



CENTRO UNIVERSITÁRIO 7 DE SETEMBRO
CURSO SISTEMAS DE INFORMAÇÃO

MATEUS CESAR GONDIM DE LIMA

**ANÁLISE COMPARATIVA ENTRE AS ABORDAGENS DE MICRO FRONTEND E
MONÓLITO VISANDO ESCALABILIDADE, FACILIDADE DE MANUTENÇÃO E
EFICIÊNCIA.**

Fortaleza
2023

MATEUS CESAR GONDIM DE LIMA

**ANÁLISE COMPARATIVA ENTRE AS ABORDAGENS DE MICROFRONTEND E
MONÓLITO VISANDO ESCALABILIDADE, FACILIDADE DE MANUTENÇÃO E
EFICIÊNCIA.**

Monografia destinada à Conclusão do
Curso de Graduação do Centro Centro
Universitário 7 de Setembro como
requisito para a obtenção do Título de
Bacharel em Sistemas de Informação.
Orientador: Professor Doutor Francisco
Cristino de França Junior

Fortaleza
2023

MATEUS CESAR GONDIM DE LIMA

**ANÁLISE COMPARATIVA ENTRE AS ABORDAGENS DE MICROFRONTEND E
MONÓLITO VISANDO ESCALABILIDADE, FACILIDADE DE MANUTENÇÃO E
EFICIÊNCIA.**

Monografia apresentada ao Centro
Universitário 7 de Setembro como
requisito parcial para obtenção do título de
Bacharel em Sistemas de Informação.

Monografia aprovada pelos seguintes professores da banca examinadora, na
data: ____/____/____

Prof. Doutor Francisco Cristino de França Júnior
Centro Universitário 7 de Setembro – Uni7

Prof. MSc Marcelo Bezerra de Alcântara
Centro Universitário 7 de Setembro – Uni7

Prof. MSc. Alan Bessa Gomes Peixoto
Centro Universitário 7 de Setembro – Uni7

Prof. Doutor. Luciano Comin Nunes
Coordenador
Centro Universitário 7 de Setembro – Uni7

AGRADECIMENTOS

Agradeço primeiro a Deus por ter me mantido na trilha certa e cuidado de mim durante toda minha jornada chamada vida.

Aos meus pais Paulo César e Maria Adriana Gondim que sempre estiveram ao meu lado me apoiando ao longo de toda a minha trajetória.

Agradeço ao meu orientador Francisco Cristino de França Júnior por aceitar conduzir o meu trabalho de pesquisa e me orientar de forma excepcional.

Também agradeço aos meus amigos na instituição em especial o Adamor Henner que sempre me ofereceu carona para a instituição de ensino e nunca cobrou nada por isso, e também por toda parceria desempenhada durante todo o curso, onde atuamos juntos nos estágios 2 e 3 criando projetos do zero.

A todos os meus professores do curso de Sistemas de Informação do Centro Universitário 7 de Setembro pela excelência da qualidade técnica de cada um.

Agradecimento especial ao nosso eterno coordenador Marum Simão Filho, onde exerceu sua profissão com excelência, fazendo o possível e impossível por todos que tiveram o privilégio de cruzar a jornada com a dele.

Agradeço a todos que contribuíram direta ou indiretamente para realização deste trabalho.

RESUMO

Este Trabalho de Conclusão de Curso (TCC) promove uma análise comparativa abrangente entre sistemas monolíticos e microfrontends, com foco especial nos pilares fundamentais de escalabilidade, manutenibilidade e eficiência. A pesquisa minuciosamente explora as nuances de cada arquitetura, elucidando suas vantagens e desvantagens de modo a proporcionar uma compreensão aprofundada sobre suas implicações no ciclo de desenvolvimento, na capacidade de expansão e no desempenho global dos sistemas. O estudo visa oferecer insights valiosos para profissionais e pesquisadores, contribuindo para a tomada de decisões informadas no âmbito do desenvolvimento de software e arquitetura de sistemas.

Palavras-chave: Arquitetura, Desenvolvimento Frontend, Microfrontend, Monólito, Interface Visual, Sistemas.

ABSTRACT

This Undergraduate Thesis (TCC) conducts a comprehensive comparative analysis between monolithic systems and microfrontends, with a special focus on the fundamental pillars of scalability, maintainability, and efficiency. The research meticulously explores the nuances of each architecture, elucidating their advantages and disadvantages to provide an in-depth understanding of their implications on the development cycle, expansion capabilities, and overall system performance. The study aims to offer valuable insights for professionals and researchers, contributing to informed decision-making in the field of software development and system architecture.

Keywords: Architecture, Frontend Development, Microfrontends, React, Single-SPA, Monolith, Visual Interface, Systems.

LISTA DE FIGURAS

Figura 1 - Arquitetura monolítica.....	19
Figura 2 - Complexidade x Produtividade entre um sistema monolítico e microsserviços.....	21
Figura 3 - Cada Microfrontend é implantado na produção de forma independente.	23
Figura 4 - Exemplo de aplicação que usa arquitetura de Microfrontend.....	25
Figura 5 - Evolução da arquitetura dos projetos.....	26
Figura 6 - Arquitetura do protótipo monolítico.....	30
Figura 7 - Monólito: organização das pastas do protótipo monolítico.....	31
Figura 8 - Monólito: interface visual.....	32
Figura 9 - Microfrontend: criação do projeto utilizando Single-SPA.....	34
Figura 10 - Microfrontend: arquitetura geral do protótipo da aplicação com a arquitetura de microfrontend.....	35
Figura 11 - Microfrontend: todo-composer.....	36
Figura 12 - Microfrontend: exemplo de exportação de componente para consumo por outro MFE.....	37
Figura 14 - Microfrontend: todo-header.....	38

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
MFEs	Microfrontends
SPA	Single Page Application
SRP	Single Responsibility Principle
OCP	Open Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle

SUMÁRIO

1 INTRODUÇÃO.....	15
4.1 TECNOLOGIAS E FERRAMENTAS ASSOCIADAS A MICROFRONTENDS.....	26
4.1.1 SINGLE-SPA.....	26
4.1.2 WEBPACK MODULE FEDERATION.....	28
5 PROTÓTIPO FUNCIONAL.....	28
5.1 PROTÓTIPO MONOLÍTICO.....	29
5.2.1 REACT-MONOLITH-TODO.....	30
5.2 PROTÓTIPO DOS MICROFRONTENDS.....	34
5.2.1 TODO-COMPOSER.....	36
5.2.2 REACT-TODO-STYLEGUIDE.....	36
5.2.3 REACT-TODO-HEADER.....	38
5.2.4 REACT-TODO-LIST.....	38
5.2.5 REACT-TODO-UTIL-STATE.....	39
6 AVALIAÇÃO COMPARATIVA ENTRE MICROFRONTEND E MONÓLITO.....	40
7 CONCLUSÃO.....	43
7.1 TRABALHOS FUTUROS.....	44
APÊNDICE A:.....	49

1 INTRODUÇÃO

O desenvolvimento de aplicações passaram por inúmeras mudanças ao decorrer do tempo e nos anos 1980/1990 passou-se a ter toda a aplicação ou sistema em uma única fonte, onde o backend, frontend e banco de dados se mesclam formando o que chamamos de monólito (LEONARDO, 2020).

Com o crescimento desenfreado das aplicações alguns problemas foram aparecendo e tornou-se necessário construir sistemas cada vez mais desacoplados, pois através do desacoplamento nasce a possibilidade de ter grandes times separados agindo em cada camada da aplicação que está sendo desenvolvida e também o baixo acoplamento ajuda a isolar o comportamento de um componente de outros que dependem dele, aumentando a resiliência, agilidade e manutenibilidade de um sistema (AWS, 2023), porém em detrimento ao que foi apresentado até o momento, há o aumento da complexidade e custos do software.

Um estilo arquitetônico que ganhou bastante popularidade no presente século foi o de microsserviços, o qual é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando com mecanismos leves (LEWIS; FOWLER, 2014) e também tornou-se base para a criação de um modelo arquitetônico utilizado para construção de interfaces de usuário onde cada interface é dividida em partes menores e independentes chamadas de microfrontends.

Microfrontends foram introduzidos em 2016 que foi quando o termo ganhou notoriedade pela primeira vez no Technology Radar da *ThoughtWorks* (MEDIUM, 2018). A arquitetura de microfrontends tem como objetivo aplicar os conceitos de microsserviços para a camada de apresentação das aplicações. Praticamente o sistema *frontend* é dividido em outras partes menores como uma navbar ou uma section completa e essas partes são unidas em client-side, como o navegador do usuário, *edge-side*, como uma *CDN* ou *Content Delivery Network*, ou *server-side*, como um servidor na nuvem, produzindo um resultado único e coeso". (RICARDO, 2021, p.1)

Afinal, aplicações *frontend* que possuem sistemas com arquitetura de microfrontends serão sempre superiores quando comparados com sistemas que possuem arquitetura monolítica?

O objetivo geral deste trabalho visa apresentar uma análise comparativa entre os modelos arquiteturais monólito e microfrontends visando melhorar o entendimento em relação aos modelos focalizando questões sobre facilidade de manutenção, eficiência e escalabilidade.

Como objetivos específicos, tem-se:

- Apresentar o conceito da Arquitetura de Software, Arquitetura Monolítica e Arquitetura de microfrontends.
- Avaliar o nível de manutenibilidade, escalabilidade e eficiência da arquitetura monolítica.
- Avaliar o nível de manutenibilidade, eficiência e escalabilidade da arquitetura de microfrontends.

Comparar arquiteturas monolíticas e microfrontends é importante por várias razões, principalmente porque cada abordagem tem vantagens e desvantagens que podem afetar a eficiência e a manutenção de um sistema. A escolha entre arquiteturas monolíticas e microfrontends dependerá das necessidades específicas do projeto, tamanho da equipe, requisitos de escalabilidade e outros fatores. Uma comparação cuidadosa ajuda na tomada de decisões informadas para escolher a arquitetura mais adequada.

Do ponto de vista metodológico, este trabalho consiste em um estudo teórico reflexivo de revisão bibliográfica e apresentação de um protótipo para exemplificar o uso comparativo das duas abordagens.

Por fim, o estudo revela com base na análise teórica e comparativa empírica, que apesar das circunstâncias históricas de utilidade dos produtos monolíticos, diante da grande gama de possibilidades de plataformas, recursos tecnológicos e especialização das linguagens, não se pode abrir mão dos novos conceitos de microserviços aplicados, especialmente, nos microfrontends.

2 ARQUITETURA DE SOFTWARE

O conceito de arquitetura de software surgiu em um dos artigos publicados por Edsger Dijkstra em 1968 sobre o design de um sistema chamado "THE" (DIJKSTRA, 1968), embora o termo arquitetura de software não tenha sido mencionado nesse artigo, Dijkstra enfatiza a importância de uma estrutura de software clara e bem definida para garantir a eficácia e a confiabilidade do sistema.

O rápido avanço das tecnologias, como indicado por Moore (1965, p. 4), resultou em um aumento exponencial na capacidade de processamento das máquinas. Esse aumento tem um impacto direto no tamanho dos projetos, como observado pela Unyleia (s.d.), à medida que o desenvolvimento de software progride, resultando em um aumento na complexidade e desafios de projeto. Isso frequentemente leva a situações em que as estruturas de dados e algoritmos se tornam inadequados, tornando essencial a organização e a criação de padrões que promovam maior confiabilidade, manutenibilidade, escalabilidade e eficiência.

De acordo com Pressman (1995, p.6), na época a indústria de software estava passando por um período crítico que ficou conhecido como "crise de software", que teve seu início em meados de 1960 quando os programas existentes tornaram-se difíceis de serem mantidos devido a quantidade de instruções que possuíam, e o esforço despendido na manutenção de software começou a absorver recursos em índices alarmantes (PRESSMAN, 1995, p.10).

Com isso, cada vez mais foram surgindo estudos sobre padrões que tornavam o processo de desenvolvimento de sistema mais eficaz, visando melhorar questões de manutenibilidade, escalabilidade e eficiência. Algo que é capaz de adicionar esses atributos citados anteriormente relacionados a qualidade nos projetos é uma boa arquitetura de software, caso contrário, será mais lento e mais caro adicionar novas capacidades no futuro (FOWLER, 2019).

A definição para o termo arquitetura de software é bastante ampla, onde há uma dificuldade até de criar uma definição coerente, porém para Bass, Clements e Kazman (2015):

Sistemas de software são construídos para satisfazer os objetivos de negócios das organizações. A arquitetura é a ponte entre esses objetivos (frequentemente abstratos) e o sistema resultante (concreto). Enquanto o

caminho de objetivos abstratos para sistemas concretos possa ser complexo, a boa notícia é que a arquitetura de software pode ser projetada, analisada, documentada e implementada usando técnicas conhecidas que apoiarão a realização desses objetivos de negócios e missão. (BASS; CLEMENTS; KAZMAN, 2015, tradução nossa)

Com a definição da citação anterior, temos que para uma organização conseguir alcançar seus objetivos a arquitetura torna-se imprescindível, fazendo o elo entre os objetivos e o sistema resultante. Já para Fowler (2019), a arquitetura é, de fato, sobre coisas importantes, qualquer que seja a natureza dessas coisas. A observação de Fowler torna o conceito de arquitetura de software bastante abrangente, mas Terra e Valente (2008) dizem que a arquitetura de software é geralmente definida como um conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software.

De acordo com as citações anteriores percebemos que o conceito de arquitetura ainda não é algo bem definido e cada pessoa tem sua forma de definir, porém ao compararmos as citações conseguimos enxergar que ambas possuem um ponto em comum que é a preocupação em ter um sistema bem arquitetado para que a organização consiga atingir seus objetivos da melhor maneira, evitando custos, atrasos e prejuízos gerados por um sistema que teve sua arquitetura mal planejada durante o início do projeto.

Para Fowler (2019):

A arquitetura pobre é um dos principais contribuintes para o crescimento do lixo – elementos do software que impedem a capacidade dos desenvolvedores de entendê-lo. Software que contém muito lixo é muito mais difícil de modificar, levando a recursos que chegam mais lentamente e com mais defeitos. (FOWLER, p.1, 2019)

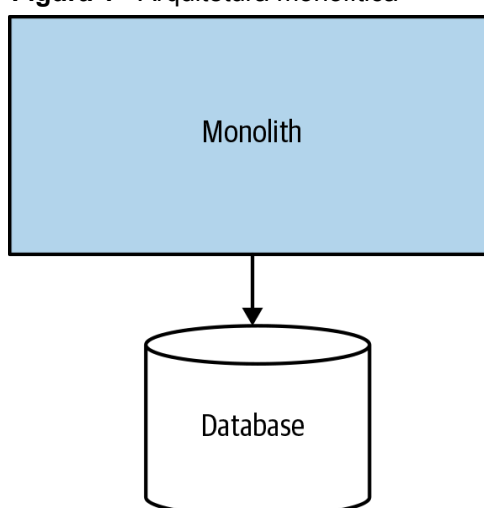
Com base nessa abordagem anterior o autor defende que o principal motivo dos desenvolvedores não entenderem o código é o crescimento do lixo, onde o mesmo é gerado através de uma arquitetura "pobre" e isso implica diretamente na dificuldade da modificação deste código, levando os recursos a chegarem mais lentamente (FOWLER, 2019).

3 ARQUITETURA MONOLÍTICA

Sistemas monolíticos são os mais utilizados no mundo corporativo, onde possui como característica a existência de uma única unidade de processamento com múltiplas responsabilidades de negócio (GOMES, 2019), ou seja, todas as funções estão em um único pacote a ser distribuído ao cliente. (SANTOS, 2021).

A Figura 1 mostra o funcionamento de uma estrutura monolítica, onde uma aplicação inteira concentra-se em um único módulo que se comunica com um banco de dados, onde traz consigo diversas vantagens e desvantagens. Como vantagens temos facilidade de implementação, implantação (*deploy*), velocidade de desenvolvimento nos momentos iniciais implicando em um retorno de resultado mais rápidos aos clientes, dentre outros inúmeros benefícios. Com tudo, de acordo com o artigo publicado na Opus Software (2021), o passar do tempo e crescimento do projeto em si pode trazer consigo problemas, como: alto acoplamento, códigos difíceis de entender, aumento da complexidade, alta dependência de componentes de código e falta de flexibilidade. Entretanto, através de um desenvolvimento padronizado e seguindo boas práticas, é possível alcançar com excelência todos os pontos citados anteriormente garantindo flexibilidade, manutenibilidade, diminuição da complexidade e redução do acoplamento.

Figura 1 - Arquitetura monolítica



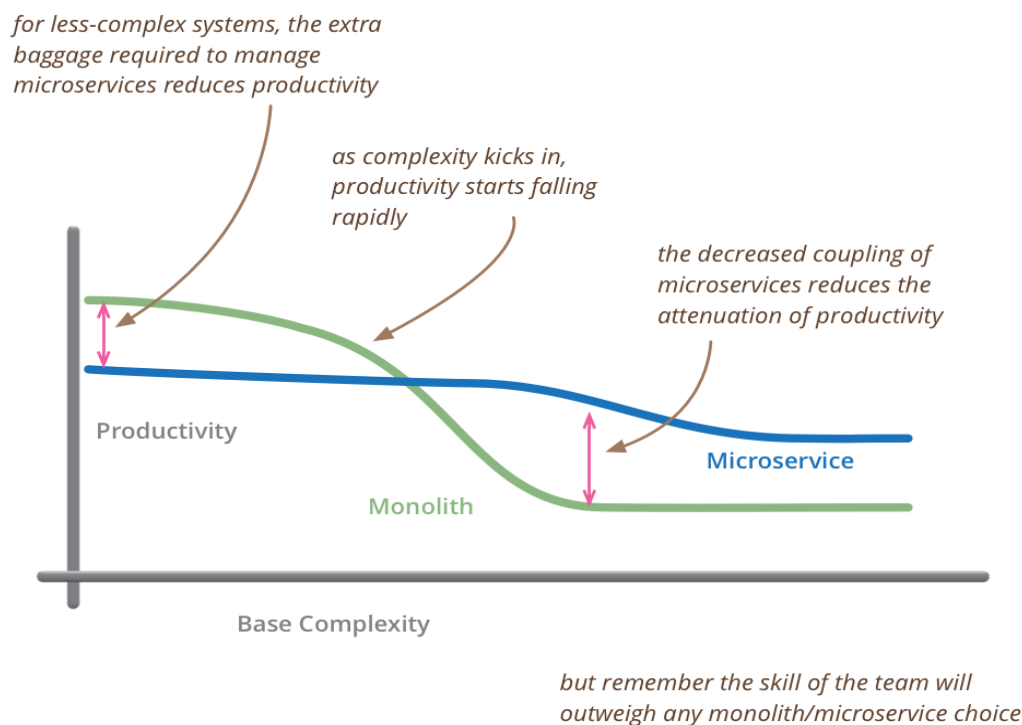
Fonte: Newman (2019)

Caso os pontos anteriores não sejam alcançados, isso pode ocasionar muitos prejuízos aos clientes, já que, novas *features* e atualizações serão cada vez mais complexas de serem implementadas e enviadas para produção, e Newman (2015) acentua isso, informando que atualizações relacionadas a utilização de uma nova linguagem de programação, banco de dados ou framework é capaz de impactar muito todo o sistema, seja positivamente ou negativamente.

Para Gonçalves (2022) falar de sistema monolítico é um pouco estranho devido ao advento dos microsserviços, onde ao citar o termo "monólito" que deveria referir-se a unidade de *deployment*, acabou se tornando de uma maneira equivocada sinônimo de aplicações legadas, ou seja, quando falamos de monólitos estamos nos referindo a algo antigo ou inferior em relação às arquiteturas que possuímos hoje, porém um sistema monolítico nem sempre é um vilão ou algo ruim a se utilizar, muitas vezes dependendo do contexto é melhor utilizar um sistema monolítico do que um sistema baseado na arquitetura de microsserviços, onde muitas vezes é adicionado muita complexidade ao sistema por completo sem necessidade.

Quase todas as histórias de sucesso de microsserviços começaram com um monólito que ficou grande demais e foi desmembrado (FOWLER, 2015). Conforme o que foi dito anteriormente por Fowler no artigo intitulado "*Monolith First*", a construção de um sistema baseado na arquitetura monolítica ainda é a melhor opção para iniciar um novo projeto, mesmo que saibam que a aplicação será grande o suficiente e que deverá ser utilizado microsserviços no futuro. No mesmo artigo, Fowler (2015) afirma que as histórias bem sucedidas de microsserviços começaram com um monólito que ficou grande demais e tornou-se necessário dividi-lo, como também houve muitos casos de sistemas que foram construídos como microsserviços desde a sua concepção e acabaram tornando-se um grande fracasso através de graves problemas que surgem com o tempo.

Figura 2 - Complexidade x Produtividade entre um sistema monolítico e microsserviços.



Fonte: Martin Fowler (2018)

Conforme ilustrado pela Figura 2, a produtividade no início de um projeto em que inicializa-se com a arquitetura monolítica é bem superior quando se comparado com um projeto que utiliza a arquitetura de microsserviços, e isso mostra que por mais que a arquitetura de microsserviços tenha surgido para lidarmos com a complexidades de um sistema grande e robusto, essa arquitetura por si só já adiciona seu próprio conjunto de complexidades (FOWLER, 2015), e isso implica diretamente na principal diretriz de Fowler (2015) sobre o assunto, onde o mesmo informa que microsserviços devem ser desconsiderados, a menos que o sistema que está sendo construído seja complexo demais para ser gerenciado como um monólito e para Newman (2015) microsserviços não são uma solução que vai resolver todos seus problemas, na verdade, caso não haja os devidos cuidados, pode-se tornar uma péssima escolha, pois carregam a complexidades associadas aos sistemas distribuídos (NEWMAN, 2015, p. 33).

De acordo com Marques (2022) sistemas monolíticos com o decorrer do tempo tendem a tornar a questão da manutenibilidade mais complexa, pois como tudo se concentra em um único módulo, o alto acoplamento faz com que torne-se

por muitas vezes inviável dependendo de como o sistema foi planejado. No início do projeto é tudo bastante simples, tudo concentra-se em um único lugar, mas como citado anteriormente, a partir do momento em que a base de código aumenta e o número de colaboradores trabalhando no sistema aumenta, modificações vão ficando cada vez mais inviáveis, pois uma pequena alteração em um ponto do código pode trazer vários problemas em outras partes do projeto.

O sistema monolítico em si nos dias atuais não é um problema, continuam sendo uma ótima escolha para sistemas de sucesso, porém o problema pode ser a forma de como o mesmo foi construído, em poucas palavras, o que pode ser o pai de todos os problemas em um sistema monolítico quando trata-se de manutenibilidade é um código mal implementado (SECCO, 2019) e uma arquitetura mal planejada.

De acordo com Secco (2019), sistemas monolíticos possuem facilidade de se escalar verticalmente, que implica na capacidade de aumentarmos os recursos do servidor que sustenta a aplicação, onde pode-se aumentar memória e números de CPUs por exemplo. Outra forma de escalabilidade que o sistema monolítico possui também é a horizontal, que consiste em dividir a carga em múltiplos servidores, adicionando mais nós (SECCO, 2019), porém, sistemas monolíticos são muito complicados de escalar horizontalmente devido o sistema exigir que os processos sejam executados de forma sequencial.

Sistemas monolíticos são eficientes e por muitas vezes quando comparados com Microfrontend possuem um desempenho melhor devido que tudo concentra-se em um único módulo, facilitando a sua comunicação e como todos os componentes estão interligados, a comunicação entre eles é direta e rápida (SILVA, 2023). Essa centralização permite uma melhor otimização de desempenho, pois o código é executado em um único ambiente, minimizando a latência e as interações entre os componentes (UZEDA, 2023).

Desenvolver um sistema monolítico oferece diversas vantagens e desvantagens, tais como, dificuldade de entendimento da aplicação, escalabilidade horizontal, demora na implantação, facilidade de deploy, (MELLA; MÁRQUEZ; ASTUDILLO, 2019).

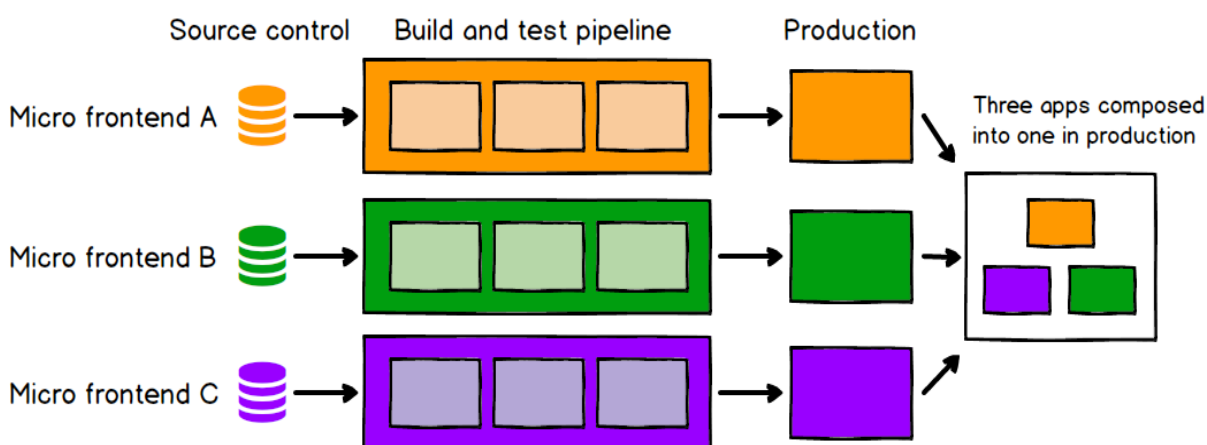
4 ARQUITETURA DE MICROFRONTENDS

O desenvolvimento web moderno está cada vez mais dando atenção aos detalhes de arquitetura, onde vemos as aplicações relacionadas às interfaces de usuário fortemente robustas necessitando de padrões para decompor a aplicação em pedaços menores e mais simples que podem ser desenvolvidos, testados e implantados de forma independente enquanto ainda aparecem para os clientes como um produto único e coeso (JACKSON, 2019).

Microfrontend é um estilo arquitetônico onde aplicativos front-end são entregues de forma independente e são compostos em um todo maior (JACKSON, 2019) onde é capaz de fornecer um conjunto de benefícios e assim como microserviços inserem um conjunto de complexidades também, porém para Jackson (2019) alguns riscos podem ser gerenciados de uma melhor maneira e que os benefícios como atualizações incrementais, bases de código simples e desacopladas, implantação independente e equipes autônomas superam os custos.

Jackson (2019) também resumiu sua explicação sobre microfrontend, informando que:

Figura 3 - Cada Microfrontend é implantado na produção de forma independente



Fonte: Martin Fowler (2019)

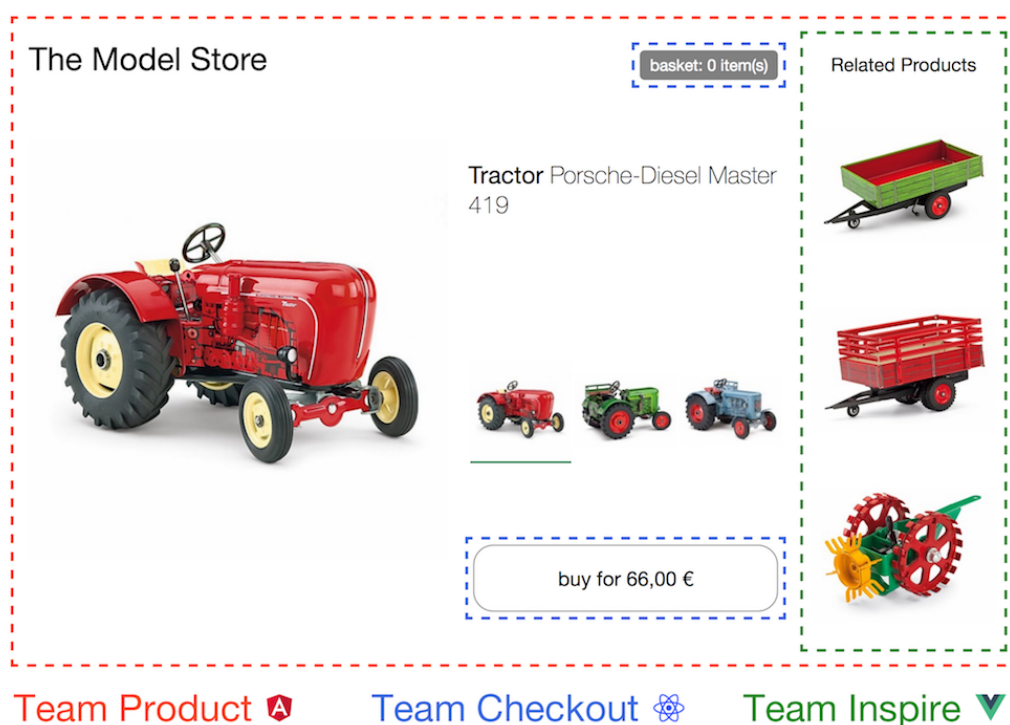
A Figura 3 aponta muito bem como funciona a arquitetura de Micro frontend, onde temos que cada Micro frontend é desenvolvido de maneira independente, muitas vezes por equipes distintas, e isso permite uma abordagem descentralizada

no desenvolvimento, e diferentes partes da aplicação podem evoluir separadamente e que cada parte pode ser responsável por uma funcionalidade específica da interface do usuário, seguindo o que foi dito no início deste tópico: as aplicações podem ser desenvolvidas, testadas e implantadas de forma independente. Cada um desses fragmentos estão em projetos diferentes, em repositórios diferentes, sendo gerenciados por equipes diferentes. Cada um com seus contextos, tecnologias e forma de trabalhar (CARLA, 2019).

Um fator muito interessante também relacionado a abordagem de Microfrontend é a possibilidade da utilização de frameworks diversificados, ou seja, cada Microfrontend pode ser desenvolvido usando tecnologias, frameworks e linguagens diferentes que possibilita o time escolher qual a melhor tecnologia para determinada parte em relação a toda aplicação, e que no final irão comunicar-se e continuará sendo um produto único e coeso.

A Figura 4 a seguir expõe como funciona a aplicação com o uso de diferentes frameworks na sua construção com o time Product utilizando o Framework Angular, time Checkout utilizando React e o Time Inspire utilizando Vue JS.

Figura 4 - Exemplo de aplicação que usa arquitetura de Microfrontend

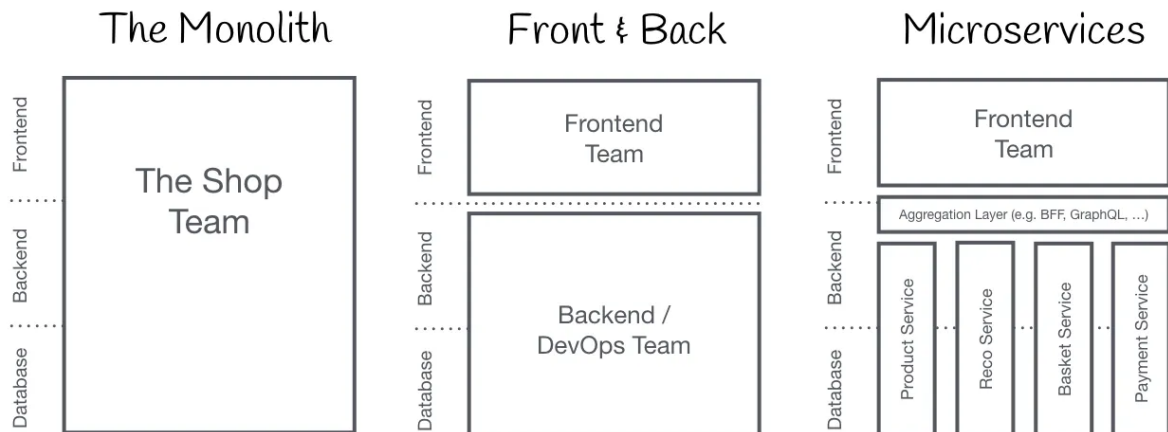


Fonte: Tautorn (2023)

A Figura 4 representa uma página de venda de tratores e algumas peças complementares. A página está dividida em equipes diferentes, onde cada fragmento é de responsabilidade de uma determinada equipe, ou seja, por mais que o usuário veja algo único e coeso, na verdade por debaixo dos panos tudo é fragmentado e de responsabilidade de diversas equipes que no final realizam a integração desses microfrontends tornando tudo como se fosse um único sistema, até porque o usuário não precisa saber como funciona por debaixo dos panos, apenas que todas as funcionalidades estejam trabalhando corretamente e em harmonia.

A Figura 5 mostra a evolução da arquitetura dos projetos focalizando em trazer de forma simples uma compreensão melhor sobre o assunto e de como as arquiteturas evoluíram com o decorrer do tempo e aos poucos foram segmentando-se, Carla (2016) complementa que:

Figura 5 - Evolução da arquitetura dos projetos



Fonte: Tautorn (2023)

Todo um sistema era desenvolvido em um único repositório, tendo front-end e back-end em um só lugar. O que tornava a complexidade muito alta e dificultava a separação das tarefas entre os desenvolvedores. Nessa época, praticamente os desenvolvedores eram obrigados a ser fullstack. Com o tempo, houve uma separação entre o back-end e o front-end, permitindo que eles trabalhassem de forma independente e se integrassem de maneira mais organizada. (CARLA, 2016)

4.1 TECNOLOGIAS E FERRAMENTAS ASSOCIADAS A MICROFRONTENDS

Nesta seção serão apresentadas ferramentas que são muito utilizadas para construção de projetos com arquitetura de microfrontends. Ferramentas que possuem como intuito facilitar o desenvolvimento e construção de projetos.

4.1.1 SINGLE-SPA

Single-SPA é uma biblioteca JavaScript de código aberto que permite a construção de aplicações de página única (SPA) compostas por microfrontends

independentes. Ele facilita a integração e a orquestração de várias partes da aplicação que são desenvolvidas, implantadas e mantidas de forma independente.

Single-SPA se inspira nos ciclos de vida dos componentes dos frameworks modernos, abstraindo os ciclos de vida para aplicativos inteiros. Surgido do desejo da Canopy de usar React + react-router em vez de ficar preso para sempre em nossa aplicação AngularJS + ui-router, single-spa é agora uma biblioteca madura que possibilita a arquitetura de microfrontends no frontend. Microfrontends oferecem muitos benefícios, como implantações independentes, migração e experimentação, e aplicativos resilientes. (SINGLE-SPA, 2023).

De acordo com a documentação disponibilizada no site da biblioteca single-spa (2023), existem três tipos de microfrontends:

- **single-spa:** Microfrontend que renderiza componentes por rotas específicas
- **single-spa parcels:** Microfrontend que renderiza componentes sem rotas controladas
- **utility-modules:** Microfrontend que exporta a lógica javascript sem renderizar componentes para que os outros microfrontend consumam removendo assim algumas responsabilidades dos mesmos. Um exemplo comum do uso do utility modules incluem guias de estilos (CSS, Styled-components, SASS, Tailwind), auxiliares de autenticação, auxiliares de API e componentes estilizados.

A documentação disponibilizada pelo Single-SPA (2023), informa que utilizar *single-spa* proporciona diversos benefícios, tais como:

- Utilizar vários frameworks na mesma página sem a necessidade de recarregamento da página (React, AngularJS, Angular, Ember, ou qualquer que seja o framework que você estiver usando).
- Microsserviços front-end de forma independente.
- Escrever código usando um novo framework sem ter que reescrever o app existente.
- Carregamento tardio de código para melhorar o tempo de carga inicial.

4.1.2 WEBPACK MODULE FEDERATION

De acordo com a documentação disponibilizada pelo site do Webpack (2023), o Webpack Module Federation, introduzido no Webpack 5 em 2020, é uma funcionalidade que permite o compartilhamento dinâmico de módulos entre diferentes aplicações JavaScript. Essa abordagem desacopla aplicações front end, permitindo que diferentes partes da interface do usuário sejam desenvolvidas e mantidas independentemente.

Já o artigo publicado da Module Federation (2023), informa que o código pode ser dividido em módulos menores, implementáveis de forma independente, que podem ser carregados sob demanda quando necessário. Isso permite que microfrontends sejam desenvolvidos e implantados de forma independente, o que reduz a coordenação entre equipes e permite ciclos de desenvolvimento mais rápidos.

O Module Federation é independente de frameworks específicos e pode ser configurado no Webpack, definindo remotes (aplicações que fornecem módulos compartilhados) e entry points para os módulos compartilhados. Ele facilita o carregamento dinâmico de módulos remotos, suporta sistemas de roteamento dinâmico e isola o escopo de estilos para evitar conflitos. Essa funcionalidade se alinha com arquiteturas de microfrontends, oferecendo flexibilidade e escalabilidade na construção de sistemas front end.

5 PROTÓTIPO FUNCIONAL

Os protótipos desenvolvidos neste trabalho são um pequeno sistema para inserção, remoção e edição de atividades relacionadas ao dia a dia, onde um dos protótipos é um sistema *frontend* monolítico construído utilizando React como framework, e o segundo é um sistema criado utilizando arquitetura de microfrontends com Single-SPA e React. O objetivo geral dos protótipos é mostrar o nível de complexidade que um sistema de microfrontends é capaz de inserir em uma aplicação de pequeno porte e com isso também expôr que um sistema monolítico

perante todas as tecnologias que possuímos hoje ainda é algo que deve ser utilizado mediante o contexto da aplicação.

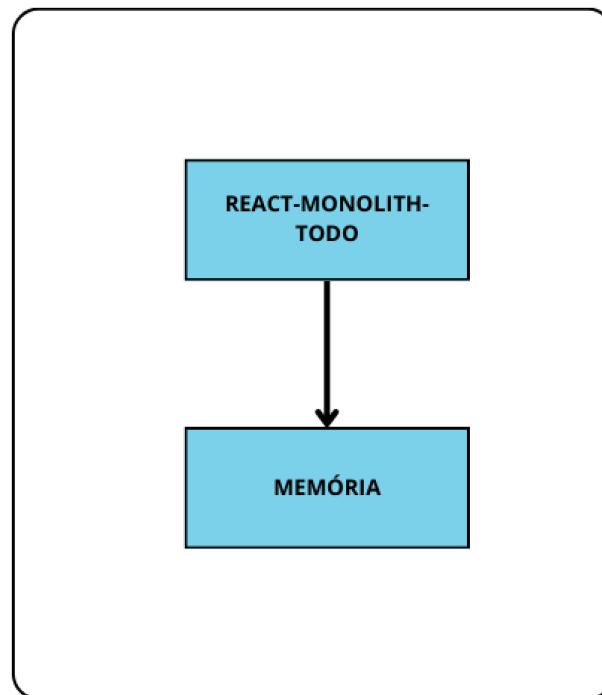
O código fonte das aplicações desenvolvidas encontram-se disponíveis no Github <https://github.com/mateuscesarglima?tab=repositories>, dos quais `todo-composer`, `react-todo-header`, `react-todo-styleguide`, `react-todo-list` e `react-todo-util-state` são referentes a aplicação desenvolvida utilizando `single-spa` e o `react-monolith-todo` é a aplicação monolítica desenvolvida.

O design da aplicação foi criado pela Rocketseat (2023), onde o acesso se dá através do Figma <https://www.figma.com/@rocketseat>.

5.1 PROTÓTIPO MONOLÍTICO

O protótipo foi desenvolvido utilizando javascript como linguagem de programação e React como *framework*, onde todo o gerenciamento de dados está sendo feito na memória através do armazenamento das informações em um *state* que é um *hook* do React. Um *Hook* é uma função especial que te permite utilizar recursos do React, que no caso do protótipo é o *useState*, que é um Hook que te permite adicionar o state do React a um componente de função (REACT, 2023).

Figura 6 - Arquitetura do protótipo monolítico



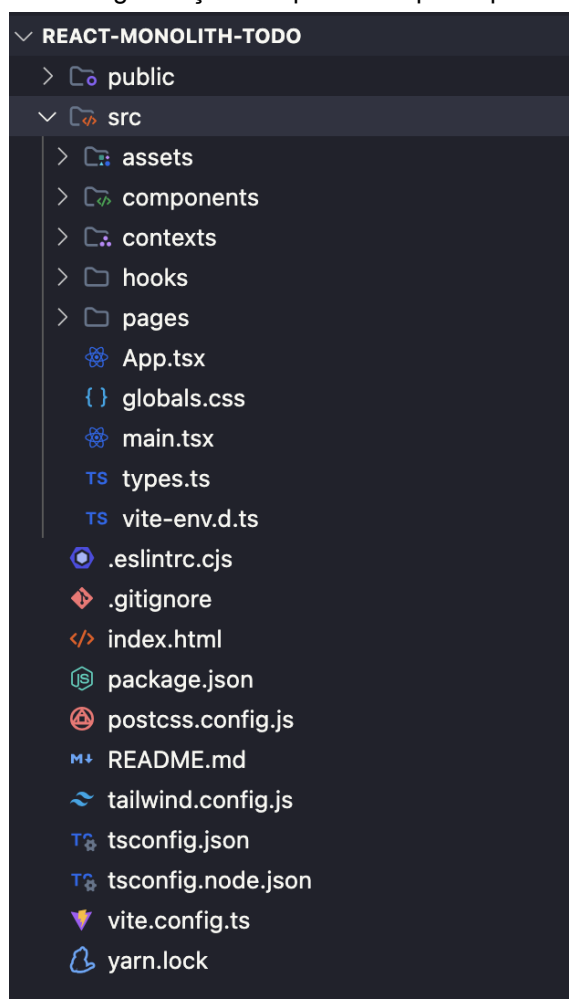
Fonte: Pelo Autor (2023)

A Figura 6 mostra como está organizada a arquitetura do protótipo monolítico desenvolvido para o trabalho em questão, trazendo uma visão mais simples e explicativa visando uma melhor compreensão da forma de como está organizada a aplicação monolítica.

5.2.1 REACT-MONOLITH-TODO

Esse protótipo consiste em uma aplicação que permite o registro organizado de atividades a serem realizadas, esse tipo de aplicação é geralmente usado para planejar, priorizar e monitorar o progresso em tarefas diárias ou projetos. A Figura 6 mostra a arquitetura monolítica da interface de usuário do protótipo em questão, trazendo uma visão semelhante a Figura 2, onde a única diferença apresenta-se na forma em que os dados são armazenados, que no caso do protótipo está sendo na memória, onde ao reiniciar a aplicação todos os dados que foram salvos serão perdidos.

Figura 7 - Monólito: organização das pastas do protótipo monolítico



Fonte: Pelo Autor (2023)

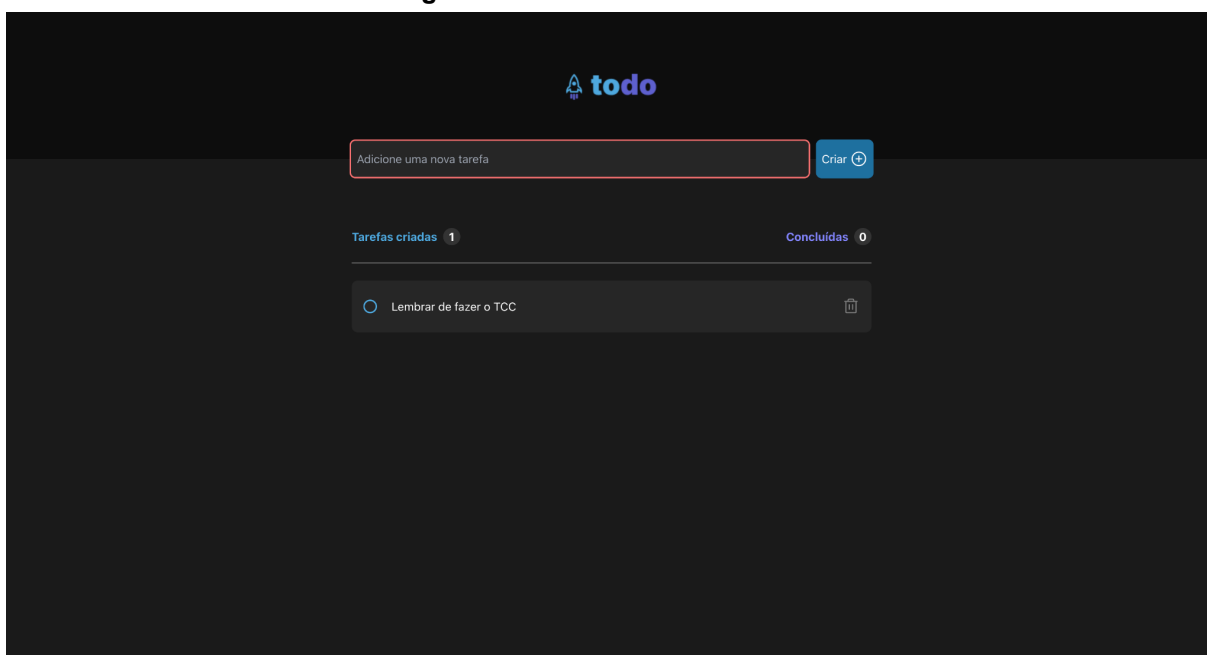
A organização das pastas do protótipo monolítico está presente na Figura 6, a qual toda a aplicação concentra-se em apenas um projeto e tanto a estilização, contexto, componentes e telas estão inseridas e serão enviadas ao mesmo bundle (arquivo que é consumido pelo browser). A forma como o projeto é organizado pode facilitar a questão da divisão de responsabilidades entrando um pouco no quesito dos princípios *S.O.L.I.D*, que é um acrônimo para:

- Single Responsibility Principle (SRP) – Princípio da Responsabilidade Única
- Open Closed Principle (OCP) – Princípio Aberto-fechado

- Liskov Substitution Principle (LSP) – Princípio da Substituição de Liskov
- Interface Segregation Principle (ISP) – Princípio da Segregação de Interface
- Dependency Inversion Principle (DIP) – Princípio da Inversão de Dependências

Ao seguir os princípios S.O.L.I.D, é possível obter o máximo de controle possível sobre a aplicação que está sendo desenvolvida e facilitar questões de manutenibilidade e eficiência de acordo com o crescimento do projeto. Um padrão que facilita essa organização é o dumb (presentational components) e o smart (containers), o qual são uma forma de compreender essas responsabilidades, permitindo maior reusabilidade e manutenibilidade do código (NCB, 2018).

A disponibilização dessa aplicação pode ser realizada através de um único *deploy*, uma vez que, todos seus componentes e funcionalidades concentram-se em um único projeto, diferentemente de um projeto que contém a arquitetura de microfrontends onde cada microfrontend possui seu *deploy* independente que ao final comunicam-se de forma conjunta e coesa, cada microfrontend com suas devidas responsabilidades.

Figura 8 - Monólito: interface visual

Fonte: Pelo Autor (2023)

A Figura 8 mostra detalhadamente como a aplicação monolítica encontra-se ao fim de sua implementação, realizando normalmente as ações de criar a tarefa, atualizar o status e também a remoção da atividade da lista de atividades ao clicar no ícone da lixeira.

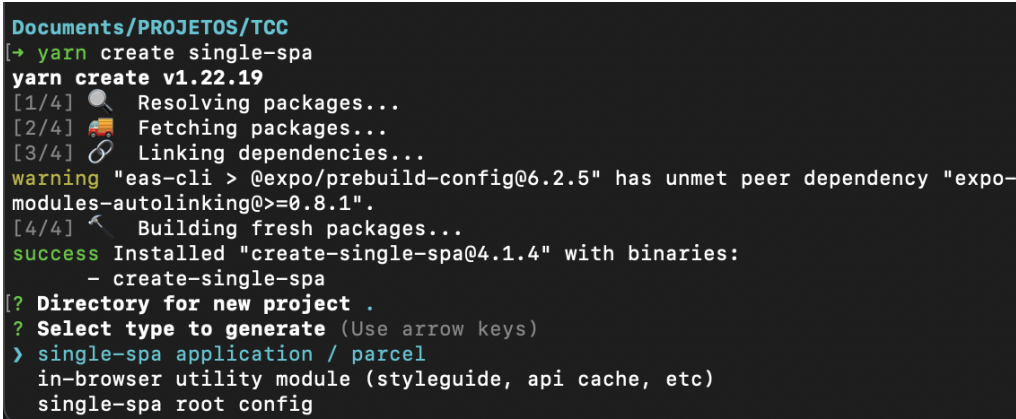
Neste ponto é realizada a finalização da apresentação do protótipo monolítico desenvolvido visando simplificar a compreensão de como funciona uma interface visual monolítica dinâmica. Seu desenvolvimento foi bastante simples e para pessoas iniciantes a curva de aprendizado é baixa, uma vez que, ao compararmos com microfrontends, é realizado bem menos configurações para que o projeto possa funcionar, e dependendo da forma de como o desenvolvimento é realizado, o software pode alcançar níveis de eficiência, manutenibilidade e escalabilidade altos, como também esses níveis podem ser muito baixos dificultando o crescimento do projeto e também podendo trazer riscos para os clientes.

5.2 PROTÓTIPO DOS MICROFRONTENDS

O protótipo utilizando a arquitetura de microfrontends foi desenvolvido utilizando dois frameworks, que são: *Single-SPA* e *React*. O Single-SPA é responsável totalmente por criar e gerenciar todos os microfrontends relacionados com o projeto ao todo.

Para iniciar a criação dos microfrontends foi utilizado o comando *yarn create single-spa*, porém esse não é o único comando capaz de realizar a criação dos microfrontends. Após inserir esse comando no terminal tendo o node instalado e nesse caso também o *yarn*, irá aparecer um menu no terminal perguntando qual o diretório aquele microfrontend em si será criado, e logo após qual o tipo do microfrontend.

Figura 9 - Microfrontend: criação do projeto utilizando Single-SPA



```
Documents/PROJETOS/TCC
[→ yarn create single-spa
yarn create v1.22.19
[1/4] 🔍 Resolving packages...
[2/4] 📦 Fetching packages...
[3/4] 🔗 Linking dependencies...
warning "eas-cli > @expo/prebuild-config@6.2.5" has unmet peer dependency "expo-modules-autolinking@>=0.8.1".
[4/4] ⚙ Building fresh packages...
success Installed "create-single-spa@4.1.4" with binaries:
- create-single-spa
[? Directory for new project .
? Select type to generate (Use arrow keys)
> single-spa application / parcel
in-browser utility module (styleguide, api cache, etc)
single-spa root config
```

Fonte: Pelo Autor (2023)

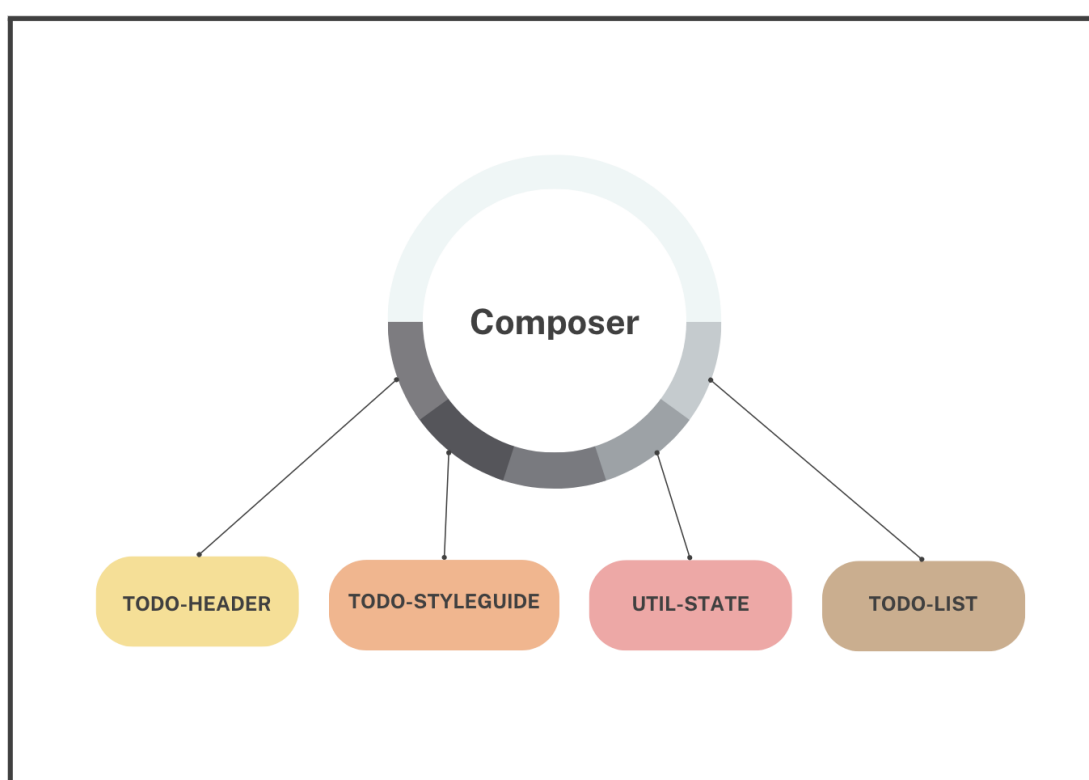
Conforme citado na seção 4.1.1, existem 3 tipos de microfrontend para você selecionar no menu de criação que aparece através da inserção e a Figura 9 mostra os 3 tipos no terminal, que são:

- **single-spa application / parcel:** template de microfrontend responsável por renderizar os componentes.
- **in-browser utility module (styleguide, api cache, etc):** template do microfrontend responsável por conter componentes estilizados,

comunicação com endpoints, dentre outras funcionalidades que podem ser importadas por outros microfrontends.

- **single-spa root config:** template do principal microfrontend da aplicação, responsável por realizar a importação dos MFEs e também seus respectivos registros.

Figura 10 - Microfrontend: arquitetura geral do protótipo da aplicação com a arquitetura de microfrontend



Fonte: Pelo Autor (2023)

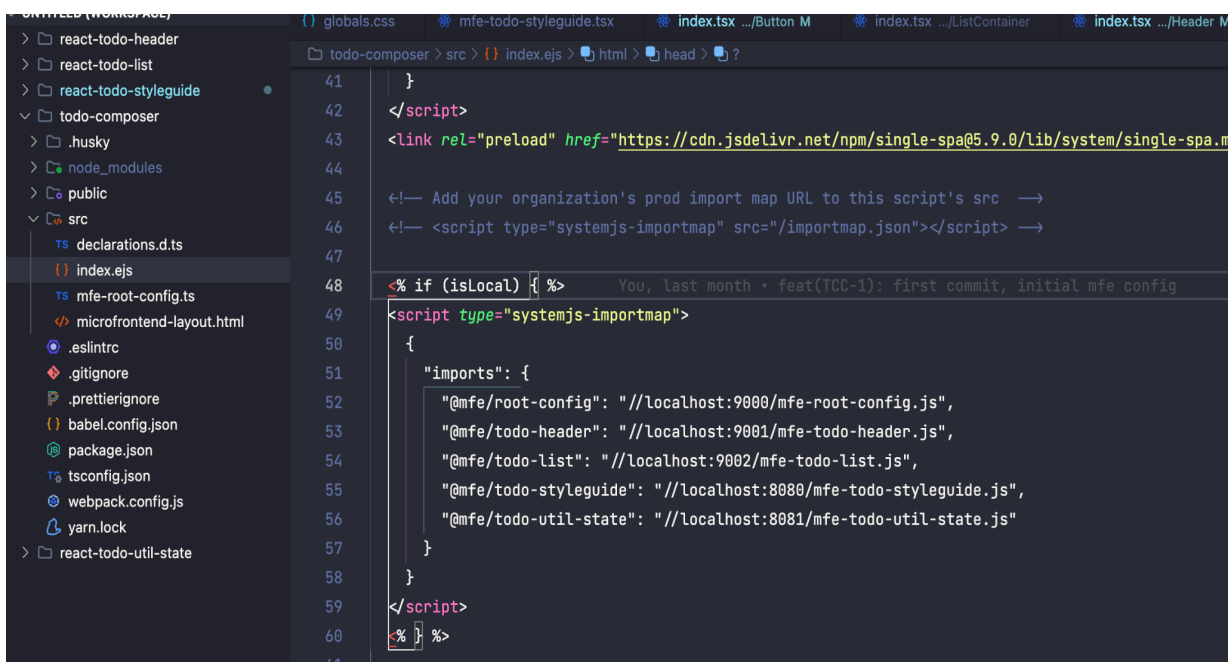
A arquitetura do protótipo de microfrontends Figura 10, traz a forma de como está estruturada a aplicação, onde o composer é a aplicação que fornece todo o gerenciamento dos microfrontends e sua comunicação. O composer é o "root-config", ou seja, o projeto responsável por ter as principais configurações que englobam os principais microfrontends envolvidos na aplicação, onde é realizado a renderização da página HTML e o JavaScript que registra os aplicativos (SINGLE-SPA, 2023).

5.2.1 TODO-COMPOSER

Como citado no tópico anterior, o root-config que no protótipo em questão é chamado de composer, é o coração de um projeto que utiliza-se da arquitetura de microfrontend com single-spa, que de acordo com a documentação do single-spa (2023) é responsável por realizar renderização do HTML e Javascript do registro das aplicações relacionadas.

O composer tem como arquivo principal o "index.ejs", nele concentra-se as importações dos microfrontends que estarão sendo utilizados no projeto, conforme a Figura 11.

Figura 11 - Microfrontend: todo-composer



```
41 }
42 </script>
43 <link rel="preload" href="https://cdn.jsdelivr.net/npm/single-spa@5.9.0/lib/system/single-spa.m
44
45 <!-- Add your organization's prod import map URL to this script's src -->
46 <!-- <script type="systemjs-importmap" src="/importmap.json"></script> -->
47
48 <% if (isLocal) { %> You, last month + feat(TCC-1): first commit, initial mfe config
49 <script type="systemjs-importmap">
50 {
51   "imports": {
52     "@mfe/root-config": "//localhost:9000/mfe-root-config.js",
53     "@mfe/todo-header": "//localhost:9001/mfe-todo-header.js",
54     "@mfe/todo-list": "//localhost:9002/mfe-todo-list.js",
55     "@mfe/todo-styleguide": "//localhost:8080/mfe-todo-styleguide.js",
56     "@mfe/todo-util-state": "//localhost:8081/mfe-todo-util-state.js"
57   }
58 }
59 </script>
60 <% } %>
61
```

Fonte: Pelo Autor (2023)

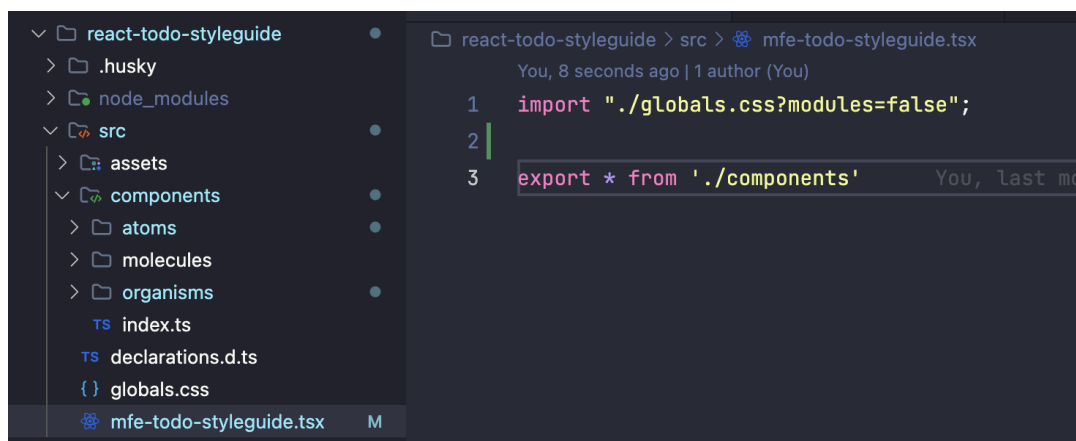
A Figura 11 mostra que dentro da tag script é realizada a importação de todos os MFEs que deverão ser utilizados, no caso do protótipo realizamos a importação *todo-header*, *todo-list*, *todo-styleguide* e *todo-util-state*.

5.2.2 REACT-TODO-STYLEGUIDE

Módulo utilitário é um módulo JavaScript no navegador que não é uma aplicação single-spa ou um pacote. Em outras palavras, seu único propósito é exportar funcionalidades para que outros microfrontends possam importar. Exemplos comuns de módulos utilitários incluem guias de estilo, auxiliares de autenticação e auxiliares de API. Esses módulos não precisam ser registrados no single-spa, mas são importantes para manter a consistência entre várias aplicações e pacotes single-spa. (SINGLE-SPA, 2023).

O *react-todo-styleguide* é um módulo utilitário para fornecer componentes para os MFEs utilizados, como o *react-todo-header* e o *react-todo-list*. Todos os componentes são exportados por esse microfrontend e todos os MFEs que precisam utilizá-los basta apenas realizar a importação do componente que deseja, mediante os componentes estarem sendo exportados da maneira correta.

Figura 12 - Microfrontend: exemplo de exportação de componente para consumo por outro MFE



Fonte: Pelo Autor (2023)

A Figura 12 mostra a forma de como os componentes que estão sendo utilizados na aplicação estão sendo exportados pelo microfrontend *react-todo-styleguide*, onde qualquer outro microfrontend que deseja utilizar-se desses componentes basta importar o microfrontend e o componente que deseja.

Figura 13 - Microfrontend: explicação sobre a importação do componente Header

```
import { Header } from '@mfe/todo-styleguide'
```

Fonte: Pelo Autor (2023)

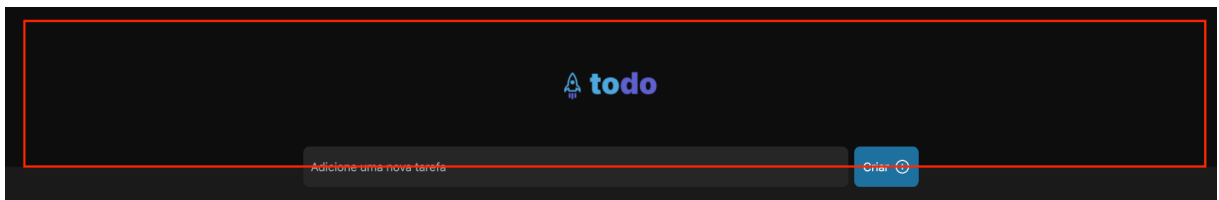
De acordo com a Figura 13 temos de forma detalhada como pode ser realizado a importação dos componentes que precisam ser utilizados no projeto. É importante salientar que essa não é a maneira mais adequada de realizar a importação dos componentes, uma vez que, a documentação disponibilizada pelo site do framework recomenda que seja realizado a disponibilização dos componentes através de uma biblioteca que possa ser instalada no microfrontend que necessita utilizar o componente ou alguma função utilitária.

5.2.3 REACT-TODO-HEADER

Aplicação que contém apenas o Header como forma de mostrar a comunicação e o enquadramento dos componentes utilizando a arquitetura de microfrontend. A Figura 14 mostra detalhadamente o componente está sendo importado diretamente do microfrontend *react-todo-styleguide* e a Figura 13 mostra como realizar essa importação, realizando através da importação do MFE e entre as chaves pegando o utilitário desejado e também, de acordo com a documentação do Single-SPA

É permitido realizar a importação e exportação de funções, componentes, lógica, dados, emissores de eventos e variáveis de ambiente entre seus microfrontends que estão em repositórios Git e pacotes JavaScript diferentes. Cada microfrontend deve ter um único arquivo de entrada que serve como a "interface pública" que controla o que é exposto fora do microfrontend. (SINGLE-SPA, 2023).

Figura 14 - Microfrontend: todo-header



Fonte: Pelo Autor (2023)

5.2.4 REACT-TODO-LIST

Microfrontend responsável por renderizar a lista de atividades e também por disparar as ações de adição, remoção e edição das atividades. Nesse MFE estará

contido os componentes de Input e o botão, onde o botão com o texto "Criar" será responsável por disparar a ação de adição. Todos os componentes também estarão sendo importados diretamente do MFE react-todo-styleguide.

5.2.5 REACT-TODO-UTIL-STATE

Essa aplicação é também um utility-module que é responsável inteiramente por lidar com os estados da aplicação, ela fornece e gerencia o armazenamento dos dados, inserção, remoção e edição.

Para o gerenciamento foi utilizado uma biblioteca chamada "zustand", que de acordo com a documentação da aplicação em seu site oficial (2023), essa biblioteca é uma:

Solução de gerenciamento de estado básica, rápida e escalável, utilizando princípios simplificados do Flux. Possui uma API confortável baseada em hooks, sem ser excessivamente cheia de boilerplate ou opinativa. Não a subestime por sua aparência fofa. Ela tem garras afiadas e por muito tempo foi dedicada a lidar com armadilhas comuns, como o temido problema dos "zumbis" (zombie child), concorrência no React e perda de contexto entre renderizadores mistos. Pode ser o gerenciador de estado no espaço do React que acerta em todos esses aspectos. (ZUSTAND, 2023).

No caso do react-todo-util-state foi realizada a criação da store, que de acordo com a documentação oficial do Zustand (2023) é um hook onde pode ser inserido qualquer coisa como: primitivos, objetos e funções. A função 'set' mescla o estado. O código fonte relacionado a implementação principal do gerenciador de estados encontra-se no apêndice A.

Neste tópico é realizada a finalização da apresentação do protótipo baseado na arquitetura de microfrontends utilizando o framework Single-SPA. Este protótipo foi apresentado visando uma melhor compreensão e também uma melhor visualização da diferença entre a utilização da arquitetura monolítica e arquitetura de microfrontends no desenvolvimento de software no mundo contemporâneo.

É nítido o tanto de complexidade que é inserida em um projeto envolvendo a arquitetura de microfrontends, onde qualquer simples projeto desenvolvido torna-se um pouco mais confuso e a curva de aprendizado aumenta bastante, porém isso não é algo que deve desmotivar o desenvolvimento de software utilizando esta

arquitetura, pois ela também traz inúmeros benefícios, e neste ponto, não difere da arquitetura monolítica que também possui seus prós e contras.

6 AVALIAÇÃO COMPARATIVA ENTRE MICROFRONTEND E MONÓLITO

Nesta seção será apresentada uma análise comparativa através de um quadro comparativo visando aspectos de eficiência, facilidade de manutenção, escalabilidade e suas respectivas vantagens e desvantagens, trazendo uma visão mais detalhada sobre cada um desses pontos relacionados com essas duas arquiteturas muito importantes para o desenvolvimento de software.

Quadro 1 - Vantagens

QUESITOS	ARQUITETURA MONOLÍTICA	ARQUITETURA DE MICROFRONTENDS
EFICIÊNCIA	As operações podem ser mais eficientes, pois tudo está integrado em um único contexto. Menos complexidade na comunicação interna. Desenvolvimento e manutenção podem ser mais diretos em projetos menores.	Eficiência no desenvolvimento, pois diferentes equipes podem trabalhar de forma independente em microfrontends específicos. Reusabilidade de componentes em diferentes partes da aplicação.
FACILIDADE DE MANUTENÇÃO	Manutenção mais direta em projetos menores. Mudanças podem ser aplicadas globalmente.	Facilita a manutenção, pois diferentes partes da aplicação podem ser atualizadas e mantidas de forma independente. Problemas em um microfrontend não afetam globalmente a aplicação

ESCALABILIDADE	Escalabilidade vertical é mais direta, adicionando mais recursos ao servidor existente. Menos complexidade na gestão de comunicação, pois todos os componentes estão no mesmo contexto.	Facilita a escalabilidade horizontal, pois diferentes partes da aplicação podem ser escaladas independentemente. Novas funcionalidades podem ser desenvolvidas e implementadas em módulos separados.
-----------------------	---	--

Fonte: Pelo Autor (2023)

Ao comparar as abordagens de microfrontend e monolito visando as vantagens em termos de escalabilidade, facilidade de manutenção e eficiência, é evidente que ambas apresentam vantagens únicas. Enquanto os microfrontends destacam-se pela modularidade e flexibilidade, os monólitos destacam-se por eficiência no desenvolvimento, escalabilidade vertical e menor complexidade durante a fase inicial do projeto.

Quadro 2 - Desvantagens

QUESITOS	ARQUITETURA MONOLÍTICA	ARQUITETURA DE MICROFRONTENDS
EFICIÊNCIA	Maior complexidade e potencial para ineficiência à medida que a aplicação cresce. Atualizações podem afetar a aplicação como um todo, mesmo partes não relacionadas.	Introduz um certo overhead na comunicação entre os microfrontends. Pode exigir esforço adicional para garantir a consistência da interface do usuário.

FACILIDADE DE MANUTENÇÃO	Dificuldade de manter e atualizar partes específicas sem afetar a aplicação como um todo. A complexidade aumenta à medida que a aplicação cresce.	Pode ser desafiador garantir consistência na interface do usuário. A necessidade de gerenciar várias dependências pode aumentar a complexidade.
ESCALABILIDADE	Pode ser difícil escalar partes específicas da aplicação sem aumentar os recursos globais do servidor. Mudanças e atualizações podem exigir a implantação de toda a aplicação.	Introduz complexidade na gestão de comunicação entre os microfrontends, e o overhead de carregamento dinâmico pode impactar o desempenho em determinados casos.

Fonte: Pelo Autor (2023)

Ao finalizar a análise comparativa entre microfrontends e monólitos, torna-se evidente que ambas as arquiteturas apresentam seus desafios e limitações significativas em relação à escalabilidade, facilidade de manutenção e eficiência. Esses aspectos críticos devem ser ponderados cuidadosamente pelos desenvolvedores e gestores de TI ao decidir pela adoção de uma dessas abordagens arquiteturais.

Em suma, tanto microfrontends quanto monólitos apresentam desafios substanciais em relação à escalabilidade, facilidade de manutenção e eficiência. A escolha entre essas arquiteturas deve ser cuidadosamente alinhada aos requisitos específicos do projeto e da organização, reconhecendo que cada abordagem possui suas vantagens e desvantagens inerentes.

7 CONCLUSÃO

Uma análise comparativa entre monólitos e microfrontends revela nuances essenciais relacionadas à escalabilidade, eficiência e facilidade de manutenção em arquiteturas de software. Ambas as abordagens apresentam vantagens e desafios, e a escolha entre elas depende dos requisitos específicos do projeto.

No aspecto de escalabilidade, os microfrontends destacam-se por sua capacidade de escalar de maneira mais eficiente. Ao adotar uma abordagem de microfrontends, é possível dividir a interface do usuário em módulos independentes, facilitando a implementação de atualizações e o dimensionamento horizontal. Cada microfrontend pode ser desenvolvido, testado e implantado independentemente, permitindo uma escalabilidade mais granular e flexível em comparação com os monólitos.

Os monólitos, por outro lado, podem encontrar desafios quando se trata de escalabilidade. O aumento da carga pode resultar em requisitos adicionais de recursos, e a escala horizontal pode ser mais complexa devido à natureza integrada do sistema.

No que diz respeito à eficiência, os microfrontends proporcionam uma vantagem notável. Com a capacidade de desenvolver, testar e implantar módulos de front-end de forma independente, as equipes podem adotar tecnologias mais recentes e otimizar cada microfrontend para sua finalidade específica. Isso permite um melhor aproveitamento de recursos e um desempenho mais eficiente.

Os monólitos podem enfrentar desafios relacionados à eficiência, pois todas as partes da aplicação compartilham o mesmo contexto e dependem da mesma base de código. Isso pode resultar em desafios ao implementar atualizações específicas ou ao otimizar partes individuais do sistema.

A facilidade de manutenção é um ponto crucial na escolha entre monólitos e microfrontends. Os microfrontends, ao permitirem o desenvolvimento independente de módulos, facilitam a manutenção e a evolução contínua. As equipes podem atualizar, corrigir bugs e expandir funcionalidades sem afetar o restante do sistema, facilitando a manutenção a longo prazo.

Os monólitos podem ser mais desafiadores nesse aspecto, especialmente à medida que o código cresce. Modificações em uma parte do sistema podem ter efeitos colaterais inesperados em outras áreas, e a complexidade geral pode tornar a manutenção mais difícil.

Em última análise, a escolha entre monólitos e microfrontends depende das necessidades específicas do projeto e das preferências da equipe de desenvolvimento. Os microfrontends oferecem escalabilidade, eficiência e facilidade de manutenção notáveis, especialmente em projetos grandes e complexos. No entanto, os monólitos podem ser mais adequados para aplicações menores, onde a simplicidade e a coesão são mais valorizadas do que a escalabilidade granular.

Independentemente da escolha, é crucial adotar práticas sólidas de desenvolvimento, testes e integração contínua para garantir um ciclo de vida de desenvolvimento suave e um software robusto e de alta qualidade.

7.1 TRABALHOS FUTUROS

Para ampliar a compreensão e relevância das conclusões apresentadas nesta pesquisa, sugere-se a realização de trabalhos futuros que se aprofundem em áreas específicas e considerem cenários mais complexos. Dentre as possíveis direções para estudos subsequentes, destacam-se:

Validação em Ambientes Produtivos Reais:

- Conduzir análises adicionais em ambientes de produção de organizações de médio e grande porte para validar as descobertas desta pesquisa.
- Investigar como as arquiteturas de microfrontend e monolito se comportam em situações de carga intensiva e sob demandas variáveis em tempo real.

Avaliação de Desempenho em Projetos de Grande Escala:

- Estender a análise comparativa para projetos de maior escala, levando em consideração a complexidade de sistemas distribuídos em organizações robustas.
- Examinar o desempenho das arquiteturas em contextos que envolvam integração com sistemas legados e serviços externos.

Inclusão de Aspectos de Segurança:

- Introduzir uma análise mais aprofundada sobre as implicações de segurança associadas a cada arquitetura, considerando as vulnerabilidades específicas de microfrontend e monolito.

REFERÊNCIAS

ANGULAR. **Angular**. Disponível em: <<https://angular.io/>>. Acesso em: 27 nov. 2023.

ARAUJO, J. **Atomic Design: o que é, como surgiu e sua importância para a criação do Design System**. Disponível em: <<https://medium.com/pretux/atomic-design-o-que-%C3%A9-como-surgiu-e-sua-import%C3%A2ncia-para-a-cria%C3%A7%C3%A3o-do-design-system-e3ac7b5aca2c>>.

AWS. **Monolítico x microsserviços — Diferença entre arquiteturas de desenvolvimento de software — AWS**. Disponível em: <<https://aws.amazon.com/pt/compare/the-difference-between-monolithic-and-microservices-architecture/>>. Acesso em: 10 set. 2023.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. [s.l.] Addison-Wesley, 2012.

CARLA, L. **Micro Frontends**. Disponível em: <<https://medium.com/@lorenamelor/micro-frontends-ac0e5d87582a>>. Acesso em: 27 nov. 2023.

DIJKSTRA, E. W. The structure of the “THE”-multiprogramming system. **Communications of the ACM**, v. 11, n. 5, p. 341–346, maio 1968.

FOWLER, M. **bliki: MonolithFirst**. Disponível em: <<https://martinfowler.com/bliki/MonolithFirst.html>>. Acesso em: 30 out. 2023.

FOWLER, M. **Software Architecture Guide**. Disponível em: <<https://martinfowler.com/architecture/>>. Acesso em: 2 set. 2023.

FROST, B. **Atomic Design Methodology | Atomic Design by Brad Frost**. Disponível em: <<https://atomicdesign.bradfrost.com/chapter-2/>>. Acesso em: 29 nov. 2023.

GEERS, M. **Micro Frontends - extending the microservice idea to frontend development**. Disponível em: <<https://micro-frontends.org/>>. Acesso em: 13 nov. 2023.

GOMES, T. H. F. **Refatoração de arquiteturas monolíticas em microserviços no contexto de desenvolvimento de software global.** Disponível em: <<https://repository.ufrpe.br/handle/123456789/1924>>. Acesso em: 15 set. 2023.

GOMES, V. **Micro-frontend o que é isso?** Disponível em: <<https://medium.com/iclubs/o-que-s%C3%A3o-micro-frontends-5e83b91ad45d>>. Acesso em: 13 nov. 2023.

GONÇALVES, M. M. **Distribuindo Monólitos: SOA, Macro, Mini, Micro e Nano Serviços.** Disponível em: <<https://medium.com/@marcelomg21/distribuindo-mon%C3%B3litos-soa-macro-mini-micro-e-nano-servi%C3%A7os-8057b9d4cf81>>. Acesso em: 8 out. 2023.

HAWKINS, T. **Developing and Deploying Micro-Frontends with Single-Spa.** Disponível em: <<https://medium.com/swlh/developing-and-deploying-micro-frontends-with-single-spa-c8b49f2a1b1d>>. Acesso em: 13 nov. 2023.

JACKSON, C. **Micro Frontends.** Disponível em: <<https://martinfowler.com/articles/micro-frontends.html>>. Acesso em: 13 nov. 2023.

LEWIS, J.; FOWLER, M. **Microservices.** Disponível em: <<https://martinfowler.com/articles/microservices.html>>. Acesso em: 10 set. 2023.

LYNCAS. **Escalabilidade em TI: O que é e por que implementar?** Disponível em: <<https://lyncas.net/escalabilidade-em-ti-o-que-e-e-quais-as-vantagens/>>. Acesso em: 12 nov. 2023.

MARQUES, R. **Arquitetura de microsserviços ou monolítica: qual devo usar?** Disponível em: <<https://www.cedrotech.com/blog/arquitetura-de-microsservicos-ou-monolitica/>>. Acesso em: 30 dez. 2023.

META. **useState – React.** Disponível em: <<https://pt-br.react.dev/reference/react/useState>>. Acesso em: 27 nov. 2023.

MODULE FEDERATION. **What is Module Federation? :: Module Federation Documentation.** Disponível em: <<https://module-federation.io/docs/en/mf-docs/0.2/getting-started/>>. Acesso em: 27 nov. 2023.

MOORE, G. E. **Cramming more components onto integrated circuits.** New York: Mcgraw-Hill, 1965.

NCB{CODE}. **Componentes React: Dumb vs Smart.** Disponível em: <<https://medium.com/@ncbcode/componentes-react-dumb-vs-smart-830a7227e7d3>>. Acesso em: 29 nov. 2023.

NEWMAN, S. **Building microservices.** Sebastopol, Ca: O'reilly Media, 2015. p. 20, 33

NEWMAN, S. **Monolith to microservices : evolutionary patterns to transform your monolith.** Sebastopol, Ca: O'reilly Media, Inc, 2019.

PRESSMAN, R. S. **Engenharia de software.** 3. ed. [s.l.] Pearson, 1995.

ROCHA, W. N. **Micro Frontend Architecture: Vantagens e Desvantagens.** Disponível em: <<https://medium.com/@wr.ergon/micro-frontend-architecture-vantagens-e-desvantagens-13663b2eff57>>. Acesso em: 30 dez. 2023.

SANTOS, P. **Arquitetura de Microserviços x Monolítica: Vantagens e desvantagens.** Disponível em: <<https://arphoenix.com.br/quais-as-diferencas-entre-arquitetura-monolitica-e-microservicos-suas-vantagens-e-desvantagens/>>. Acesso em: 15 set. 2023.

SECCO, A. **Monolitos não escalam?** Disponível em: <<https://andresecco.com.br/2019/09/monolitos-nao-escalam/>>. Acesso em: 30 out. 2023.

SILVA, W. **Sistemas Monolíticos: O que são, vantagens e desvantagens.** Disponível em: <<https://www.cherrypickintegration.com/sistemas-monoliticos/>>. Acesso em: 12 nov. 2023.

SINGLE SPA. **Getting Started with single-spa | single-spa.** Disponível em: <<https://single-spa.js.org/docs/getting-started-overview>>. Acesso em: 13 nov. 2023.

SINGLE-SPA. **The Recommended Setup | single-spa.** Disponível em: <<https://single-spa.js.org/docs/recommended-setup#utility-modules-styleguide-api-etc>>.

Acesso em: 13 nov. 2023.

SOFTDESIGN; CAMELO, R. **Monólitos, Serviços e Microserviços: impactos nos negócios.** Disponível em:

<<https://softdesign.com.br/blog/monolitos-servicos-e-microservicos-impactos-nos-negocios/>>. Acesso em: 11 nov. 2023.

SOFTWARE, O. **Micro Serviços: qual a diferença para o monolito?** Disponível em:

<<https://www.opus-software.com.br/insights/micro-servicos/>>. Acesso em: 30 out. 2023.

TERRA, R.; VALENTE, M. T. **Verificação Estática de Arquiteturas de Software utilizando Restrições de Dependência.** Disponível em:

<<https://sol.sbc.org.br/index.php/sbcars/article/view/24715>>. Acesso em: 2 set. 2023.

UZEDA, D. **Microserviços vs arquitetura monolítica: como alcançar a eficiência operacional?** - HostDime. Disponível em:

<<https://www.hostdime.com.br/microservicos-vs-arquitetura-monolitica-como-alcancar-a-eficiencia-operacional/>>. Acesso em: 12 nov. 2023.

WEBPACK. **Module Federation.** Disponível em:

<<https://webpack.js.org/concepts/module-federation/>>. Acesso em: 27 nov. 2023.

YOU, E. **Vue.js.** Disponível em: <<https://vuejs.org/>>. Acesso em: 27 nov. 2023.

ZUSTAND. **Zustand Documentation.** Disponível em:

<<https://docs.pmnd.rs/zustand/getting-started/introduction>>. Acesso em: 5 dez. 2023.

APÊNDICE A:

react-todo-util-state

```

import { create } from "zustand";
import { combine } from 'zustand/middleware'

export type TaskProperties = {
  id: number;
  title: string;
  completed: boolean;
  description: string;
  tasks: TaskProperties[];
}

export type TaskMethods = {
  addTask: (task: TaskProperties) => void;
  removeTask: (id: number) => void;
  updateTask: (id: number) => void;
}

const store = combine<TaskProperties, TaskMethods>({
  tasks: [],
  completed: false,
  description: "",
  id: 0,
  title: ""
}, (set) => ({
  addTask: (task) => set((state) => ({ tasks: [...state.tasks, task] })),
  removeTask: (id) => set((state) => ({ tasks: state.tasks.filter((task) => task.id !== id) })),
  updateTask: (id) => set((state) =>
    ({ tasks: state.tasks.map((task) => task.id === id ? { ...task, completed: !task.completed } : task)
      }))
}));

export const useStore = create(store);

```