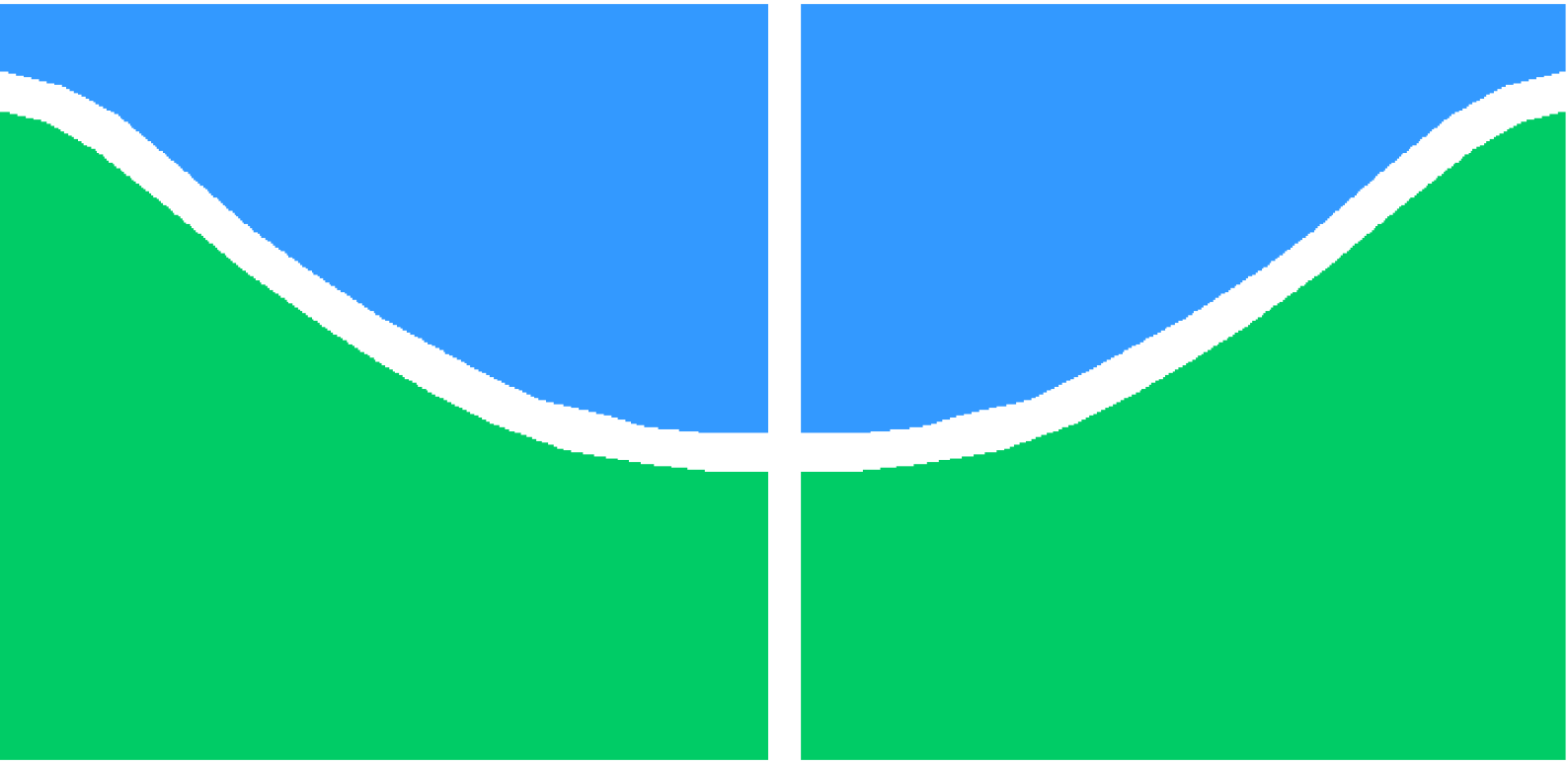


Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Microfrontends com Web Components e WebPack: Uma abordagem de implementação agnóstica em relação ao framework

Autor: Rhuan Carlos Pereira de Queiroz
Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF
2023

Rhuan Carlos Pereira de Queiroz

Microfrontends com Web Components e WebPack: Uma abordagem de implementação agnóstica em relação ao framework

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2023

Rhuan Carlos Pereira de Queiroz

Microfrontends com Web Components e WebPack: Uma abordagem de implementação agnóstica em relação ao framework/ Rhuan Carlos Pereira de Queiroz. – Brasília, DF, 2023-

90 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2023.

1. Microfrontends. 2. Arquitetura de Software. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Microfrontends com Web Components e WebPack: Uma abordagem de implementação agnóstica em relação ao framework

CDU 02:141:005.6

Rhuan Carlos Pereira de Queiroz

Microfrontends com Web Components e WebPack: Uma abordagem de implementação agnóstica em relação ao framework

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 24 de fevereiro de 2023:

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dr. Fernando Willian Cruz
Convidado 1

Profa. Dra. Milene Serrano
Convidado 2

Brasília, DF
2023

Este trabalho é dedicado a todos que contribuíram,
positivamente, em minha vida, a fim de chegar onde estou hoje.
Cada ensinamento, risada, dica e expressão de afeto.

Agradecimentos

Agradeço, primeiramente, a Deus, por ter me ajudado até aqui.

Agradeço aos meus pais, por todo apoio financeiro, emocional e afetivo que me deram durante toda minha vida.

As minhas irmãs, por fazerem meus dias mais divertidos.

Ao meu orientador, por todo apoio e aceitação do tema proposto, bem como por toda ajuda e orientação.

Aos meus amigos, colegas de graduação e de trabalho, por todo apoio, ajuda e conhecimento compartilhado. Além dos bons momentos de diversão.

"There are only two hard things in Computer Science:
cache invalidation and naming things."
(Phil Karlton)

Resumo

A arquitetura de software se propõe a tomar decisões focadas em um objetivo. Sistemas podem implementar *microfrontends*, uma arquitetura que tem como um dos seus objetivos permitir que times diferentes desenvolvam aplicações de forma paralela. Times que desejam utilizar tecnologias diferentes enfrentam o problema da incompatibilidade e da integração dos diferentes componentes. Além do mais, existem diferentes abordagens de implementar esta arquitetura, e não há um padrão definido acerca de qual usar, o que agrega mais complexidade. Este trabalho define uma abordagem combinada utilizando *Web Components* e *WebPack Module Federation*, que tem como principal objetivo ser agnóstica em relação ao *framework*. Para isso, foi realizada uma pesquisa exploratória aplicada, com o intuito de definir essa abordagem de implementação, como também, explorar a abordagem em uma situação com os três principais frameworks de frontend: *React*, *Angular* e *Vue.js*. Visto que isolar as folhas de estilo CSS é um problema conhecido na comunidade, também foi explorado a funcionalidade de *shadow* DOM dos *Web Components*. Por fim, foi possível concluir que a abordagem combinada definida é funcional e pode ser utilizada em aplicações que utilizam a API nativa do DOM ou *frameworks* que fazem uso da mesma.

Palavras-chave: Microfrontends. Arquitetura de Software. Aplicações Web.

Abstract

Software Architecture propose itself to make decisions focused on a objective. Systems can implement microfrontends, an architecture that has as one of its main purposes to enable different teams to develop applications in parallel to each other. Teams that wish to use different technologies face the problems of incompatibility and integration between an ample range of components. Besides, there are diverse approaches in this architecture implementation, which does not have a established pattern of how to be used and thus complicating the process even more. The aim of this work is to define a combined approach using Web Components and WebPack Module Federation, in order to turn this methodology framework-agnostic. To achieve this, a exploratory research was applied to define this implementation approach, as well as to explore other scenary situations, with the other three mains frontend frameworks: React, Angular and Vue.js. Since isolating Cascading Style Sheets (CSS) is a known problem in the community, the shadow DOM functionality of the web components it was also explored. Finally, it was possible to conclude that the combined approach defined is functional and can be used in applications that use the native API of the DOM or frameworks that use it too.

Key-words: Microfrontends. Software Architecture. Web Applications.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

API	*Application Programming Interface*
CDN	*Content Delivery Network*
CSS	*Cascading Style Sheets*
DOM	*Document Object Model*
DNS	*Domain Name System*
ES6	*ECMAScript 6*
ESI	*Edge Side Includes*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
MPA	*Multi Page Application*
PoC	*Proof of Concept*
SPA	*Single Page Application*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*

Sumário

1 Introdução

A arquitetura de software de um sistema pode ser definida como o conjunto de estruturas necessárias para definir o sistema, além de seus elementos, propriedades e relacionamentos. (BASS; CLEMENTS; KAZMAN, 2012) Existem várias definições para o termo arquitetura de software, mas dentre elas, todas consideram que há um conjunto de decisões que devem ser tomadas para definir o sistema levando em conta objetivos, restrições e atributos de qualidade.

Existem várias arquiteturas no mercado e na comunidade de software, dentre as que utilizam o padrão Cliente-Servidor, ou seja, o cliente envia uma solicitação para o servidor, que responde a essa solicitação, pode-se citar a de microsserviços. Esta arquitetura é uma das mais comentadas nos últimos anos, ainda mais com o crescimento de aplicações em nuvem.

Os microsserviços são conhecidos pela separação do problema em vários componentes, cada um com seu propósito, tecnologia e funcionalidades, os times de desenvolvimento não precisam ser os mesmos, o ambiente de implantação também pode ser totalmente diferente. É possível também escalar cada componente, que recebe o nome de serviço, de maneira independente dado as necessidades do sistema, número de usuários, tempo de computação, etc. (ALMEIDA, 2021)

Embora a arquitetura associada ao *backend* de um sistema tenha sido subdividida em vários componentes, em muitos projetos de software, o *frontend* permaneceu como um monólito. Na última década surgiu uma arquitetura que é análoga a de microsserviços, ou seja, a camada de apresentação passa a ser dividida em componentes, permitindo tecnologias diferentes, equipes de desenvolvimento diferentes, e o principal, propósitos diferentes.

Essa arquitetura é chamada de *microfrontends*, há na literatura e no mercado muitas abordagens que implementam essa arquitetura, ela compartilha algumas semelhanças com a de microsserviços, não há uma padronização do tamanho do componente, não há também um padrão de como esses componentes são separados. Uma das grandes distinções da arquitetura de *microfrontends* é que no final todos os componentes são integrados em um componente central, ou seja, o usuário final pode não saber se a aplicação utiliza essa arquitetura ou não.

Um dos principais problemas que essa arquitetura se propõe a resolver é permitir que vários times de desenvolvimento possam trabalhar em paralelo e descentralizados, ou seja, não há necessidade que todos os times utilizem o mesmo repositório de código, nem que todos os times utilizem a mesma tecnologia. Combinar diferentes tecnologias pode

não ser uma tarefa fácil, este trabalho se propõe a apresentar uma abordagem combinada que possa facilitar essa tarefa.

1.1 Justificativa

Há na literatura muitas abordagens para a implementação da arquitetura de *microfrontends*, dentre elas é possível listar: *iFrames*, *JavaScript*, *Web Components* e *Module Federation*. Essas abordagens possuem vantagens e desvantagens, além do mais, algumas delas resolvem facilmente problemas encontrados durante a implementação de *microfrontends*.

Dentre essas abordagens, existem as que são mais simples de serem implementadas, como é o caso da abordagem com *iFrames*, enquanto outras abordagens, como a de *Module Federation*, são mais complexas de serem implementadas, mas que permitem carregamento dinâmico de módulos, compartilhamento de dependências e outras funcionalidades.

Já a abordagem com *Web Components* é uma abordagem muito próxima da abordagem com *JavaScript*, porém com a vantagem de ser mais simples de ser implementada, além de utilizar a API nativa de elementos HTML. Bem como, as especificações de *Web Components* que podem evitar conflitos de estilos e permitem algum nível de comunicação entre os componentes.

Algumas dessas abordagens podem também ser combinadas em algum nível, de modo que com isso é possível obter novas abordagens e combinando seus benefícios. Com esse trabalho, vai ser possível explorar e descrever uma dessas combinações, essa pode vir a ser adotada como abordagem principal em projetos futuros da comunidade de engenharia de software.

Um dos principais problemas que esta abordagem combinada se propõe a resolver é ser agnóstica ao *framework*, ou seja, não depender de um *framework* específico para a implementação de *microfrontends*. Carregando consigo a possibilidade de trazer mais autonomia e independência para os times de desenvolvimento, semelhante ao que já é possível com a arquitetura de microsserviços.

Além disso, foi possível observar no repositório de exemplos do *plugin Module Federation* uma preocupação da comunidade com relação a integração de *microfrontends* feitos em *frameworks* diferentes. Isso pode ser considerado um ponto positivo, visto que uma abordagem agnóstica ao *framework* pode facilitar ainda mais esse tipo de integração.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho tem como objetivo geral explorar a abordagem combinada de implementação de *microfrontends* utilizando *Web Components* e *Module Federation*, com foco na criação de uma interface que seja agnóstica em relação ao *framework frontend* que está sendo utilizado em ambos os lados.

1.2.2 Objetivos Específicos

Com foco no objetivo geral é possível listar os seguintes objetivos específicos:

- Definir a interface de abstração com o que é necessário para a sua utilização;
- Compor uma página Web utilizando a interface e três *frameworks frontend*, sendo eles: *React*, *Angular* e *Vue.js*;
- Aplicar *Shadow* DOM em conjunto com a interface para obter encapsulamento dos *microfrontends*;
- Desenvolver nativamente com a interface e incorporar na página criada anteriormente.

2 Fundamentação Teórica

Ao longo deste capítulo serão explorados assuntos necessários para o embasamento do trabalho, algumas definições teóricas importantes, tais como conceitos de Web, já que é o conceito principal que permeia este trabalho. Os assuntos a serem explorados são:

1. Fundamentos de Web;
2. Arquiteturas de aplicações Web;
3. *Frontend*;
4. *Microfrontends*;

2.1 Fundamentos de Web

Esta seção tem como objetivo abordar alguns tópicos gerais sobre como as aplicações Web funcionam e são construídas. Os assuntos variam da forma como elas são estruturadas, até como os componentes que as compõem se comunicam.

2.1.1 Comunicação na Web

A Web é baseada em acessar arquivos remotamente, ou seja, acessar arquivos de um sistema de arquivos de um servidor. A maneira mais simples de se referir a um documento na Web é uma referência chamada URL (*Uniform Resource Locator*), essa referência especifica onde um documento está localizado, combinando geralmente o DNS (*Domain Name System*) do servidor com o nome do arquivo. (TANENBAUM; STEEN, 2013)

Essa referência também contém o protocolo para transferir o arquivo pela rede, os mais comuns e utilizados são HTTP e HTTPS, o segundo é idêntico ao primeiro, porém, sobre uma camada de segurança. O protocolo HTTP suporta uma grande variedade de tipos de arquivos, os seis principais tipos de acordo com Melnikov e Kucherawy (s.d.) são: texto, fonte, imagem, áudio, vídeo e aplicação.

A aplicação central em todo esse processo é o navegador, esse que é responsável por carregar os documentos, efetuar a comunicação com o servidor, efetuar a renderização dos documentos e de outros arquivos necessários. Bem como, são eles quem executam as aplicações Web modernas, graças a interpretação de JavaScript.

2.1.2 Document Object Model

O DOM (*Document Object Model*) é uma interface de programação (API) que permite que os documentos sejam manipulados. Essa interface é baseada em objetos, que são representações de elementos do documento, como por exemplo, tags HTML. Esses objetos são manipulados por meio de funções e propriedades, que são definidas pela interface. (HéGARET; W3C, 2004)

2.1.3 Multi Page Application

Trata-se do modelo referido como tradicional de aplicações Web, comumente referida pelo acrônimo MPA, onde as páginas são completamente recarregadas sempre que há uma interação do usuário, a interação desencadeia uma troca de dados entre o servidor, essas respostas vindas do servidor são páginas novas com os dados. (ALMEIDA, 2021)

2.1.4 Single Page Application

Em oposição ao modelo tradicional, as aplicações que são construídas como *Single Page Application* (SPA) são aplicações que são compostas de uma única página, ou seja, somente uma página é carregada do servidor. As interações com o usuário são realizadas através de novas requisições e alterações diretas no DOM. (ALMEIDA, 2021)

Por conta da particularidade, quando comparado com o outro modelo, este modelo é facilmente identificado, as páginas serem recarregadas a cada interação do usuário são nítidas quando comparadas com as aplicações SPA.

Essas aplicações são possíveis graças aos *frameworks* que abstraem as alterações no DOM, como o *React*, o *Angular* e o *Vue*, por exemplo. De outro modo, a utilização de *frameworks* não é mandatória, visto que é possível implementar as alterações de forma nativa utilizando apenas JavaScript.

2.2 Arquiteturas de Aplicações Web

Essa seção é dedicada a explorar e descrever algumas das arquiteturas que são comumente utilizadas em aplicações Web. De certa forma, algumas delas são importantes para entender o problema a que *microfrontends* se propõe a resolver. Além disso, é possível notar uma evolução das arquiteturas, que começa com a monolítica e, após algumas modificações, se torna baseada em microsserviços.

2.2.1 Cliente e Servidor

Essa arquitetura é a base para as aplicações Web, pois se comportam como um sistema distribuído, sendo possível definir dois grupos de componentes: o cliente e o servidor. O servidor é um processo implementando um serviço, e o cliente é um processo que requisita um serviço de um servidor por meio de uma requisição e espera pela resposta. (TANENBAUM; STEEN, 2013)

2.2.2 Arquitetura em Camadas

Um dos padrões de arquitetura muito utilizado em aplicações Web é a arquitetura em camadas, segundo Bass, Clements e Kazman (2012), essa arquitetura é muito utilizada quando as relações entre os componentes são estritamente uni-direcionais, sendo assim, uma estrutura em camadas, onde uma camada só pode utilizar as camadas abaixo dela.

As aplicações Web geralmente possuem componentes ou módulos que podem ser separados em três camadas: a camada de apresentação, a camada de aplicação ou negócios e a camada de dados ou persistência. A camada de apresentação é responsável por exibir os dados para o usuário, a camada de aplicação é responsável por processar os dados e a camada de dados é responsável por armazenar os dados.

2.2.3 Monólito

Uma arquitetura clássica, marcada pela simplicidade e facilidade de implementação, é a monolítica, uma aplicação em camadas que compartilha desde a base de código até os recursos de computação, geralmente constituída de um único processo. (ALMEIDA, 2021)

As aplicações que seguem essa arquitetura, comumente, são *websites* MPA, pois como é organizada em um único processo, toda interação do usuário com a interface resultará em uma requisição e uma nova página será carregada.

2.2.4 Backend e Frontend

A seguinte arquitetura é uma evolução da monolítica, a principal diferença reside no fato de que nessa arquitetura não há apenas um componente, mas no mínimo três, sendo o *backend*, o *frontend* e o cliente. Em ambos o cliente mantém suas responsabilidades, já o monólito é separado em dois componentes, o *backend* e o *frontend*.

Essa alteração faz com que surjam duas bases de código, e análogo à arquitetura em camadas, o *frontend* é o que se encontra na camada de interface, e o *backend* engloba as camadas restantes, ou seja, as camadas de negócios e de persistência.

Nessa arquitetura se populariza o *frontend* como o componente que é responsável pela interface e interação com o usuário. As aplicações passam a adotar em grande maioria o estilo de páginas únicas (SPA). Com isso, os *frameworks* de *frontend* passam a ser amplamente utilizados.

Já o *backend*, segue com parte das responsabilidades que já haviam na monolítica, ou seja, é responsável pela lógica de negócios, manipulação dos dados, validações dos dados. O protocolo de comunicação entre o *backend* e o *frontend* mais utilizado é o HTTPS, que é baseado no protocolo mais utilizado em toda a Web o HTTP.(TANENBAUM; STEEN, 2013)

Uma das principais vantagens é que o desenvolvimento das bases de código é independente pois elas estão separadas, na prática, isso permite que novas funcionalidades sejam desenvolvidas separadamente. De modo semelhante, o *backend* estando separado do *frontend*, permite também o reúso do *backend* para outras aplicações.(ALMEIDA, 2021)

2.2.5 Microsserviços

A seguinte arquitetura é a primeira a criar um grande contraste com a monolítica, segundo Fowler (2014), embora não haja uma definição formal de microsserviços, no geral eles compartilham características. A primeira grande característica é que a lógica é dividida em vários processos, estes que se tornam componentes.

A segunda característica, embora seja quase uma consequência da primeira, é que os componentes por serem independentes, podem ser desenvolvidos de formas independentes, com tecnologias e implantações diferentes (ALMEIDA, 2021).

2.3 Frontend

Ao longo dessa seção serão descritas tópicos e assuntos que podem ser utilizados junto com os *frontends*, alguns desses assuntos se referem a *frameworks* utilizados para desenvolver os componentes dessa camada, como o *React*, o *Angular* e o *Vue*, por exemplo. Também inclusas ferramentas auxiliares como o *Webpack*, além dos *Web Components* que podem ser utilizados em conjunto para o desenvolvimento dessas aplicações.

2.3.1 Webpack

Segundo a documentação do *Webpack*, é um empacotador de código estático, que é utilizado em aplicações JavaScript modernas (WEBPACK. . . , 2022). Esta dependência é padrão nos principais *frameworks*, e é responsável por gerar o código final que será carregado pelo navegador, com ela é possível resolver corretamente as dependências a outra bibliotecas e incluí-las na versão final.

Suas principais responsabilidades são: definir quais serão os arquivos na versão de distribuição, permitir que diferentes tipos de recursos: imagens, fontes, e CSS sejam inclusos como arquivos estáticos, permitindo que o navegador carregue os recursos de forma mais eficiente.

Além do mais, desempenha um papel importante na performance das aplicações Web, visto que com ele é possível reduzir as dependências inutilizadas, o que consequentemente produz aplicações mais leves. Isso aliado com os *plugins* que permitem combinações ainda mais variadas, aumentando em muito a utilidade do *Webpack*.

2.3.2 Web Components

De acordo com a própria organização de *Web Components*, se define como um conjunto de APIs Web que permitem criar novas, customizadas, reusáveis e encapsuladas tags HTML para uso em aplicações Web. Esses componentes funcionam em diferentes navegadores modernos, e podem ser utilizados com quaisquer bibliotecas ou *frameworks* que funcionem com HTML (WEB. . . , 2022).

A maior vantagem deles é serem baseados nas especificações e padronizações da própria W3C (*World Wide Web Consortium*), ou seja, o que é possível fazer com HTML é possível ainda mais com *Web Components*. Sendo que eles são baseados em quatro principais especificações: *Custom Elements*, *Shadow DOM*, *ES Modules* e *HTML Template*.

A primeira dessas especificações é a *Custom Elements*, que é uma especificação que define a API que permite a criação de componentes customizados, essa API é utilizada para que os novos componentes sejam registrados para uso no DOM. A segunda é a *Shadow DOM*, uma API que permite encapsular os estilos de um componente em uma versão isolada do DOM, é imprescindível visto que a estilização em aplicações Web é em escopo global.

2.4 Microfrontends

A seguinte arquitetura, que é o foco principal deste trabalho, foi criada depois da arquitetura de microsserviços. De forma estrutural, embora a camada do *backend* tenha sido dividida em serviços menores, o *frontend* permaneceu como um monólito. Esses monólitos se tornam complicados para desenvolver e manter, pois ao longo do tempo a base de código tendem a ficar muito grande e complexa.

Surgiu então a arquitetura de *microfrontends*, que consiste principalmente em decompor o monólito de *frontend* em vários componentes menores, que são chamados de *microfrontends*. Jackson (2019) define essa técnica como: "Um estilo arquitetural onde aplicações *frontend* com entregas independentes são compostas em um todo maior", ou

seja, cada *microfrontend* é um componente independente, cada um com seu próprio fluxo de desenvolvimento, testes e *deploy* (JACKSON, 2019).

Algumas literaturas se referem a essa técnica, como sendo microsserviços no lado do *frontend*, sendo assim compartilham características em comum, e muitas das vantagens. Algumas vantagens: implantações independentes, bases de código menores e times independentes. Silva (2021) lista algumas empresas no mercado que já utilizam essa técnica como : Spotify, Zalando, Dazn, Ikea, Facebook.

Os diferentes estilos arquiteturais resolvem problemas, mas são baseados em *trade-off*, ou seja, algum aspecto do software, algum atributo de qualidade pode ser prejudicado para atingir os benefícios almejados.

2.4.1 Vantagens

2.4.1.1 Times Verticais

Com a quebra do monolito do *frontend*, é possível organizar os times em equipes denominadas verticais, esse modelo se opõe ao modelo de times horizontais, onde cada time é composto por membros que compartilha habilidades técnicas em comum, como por exemplo, o time de *backend* tem domínio dos conceitos de construção de *backends*, acesso e manipulação em bases de dados. Já o time de *frontend* tem como foco o desenvolvimento de interfaces e estilização das páginas.

Um time vertical é um time capaz de agregar valor a uma funcionalidade do negócio de ponta a ponta, esses times tem posse de tudo que eles precisam para entregar valor aos usuários finais (JACKSON, 2019). Os times podem ser compostos por desenvolvedores *frontend*, *backend*, *designers*, arquitetos, analistas de testes e outros *stakeholders*, todos com um propósito comum, esses times multi-disciplinares, tem não só o benefício da escalabilidade, como também promovem a criação de soluções mais criativas, efetivas e voltadas para o usuário (KLIMM, 2021).

2.4.1.2 Implantações e Entregas Independentes

Os times por serem verticais podem desenvolver e entregar de forma autônoma, sem gerar impactos em outros times (SILVA, 2021). Essa característica é semelhante ao que ocorre com os microsserviços, onde cada um possui sua própria pipeline de desenvolvimento, testes e implantação (ALMEIDA, 2021).

É possível diminuir os riscos associados a uma entrega, assim como realizar entregas mais rapidamente e com menos dependências entre os times (JACKSON, 2019). Assim como, segundo Prajwal, Parekh e Shettar (2021), os times podem atingir o desenvolvimento de funcionalidades rapidamente seguindo uma arquitetura sem compartilhamento.

2.4.1.3 Agnóstico em relação à tecnologia

Os microsserviços introduzem essa abordagem, pois permitem que cada serviço seja desenvolvido em uma tecnologia diferente, de forma análoga, os *microfrontends* também seguem a mesma linha, é possível desenvolver as pequenas partes em tecnologias distintas. É possível que um time decida as tecnologias para desenvolver uma funcionalidade específica do negócio, outrossim, os times podem concordar em utilizar as mesmas boas práticas (KLIMM, 2021).

No entanto, mesmo com as tecnologias sendo diferentes, essas aplicações, por ainda serem aplicações Web, se baseiam em tecnologias, protocolos e padrões da Web. No que tange os *microfrontends*, esse agnosticismo se refere mais ao uso de diferentes *frameworks* de *frontend*, pré-processadores de CSS e paradigmas (KLIMM, 2021).

2.4.1.4 Isolamento de Falhas

Lidar com erros em tempo de execução é um problema já conhecido:

> Uma das vantagens importantes de *microfrontends* em relação aos monólitos, é que em caso de erro a aplicação toda não precisa ser desabilitada. É possível detectar em qual módulo ocorreu o erro e uma correção apropriada pode ser feita naquele módulo. (PRAJWAL; PAREKH; SHETTAR, 2021, tradução nossa) [1]

De modo semelhante, a falha em um *microfrontend* não vai afetar os outros, além de existir a possibilidade de exibir conteúdos alternativos para o usuário (SILVA, 2021). A depender do tipo de erro, existem outras maneiras de tratar o mesmo, um erro de conexão, pode ser resolvido por uma nova tentativa de conexão, ou por uma página de erro.

Por outro lado, como cada componente tem seu fluxo de implantação, é possível que cada parte da aplicação esteja geograficamente isolada das outras, ou seja, a depender do tipo de falha, ainda sim pode haver uma parte da aplicação que estará funcionando.

2.4.1.5 Reusabilidade

O conceito de fragmentos, descrito por Klimm (2021): são *microfrontends* autocontidos, e usáveis em outros *microfrontends*. É possível que os times criem *microfrontends* desse tipo, que implementam alguma funcionalidade específica, e que graças à arquitetura distribuída, possam ser reutilizados em outras aplicações. Dessa forma, é possível salvar tempo e esforço, ao invés de recriá-los (ALMEIDA, 2021).

[1] No original: One of the important advantages of micro-frontends over monoliths is that in case of any error, the entire application need not be turned down. It is possible to detect in which module the error has occurred, and an appropriate fix can be made to that specific module.

2.4.2 Desvantagens

2.4.2.1 Complexidade

No quesito complexidade, os *microfrontends* tem uma complexidade acima a de microsserviços, que por sua vez, é maior do que comparada aos monólitos. Silva (2021) cita alguns principais problemas encontrados, sendo eles: organizar os times, integrar as várias tecnologias, compor os módulos distribuídos e monitorá-los.

Embora os *microfrontends*, por serem menores, tendem a ser mais fáceis de manter, Klimm (2021) aborda que é introduzida uma complexidade na camada organizacional da arquitetura. Algo semelhante ocorre com os microsserviços, é necessário administrar e organizar: mais repositórios, ferramentas, *pipelines*, servidores e domínios (JACKSON, 2019).

2.4.2.2 Performance

Com o fato dos *microfrontends* serem agnósticos em relação aos *frameworks*, segundo Silva (2021), essa característica contribui para o aumento do *payload* e a latência dos recursos a serem baixados. Mesmo com estratégias de *cache*, o primeiro carregamento da página ainda é muito lento quando comparado com outras abordagens (PRAJWAL; PAREKH; SHETTAR, 2021).

Existem técnicas que podem ser usadas para melhorar o desempenho. Almeida (2021) descreve uma abordagem para isso que é utilizar apenas um *microfrontend* por página. Com esse tipo de abordagem, o navegador carrega somente o código necessário para aquela página. Da mesma maneira, é possível conceber várias páginas que utilizam componentes em comum, por exemplo, um cabeçalho e um rodapé, embora o usuário navegue por novas páginas, a estrutura base da aplicação permanece a mesma.

As abordagens de implementação de *microfrontends* abordam esse problema de maneira diferente, em algumas delas é possível utilizar técnicas mais eficazes para otimizar o desempenho, como é o caso do *Webpack*, que permite que código seja carregado sob demanda. Esse problema de performance requer ainda mais atenção quando se possui um público que utiliza dispositivos móveis, visto que as velocidades de conexão nesses dispositivos tendem a ser mais baixas, o que pode agravar o desempenho.

2.4.2.3 Redundância

Com o desenvolvimento descentralizado, é muito comum que times implementem funcionalidades semelhantes, ainda mais por se tratar e aplicações Web, Prajwal, Parekh e Shettar (2021) aponta que os times independentes trabalhando de forma paralela podem gerar redundâncias em seus códigos CSS e JavaScript, que por sua vez impactam na performance da aplicação.

Muitas dessas duplicatas de código não podem ser evitadas, algumas delas são arquivos de configuração para desenvolvimento e até construção, de outro modo, existem duplicatas que até poderiam ser evitadas com a criação de bibliotecas. Mas, Silva (2021) defende que os vários *microfrontends* compartilhando dependências em comum contribui para um novo problema, o aumento do acoplamento.

2.5 Abordagens de implementação

Ao longo dessa seção serão apresentadas algumas das abordagens utilizadas para implementar a arquitetura de *microfrontends*. Não existe uma abordagem padronizada na indústria, segundo Klimm (2021), o que requer diversas pesquisas e validações do sistema ao longo do desenvolvimento.

2.5.1 Tempo de Construção

Essa abordagem considera os *microfrontends* como pacotes que são publicados e depois incluídos como dependências em um componente responsável pela disponibilidade da aplicação (ALMEIDA, 2021). Essa abordagem não utiliza os principais conceitos da arquitetura, visto que possui apenas um componente no final, mas ainda mantém o paralelismo na etapa de desenvolvimento.

A principal desvantagem desse modelo, é o alto nível de acoplamento, pois o componente final deve ser recompilado sempre que houver uma alteração em suas dependências, revertendo a aplicação para um monólito.

2.5.2 Tempo de Execução

Diferente da anterior, os *microfrontends* são integrados em tempo de execução, o que contribui para diversos tipos de abordagens de composição, as três principais na literatura são: *server-side*, *edge-side* e *client-side*.

2.5.2.1 Server-side

Segundo Jackson (2019) essa abordagem consiste em renderizar HTML do servidor utilizando templates ou fragmentos do HTML, um arquivo principal com os elementos em comum da aplicação, e então os conteúdos específicos das páginas são integrados de outros arquivos.

Por se tratar de uma abordagem MPA, a performance é melhorada, Almeida (2021) aponta que as páginas são carregadas mais rapidamente e também não é necessário o uso excessivo de JavaScript para renderizar os componentes. Além disso, nessa abordagem os

dados já são carregados no HTML, poupando as requisições que seriam necessárias em aplicações SPA.

2.5.2.2 Edge-Side

De maneira semelhante a *server-side*, essa abordagem utiliza também templates e fragmentos do HTML, mas em nível de CDN (*Content Delivery Network*). Uma CDN é um serviço que permite que o conteúdo de um site seja distribuído por diversos servidores em diferentes locais geográficos, dessa forma, o usuário pode acessar o site mais próximo de sua localização, melhorando a performance da aplicação.

Quanto a composição dos *microfrontends*, a abordagem é semelhante a *server-side*, mas ocorre nesses tipos de serviço graças a uma linguagem de marcação chamada ESI (*Edge Side Includes*), desenvolvida por Akamai (ALMEIDA, 2021).

Um dos problemas dessa abordagem segundo Almeida (2021) é a falta de suporte a ESI nos provedores de CDN. Um outro problema a se observar é que o uso de CDN é indicado para aplicações com muito conteúdo estático, pois conseguem utilizar melhor as técnicas de cache desse tipo de serviço.

2.5.2.3 Client-Side

Essa abordagem é semelhante ao que ocorre nas aplicações SPA, ou seja, as alterações são feitas diretamente no navegador, onde a página reage às interações do usuário quase imediatamente (SILVA, 2021).

2.5.2.3.1 Rotas Distribuídas

Segundo Yang, Liu e Su (2019), essa abordagem consiste em distribuir diferentes serviços em diferentes e independentes aplicações *frontend* via rotas. Normalmente é implementada com *proxy* reverso em um servidor HTTP. Ainda relata que a abordagem parece uma coleção de *frontends* agrupados e se parecem como uma única aplicação, no processo de roteamento, é comum a necessidade de recarregar a página, com isso as aplicações perdem o controle da página.

Um dos problemas dessa abordagem é a duplicação de código, desde a estilização até as lógicas de negócio, dado que os componentes podem não ser granulares o suficiente. Visto que as dependências não são compartilhadas o navegador tem que carregar todas as dependências a cada atualização da página (ALMEIDA, 2021).

2.5.2.3.2 IFrames

Cada time desenvolve sua solução em uma aplicação completamente isolada e independente, elas são integradas posteriormente utilizando a tag *iframe* do HTML, per-

mitindo combinar uma página com outra (SILVA, 2021). Segundo Yang, Liu e Su (2019), cada módulo pode utilizar um *framework*, sem necessidade de coordenar com outros times, mas ainda conseguem utilizar APIs nativas para interagir com outros módulos.

A fácil maneira de isolar os componentes utilizando *iframes* tende a torná-los menos flexíveis que outras abordagens, pois pode ser difícil construir integrações entre as diferentes partes da aplicação. Essa abordagem apresenta desafios para tornar a página mais dinâmica e responsiva (JACKSON, 2019).

2.5.2.3.3 JavaScript

Baseada em uma técnica comum no desenvolvimento Web, os *microfrontends* são integrados por meio de tags script no DOM da página (ALMEIDA, 2021). A aplicação contêiner determina quais *microfrontends* devem ser montados, e faz a chamada de uma função para dizer onde um *microfrontend* deve se montar (JACKSON, 2019).

Jackson (2019) ainda diz que cada *microfrontend*, diferente da integração em tempo de construção, pode ser implantado de forma independente, e diferente da abordagem com *iframes*, os *microfrontends* podem ser melhor integrados.

2.5.2.3.4 *Web Components*

Nessa abordagem, cada time desenvolve sua solução como um *web component*, de forma isolada, esses componentes são reusáveis, e podem ser importados, de maneira semelhante ao que a abordagem anterior faz, pela aplicação contêiner e registrados para uso sob demanda na página (SILVA, 2021).

Segundo Poloskei e Bub (2021), esses componentes podem estender os hooks dos *custom elements*, é possível atribuir atributos HTML aos mesmos e eventos são disparados se houver uma manipulação no DOM. De outro modo, é possível utilizar os quatro conceitos principais de *Web Components*, o que permite melhorar o desenvolvimento com essa abordagem.

2.5.2.3.5 *Webpack Module Federation*

A abordagem com o *plugin Module Federation*, presente a partir das versões do *WebPack 5*, permite que módulos remotos sejam importados e integrados em uma aplicação. O código importado pode ser utilizado com um *import* ES6 como se o código estivesse presente no repositório (SILVA, 2021).

Segundo a documentação do *plugin Module federation*[2], é possível expor e usar qualquer tipo de módulo que o *webpack* suporta, CSS, códigos, imagens. Além disso,

[2] (MODULE..., 2022)

carregar módulos remotos é considerado uma operação assíncrona, não é possível utilizar módulos remotos sem carregamento em *chunks*.

Chunks são partes menores do *bundle* a ser carregado, o *bundle*, por sua vez, se refere ao código que resulta do empacotamento da aplicação. O carregamento em *chunks* permite que essas partes sejam carregadas de forma assíncrona, sendo as mesmas combinadas posteriormente.

Dessa forma, os *microfrontends* são desenvolvidos, construídos e implantados de maneira independente como módulos remotos do *webpack*, e são importados e integrados na aplicação contêiner. Por outro lado, Silva (2021) aponta que o *webpack* por si só não pode ser usado para integrar componentes agnósticos desenvolvidos em diferentes *frameworks*.

Segundo suas observações, esse *plugin* é uma nova abordagem de carregar módulos dinamicamente de maneira remota (agindo como uma camada de transporte). De modo semelhante, é uma solução otimizada em termos de performance e escalabilidade quando comparado com arquivos externos ou *import maps*.

3 Metodologia

Este capítulo descreve a metodologia utilizada nesse trabalho, com foco na completude dos objetivos propostos. O trabalho se trata de uma pesquisa aplicada, exploratória e qualitativa.

3.1 Levantamento teórico

O levantamento teórico foi realizado com intuito final da construção do referencial teórico, algumas das fontes utilizadas são conhecidas no mercado de desenvolvimento de software, em especial os que utilizam a arquitetura de *microfrontends*: Martin Fowler, documentação do *Webpack* e documentação de *Web Components*.

Os seguintes trabalhos foram levantados por meio de contato direto com pesquisadores da área de *microfrontends*:

- ALMEIDA, D. *Micro Frontends para aplicações Web*. Dissertação (Mestrado) — Instituto Superior de Engenharia do Porto, 2021.
- SILVA, R. da. *A Micro Frontends Solution - Analyzing quality attributes*. Dissertação (Mestrado) — Instituto Superior de Engenharia do Porto, 2021.
- KLIMM, M. C. *Design Systems for Micro Frontends*: An investigation into the development of framework-agnostic design systems using svelte and tailwind css. Dissertação — TH Köln - University of Applied Sciences, 2021.
- PRAJWAL, Y. R.; PAREKH, J. V.; SHETTAR, D. R. A brief review of micro-frontends. *United International Journal for Research & Technology*, v. 2, n. 8, 2021.
- YANG, C.; LIU, C.; SU, Z. Research and application of micro frontends. *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, v. 490, p. 062082, apr 2019. Disponível em: <https://doi.org/10.1088/1757-899x/490/6/062082>.
- POLOSKEI, I.; BUB, U. Enterprise-level migration to micro frontends in a multi-vendor environment. *Acta Polytechnica Hungarica*, v. 18, p. 7–25, 01 2021.

Outras literaturas relevantes para o trabalho foram: (TANENBAUM; STEEN, 2013) e (BASS; CLEMENTS; KAZMAN, 2012). Referências estas conhecidas nas áreas de sistemas distribuídos e arquitetura de software, respectivamente.

O referencial teórico foi construído com foco na contextualização do assunto, dando ênfase na progressão das arquiteturas anteriores à de *microfrontends*, como uma progressão partindo da arquitetura monolítica até a arquitetura de *microfrontends*.

De outro modo, foram abordados conceitos importantes da arquitetura de aplicações Web, além da definição da arquitetura de *microfrontends*, suas principais características, vantagens, desvantagens e principais abordagens de implementação.

3.2 Definição da Abordagem

Propõe-se a definir como as abordagens serão combinadas em uma nova, bem como a interface que será utilizada para que seja possível um *microfrontend* incorporar outro *microfrontend*. Além disso, descreve como e quais os parâmetros necessários para que a abordagem seja implementada.

A abordagem combina *Web Components* e *WebPack Module Federation*, que foi escolhida almejando as vantagens que cada uma dessas abordagens oferece. Sendo as principais vantagens: agnóstico ao *framework*, otimização de carregamento, independência de implantação, isolamento de CSS e compartilhamento de contexto.

Os objetivos a serem alcançados com esta abordagem, que sintetizam os objetivos específicos do trabalho são:

- Combinação das abordagens de *Web Components* e *WebPack Module Federation*;
- A abordagem ser agnóstica ao *framework*;
- A utilização das especificações de *Web Components*;
- Integração nativa, sem depender de *frameworks* de *frontend*.

3.3 Provas de conceito

A justificativa para a escolha da metodologia de provas de conceito, do inglês *proof of concept* (PoC), é que se mostra uma técnica adequada para validar a abordagem proposta, visto sua capacidade para validar a viabilidade de uma solução, por facilitar a validação de propostas de arquiteturas e de tecnologias dado o desenvolvimento da mesma em baixa escala e com objetivos bem definidos.

O foco principal das provas de conceito é validar a viabilidade de implementação de *microfrontends* utilizando a abordagem. Para isso, cada uma delas foi definida com base em um objetivo dentre os definidos anteriormente para a abordagem. Espera-se que

elas provem a validade e o devido cumprimento de cada objetivo por meio dos requisitos e arquitetura que serão definidos.

Cada prova de conceito possuirá sua própria arquitetura, muito embora elas sejam incrementais, ou seja, cada uma delas é uma evolução da prova de conceito anterior. As provas de conceito são listadas a seguir e abordadas nos capítulos seguintes:

- **PoC 1 - Abordagem de implementação combinada:** o foco principal dessa validação é a abordagem em si, de maneira que seja possível incorporar um *microfrontend* em outro *microfrontend*;
- **PoC 2 - Aplicação *multi-framework* com abordagem agnóstica:** com foco em validar o atributo de agnosticidade da abordagem, de modo que a interface permita importação independente do *framework* em uso, será implementada utilizando os *frameworks React*, *Angular* e *Vue*;
- **PoC 3 - Utilizando especificações de Web Components:** Utilizar *Shadow DOM* nos *microfrontends*, com o intuito de isolar o CSS, validando uma das funcionalidades principais de *Web Components*;
- **PoC 4 - *Microfrontend* nativo:** o foco desta prova é validar também agnosticidade da abordagem, de modo que seja possível um *microfrontend* desenvolvido totalmente utilizando a API nativa.

Além disso, para o desenvolvimento das provas de conceito, foram utilizadas os exemplos disponibilizados pela comunidade do *WebPack Module Federation*, que podem ser encontrados no repositório https://github.com/module-federation/module-federation-examples.

3.4 Ferramentas

As ferramentas utilizadas nesse trabalho e principalmente para o desenvolvimento das provas de conceito são:

3.4.1 Chrome DevTools

O *Chrome DevTools* é uma ferramenta de desenvolvimento web, que permite a inspeção de elementos e a depuração de código, o uso dessa ferramenta se dá devido a sua popularidade e facilidade de uso, além de fornecer diversas funcionalidades que auxiliam no desenvolvimento de aplicações web.

3.4.2 CodeSnap

O *CodeSnap* é uma extensão do editor de textos *Visual Studio Code*, que permite gerar imagens de código, o uso dessa ferramenta se dá devido a facilidade de representar trechos de código de maneira visual.

3.4.3 Draw.io

É uma aplicação web que permite a criação de diagramas, será utilizada para a criação dos diversos diagramas que serão utilizados no trabalho. O seu uso se dá devido a facilidade de acesso além de ser uma ferramenta gratuita.

3.4.4 Git

O *Git* é um sistema de controle de versão distribuído, que permite o versionamento de código fonte, o uso dessa ferramenta se dá devido a sua popularidade na comunidade de desenvolvimento de software e a familiaridade de uso.

3.4.5 GitHub

O *GitHub* é uma plataforma de hospedagem de código-fonte com controle de versão usando o *Git*, o uso dessa ferramenta se dá devido também a popularidade na comunidade de desenvolvimento de software, e a principal razão é a facilidade de acesso e a possibilidade de hospedar e disponibilizar os códigos fonte.

3.4.6 Lerna

O *Lerna* é um gerenciador de pacotes para projetos *JavaScript*, que são mantidos em um único repositório *Git*, o uso dessa ferramenta se dá devido as provas de conceito serem desenvolvidas em modelo monorepo, e o *Lerna* é uma ferramenta que facilita o gerenciamento de pacotes em um repositório monorepo.

3.4.7 Visual Studio Code

O *Visual Studio Code* é um editor de código-fonte desenvolvido pela *Microsoft*, o uso dessa ferramenta se dá devido a sua popularidade na comunidade de desenvolvimento de software, além de ser uma ferramenta gratuita, de fácil acesso e possuir extensões que facilitam o desenvolvimento, principal de aplicações web.

3.4.8 Yarn

O *Yarn* é um dos gerenciadores de pacotes para a linguagem de programação *JavaScript*, o uso dessa ferramenta se dá devido acessar o repositório comum (NPM *Registry*) de pacotes para a linguagem, lá é possível encontrar diversos pacotes, dos quais alguns são utilizados no desenvolvimento das provas de conceito.

4 Definição da abordagem

4.1 Integração de *microfrontends*

A implementação da arquitetura de *microfrontends* combinando *WebPack Module Federation* e *Web Components*, pode ser feita utilizando o *Webpack* como camada de transporte para a realização das importações remotas dos módulos, e a tecnologia de *Web Components* responsável pela abstração do *microfrontend* como uma tag HTML customizada.

Essa abordagem continua sendo considerada uma abordagem de renderização client-side e em tempo de execução, pois os módulos remotos são carregados e executados em tempo de execução, além disso, são carregados a partir de um frontend principal no navegador e não em um servidor.

Além do mais, o *plugin* Module Federation do *Webpack* presume dois tipos de importação remota de módulos, a importação estática e a dinâmica. Para esta abordagem, será utilizada a importação dinâmica, pois permite a importação de módulos em tempo de execução utilizando apenas a interface definida pelo *plugin*.

Código 1 – Configuração *Plugin Module Federation*

```
new ModuleFederationPlugin({
  name: "...",
  filename: "...",
  exposes: [...],
  // ...
}),
```

O Código 1 é um exemplo de configuração do *plugin*, nele é possível identificar três principais parâmetros:

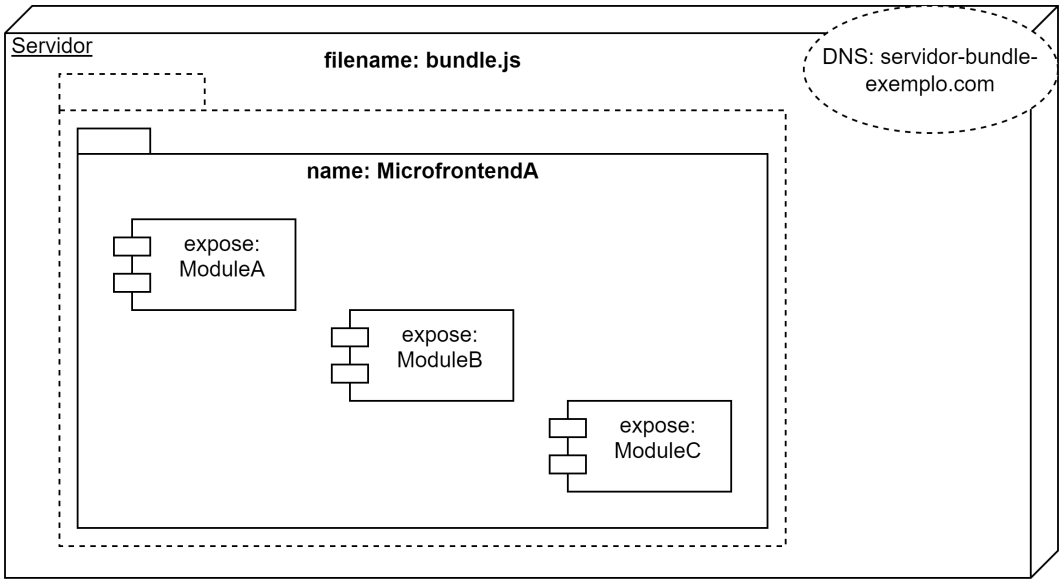
- **name**: Nome do contêiner que será exposto para a importação remota.
- **filename**: Nome do arquivo que será gerado contendo o contêiner.
- **exposes**: Lista de módulos que serão expostos para a importação remota naquele contêiner.

A importação remota é feita utilizando a URL em que o módulo está hospedado, junto com o nome do arquivo (*filename*), o nome do contêiner (*name*) a ser carregado

pelo host e o nome do módulo que será importado. Como se pode ver, é possível que um contêiner possua mais de um módulo exposto para importação remota.

A Figura 1 é um diagrama de implantação de um *microfrontend* utilizando o *plugin*. Nela é possível identificar os três itens e como eles são organizados.

Figura 1 – Diagrama de Implantação de um *microfrontend* com *Webpack Module Federation*



Fonte: Autor.

Já os *Web Components* precisam ser registrados no DOM, por meio da API de *Custom Elements* para que possam ser utilizados. Para isso, nas abordagens que implementam *microfrontends* com *Web Components*, os componentes são registrados no arquivo remoto que é carregado pelo *frontend* principal. Para a abordagem proposta nesse trabalho, os componentes serão registrados nos módulos que serão importados remotamente, seguindo o que já ocorre na abordagem utilizando *Web Components*.

De forma geral, a interface da abordagem consiste em quatro parâmetros, os três apresentados anteriormente, necessários para o carregamento remoto dos módulos: `url`, `name` e `module`, e o quarto, `tag-name`, que é o nome da tag HTML customizada que foi definida para o *microfrontend*. A `url` se refere à combinação do domínio, caminho e `filename` oriundo da configuração do *plugin Module Federation*.

Os módulos que serão importados poderão estar definindo um ou mais *microfrontends* como tags HTML customizadas, por isso, o `tag-name` é necessário para identificar qual *microfrontend* será importado, inclusive entre módulos e contêiners diferentes, visto que a declaração de *Custom Elements* é global para todo o DOM.

4.2 *Shell*

Shell, root, host ou aplicação contêiner (JACKSON, 2019), nomes dados para o *frontend* principal, responsável por carregar os módulos remotos. Além disso, para o usuário final, a URL exibida no navegador é a do *Shell*, e não a dos módulos remotos.

Esse comportamento é oriundo do fato de uma arquitetura com *microfrontends* ser um sistema distribuído, que segundo (TANENBAUM; STEEN, 2013), é um sistema composto por vários computadores independentes que se revelam aos seus usuários como um único e coerente sistema.

De forma semelhante, tanto a abordagem de *microfrontends* com *Web Components* quanto a com *Webpack Module Federation*, possuem esse *frontend* principal, a abordagem combinada também possuirá, e será referida ao longo deste trabalho como *Shell*. Nesta abordagem, o *Shell* se trata de uma aplicação SPA, com a particularidade de ser um *host Module Federation*, ou seja um contêiner que carregará os outros módulos remotos.

4.3 Provas de Conceito

Os seguintes capítulos apresentam as provas de conceito, desde a sua concepção até a sua implementação e posterior análise. As provas não buscam ser uma implementação completa, mas sim um exemplo de como a abordagem proposta pode ser utilizada. O código fonte das provas de conceito pode ser encontrado no repositório *GitHub* https://github.com/Rhuancpq/microfrontends-wc-webpack-mfe.

5 PoC 1 - Abordagem de implementação combinada

A primeira prova de conceito consiste em uma aplicação SPA (Single Page Application) *com* microfrontends, contendo somente uma página, que é a página inicial. A aplicação *Shell* será responsável por uma estrutura básica da aplicação, contendo uma seção e o conteúdo desta seção será um *microfrontend*. A aplicação *Shell* será responsável por carregar o *microfrontend* e incorporá-lo em sua estrutura.

5.1 Objetivos

O objetivo desta prova de conceito é validar inicialmente se é possível combinar as abordagens de *Web Components* e *WebPack Module Federation*, de modo que seja possível incorporar um *microfrontend* em outro *microfrontend*.

Esta prova de conceito ao utilizar a abordagem combinada, consegue atingir o objetivo inicial de a combinação das abordagens ser factível e funcional, validando não só o funcionamento da abordagem, como também que deve ser possível utilizá-la para as demais provas de conceito.

De modo semelhante, essa prova de conceito também colabora para o cumprimento do terceiro objetivo, que está relacionado com a utilização das especificações de *Web Components* na abordagem combinada, visto que para a declaração de um Web Component é necessário utilizar a especificação de Custom Elements. O uso de WebPack também gera um módulo JavaScript, o que tem relação com a especificação de ES Modules.

5.2 Requisitos

A seguir são listados os requisitos que devem ser atendidos pela prova de conceito 1:

- A aplicação *Shell* deve ser uma SPA (Single Page Application).
- A aplicação *Shell* deve ser responsável por incorporar o *microfrontend* em sua estrutura.
- O *microfrontend* deve ser um Web Component.
- O *microfrontend* deve ser incorporado como uma tag HTML customizada.

- O *microfrontend* estará contido em um módulo remoto *WebPack Module Federation*.
- O *microfrontend* também deve utilizar técnicas de SPA (Single Page Application).
- A interface deverá ter um botão que ao ser clicado irá carregar e exibir o *microfrontend*.
- A seção carregada deverá ser um formulário com um campo de texto e um botão.
- Ao confirmar o formulário, deverá ser apresentada um alerta com o conteúdo do campo de texto.

5.3 Arquitetura

A arquitetura da prova de conceito 1 consiste em dois componentes, sendo um deles o componente *Shell* e o outro um *microfrontend*. Eles serão desenvolvidos utilizando o *framework React*, e a abordagem de implementação dos *microfrontends* utilizará a abordagem proposta no capítulo anterior. É considerada a mais simples dentre as provas de conceito, pois possui menos componentes e recursos explorados.

5.4 Pacotes Utilizados

A seguir são listados os pacotes utilizados na prova de conceito 1 e suas respectivas versões:

Tabela 1 – Pacotes utilizados na prova de conceito 1

Pacote	**Versão**
@babel/core	7.18.9
@babel/preset-react	7.18.6
babel-loader	8.2.5
html-webpack-plugin	5.5.0
serve	13.0.4
webpack	5.75.0
webpack-cli	5.0.0
webpack-dev-server	4.11.1
react	16.13.0
react-dom	16.13.0

Fonte: Autor.

5.4.1 Pacotes notáveis

A seguir são listados os principais pacotes utilizados na prova de conceito 1 e suas respectivas funções:

- **webpack**: É um empacotador de módulos, que permite a utilização de módulos JavaScript, CSS, imagens, fontes, etc. Ele também permite a utilização de plugins, que são extensões que podem ser utilizadas para adicionar funcionalidades ao empacotador. O principal plugin utilizado na prova de conceito 1 é o *webpack Module Federation*, que permite a utilização de módulos remotos.

- **webpack-dev-server**: É um servidor de desenvolvimento que permite a execução de uma aplicação web em um servidor local, permitindo a utilização de recursos como hot reload, que permite a atualização automática da aplicação web quando um arquivo é alterado.

- **html-webpack-plugin**: É um plugin que permite a utilização de um template HTML para a geração do arquivo HTML final.

- **react**: É uma biblioteca JavaScript para a criação de interfaces. Neste projeto, será utilizado para a criação dos componentes das provas de conceito.

- **react-dom**: É uma biblioteca JavaScript que permite a renderização de componentes React em um documento HTML.

5.5 Desenvolvimento

O desenvolvimento da prova de conceito 1 pode ser dividido em três etapas, sendo elas: configuração do ambiente, desenvolvimento da interface de integração e desenvolvimento dos *microfrontends*.

O desenvolvimento foi realizado baseado no exemplo **dynamic-system-host** disponível no repositório de exemplos do *webpack Module Federation* Plugin. [1]

5.5.1 Configuração do ambiente

A primeira etapa do desenvolvimento da prova de conceito 1 consiste na configuração do ambiente de desenvolvimento. Para isso, foi criado um diretório chamado poc1, e dentro dele foram criados dois diretórios, um chamado *shell* e outro chamado *microfrontend*. Esses diretórios serão utilizados para armazenar os códigos fonte da aplicação *Shell* e do *microfrontend*, respectivamente.

Além disso, foi criado um arquivo `package.json` (Figura 2) na raiz do diretório poc1, que contém as informações do projeto e algumas configurações para o pacote Lerna, que foi utilizado principalmente para a execução dos componentes de forma simplificada.

[1] Disponível em: <https://github.com/module-federation/module-federation-examples/tree/master/dynamic-system-host>. Acesso em: 19 nov. 2022.

Figura 2 – Arquivo package.json na raiz do diretório poc1

```
{
  "private": true,
  "scripts": {
    "start": "lerna run --scope @poc-1/* --parallel start",
    "build": "lerna run --scope @poc-1/* build",
    "serve": "lerna run --scope @poc-1/* --parallel serve",
    "clean": "lerna run --scope @poc-1/* --parallel clean"
  },
  "devDependencies": {
    "lerna": "3.22.1"
  },
  "workspaces": {
    "packages": [
      "shell",
      "microfrontend"
    ]
  }
}
```

Fonte: Autor.

Nesta etapa os pacotes foram instalados e atualizados, para isso foi utilizado o pacote `yarn`, que é um dos gerenciadores de pacotes mais utilizados do Node.js. Cada um dos diretórios contém um arquivo chamado `package.json`, que contém as informações do projeto e as dependências do mesmo. Para a instalação das dependências foi utilizado o comando `yarn` e para a execução dos componentes foi utilizado o comando `yarn start`.

5.5.2 Desenvolvimento da interface de integração

A interface de integração seguindo a abordagem proposta no capítulo anterior foi desenvolvida em três passos principais, sendo eles: a configuração do plugin Module Federation, a criação da tag customizada e a alteração do código de integração no componente *Shell*.

O plugin precisa ser configurado no arquivo de configuração do webpack em cada um dos componentes, cada diretório contém um arquivo chamado webpack.config.js, que é o arquivo mencionado. A configuração do *plugin Module Federation* é feita através da inclusão de uma nova instância do plugin no objeto plugins do arquivo de configuração do webpack.

Para o componente *Shell*, o arquivo webpack.config.js (Figura 3) foi alterado e se assemelha muito com a configuração no *microfrontend*, as diferenças são: a propriedade `filename` e o objeto `exposes` que mapeia o caminho módulo exposto para o caminho remoto desse módulo (Figura 4).

A chave `shared` contém um mapa com os pacotes que serão compartilhados entre os componentes, nesse caso, o *React* e o *ReactDOM*. Essa configuração tem o objetivo de evitar que o mesmo pacote seja carregado duas vezes, o que pode causar problemas de

performance (MODULE. . . , 2022).

Figura 3 – Arquivo webpack.config.js no diretório *shell*

Fonte: Autor.

Figura 4 – Arquivo webpack.config.js no diretório *microfrontend*

Fonte: Autor.

A criação da tag customizada é feita no arquivo `WebComponent.js`, que é o arquivo que foi exposto para importação remota. Esse arquivo é responsável por definir a tag customizada e carregar o *microfrontend*. A tag customizada é definida na linha 11 (Figura 5) e o elemento do DOM definido para o carregamento do *ReactDOM* do *micro-*

frontend é a própria tag customizada, o que é possível ver na linha 7 dentro do método `connectedCallback` do ciclo de vida de elementos HTML.

Figura 5 – Arquivo WebComponent.js do componente *microfrontend*

```
1  import App from "./App";
2  import React from "react";
3  import ReactDOM from "react-dom";
4
5  class Microfrontend extends HTMLElement {
6    connectedCallback() {
7      ReactDOM.render(<App />, this);
8    }
9  }
10
11 customElements.define("micro-frontend", Microfrontend);
```

Fonte: Autor.

A alteração do código de integração no componente *Shell* foi realizada no componente `System` no arquivo `App.js`, que é o componente responsável por carregar, importar e renderizar os *microfrontends*. Este componente foi implementado no exemplo do **dynamic-system-host** do repositório de exemplos do *Module Federation*.

O código desse componente foi alterado para carregar o *microfrontend* através da tag customizada, a alteração pode ser vista na linha 26 (Figura 6). Onde é possível ver que a tag é injetada no DOM através da propriedade `setDangerouslyInnerHTML` e o código HTML é o código da tag customizada.

As outras propriedades desse componente são os três parâmetros necessários para que o plugin importe o módulo remoto que contém a declaração da tag customizada. Sendo eles `remote`, `module` e `url`, possuem nomes diferentes mas são análogos aos definidos na configuração do webpack no componente *microfrontend* (Figura 4). Onde a função `loadComponent` é a responsável por carregar o módulo remoto e disponibilizá-lo para uso.

5.5.3 Desenvolvimento dos *microfrontends*

Para o devido cumprimento dos requisitos da PoC foram desenvolvidos dois *microfrontends*, sendo um deles o componente *Shell* e o outro o componente *microfrontend*.

O objetivo do componente *Shell* é carregar o *microfrontend*, e o faz por meio da função `setMicrofrontend` definindo os parâmetros necessários para carregar o *microfrontend*, ela é chamada no arquivo `App.js` do componente *Shell* (Figura 7). Essa função é

Figura 6 – Componente React System do *Shell*

```
function System(props) {
  const {
    system,
    system: { remote, url, module, tag },
  } = props;

  const [loading, setLoading] = useState(true);

  useEffect(() => {
    if (remote && url && module)
      loadComponent(remote, "default", module, url)().then(() => {
        setLoading(false);
      });

    return () => {};
  }, [remote, url, module]);

  if (!system || !remote || !url || !module) {
    return <h2>Nenhum microfrontend especificado</h2>;
  }

  if (loading) {
    return <h2>Loading...</h2>;
  }

  return <div dangerouslySetInnerHTML={{ __html: `<${tag} />` }}></div>;
}
```

Fonte: Autor.

chamada no evento `onClick` do elemento `button` do componente *Shell*, criando assim o comportamento de carregar o *microfrontend*.

Já no componente *microfrontend*, o formulário de texto foi implementado no componente *React* `Form` (Figura 8). Sendo possível visualizar a implementação do controle de estado do campo de texto e o devido acionamento do alerta com o conteúdo do campo de texto após o evento de submissão.

Figura 7 – Definição do componente a ser carregado

```
function App() {
  const [system, setSystem] = React.useState({});

  function setMicrofrontend() {
    setSystem({
      remote: "microfrontend",
      url: "http://localhost:3002/bundle.js",
      module: "./Microfrontend",
      tag: "micro-frontend",
    });
  }
  ...
```

Fonte: Autor.

Figura 8 – Componente *React Form*

```
const lorem =
  "Lorem ipsum dolor sit amet, consectetur adipiscing elit." +
  "Phasellus aliquet lobortis tempor.";

function Form() {
  const [text, setText] = useState(lorem);
  const modalOpen = useCallback(() => {
    alert(text);
  }, []);

  return (
    <div>
      <textarea
        rows="4"
        cols="50"
        value={text}
        onChange={(e) => setText(e.target.value)}
      ></textarea>
      <div style={{ marginTop: "1em" }}>
        <button onClick={modalOpen}>Submeter Formulário</button>
      </div>
    </div>
  );
}
```

Fonte: Autor.

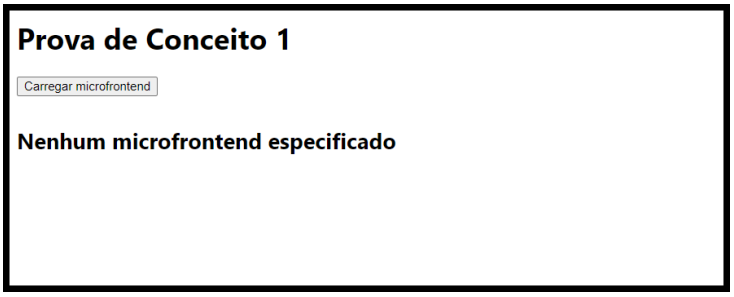
Obtendo assim, três estados principais da aplicação, sendo eles: inicial, *microfrontend* carregado e alerta disparado. O estado inicial é o estado inicial da aplicação, onde o botão de carregar o *microfrontend* está habilitado e o *microfrontend* não foi carregado (Figura 9).

O estado de *microfrontend* carregado é o estado em que o *microfrontend* foi carregado e o formulário de texto está disponível para ser preenchido (Figura 9). E o estado de alerta disparado é o estado em que o formulário foi preenchido e o botão de submeter o formulário foi clicado, disparando assim o alerta com o texto preenchido no formulário (Figura 9).

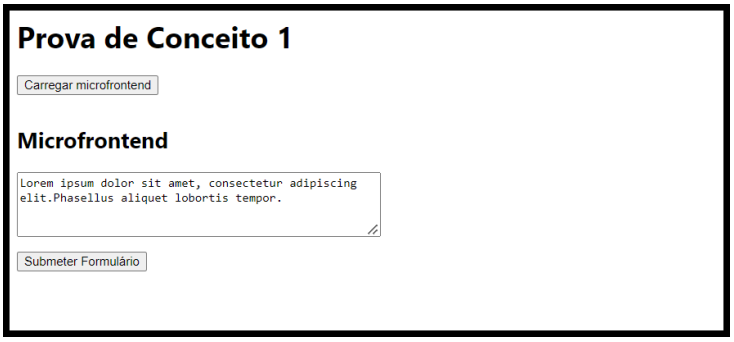
5.6 Resultados

Quanto aos requisitos, a primeira prova de conceito foi concluída com sucesso, essa seção é dedicada a apresentação das descobertas e resultados obtidos durante o processo de desenvolvimento.

Figura 9 – Estados da PoC 1



(a) Estado inicial



(b) *microfrontend* carregado

(c) Alerta disparado

Fonte: Autor.

5.6.1 Factibilidade da abordagem

A PoC 1 foi desenvolvida com o principal objetivo de demonstrar a factibilidade da abordagem proposta, e se seria possível carregar um *microfrontend* em outro *microfrontend* e se as seguintes provas de conceito seriam possíveis de serem implementadas.

A abordagem se mostrou funcional para o que foi proposto, sendo implementada com os principais conceitos propostos, como a utilização do componente *Shell* para carregar o *microfrontend*, a utilização do *plugin Webpack Module Federation* para o carregamento remoto dos módulos e a utilização da primeira especificação de *Web Components*, para a definição de tags customizadas.

Mostra também que a abordagem funciona para o carregamento de *microfrontends* empregando o *framework React*, e ainda visando a utilização de principais conceitos do *React*, como o controle de estado e o uso de eventos. Com isso, pode ser possível explorar a implementação de aplicações utilizando *microfrontends* e a abordagem proposta

empregando o *framework React*.

5.6.2 Definição da tag customizada

Ao longo da prova de conceito, foi possível observar que a definição da tag customizada pode ser feita de duas formas, sendo elas: adefinição da tag customizada no componente *microfrontend* e a definição da tag customizada no componente *Shell*.

Na primeira forma, que é o caso da PoC 1, a tag customizada é definida no arquivo `WebComponents.js` do componente *microfrontend*, e é carregado no componente *Shell*. O módulo então utiliza um conceito importante chamado de *side effects*, que é a execução de código que não retorna um valor, mas que tem um efeito colateral, como por exemplo, a definição de uma tag customizada.

A segunda forma, consistiria na definição da tag customizada no componente *Shell*, sendo que a classe `Microfrontend` definida na Figura 5 poderia ser exportada e carregada como um módulo, e o *Shell* poderia definir a tag customizada logo após o carregamento do módulo.

A implementação dessa segunda forma pode ser vista na Figura 10, onde o *hook* `useEffect` é utilizado para definir a tag customizada no componente *Shell*, após o carregamento do módulo, uma alteração do componente `System` observado anteriormente na Figura 6.

Figura 10 – PoC 1 - Definição da tag customizada no componente *Shell*

```
useEffect(() => {
  if (remote && url && module)
    loadComponent(remote, "default", module, url)().then((module) => {
      customElements.define(tag, module.default);
      setLoading(false);
    });
...
```

Fonte: Autor.

Não é possível afirmar se há alguma diferença significativa entre as duas formas de definição da tag customizada, mas é possível afirmar que na segunda, o componente *Shell* por estar responsável por algo novo, pode ter sua coesão afetada. Mas ao mesmo tempo, permite que o componente *Shell* tenha mais controle sobre as tags que foram definidas e pode evitar conflitos de nome, já que cada tag customizada deve ser única.

6 PoC 2 - Aplicação multi-framework com abordagem agnóstica

A segunda prova de conceito consiste numa aplicação SPA (Single Page Application) com quatro componentes, sendo um deles o componente Shell e os outros três *microfrontends*. A aplicação Shell permanece com as responsabilidades da primeira prova de conceito, sendo responsável por carregar os *microfrontends* e incorporá-los em sua estrutura.

Tanto a aplicação Shell, quanto o *microfrontend* da prova de conceito 1, serão utilizados, com as devidas adaptações, para a segunda prova de conceito. O *microfrontend* será adaptado para se tornar a seção de conteúdo adicional da aplicação (Aside).

6.1 Objetivos

O foco principal desta prova de conceito é a construção de componentes com tecnologias diferentes, mas ainda assim, por meio da mesma abordagem combinada, ser possível utilizá-los em conjunto, isso condiz com o segundo objetivo da abordagem e assim como na primeira prova, o terceiro objetivo também é incluído. Além disso, essa prova também colabora para o cumprimento do primeiro objetivo, que está atrelado à utilização da abordagem combinada, visto que tem estrutura semelhante a primeira.

Com isso será possível validar se a abordagem combinada é capaz de realmente atender a necessidade de ser agnóstica a tecnologias, de modo que o componente Shell não precise saber, previamente, das tecnologias utilizadas pelos *microfrontends* que serão carregados.

Um outro objetivo desta prova de conceito é validar a utilização da abordagem próxima aos usos reais, visto que a primeira prova de conceito foi mais simples, a segunda se propõe a explorar diferentes elementos HTML e estilizações CSS mais complexas. Nessa linha, os componentes serão carregados logo após o carregamento da página, sem a necessidade de interação do usuário, o que é um cenário mais próximo do uso real dessa arquitetura.

6.2 Requisitos

A seguir os requisitos da segunda prova de conceito:

- A interface da abordagem deve ser suficiente para que o componente Shell possa incorporar os *microfrontends*.
- A aplicação Shell carregará os componentes logo após o carregamento da página.
- Os *microfrontends* devem ser Web Components.
- Os *microfrontends* devem ser incorporados como tags HTML customizadas.
- Os *microfrontends* estarão contidos cada um em módulo remoto WebPack Module Federation.
- Os *microfrontends* também devem utilizar técnicas de SPA.
- As seções devem possuir estados de carregamento e erro.
- O conteúdo principal deve ser um texto em uma seção com borda pontilhada.
- O cabeçalho da aplicação deve possuir uma lista de links enumerados.

6.3 Arquitetura

A aplicação web é composta de três *microfrontends*, sendo um desenvolvido em *React*, outro em *Angular* e o último em *Vue*. A aplicação Shell será evoluída da prova de conceito anterior para comportar a nova estrutura.

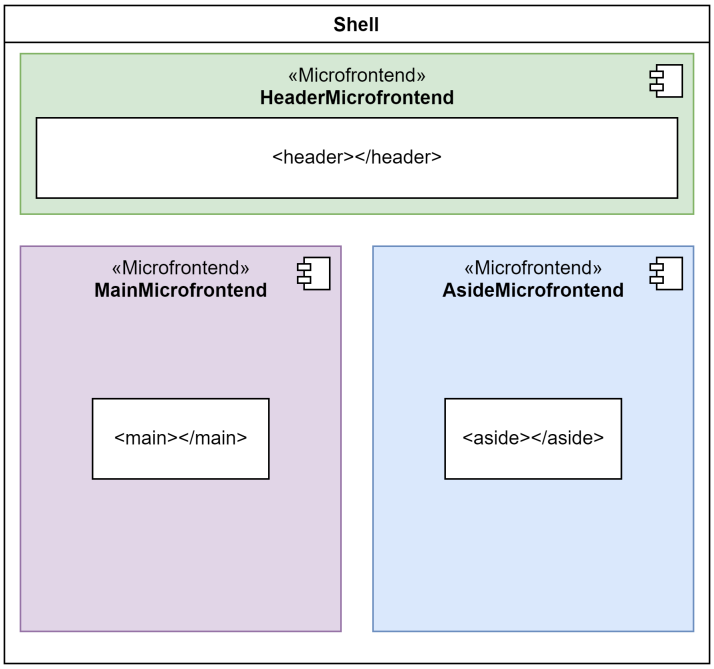
Os *microfrontends* representam elementos do leiaute da aplicação, como pode ser visto na Figura 11, sendo os três:

- **Header**: Representa o cabeçalho da aplicação, será desenvolvido utilizando o *framework Vue*;
- **Main**: Representa o conteúdo principal da aplicação, será desenvolvido utilizando o *framework Angular*;
- **Aside**: Representa a seção de conteúdo adicional da aplicação, será utilizado o componente da prova de conceito anterior.

6.4 Pacotes Utilizados

A seguir são listados os pacotes utilizados nos componentes da prova de conceito 2:

Figura 11 – Leiaute da aplicação da prova de conceito 2



Fonte: Autor.

Tabela 2 – Pacotes utilizados nos componentes Shell e Aside

Pacote	**Versão**
@babel/core	7.18.9
@babel/preset-react	7.18.6
babel-loader	8.2.5
html-webpack-plugin	5.5.0
serve	13.0.4
webpack	5.75.0
webpack-cli	5.0.0
webpack-dev-server	4.11.1
react	16.13.0
react-dom	16.13.0

Fonte: Autor.

6.4.1 Pacotes notáveis

Com a criação de novos *microfrontends* com *frameworks* diferentes, novos pacotes foram utilizados. A seguir são listados os pacotes que merecem destaque:

- ***Angular***: O *framework Angular* foi utilizado para o desenvolvimento do componente Main na versão `14.X`;
- **custom-webpack**: O pacote é um plugin do *Angular* que permite a customização das configurações do *Webpack*;

Tabela 3 – Pacotes utilizados no componente Header

Pacote	Versão
@babel/core	7.18.9
@vue/compiler-sfc	3.2.37
babel-loader	8.2.5
css-loader	6.7.1
file-loader	6.2.0
html-webpack-plugin	5.5.0
mini-css-extract-plugin	2.6.1
serve	13.0.4
url-loader	4.1.1
vue	3.0.11
vue-loader	16.8.3
webpack	5.75.0
webpack-cli	5.0.0
webpack-dev-server	4.11.1

Fonte: Autor.

- **elements**: O pacote é um plugin do *Angular* que permite a criação de Web Components utilizando o *Angular*;
- **mini-css-extract-plugin**: O pacote é um plugin do *Webpack* que permite a extração de CSS em arquivos separados;
- **Typescript**: O pacote é uma linguagem de programação baseada em Javascript que permite a tipagem de variáveis, é por padrão utilizado pelo *Angular*;
- ***Vue***: O *framework Vue* foi utilizado para o desenvolvimento do componente Header na versão `3.X`;

6.5 Desenvolvimento

O desenvolvimento dos componentes da prova de conceito 2 foi realizado também baseado nos exemplos disponibilizados no repositório de exemplos do *Webpack Module Federation*. O componente Header foi desenvolvido baseado no exemplo **vue3-demo**[1] e o componenteMain foi baseado no exemplo **angular14-react**[2]

O início do desenvolvimento se dá na implementação da interface da abordagem nos novos componentes, seguido pela implementação da integração no componente Shell, e por fim o desenvolvimento de fato das funcionalidades dos componentes.

[1] Disponível em: <https://github.com/module-federation/module-federation-examples/tree/master/vue3-demo>. Acesso em: 20 nov. 2022.

[2] Disponível em: <https://github.com/module-federation/module-federation-examples/tree/master/angular14-react>. Acesso em: 20 nov. 2022.

Tabela 4 – Pacotes utilizados no componente Main

Pacote	Versão
@angular-builders/custom-webpack	14.0.1
@angular-devkit/build-angular	14.2.2
@angular/animations	14.2.0
@angular/cli	14.2.2
@angular/common	14.2.0
@angular/compiler	14.2.0
@angular/compiler-cli	14.2.0
@angular/core	14.2.0
@angular/elements	14.2.11
@angular/forms	14.2.0
@angular/platform-browser	14.2.0
@angular/platform-browser-dynamic	14.2.0
@angular/router	14.2.0
@ngxs/devtools-plugin	3.7.5
@ngxs/store	3.7.5
@types/jasmine	4.0.0
bootstrap	5.2.2
jasmine-core	4.3.0
karma	6.4.0
karma-chrome-launcher	3.1.0
karma-coverage	2.2.0
karma-jasmine	5.1.0
karma-jasmine-html-reporter	2.0.0
rxjs	7.5.0
tslib	2.3.0
typescript	4.7.2
webpack	5.74.0
zone.js	0.11.4

Fonte: Autor.

6.5.1 Cabeçalho

A implementação da abordagem no *microfrontend* Header foi muito semelhante ao que aconteceu no *microfrontend* da PoC 1, embora utilizando o *framework Vue.js* foi criado um arquivo `WebComponent.js` (Figura 12) com a definição da classe e a declaração da tag customizada.

As configurações relacionadas ao plugin *Module Federation* foram realizadas também nesse componente, no arquivo `webpack.config.js`. Os parâmetros do plugin foram definidos no arquivo `webpack.config.js`, de maneira semelhante ao que ocorreu na prova de conceito anterior, o que é possível ver na Figura 13.

O conteúdo do componente Header é definido no arquivo `HeaderComponent.vue` (Figura 14), que é o arquivo que contém a definição do componente e a implementação

Figura 12 – Arquivo WebComponent.js do componente Header

```
import { createApp } from "vue";
import App from "./App.vue";

class WebComponent extends HTMLElement {
  constructor() {
    super();
  }

  connectedCallback() {
    const app = createApp(App);
    app.mount(this);
  }
}

customElements.define("header-microfrontend", WebComponent);
```

Fonte: Autor.

Figura 13 – Configuração do *Plugin Module Federation* para o componente Header

```
plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
    }),
    new ModuleFederationPlugin({
      name: "HeaderMicrofrontend",
      filename: "bundle.js",
      exposes: {
        "./Microfrontend": "./src/WebComponent.js",
      },
      shared: {
        vue: {
          singleton: true,
        },
      },
    }),
    ...
```

Fonte: Autor.

Figura 14 – Componente Vue Header

```
1  <template>
2    <div class="header">
3      <div class="header-links">
4        <a href="">Link 1</a>
5        <a href="">Link 2</a>
6        <a href="">Link 3</a>
7        <a href="">Link 4</a>
8        <a href="">Link 5</a>
9      </div>
10   </div>
11 </template>
12
13 <style scoped>
14 .header {
15   background-color: palegreen;
16   height: 60px;
17   width: 100%;
18 }
19
20 .header-links {
21   padding: 16px;
22 }
23
24 .header-links a {
25   margin-right: 16px;
26 }
27 </style>
28
```

Fonte: Autor.

das funcionalidades.

6.5.2 Conteúdo Principal

Diferente dos componentes criados até então, o componente Main é o que mais se distingue dos demais, não pela tecnologia utilizada, mas pela forma como o *framework Angular* implementa a injeção como tag customizada.

No componente Main, a tag customizada é definida no arquivo `content.module.ts` (Figura 15), que é o arquivo responsável por definir o módulo do componente. A tag customizada é definida utilizando a função `createCustomElement` do pacote *elements* do *Angular*, que recebe como parâmetro o componente que será usado para a criação da classe que, posteriormente, será utilizada para a criação da tag customizada na linha 14.

Já quanto as configurações do plugin *Module Federation*, de forma semelhante aos outros componentes, as configurações foram realizadas no arquivo `webpack.config.js` (Figura 16), onde é possível ver os parâmetros `name` e `exposes`.

Figura 15 – Definição da tag customizada no componente Main

```
import { NgModule, Injector } from '@angular/core';
import { ContentComponent } from './content.component';
import { createCustomElement } from '@angular/elements';

@NgModule({
  declarations: [ContentComponent],
  exports: [ContentComponent],
})
export class ContentModule {
  constructor(private injector: Injector) {
    const el = createCustomElement(ContentComponent, {
      injector: this.injector,
    });
    customElements.define('main-microfrontend', el);
  }

  ngDoBootstrap() {}
}
```

Fonte: Autor.

Figura 16 – Definição da tag customizada no componente Main

```
plugins: [
    new container.ModuleFederationPlugin({
      name: 'MainMicrofrontend',
      filename: 'bundle.js',
      exposes: {
        './Microfrontend': './src/bootstrap.ts',
      },
    }),
  ],
```

Fonte: Autor.

O módulo que é exposto pelo componente Main é o `bootstrap.ts` (Figura 17) que é o arquivo que inicializa o *framework Angular*, de forma bem diferente do que ocorre nos componentes Header e Aside, onde o arquivo que é exposto é o responsável por definir a tag customizada.

Na Figura 18 é possível ver o HTML implementado para o conteúdo principal da aplicação, que é uma citação de um texto genérico em latim. Junto com os estilos definidos no arquivo `content.component.scss` (Figura 19).

Figura 17 – Arquivo bootstrap.ts do componente Main

```ts
import './polyfills';

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { WebComponentModule } from './app/web-component.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic()
  .bootstrapModule(WebComponentModule)
  .catch((err) => console.error(err));

```

Fonte: Autor.

Figura 18 – HTML do componente Main

```html
<blockquote class="box-text">
  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean fermentum
  egestas sodales. Praesent non dui ut quam fermentum fermentum. Sed non tellus
  ut nisl volutpat iaculis vitae in mauris. Nullam scelerisque ipsum non justo
  ornare commodo. Phasellus placerat tortor in ultrices facilisis. Lorem ipsum
  dolor sit amet, consectetur adipiscing elit. Duis id tortor sit amet leo
  lobortis bibendum. Etiam nisi risus, interdum et volutpat vitae, feugiat vitae
  est. Vestibulum eu erat in lectus convallis mattis sit amet et nunc. Donec
  egestas vehicula purus eget finibus.
</blockquote>
```

Fonte: Autor.

Figura 19 – Estilos do componente Main

```scss
.box-text {
  border: 8px dashed purple;
  padding: 8px;
  font-size: medium;
  font-family: "Brush Script MT", cursive;
}
```

Fonte: Autor.

6.5.3 Shell

O componente Shell, responsável por integrar os demais componentes, sofreu uma grande alteração da sua estrutura, de forma a comportar o leiaute da aplicação. A integração continua sendo realizada pelo componente *React* `System`, que recebe os parâmetros necessários para o carregamento e renderização dos componentes.

Figura 20 – Interfaces de integração do componente Shell

Fonte: Autor.

É possível ver na Figura 20 que o componente Shell define os parâmetros para a integração com os demais componentes, de acordo com a abordagem proposta. O novo leiaute da aplicação é definido no arquivo `App.js` (Figura 21).

6.5.4 Aside

O componente Aside, responsável pelo conteúdo adicional da aplicação, é o *microfrontend* da prova de conceito anterior, as únicas alterações realizadas foram na interface de integração, recebendo outro `name` (`AsideMicrofrontend`) e `tag-name` (`aside-microfrontend`).

6.6 Resultados

A segunda prova de conceito foi implementada e os objetivos propostos foram alcançados. A abordagem se mostrou realmente viável e agnóstica ao *framework* utilizado,

Figura 21 – Componente React App do *microfrontend Shell*

```
function App() {
  return (
    <div className="app-container">
      <div>
        <System system={HeaderMicrofrontend} />
      </div>
      <h1 className="title">Prova de Conceito 2</h1>
      <div className="box">
        <div className="main-box">
          <System system={MainMicrofrontend} />
        </div>
        <div className="aside-box">
          <System system={AsideMicrofrontend} />
        </div>
      </div>
    </div>
  );
}
```

Fonte: Autor.

de modo que o componente Shell não precisa conhecer os detalhes de implementação dos demais componentes.

É possível ver na Figura 22 a aplicação final, com o leiaute definido anteriormente e os componentes Header, Aside e Main integrados.

Figura 22 – Prova de conceito 2

Fonte: Autor.

6.6.1 Diferenças entre as tecnologias

Diferente da primeira prova de conceito, onde foi utilizado somente o *framework React*, a segunda prova de conceito combina três *frameworks* diferentes. Com isso é possível verificar as diferenças entre as tecnologias.

A que leva o maior destaque nesse quesito é o *framework Angular*, que possui uma estrutura de arquivos bem diferente das demais tecnologias, além de possuir uma interface própria para a definição de tags customizadas.

Além disso, cada *framework* possui uma ou mais formas diferentes de definir os estilos dos componentes, o que pode dificultar a integração em uma página só.

6.6.2 Conflitos de CSS

Um dos problemas encontrados durante a implementação foi o conflito de estilos entre os componentes. Quando o componente Main utiliza a classe CSS `.box` que é o mesmo nome de uma classe CSS definida no componente Shell, a aplicação apresenta-se como na Figura 23.

Figura 23 – Conflito de estilos entre os componentes Shell e Main

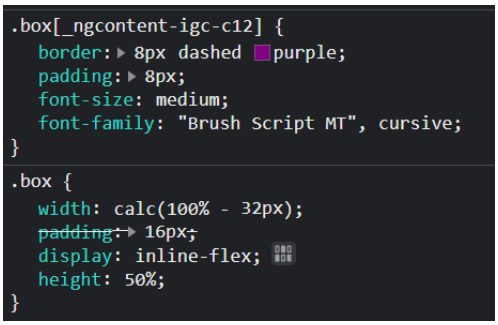
Fonte: Autor.

Inspecionando os elementos da página, é possível ver que o elemento `blockquote` possui a classe CSS `.box` e ambos os estilos são aplicados ao elemento, como mostrado na Figura 24. A gravidade de um conflito de CSS vai depender do estilo que está sendo aplicado, podendo causar problemas de visualização ou até mesmo de funcionalidade. De qualquer modo, é um problema que deve ser evitado.

Para evitar esse tipo de problema, uma das técnicas mais fáceis é utilizar um prefixo ou sufixo para classes CSS, como por exemplo `.box-main` e `.box-shell`. Outra técnica é utilizar *shadow DOM*, que é uma das especificações de Web Components.

É possível observar também que o *framework Vue* possui a configuração `scoped` para definir estilos que só serão aplicados ao componente, evitando conflitos com outros componentes.

Figura 24 – Conflito de estilos inspecionado no navegador



Fonte: Autor.

7 PoC 3 - Utilizado especificações de Web Components

A terceira prova de conceito é uma evolução da segunda, uma evolução que não afeta a arquitetura da aplicação, mas sim busca utilizar mais recursos disponíveis na especificação de *Web Components*.

7.1 Objetivos

O objetivo principal dessa prova de conceito é validar o terceiro objetivo da definição da abordagem combinada, relacionado ao uso das quatro especificações de *Web Components*: *Custom Elements*, *ES Modules*, *Shadow DOM* e *HTML Templates*, sendo que as duas primeiras já utilizadas tanto na primeira quanto na segunda prova de conceito.

O recurso de *Shadow DOM* é utilizado para encapsular os estilos do *microfrontend*, de modo que não afetem o restante da aplicação. Já a especificação de *HTML Templates*, que permite a criação de templates HTML, também será utilizada, de modo que o componente responsável pelo conteúdo principal da aplicação possa fazer uso de um template HTML definido no componente Shell.

7.2 Requisitos

A seguir são listados os requisitos que devem ser atendidos pela prova de conceito 3, os quais são os mesmos da prova de conceito 2, com a adição de dois novos requisitos:

- Os *microfrontends* devem utilizar o recurso de *Shadow DOM*.
- O *microfrontend* responsável pelo conteúdo adicional deverá permitir que o conteúdo seja alternado entre duas opções: o conteúdo definido pelo *microfrontend* e o conteúdo definido pela aplicação Shell.

7.3 Arquitetura

A arquitetura da prova de conceito 3 é a mesma da prova de conceito 2, o mesmo segue para os principais pacotes e componentes. A única diferença é que os *microfrontends* agora passam a utilizar o recurso de *Shadow DOM*, encapsulando os estilos do *microfrontend*.

7.4 Desenvolvimento

Os pontos principais do desenvolvimento são a utilização do recurso de *Shadow DOM* e a implementação da alternância de conteúdo. A utilização do recurso de *Shadow DOM* foi feita em todos os *microfrontends*, com exceção da aplicação Shell.

7.4.1 Utilização do recurso de *Shadow DOM*

A configuração do recurso no componente Main foi feita por meio da propriedade `ViewEncapsulation.ShadowDom` do pacote `angular/core`, que é responsável por encapsular os estilos do componente. A configuração foi feita no arquivo `content.component.ts` (Figura 25)

Figura 25 – Habilitando *Shadow DOM* no componente Main

Fonte: Autor.

A configuração do recurso no componente Aside foi feita por meio da propriedade `attachShadow` do objeto `HTMLElement`, que é responsável por criar uma instância do *Shadow DOM* no elemento HTML, o elemento em questão é a tag customizada criada para o componente Aside (Figura 26).

Já nos componentes Header, a configuração ficou similar, com a diferença de que no componente Header foi necessário copiar o elemento `link` (Figura 27), que importa o arquivo de estilos do componente, para que o mesmo fosse importado no contexto do *Shadow DOM*. Além disso, o *plugin* `MiniCssExtractPlugin` foi configurado para que a tag `link`, tivesse a propriedade `id` para que o mesmo fosse referenciado na montagem do *component* customizado (Figura 28).

7.4.2 Alternância de conteúdo

A alternância de conteúdo foi feita por meio da criação de um elemento `template` n aplicação Shell, e no componente Aside foi implementada uma função que faz a troca

Figura 26 – Habilitando *Shadow DOM* no componente Aside

```
class Microfrontend extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    ReactDOM.render(<App />, this.shadowRoot);
  }
}
```

Fonte: Autor.

Figura 27 – Habilitando *Shadow DOM* no componente Header

```
class WebComponent extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const app = createApp(App);
    app.mount(this.shadowRoot);
    const styleLink = document.getElementById("header-style");
    this.shadowRoot.appendChild(styleLink.cloneNode(true));
  }
}
```

Fonte: Autor.

Figura 28 – Configuração do *plugin* `MiniCssExtractPlugin` no componente Header

```
new MiniCssExtractPlugin({
      filename: "[name].css",
      attributes: {
        id: "header-style",
      },
    }),
```

Fonte: Autor.

do conteúdo padrão pelo conteúdo definido na aplicação Shell (Figura 29), essa função é chamada a cada troca da variável que armazena o estado da alternância.

Figura 29 – Função que faz a troca de conteúdo

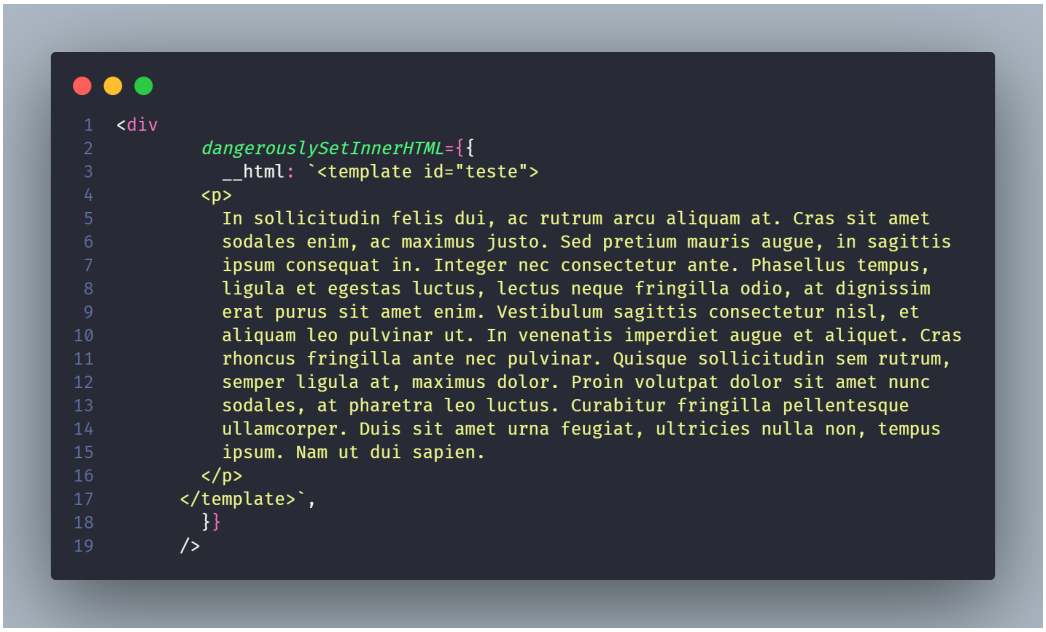
```
const loadShellContent = useCallback(() => {
    const shellContent = document.getElementById("shell-aside-content");
    if (shellContent) {
      const fragment = shellContent.content.cloneNode(true);
      let fragmentChilds = fragment.childNodes.length;

      while (fragmentChilds-- > 1) {
        rootDiv.current.appendChild(fragment.firstChild);
      }
    }
  });
```

Fonte: Autor.

Um problema que foi encontrado é que o componente Shell não estava conseguindo definir o elemento `template` com o conteúdo corretamente, o problema foi resolvido utilizando a propriedade `dangerouslySetInnerHTML` (Figura 30) do pacote `react-dom`.[1]

Figura 30 – Definição do elemento `template` no componente Shell



```
<div
        dangerouslySetInnerHTML={{
          __html: `<template id="teste">
        <p>
          In sollicitudin felis dui, ac rutrum arcu aliquam at. Cras sit amet
          sodales enim, ac maximus justo. Sed pretium mauris augue, in sagittis
          ipsum consequat in. Integer nec consectetur ante. Phasellus tempus,
          ligula et egestas luctus, lectus neque fringilla odio, at dignissim
          erat purus sit amet enim. Vestibulum sagittis consectetur nisl, et
          aliquam leo pulvinar ut. In venenatis imperdiet augue et aliquet. Cras
          rhoncus fringilla ante nec pulvinar. Quisque sollicitudin sem rutrum,
          semper ligula at, maximus dolor. Proin volutpat dolor sit amet nunc
          sodales, at pharetra leo luctus. Curabitur fringilla pellentesque
          ullamcorper. Duis sit amet urna feugiat, ultricies nulla non, tempus
          ipsum. Nam ut dui sapien.
        </p>
      </template>`,
        }}
      />
```

Fonte: Autor.

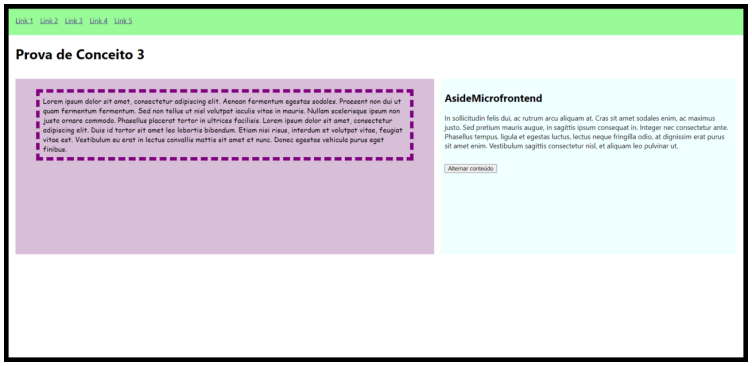
[1] De acordo com a comunidade, o problema ocorre devido a utilização da tag `template`. Disponível para acesso em: <https://github.com/preactjs/preact/issues/3444>

7.5 Resultados

A PoC 3 atendeu a todos os requisitos, na Figura 31 é possível ver a aplicação com o conteúdo alternado. Acerca da utilização de *Shadow DOM* para encapsular os estilos, foi possível verificar que os isolamentos funcionaram corretamente, o conflito de estilos que ocorreu na PoC 2 não ocorreu nesta.

Quanto às lógicas que foram implementadas no componente Header, caso o componente Aside utilizasse o mesmo plugin, seria necessário fazer a mesma configuração, para que os estilos fossem importados corretamente no contexto do *Shadow DOM*. Já o componente Main, no quesito de estilos, mostrou-se o mais simples, pois não foi necessário fazer nenhuma configuração para que os estilos fossem importados corretamente.

Figura 31 – Aplicação com o conteúdo definido pelo *microfrontend* Shell



Fonte: Autor.

Já a utilização do `template` para definir o conteúdo alternado, mostrou-se uma ferramenta simples e eficiente, embora com problemas com o `react`. Com isso, pode ser possível utilizar de lógica semelhante para a implementação de algum nível de comunicação sentido único entre os *microfrontends*, onde um componente pode definir o conteúdo de outro componente.

8 PoC 4 - Microfrontend nativo

A última prova de conceito é uma evolução da terceira, uma evolução se propõe a acrescentar novas funcionalidades à aplicação, por meio de um novo *microfrontend*. Este que será desenvolvido de forma nativa, ou seja, sem a utilização de *frameworks frontend*.

8.1 Objetivos

O objetivo principal atrelado à essa prova de conceito é validar o quarto objetivo da definição da abordagem combinada, que é o desenvolvimento nativo de *microfrontends*. Com isso, espera-se mostrar que é a abordagem combinada é capaz de suportar a API nativa JavaScript para manipulações no DOM.

De forma complementar, pode-se estender a validação da abordagem para futuros *frameworks* e tecnologias que façam uso das mesmas APIs de manipulação no DOM. Contribuindo assim para o principal objetivo da abordagem combinada, que é ser agnóstica ao *framework*. De modo que todas as provas de conceito realizadas até o momento contribuem para esse objetivo principal.

8.2 Requisitos

Os requisitos a serem adicionados na prova de conceito 3 para a construção da prova de conceito 4 são os seguintes:

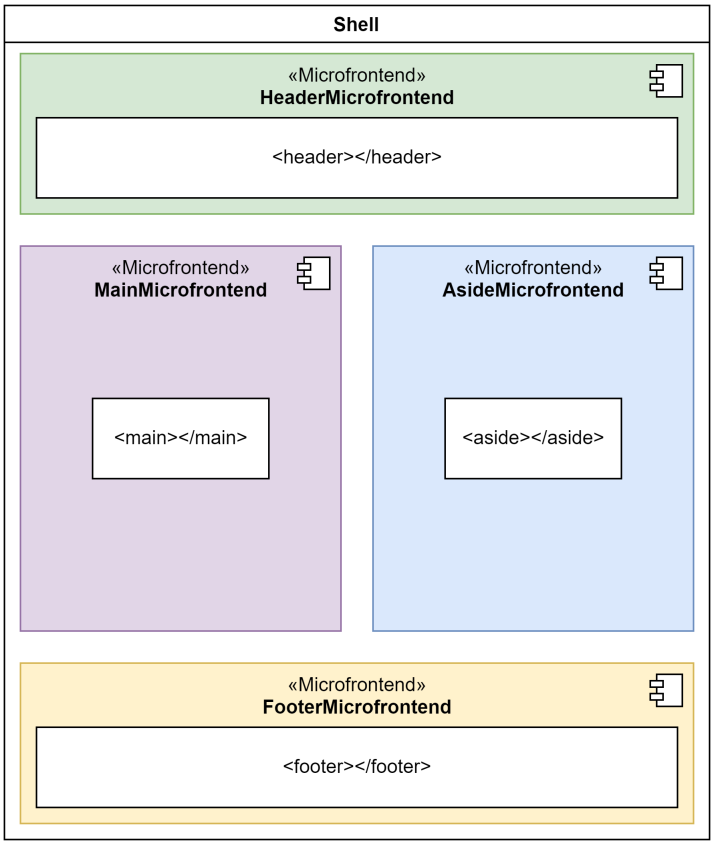
- A aplicação deverá conter um rodapé divido em três seções menores.
- O novo *microfrontend* deverá ser desenvolvido de forma nativa, sem a utilização de um *framework frontend*.
- O novo *microfrontend* deverá ser responsável por exibir o conteúdo do rodapé.

8.3 Arquitetura

A arquitetura da prova de conceito 4, que é uma evolução da prova de conceito 3, é mostrada na Figura 32.

Embora o *microfrontend* a ser desenvolvido seja nativo, ele também faz uso do *WebPack* para empacotar o código JavaScript e CSS. O plugin *Module Federation do WebPack* é utilizado em conjunto com *Web Components* para a integrar o componente nativo à aplicação Shell.

Figura 32 – Leiaute da aplicação da prova de conceito 4



Fonte: Autor.

8.4 Pacotes Utilizados

Os pacotes utilizados na prova de conceito se mantiveram os mesmos da prova de conceito 2, com a adição do componente Footer, que é o novo *microfrontend* nativo. A tabela 5 mostra os pacotes utilizados nesse componente.

8.5 Desenvolvimento

O desenvolvimento do *microfrontend* nativo não apresentou grandes diferenças em relação aos outros *microfrontends* desenvolvidos até o momento. A única diferença é como o conteúdo é inserido no DOM. O componente Footer é desenvolvido de forma nativa, sem a utilização de *frameworks frontend*. Em relação à aplicação Shell, o leiaute e os estilos da aplicação foram alterados para que o componente Footer fosse inserido corretamente.

8.5.1 Integração

A integração do componente nativo com a aplicação Shell é feita seguindo a interface definida pela abordagem. O componente nativo é empacotado como um módulo

Tabela 5 – Pacotes utilizados no componentes Footer

Pacote	Versão
@babel/core	7.18.9
babel-loader	8.2.5
css-loader	6.7.1
file-loader	6.2.0
html-webpack-plugin	5.5.0
mini-css-extract-plugin	2.6.1
serve	13.0.4
url-loader	4.1.1
webpack	5.75.0
webpack-cli	5.0.0
webpack-dev-server	4.11.1

Fonte: Autor.

remoto pelo *WebPack*, e é importado pelo componente Shell. O componente nativo é importado como um *Web Component*, e é inserido no DOM da aplicação Shell, sua interface pode ser visualiza na Figura 33.

Figura 33 – Integração do componente Footer na aplicação Shell

Fonte: Autor.

8.5.2 Estilos

Os estilos do componente Footer são definidos no arquivo *footer.css*, que é empacotado pelo *WebPack*. Por conta do shadow DOM, os estilos precisam ser carregados na subárvore DOM do componente nativo. Para isso, foi usado lógica semelhante à utilizada no componente Header, para que os estilos funcionem corretamente.

8.5.3 Ciclo de Vida

O ciclo de vida do componente customizado é definido na própria interface do DOM, é necessário a utilização dos mesmos para que o componente seja rende-

rizado corretamente. Em especial foram utilizados os eventos `connectedCallback` e `disconnectedCallback`.

O evento `connectedCallback` é disparado quando o componente é inserido no DOM da página. Nesse evento é realizada a lógica de carregar os estilos do componente, e o seu conteúdo. Já o evento `disconnectedCallback`, que é disparado quando o componente é removido do DOM, é utilizado para remover os estilos do componente.

8.6 Resultados

A aplicação referente a esta prova de conceito pode ser visualizada na Figura 34, onde é possível observar a leve mudança no leiaute da aplicação, e a inserção do novo componente Footer. Como esperado, o componente nativo funcionou corretamente, e não apresentou problemas de compatibilidade com os demais componentes.

Figura 34 – Aplicação com o footer nativo

Fonte: Autor.

As complexidades encontradas na prova de conceito 4 são referentes a implementação de interfaces Web nativas. Os *frameworks frontend* utilizados nas outras provas de conceito, por abstraírem as APIs nativas, facilitam o desenvolvimento de interfaces Web. Porém, a utilização de *frameworks* também trazem complexidades, como o aumento do tamanho do bundle, e a necessidade de manter a compatibilidade com os navegadores mais antigos.

9 Análise de Resultados e Discussão

Graças às provas de conceito, foi possível colher insumos importantes sobre a abordagem combinada, e até sobre a viabilidade da sua utilização em um cenário real. A seguir, serão discutidos os resultados obtidos no trabalho e alguns aspectos que foram observados durante a exploração da abordagem.

A segunda prova de conceito foi a mais complexa, quando comparada com as outras três. Visto que, com ela foi possível explorar a abordagem em um cenário mais próximo do desejado quanto ao uso de múltiplos *frameworks* na mesma aplicação. Além disso, foi possível visualizar os aspectos relacionados a integração dos *frameworks* com a interface base da abordagem.

Dois grandes aspectos foram observados, e que podem ser considerados como pontos de atenção para a utilização da abordagem. O primeiro deles é a necessidade de se utilizar um *framework* que seja compatível com *Web Components*, e que implemente as abstrações definidas na interface do DOM.

No que tange o primeiro aspecto, foi possível notar que a utilização do *framework Angular* sugere que alguns *frameworks* podem abordar a declaração de componentes customizados de maneira diferente. Embora não seja um problema, visto que os componentes customizados possuem escopo global.

O segundo aspecto vem da necessidade de gerenciar os estilos dos *microfrontends* e o seu relacionamento com o encapsulamento proposto com *Shadow DOM*. Como foi observado na segunda prova de conceito, foi necessário configurar o *plugin MiniCssExtractPlugin* do *Webpack* para que os estilos fossem extraídos e injetados na subárvore do *Shadow DOM* dentro do componente.

Esse último, ainda mais importante visto que os problemas de estilização podem tornar a integração dos componentes mais complexa, gerar complexidades para a manutenção, problemas relacionados a experiência do usuário e o risco da aplicação ficar inconsistente.

Com a quarta prova de conceito, foi possível observar como a integração da abordagem com os *frameworks React* e *Vue* são semelhantes a integração feita no componente nativo. Isso pode ser considerado um ponto positivo, visto que a abordagem pode não precisar implementar uma solução específica para cada *framework*.

De modo geral, é possível observar que a abordagem se baseia em alterar levemente a forma como os *frameworks* são integrados ao DOM. Sendo que para o devido funcionamento da mesma, é necessário que os *frameworks* sejam integrados por meio de

componentes customizados.

Durante as provas de conceito, foi possível notar também como a utilização da interface na aplicação base é agnóstica, em nenhum momento foi necessário algum contorno pelo lado da aplicação base para que um *microfrontend* fosse integrado, tampouco devido a tecnologia que o mesmo empregava. Isso pode permitir até customizações mais complexas, como por exemplo, carregar *microfrontends* de acordo com a resposta de uma API.

Poucos recursos do *WebPack* foram utilizados, com exceção do *plugin MiniCssExtractPlugin* e o *plugin ModuleFederationPlugin*. Este último responsável pela integração e o carregamento remoto dos módulos. Em casos de uso mais complexos, pode ser necessário utilizar outros recursos do *Webpack*, como por exemplo otimizações de performance.

A utilização de *HTML Templates* não agrega valor para a abordagem do ponto de vista de integração de *microfrontends*. Mas, pode ser útil para a criação de comunicações unilaterais e agnósticas entre os diferentes componentes. Devido a maneira como o HTML é estruturado, é recomendado que o próprio *microfrontend* defina onde colocar o conteúdo vindo de outro *microfrontend*.

10 Considerações finais

Em suma, este trabalho se dedicou a propor e explorar uma nova abordagem para a implementação de *microfrontends* utilizando *Web Components* e *WebPack Module Federation*, com foco na criação de uma interface que seja agnóstica em relação ao *framework Frontend* que está sendo utilizado, motivado também pela possibilidade de combinar abordagens já existentes, agregando as vantagens de cada uma delas.

Embora a metodologia utilizada baseada em provas de conceito tenha sido eficaz para a exploração da abordagem, é necessário que sejam realizados testes mais robustos e com maior escopo, para que se possa ter uma visão mais completa da abordagem, e principalmente para que se possa avaliar a sua viabilidade em um cenário real.

A elaboração de provas de conceitos incrementais, com um escopo bem definido para cada uma delas, foi uma estratégia utilizada para que se pudesse explorar a abordagem de forma gradativa, e que se pudesse avaliar os resultados obtidos em cada uma delas, para que se pudesse decidir se era necessário continuar a exploração ou não. Graças às provas de conceito, foi possível atingir os objetivos propostos.

Quanto aos trabalhos futuros, é possível explorar a abordagem em um cenário real, com um escopo maior, uma aplicação mais complexa, e com casos de uso mais reais. Além disso, é possível explorar a abordagem em um cenário onde a aplicação é composta por *microfrontends* e microsserviços, onde os dados exibidos são obtidos de maneira dinâmica das APIs.

Ainda sim, podem ser necessárias investigações mais aprofundadas acerca da viabilidade da abordagem, e como aplicações com múltiplos *frameworks* podem ser implementadas com a mesma. E também, os possíveis problemas que podem surgir com a utilização dessa implementação, além dos seus impactos sobre a arquitetura da aplicação como um todo.

Outro ponto que pode ser explorado é a possibilidade de utilizar a abordagem em conjunto com arquiteturas de *microfrontends* no lado do servidor. Podendo elevar o nível de interoperabilidade para além de só componentes no lado do cliente. Na mesma linha, a aplicação base pode utilizar dos conceitos de renderização no lado do servidor, para melhorar a indexação dos *microfrontends* pelos mecanismos de busca.

Por tudo isso, espera-se que esta abordagem possa ser explorada e utilizada em cenários reais, e que possa ser uma alternativa viável para a implementação de *microfrontends*. Permitindo que os times de desenvolvimento possam ter mais autonomia, e que os desenvolvedores possam escolher o *framework* que melhor se adéqua ao seu projeto e

objetivos.

Referências

ALMEIDA, D. *Micro Frontends para aplicações Web*. Dissertação (Mestrado) — Instituto Superior de Engenharia do Porto, 2021. Citado 11 vezes nas páginas 27, 32, 33, 34, 36, 37, 38, 39, 40, 41 e 43.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 3th. ed. USA: Addison-Wesley Publishing Company, 2012. ISBN 978-0321815736. Citado 3 vezes nas páginas 27, 33 e 43.

FOWLER, M. Microservices. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado na página 34.

HéGARET, P. L.; W3C. *What is the Document Object Model?* [S.l.], 2004. Disponível em: <https://www.w3.org/TR/DOM-Level-3-Core/introduction.html>. Citado na página 32.

JACKSON, C. Micro frontends. 2019. Disponível em: <https://martinfowler.com/articles/micro-frontends.html>. Citado 6 vezes nas páginas 35, 36, 38, 39, 41 e 49.

KLIMM, M. C. *Design Systems for Micro Frontends*: An investigation into the development of framework-agnostic design systems using svelte and tailwind css. Dissertação — TH Köln - University of Applied Sciences, 2021. Citado 5 vezes nas páginas 36, 37, 38, 39 e 43.

MELNIKOV, A.; KUCHERAWY, M. *Media Types*. [S.l.], s.d. Disponível em: <https://www.iana.org/assignments/media-types/media-types.xhtml>. Citado na página 31.

MODULE Federation. 2022. Disponível em: <https://webpack.js.org/concepts/module-federation/>. Citado 2 vezes nas páginas 41 e 55.

POLOSKEI, I.; BUB, U. Enterprise-level migration to micro frontends in a multi-vendor environment. *Acta Polytechnica Hungarica*, v. 18, p. 7–25, 01 2021. Citado 2 vezes nas páginas 41 e 43.

PRAJWAL, Y. R.; PAREKH, J. V.; SHETTAR, D. R. A brief review of micro-frontends. *United International Journal for Research & Technology*, v. 2, n. 8, 2021. Citado 4 vezes nas páginas 36, 37, 38 e 43.

SILVA, R. da. *A Micro Frontends Solution - Analyzing quality attributes*. Dissertação (Mestrado) — Instituto Superior de Engenharia do Porto, 2021. Citado 8 vezes nas páginas 36, 37, 38, 39, 40, 41, 42 e 43.

TANENBAUM, A.; STEEN, M. v. *Distributed Systems: Principles and Paradigms*. 2nd. ed. USA: Pearson Higher Education, 2013. ISBN 978-1292025520. Citado 5 vezes nas páginas 31, 33, 34, 43 e 49.

WEB Components - Introduction. 2022. Disponível em: <https://www.webcomponents.org/introduction>. Citado na página 35.

WEBPACK - Concepts. 2022. Disponível em: <https://webpack.js.org/concepts/>. Citado na página 34.

YANG, C.; LIU, C.; SU, Z. Research and application of micro frontends. *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, v. 490, p. 062082, apr 2019. Disponível em: <https://doi.org/10.1088/1757-899x/490/6/062082>. Citado 2 vezes nas páginas 40 e 43.