReactDOM.render para renderizar o componente App no elemento com o id ’root’. Este é um padrão comum em aplicativos React, onde a árvore de componentes é anexada ao elemento raiz na página HTML.

O componente “App.js” do “container” (código 6) começa importando diversas bibliotecas e módulos necessários, incluindo React, componentes específicos do “react-router-dom” para roteamento, estilos do Material-UI, e utilitários para geração de estilos únicos e manipulação de histórico de navegação.

Em seguida, o componente define três componentes React usando a função lazy para carregamento dinâmico. Esses componentes (CatalogLazy, AuthLazy, MemberAreaLazy) são importados dinamicamente usando a função import e representam os aplicativos que estão vindo dos demais micro frontends: catálogo, autenticação e área do membro. O Lazy Loading garante uma melhora no desempenho inicial, a otimização no tempo de carregamento e a redução no tamanho do pacote inicial. Esse componente é de suma importância, pois nele são definidos como os componentes remotes serão apresentados na aplicação completa.

A variável generateClassName é configurada para criar nomes de classe únicos, especialmente útil em ambientes de produção onde evitar conflitos de estilos é crucial. O histórico de navegação é configurado utilizando createBrowserHistory da biblioteca history. Este histórico será utilizado pelo componente Router para gerenciar a navegação do aplicativo. Dentro do componente funcional principal, um estado isSignedIn é gerenciado via useState para rastrear o status de autenticação do usuário. Um efeito colateral (useEffect) é utilizado para redirecionar o usuário para a página de área de membros quando ele se autentica.

Finalmente, o componente retorna a estrutura do aplicativo. Ele inclui o componente Router para o roteamento, StylesProvider para encapsular estilos, um componente de cabeçalho (Header) que reage à autenticação do usuário, e um componente Suspense para lidar com o carregamento assíncrono dos componentes. O componente Switch define as rotas do aplicativo, e o carregamento dinâmico (lazy) é tratado com o componente Suspense, exibindo um indicador de progresso (Progress) enquanto os componentes estão sendo carregados.

```
1 import React, { lazy, Suspense, useState, useEffect } from 'react';
2 import { Router, Route, Switch, Redirect } from 'react-router-dom';
3 import {
4   StylesProvider,
5   createGenerateClassName,
6 } from '@material-ui/core/styles';
7 import { createBrowserHistory } from 'history';
8
9 import Progress from './components/Progress';
```

```
import Header from './components/Header';

const CatalogLazy = lazy(() => import('./components/CatalogApp'));
const AuthLazy = lazy(() => import('./components/AuthApp'));
const MemberAreaLazy = lazy(() => import('./components/MemberAreaApp'))
    ;

const generateClassName = createGenerateClassName({
  productionPrefix: 'co',
});

const history = createBrowserHistory();

export default () => {
  const [isSignedIn, setIsSignedIn] = useState(false);

  useEffect(() => {
    if (isSignedIn) {
      history.push('/memberArea');
    }
  }, [isSignedIn]);
  return (
    <Router history={history}>
      <StylesProvider generateClassName={generateClassName}>
        <div>
          <Header
            onSignOut={() => setIsSignedIn(false)}
            isSignedIn={isSignedIn}
          />
          <Suspense fallback={<Progress />}>
            <Switch>
              <Route path="/auth">
                <AuthLazy onSignIn={() => setIsSignedIn(true)} />
              </Route>
              <Route path="/memberArea">
                {!isSignedIn && <Redirect to="/" />}
                <MemberAreaLazy />
              </Route>
              <Route path="/" component={CatalogLazy} />
            </Switch>
          </Suspense>
        </div>
      </StylesProvider>
    </Router>
  );
};
```

Código 6. “App.js” do módulo “container”

Foi construído para cada aplicação um código de fluxo de trabalho (workflow) para GitHub Actions, como mostrado no código 7, destinado a automatizar o processo de implantação dos aplicativos, no exemplo tem-se o workflow para a implantar o container.

```
name: deploy-container

on:
  push:
```

```
    branches:
      - main
    paths:
      - 'projetos/container/**'

defaults:
  run:
    working-directory: projetos/container

jobs:
  build:
    runs-on: macos-latest

    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm run build
        env:
          PRODUCTION_DOMAIN: ${{ secrets.PRODUCTION_DOMAIN }}

      - uses: shinyinc/action-aws-cli@v1.2
      - run: aws s3 sync dist s3://${{ secrets.AWS_S3_BUCKET_NAME }}/container/latest
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_DEFAULT_REGION: us-east-1

      - run: aws cloudfront create-invalidation --distribution-id ${{ secrets.AWS_DISTRIBUTION_ID }} --paths "/container/latest/index.html"
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          AWS_DEFAULT_REGION: us-east-1
```

Código 7. “container.yml”

O início do código define o nome do fluxo de trabalho como “deploy-container” e especifica os eventos que desencadearão a execução do workflow. Neste caso, a execução é acionada quando ocorre um “push” (envio) para o branch “main” e se as alterações ocorrem nos arquivos dentro do diretório ’projetos/container/**’.

Em seguida, são definidos alguns valores padrão para as execuções do workflow, como o diretório de trabalho (working-directory), que é configurado para ’projetos/container’. O workflow é composto por uma série de jobs (trabalhos). Neste caso, há um único job chamado “build” que será executado em uma máquina macOS. O job “build” consiste em várias etapas (steps). Primeiramente, ocorre o checkout do código-fonte usando a ação actions/checkout@v2. Em seguida, são realizados comandos npm para instalar as dependências e construir o aplicativo.

Posteriormente, são utilizadas as ações fornecidas pela shinyinc/action-aws-cli@v1.2 para interagir com a AWS (Amazon Web Services). Especificamente, os arquivos da pasta ’dist’ (resultado da construção do aplicativo) são sincronizados com um bucket S3 da AWS. Por fim, há uma ação para criar uma invalidação no CloudFront,

garantindo que as alterações sejam refletidas imediatamente.

Em todas as etapas, variáveis de ambiente são utilizadas para armazenar informações sensíveis, como chaves de acesso da AWS e domínio de produção, as quais são configuradas como "secrets" no repositório GitHub, garantindo uma prática segura de gestão de credenciais. Este workflow automatizado visa simplificar e acelerar o processo de implantação do aplicativo, integrando o desenvolvimento contínuo e a entrega contínua (CI/CD) com a infraestrutura da AWS.

5. Resultados

A revisão dos artigos proporcionou uma compreensão abrangente dos benefícios e desafios associados à escolha de implementar uma aplicação utilizando a arquitetura de micro frontends. A partir dessas análises, foi possível realizar uma avaliação das experiências que surgiram durante a fase de implementação deste trabalho.

As tabelas a seguir apresentam os diversos aspectos discutidos nos artigos, destacando quais são esses desafios e benefícios específicos que foram abordados de maneira prática na implementação deste projeto. A tabela 1 apresenta as vantagens e a tabela 2 mostra as desvantagens.

Aspecto	**Artigos cujo aspecto foi citado**
Times independentes	Peltonen et al (2021), Nascimento e Sotto (2020), Schio (2023), Pavlenko et al (2020), Linkola (2021), Zateishchikov (2023), Taibi e Mezzalira (2022)
Escalabilidade	Peltonen et al (2021), Manisto et al (2023), Zateishchikov (2023), Poloskei e Bub (2021)
Manutenibilidade	Peltonen et al (2021), Nascimento e Sotto (2020), Schio (2023), Linkola (2021), Zateishchikov (2023)
Independência Tecnológica	Peltonen et al (2021), Nascimento e Sotto (2020), Schio (2023), Manisto et al (2023), Pavlenko et al (2020), Poloskei e Bub (2021)
Implantação Independente	Peltonen et al (2021), Linkola (2021)

Tabela 1. Vantagens

Um dos aspectos mais amplamente reconhecidos como positivo na arquitetura de micro frontends é a capacidade de equipes trabalharem de forma independente em cada microfrontend. Cada microfrontend está encapsulado em sua própria pasta no repositório, possuindo uma configuração de ambiente de desenvolvimento que permite o carregamento isolado. Isso significa que cada projeto pode ser tratado como uma entidade independente, permitindo que diferentes equipes trabalhem simultaneamente em diferentes partes do sistema sem afetar as demais. Segundo a literatur, essa abordagem promove uma colaboração eficiente, onde o desenvolvimento em paralelo pode ocorrer de maneira harmoniosa, refletindo a flexibilidade e modularidade inerentes à arquitetura de micro frontends.

Para a manutenibilidade do código, foi essencial o fator da modularidade dos micro frontends. Atualizações e correções pontuais puderam ser implementadas em módulos

específicos, sem impactar outras partes da aplicação. De acordo com os artigos, isso resulta em um processo de manutenção mais ágil e menos propenso a introduzir novos problemas. Um exemplo disso é a capacidade de se realizar essas alterações nas aplicações "remotes" sem impactar o "host".

A abordagem de micro frontends facilitou a implantação independente de cada módulo. Usando ferramentas como Webpack e GitHub Actions, foi possível automatizar o processo de integração contínua, garantindo deploys rápidos e confiáveis para cada parte da aplicação.

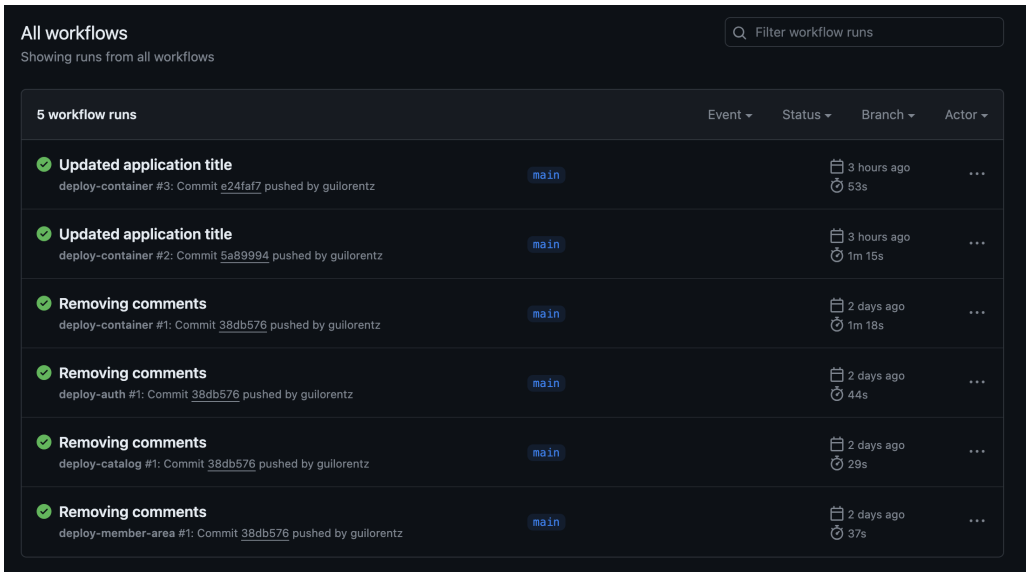

Figura 6. Workflows do Github Actions

O escopo limitado do projeto não permitiu a simulação de cenários de carga significativos para avaliar completamente a escalabilidade da aplicação em ambientes de produção. A escalabilidade é um aspecto que geralmente é mais bem avaliado em ambientes mais complexos e com uma carga de usuários mais significativa.

A independência de tecnologias entre os micro frontends é um princípio central na arquitetura, permitindo que diferentes equipes usem tecnologias e frameworks que melhor atendam às suas necessidades específicas. No entanto, devido à simplicidade do projeto, não foram exploradas amplamente as possibilidades de implementação com diversas tecnologias.

Aspecto	Artigos cujo aspecto foi citado
Aumento da Complexidade	Peltonen et al (2021), Nascimento e Sotto (2020), Zateishchikov (2023), Poloskei e Bub (2021)
Consistência de interface	Peltonen et al (2021), Nascimento e Sotto (2020), Schio (2023), Manisto et al (2023), Pavlenko et al (2020), Linkola (2021)
Compartilhamento de Dependências	Peltonen et al (2021), Schio (2023), Manisto et al (2023), Pavlenko et al (2020), Zateishchikov (2023)
Duplicação de Código	Peltonen et al (2021), Schio (2023), Manisto et al (2023), Zateishchikov (2023)
Aumento do Tamanho do Payload	Peltonen et al (2021), Nascimento e Sotto (2020), Manisto et al (2023)

Tabela 2. Desafios

Apesar dos benefícios da arquitetura de micro frontends, optou-se por uma implementação inicial mais simples para mitigar o aumento da complexidade. A escolha do Module Federation, além de ser uma solução bem documentada e com suporte ativo da comunidade, permitiu estabelecer uma governança eficiente nas interfaces de comunicação entre micro frontends. Essa abordagem estratégica visou facilitar a compreensão dos fundamentos antes de lidar com cenários mais complexos, proporcionando uma base sólida para futuras expansões.

A manutenção da consistência da interface na arquitetura de micro frontends apresentou um desafio significativo. Optou-se por implementar um sistema de design compartilhado e diretrizes rigorosas de estilo usando o Material-UI. Durante o ambiente de desenvolvimento, observou-se a criação de nomes de classes mais descritivos e extensos, o que evitava conflitos entre diferentes micro frontends. No entanto, ao migrar para o ambiente de produção, foi reparada a prática comum de gerar nomes de classes mais curtos, como “jss-1”, “jss-2” em bibliotecas CSS-in-JS como é o Material-UI. Essa abordagem, embora otimizada para desempenho, resultou em conflitos entre elementos de micro frontends diferentes, todos compartilhando nomes de classes curtos, apesar de suas propriedades distintas. Para mitigar esse problema, foi implementada uma solução pragmática utilizando o generateClassName com um prefixo específico para cada projeto, como apresentado no código 8.

```
import React from 'react';
import { Switch, Route, Router } from 'react-router-dom';
import {
  StylesProvider,
  createGenerateClassName,
} from '@material-ui/core/styles';

import Home from './components/Home';
import Pricing from './components/Pricing';

const generateClassName = createGenerateClassName({
  productionPrefix: 'ca',
});
```

```
14
15 export default ({ history }) => {
16   return (
17     <div>
18       <StylesProvider generateClassName={generateClassName}>
19         <Router history={history}>
20           <Switch>
21             <Route exact path="/pricing" component={Pricing} />
22             <Route path="/" component={Home} />
23           </Switch>
24         </Router>
25       </StylesProvider>
26     </div>
27   );
28 };
```

Código 8. "App.js" do módulo "catalog"

Essa medida garantiu que os nomes de classes gerados fossem únicos para cada micro frontend como mostra a Figura 7, evitando conflitos. Apesar dos nomes de classes mais curtos no ambiente de produção, a introdução de prefixos específicos proporcionou unicidade, contribuindo para a manutenção da integridade do estilo em cada parte da aplicação de micro frontends.

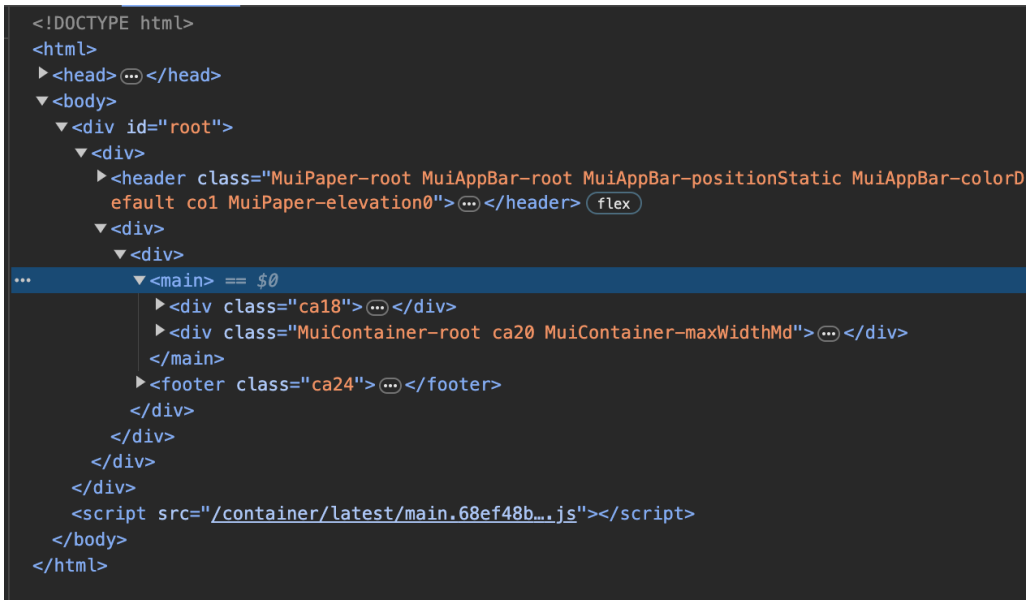

Figura 7. Exemplo da aplicação de diferentes prefixos observado no Chrome Dev Tools no ambiente de produção. Prefixo "co" para classes do container, e "ca" para classes do "catalog"

Um desafio crítico que surgiu foi a necessidade de compartilhar dependências entre os micro frontends. A utilização do Module Federation com Webpack permitiu um compartilhamento eficiente de dependências entre os diferentes módulos da aplicação. O Module Federation possibilita a inclusão de dependências compartilhadas de maneira dinâmica durante o tempo de execução. Essa prática reduziu também a duplicação de

código, proporcionando uma arquitetura mais eficiente e evitando a redundância de bibliotecas comuns em cada microfrontend.

Além disso, ao compartilhar dependências de forma dinâmica, conseguiu-se minimizar conflitos de versões entre os módulos. Isso se revelou crucial para manter a consistência e a estabilidade do sistema, uma vez que cada microfrontend podia consumir as versões necessárias das dependências, sem interferir nas outras partes da aplicação.

Para mitigar o aumento do tamanho do payload, foram adotadas práticas de otimização no empacotamento de módulos. Uma estratégia-chave foi a implementação de lazy loading na aplicação container. Ao empregar o lazy loading, foi possível o carregamento assíncrono dos micro frontends à medida que eram requisitados.

Essa abordagem permitiu evitar a carga desnecessária de módulos completos no carregamento inicial da aplicação. Ao carregar os micro frontends apenas quando necessários, pôde-se reduzir significativamente o tamanho do payload, otimizando o uso de largura de banda e melhorando o desempenho geral da aplicação, resultando em uma experiência de usuário mais eficiente.

6. Conclusão

Ao explorar a transição de uma aplicação frontend monolítica para a arquitetura de micro frontends, este estudo buscou enfrentar os desafios contemporâneos associados ao desenvolvimento ágil de aplicações web. A revisão aprofundada dos artigos proporcionou uma visão ampla das vantagens e desafios dos micro frontends, e a implementação prática desse paradigma em uma aplicação existente ofereceu insights valiosos.

Os benefícios observados, como a capacidade de times trabalharem independentemente, manutenibilidade, e implantação independente, foram tangibilizados na aplicação desenvolvida. A modularidade inerente aos micro frontends permitiu simular um cenário em que diferentes equipes pudessem colaborar de maneira eficiente, destacando a flexibilidade dessa abordagem para o desenvolvimento ágil.

A manutenção da consistência de interface revelou-se um desafio, especialmente ao lidar com bibliotecas CSS-in-JS como o Material-UI. A implementação de um sistema de design compartilhado e a introdução de prefixos únicos para cada microfrontend foram estratégias adotadas para superar esse desafio, contribuindo para uma experiência de usuário coesa.

O compartilhamento eficiente de dependências entre os micro frontends foi alcançado por meio do ModuleFederationPlugin, reduzindo a duplicação de código e minimizando conflitos de versões. A prática de lazy loading na aplicação container também contribuiu significativamente para a otimização do tamanho do payload, melhorando o desempenho geral da aplicação.

Apesar desses resultados positivos, é essencial reconhecer que a simplicidade do projeto limitou a avaliação completa de aspectos como escalabilidade em ambientes de produção e a exploração total da independência de tecnologias entre os micro frontends. Estes são tópicos que merecem uma investigação mais profunda em contextos mais complexos.

Concluindo, este trabalho proporcionou uma maior compreensão dos micro fron-

tends e sua aplicação prática. O próximo passo natural seria a expansão e refinamento desta implementação, considerando cenários mais desafiadores, inclusive com múltiplas equipes trabalhando em cada micro frontend e assim, explorando plenamente as capacidades dessa arquitetura emergente. Essa jornada, sem dúvida, abrirá novas possibilidades e desafios, contribuindo para o contínuo avanço no desenvolvimento de aplicações web modernas.

Referências

Ferracaku, J. (2021). The state of micro frontends: Challenges of applying and adopting client-side microservices.

Geers, M. (2020). *Micro Frontends in Action*. Manning, 1st edition.

Linkola, L. (2021). Design and implementation of modular frontend architecture on existing application.

Manisto, J., Tuovinen, A.-P., e Raatikainen, M. (2023). Experiences on a frameworkless micro-frontend architecture in a small organization. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 61–67.

Mezzalira, L. (2021). *Building Micro-Frontends*. O'Reilly, 1st edition.

Milic, M. e Makajic-Nikolic, D. (2022). Development of a quality-based model for software architecture optimization: A case study of monolith and microservice architectures. *Symmetry*, 14(9):1824.

Nascimento, C. H. P. e Sotto, E. C. S. (2020). Microfrontend: um estudo sobre o conceito e aplicação no frontend. *Interface Tecnológica*, 17(1):153–165.

Pavlenko, A., Askarbekuly, N., Megha, S., e Mazzara, M. (2020). Micro-frontends: application of microservices to web front-ends. *Journal of Internet Services and Information Security (JISIS),*, 10(2):49–66.

Peltonen, S., Mezzalira, L., e Taibi, D. (2021). Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. *Elsevier - Information and Software Technology*, (136).

Poloskei, I. e Bub, U. (2021). Enterprise-level migration to micro frontends in a multivendor environment. *Acta Polytechnica Hungarica*, 18(8):7–24.

Schio, A. B. (2023). Integração de módulos utilizando micro frontends na plataforma +precoce.

Taibi, D. e Mezzalira, L. (2022). Micro-frontends: Principles, implementations, and pitfalls. *ACM SIGSOFT Software Engineering Newsletter*, 47(4):25–29.

Zateishchikov, K. (2023). Scaling a software platform using micro frontends.