



Micro frontends numa aplicação de pré-contabilidade

NICOLE RODRIGUES SILVA

outubro de 2023

Micro *frontends* numa aplicação de pré-contabilidade

Nicole Rodrigues Silva

**Dissertação para obtenção do Grau de Mestre em
Engenharia Informática, Área de Especialização em
Engenharia de Software**

Orientadora: Isabel de Fátima Silva Azevedo

Porto, outubro 2023

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 16 de October de 2023

Nicole Rodrigues Silva

Resumo

O trabalho descrito neste documento foi realizado na empresa Basecone e tem como intuito explorar a arquitetura de micro *frontends*, de modo que a empresa possa futuramente aprimorar a arquitetura da sua aplicação e adotar práticas inovadoras que podem potencialmente otimizar a eficiência e flexibilidade do desenvolvimento de software.

Ao longo do projeto foi conduzida uma pesquisa no contexto dos micro *frontends*, abordando práticas, identificação de anti padrões e exploração de tecnologias associadas.

Uma arquitetura de micro *frontends* pode nem sempre se adequar em determinada situação, por isso é sempre necessário considerar-se o contexto, a organização das equipes e atributos de qualidade que importam. Desse modo, o sistema da *Basecone* é um produto *web* para automação de processos de pré-contabilidade, que suporta várias ferramentas contábilísticas e a ligação a novas ferramentas tem vindo a ser planeada. Além disso, o desenvolvimento é distribuído por equipes multidisciplinares, cada uma responsável por ferramentas contábilísticas distintas, sendo a autonomia das equipes, a facilidade de implantação, a manutenibilidade e o desempenho da solução, os fatores que se buscam enfatizar.

Para isso, foi realizada uma prova de conceito com dimensões mais reduzidas que procurou representar parte do sistema atual da empresa, o que permitiu estabelecer uma análise mais controlada, com uma comparação clara entre uma implementação com o *frontend* monolítico e os micro *frontends*.

Os resultados obtidos permitiram compreender se a arquitetura de micro *frontends* se adequa ao caso estudado e se é capaz de responder de forma mais eficaz aos desafios enfrentados pela empresa. Comparativamente à arquitetura vigente, os micro *frontends* apresentaram-se como uma solução promissora.

Espera-se que esta pesquisa não apenas beneficie a empresa Basecone, mas também sirva como referência para casos semelhantes que considerem a adoção desta opção arquitetural.

Palavras-chave: micro *frontends*, *frontend* monolítico, module federation, desenvolvimento *web*

Abstract

This document focuses on the information gathered and provided at the company Basecone. It aims to grant insights about micro frontends architecture to provide knowledge to enhance the architecture of projects and adopt innovative practices that could potentially optimize the efficiency and flexibility of software development.

Throughout the project, research was conducted in the context of micro frontends, encompassing best practices, identification of anti-patterns, and exploration of associated technologies.

For any given situation, micro frontends may not always be suitable, therefore, is pivotal to regard their substance as well as understanding how teams are coordinated and conclude which quality attributes will be emphasized. In this regard, the Basecone system is a web product for automating pre-accounting processes. It supports various accounting tools, and the integration with new tools have been under planning. Additionally, development is carried out by multidisciplinary teams, each responsible for distinct accounting tool. Team independence, ease of deployment, maintainability, and solution performance are the emphasized factors.

To address this was conducted a proof of concept on a smaller extent to represent part of the company's current system. It allowed a more controlled analysis and a transparent comparison between an implementation with monolithic frontend and micro frontends.

The obtained results helped to understand if the micro frontends architecture is suitable for the studied case and if it can better address the challenges faced by the company. Compared to the current architecture, micro frontends emerged as a promising solution.

It is hoped that this research will not only benefit the Basecone company but also serve as a reference for similar cases considering the adoption of this architectural option.

Keywords: micro frontends, monolithic frontend, module federation, web development

Agradecimentos

Gostaria de agradecer, em primeiro lugar, à minha família, nomeadamente ao meu irmão, que sempre me incentivou a continuar o curso que queria e ajudou-me a superar todas as adversidades, e aos meus pais, que se esforçaram para que nunca me faltasse nada e sempre me apoiaram nas minhas decisões.

Agradeço a todos os intervenientes deste projeto, especialmente à minha orientadora, Isabel Azevedo, que sempre se preocupou em ajudar-me quando precisava, e ao meu supervisor, Samuel Rodrigues, que me auxiliou em todo o processo e procurou clarificar todas as minhas dúvidas.

Além do meu supervisor, gostaria de agradecer a todas as pessoas da Basecone, que proporcionaram um bom ambiente de trabalho, nomeadamente a todos os integrantes da minha equipa, os Tomahawk, e também ao meu manager, Tiago Freire.

Gostaria de agradecer também ao Bino por ser o meu porto de abrigo. Estou grata por todo o suporte que me deu e pela paciência que sempre teve comigo de todas as vezes que me fui abaixo.

Agradeço a todos os que se cruzaram comigo nesta caminhada e que contribuíram de alguma forma para ser a pessoa que sou hoje.

Índice

1	Introdução	1
1.1	Contexto.....	1
1.2	Contexto Empresarial.....	2
1.3	Problema	3
1.4	Objetivo	3
1.5	Metodologia de Investigação.....	3
1.6	Estrutura do Documento	4
2	Estado da Arte	5
2.1	Metodologia	5
2.2	Arquitetura Monolítica do <i>Frontend</i>	6
2.3	Arquitetura de Micro <i>Frontends</i>	8
2.3.1	Identificação	10
2.3.2	Composição	11
2.3.3	Routing.....	13
2.3.4	Comunicação.....	14
2.4	Anti Padrões	15
2.4.1	Micro <i>Frontends</i> e Componentes	15
2.4.2	Múltiplas <i>Frameworks</i>	16
2.4.3	Dependências Excessivas	16
2.4.4	Acoplamento <i>Design-time</i>	17
2.5	Tecnologias e <i>Frameworks</i>	17
2.5.1	Single-SPA	17
2.5.2	Mooa	18
2.5.3	Module Federation	19
2.5.4	Import Maps.....	20
2.5.5	Native Federation	21
2.6	Estrutura das Equipas.....	22
2.7	Exemplos Empresariais	23
2.7.1	IKEA	23
2.7.2	DAZN	24
2.7.3	Spotify	24
2.8	Sumário	25
3	Contexto da Empresa e Problema.....	27
3.1	Empresa	27
3.2	Equipas	28
3.3	Estado Atual do Produto	29
3.3.1	Problemas	30

3.3.2	Planos.....	31
4	Análise de Valor	33
4.1	Fuzzy front end.....	33
4.2	New Concept Development Model	34
4.2.1	Identificação da Oportunidade	35
4.2.2	Análise da Oportunidade	36
4.2.3	Definição de Conceito	39
4.3	AHP.....	40
5	Projeto Base	45
5.1	Design.....	45
5.2	Descrição da <i>Stack</i> Tecnológica.....	47
5.2.1	Microserviços	47
5.2.2	<i>Frontend</i> Monolítico	47
5.3	Implementação	48
6	Requisitos	53
7	Solução	55
7.1	Design.....	55
7.2	Descrição da <i>Stack</i> Tecnológica.....	57
7.2.1	<i>Micro Frontends</i>	57
7.3	Implementação	58
7.3.1	Configuração Inicial.....	59
7.3.2	Implementação	61
7.3.3	Implantação.....	62
8	Experimentação e Avaliação	67
8.1	GQM.....	67
8.2	Ferramentas de Análise	69
8.3	Resultados das Métricas	70
8.3.1	Manutenibilidade.....	70
8.3.2	Gestão de Dependências e Versões	72
8.3.3	Performance	74
8.3.4	Deployability.....	75
8.4	Resumo dos Resultados.....	78
9	Conclusões	79
9.1	Objetivos Alcançados	79
9.2	Dificuldades e Trabalho Futuro	80
9.3	Apreciação Final	80

Apêndice A	86
------------------	----

Lista de Figuras

Figura 1 - Produto de pré-contabilidade	2
Figura 2 - Aplicação Monolítica	6
Figura 3 - Microsserviços com <i>single-page application</i>	7
Figura 4 - Microsserviços com <i>micro frontends</i>	8
Figura 5 - <i>Framework</i> de decisão de <i>micro frontends</i>	10
Figura 6 – Divisão Horizontal e Vertical	11
Figura 7 - Composição <i>server-side</i>	12
Figura 8 - Composição <i>edge-side</i>	12
Figura 9 - Composição <i>client-side</i>	13
Figura 10 - <i>Routing</i> de <i>micro frontends</i>	13
Figura 11 - Conceito de MOOA	18
Figura 12 -Comando para gerar a <i>Shell</i> dinâmica	20
Figura 13 – Desempenho de diferentes bundlers.....	22
Figura 14 - Produto de pré-contabilidade.....	27
Figura 15 – <i>Don't Repeat Yourself</i>	30
Figura 16 - Processo de Inovação.....	34
Figura 17 – New Concept Development	35
Figura 18 - <i>Microservices Adoption in 2020</i> : Benefícios no uso de microsserviços	37
Figura 19 - <i>Microservices Adoption in 2020</i> : Sistemas com microsserviços que usam uma UI monolítica	38
Figura 20 – Árvore de decisão hierárquica	40
Figura 21 - Escala fundamental	41
Figura 22 - Página de registo de faturas	46
Figura 23 - Diagrama de Componentes: microsserviços com frontend monolítico	46
Figura 24 – Página inicial.....	48
Figura 25 – Início da página de <i>Post Invoices</i>	49
Figura 26 – Parte inferior da página de <i>Post Invoices</i>	50
Figura 27 - Cálculo do VAT	50
Figura 28 - Diagrama de Casos de Uso	53
Figura 29 - Diagrama de Componentes: microsserviços com <i>micro frontends</i>	56
Figura 30 - Diagrama de Componentes: <i>micro frontends</i>	56
Figura 31 - Página de Registo de Faturas composta por <i>micro frontends</i>	58
Figura 32 – Excerto <i>app.routes.ts</i> da <i>Shell</i> : caminho para o <i>Previewer</i>	59
Figura 33 - Excerto <i>mf.manifest.json</i> da <i>Shell</i> : definição do <i>remoteEntry</i> do <i>Previewer</i>	60
Figura 34 - <i>Webpack.config.js</i> do <i>Previewer</i>	60
Figura 35 – Uso da função <i>loadManifest</i> no <i>main.ts</i> da <i>Shell</i>	61
Figura 36 - Método para espoletar um evento usando <i>Custom Events</i>	62
Figura 37 - <i>Dockerfile</i> dos <i>micro frontends</i>	62
Figura 38 - Diagrama de Implantação dos <i>micro frontends</i>	63
Figura 39 – <i>Ingress</i> no ficheiro <i>YAML</i> do <i>Invoice General</i>	64

Figura 40 - <i>Service</i> no ficheiro <i>YAML</i> do <i>Invoice General</i>	64
Figura 41 – <i>Deployment</i> no ficheiro <i>YAML</i> do <i>Invoice General</i>	65
Figura 42 - Plano <i>GQM</i>	68
Figura 43 - Definição de Complexidade do <i>SonarQube</i>	69
Figura 44 - Método <i>shareAll</i>	73
Figura 45 - Método <i>share()</i>	73
Figura 46 - Resultados do <i>Lighthouse: Frontend</i> Monolítico	74
Figura 47 - Resultado do <i>Lighthouse: Micro Frontends</i>	75

Lista de Tabelas

Tabela 1 - Comparação de critérios	41
Tabela 2 - Comparação de critérios - valores normalizados	42
Tabela 3 - Vetor de Prioridade	42
Tabela 4 - Matriz de comparação para cada critério por cada alternativa	43
Tabela 5 - Matriz alternativa e vetor final.....	44
Tabela 6 - Responsabilidade de cada equipa	48
Tabela 7 - Distribuição dos microsserviços e micro <i>frontends</i>	58
Tabela 8 - Escala de classificação da manutenibilidade	71
Tabela 9 - Análise do <i>SonarQube</i>	71
Tabela 10 - Exemplo de gestão de versões	72
Tabela 11 – Métricas de Performance	75
Tabela 12 - Duração da <i>build</i> e <i>deploy</i> de uma execução de pipeline	76
Tabela 13 - <i>Frontend</i> monolítico vs. Micro <i>Frontends</i>	78

Lista de Blocos de Código

Código 1 – Uso do <i>importmap</i> no <i>HTML</i>	20
Código 2 – Uso do <i>importmap</i> no <i>JS</i>	21
Código 3 - Comando para inicializar os projetos <i>remote</i>	59
Código 4 - Comando para inicializar o projeto da <i>Shell</i> dinâmica.....	59
Código 5 – Função <i>loadManifest</i> e seus parâmetros.....	61

Acrónimos e Símbolos

Lista de Acrónimos

ACR	Azure Container Registry
AHP	Analytic Hierarchy Process
AKS	Azure Kubernetes Service
API	Application Programming Interface
CDN	Content Delivery Network
CI/CD	<i>Continuous Integration/Continuous Delivery</i>
CSI	Client Side Includes
DRY	Don't Repeat Yourself
ECMAScript	European Computer Manufacturers Association Script
ESI	Edge Side Includes
ESModule	EcmaScript module
FCP	First Contentful Paint
FFE	Fuzzy Front End
GQM	Goal, Question, Metric
IA	Índice Aleatório
IC	Índice de Consistência
LCP	Largest Contentful Paint
MVP	Minimum Viable Product
NCD	New Concept Development
NPD	New Product Development
OCR	Optical Character Recognition
POC	Proof Of Concept
PoP	Points of Presence

RC	Rácio de Consistência
SEO	Search Engine Optimization
SPA	Single-Page Application
SSI	Server Side Includes
TBT	Total Blocking Time
UI	User Interface
UX	User experience
VAT	Value-Added Tax

Lista de Símbolos

μ	média
-------	-------

1 Introdução

Neste capítulo apresenta-se o contexto, problema, objetivo, metodologia de pesquisa e estrutura do documento.

1.1 Contexto

Micro *frontend* trata-se de um conceito aplicado na camada de *frontend*, inspirado na ideia de microsserviços. Quando se tem um *frontend* monolítico, significa que, embora o *backend* esteja fragmentado, o *frontend* permanece como uma entidade monolítica, sendo partilhado e desenvolvido por diferentes equipas. Isto torna os sistemas bastante complexos e difíceis de manter.

Assim sendo, pretende-se, com uma arquitetura de micro *frontends*, dividir esse monólito em partes de menores dimensões - micro *frontends* - de modo a permitir que várias equipas trabalhem de forma independente, minimizando a partilha de código pertencente a diferentes domínios de negócio. Portanto, no caso de equipas multidisciplinares, cada equipa é especializada numa área específica do negócio, assegurando uma funcionalidade de ponta a ponta, isto é, no seu todo, da base de dados até à interface do utilizador (Geers, no date).

Ao dividir diferentes responsabilidades em diferentes equipas, os programadores podem desenvolver, testar individualmente e implantar facilmente um recurso específico sem bloquear ou depender de outras equipas, ajudando na integração, implantação e entrega contínuas (Yang, Liu and Su, 2019).

Levando em consideração tudo isso, sabe-se que é crucial neste mercado tão competitivo encurtar o *time to market*, poder ter *feedback loops* recorrentes com o utilizador e gastar o mínimo de tempo possível resolvendo *bugs* e fazendo configurações no projeto (Mezzalana, 2021).

1.2 Contexto Empresarial

A Basecone é uma empresa que desenvolve produtos *web* para automação de processos de pré-contabilidade (Figura 1), como o reconhecimento automático de dados de faturas, sua respetiva aprovação e submissão numa determinada ferramenta de contabilidade (*Basecone*, no date a).

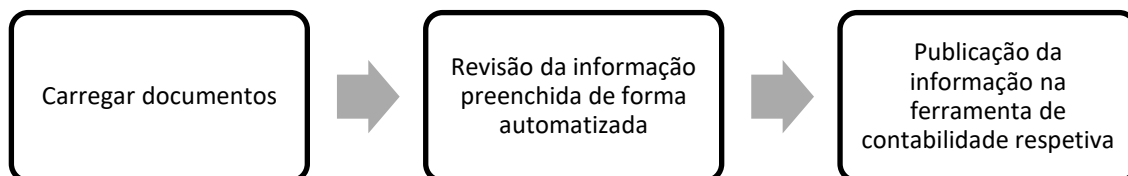


Figura 1 - Produto de pré-contabilidade

De momento, a empresa tem em mãos um problema que não é exclusivo do seu produto, mas será usado como exemplo no presente documento.

Este ano, um dos principais objetivos da empresa é capacitar as equipas para realizarem *releases* diárias, de modo a caminhar para uma abordagem de *Continuous Integration/Continuous Delivery* (integrações e entregas contínuas). Contudo, a base de código do *frontend* é partilhada por várias equipas multidisciplinares, que desenvolvem funcionalidades de acordo com a responsabilidade a elas atribuída. Esta divisão é feita com base nas várias ferramentas de contabilidade suportadas pela plataforma Basecone.

Para contextualizar, as ferramentas de contabilidade são usadas para dar suporte a sistemas contabilísticos, que são um conjunto de processos que incluem o registo da atividade financeira, como transações comerciais (como comprar ou vender mercadorias dum fornecedor ou a um cliente, respetivamente) e realização de relatórios e resumos com esses dados para análise e gestão de operações (Bragg, 2022; *AccountingTools*, 2022).

O produto da Basecone procura automatizar o procedimento de registo de documentos, ligando-se a essas ferramentas de contabilidade que lhe são externas, para submeter a informação dos documentos analisados. Assim, ao usar este produto consegue-se reduzir em parte a introdução manual dos dados nas ferramentas contabilísticas.

Como o produto em questão contempla um *frontend* monolítico, em que cada equipa é responsável por dar suporte a uma ferramenta contabilística com diferentes regras de negócio, torna-se inviável o desenvolvimento adequado neste projeto. Esta situação leva ao encontro constante de erros em ambiente de UAT (teste de aceitação do utilizador) e, por vezes, em ambiente de produção, forçando as equipas a realizarem correções em várias *bi-weekly releases* (entregas quinzenais).

Além do mais, está planeado estabelecer ligação com outras ferramentas de contabilidade, podendo conduzir ao surgimento de mais problemas de manutenibilidade, dificultando cumprir o objetivo acima referido.

Mais detalhes sobre a empresa podem ser encontrados no capítulo 3.

1.3 Problema

A empresa necessita de adquirir mais conhecimento e experiência no uso de *micro frontends*, identificando uma solução viável para a implementar. Esta iniciativa vai permitir à empresa contrariar esta limitação e aprimorar a arquitetura da aplicação, adotando práticas inovadoras que podem potencialmente otimizar a eficiência e flexibilidade do desenvolvimento de software.

1.4 Objetivo

Foi desenvolvido um projeto representativo dum em utilização na empresa com uma arquitetura de microsserviços e um *frontend* monolítico.

O trabalho visa explorar uma solução baseada em *micro frontends* que traga benefícios, sem penalizar o desempenho. Será necessário, analisar as diferentes abordagens do conceito de *micro frontends* e aplicar uma que responda a todos *drivers* arquiteturais e se enquadre na situação.

Pretende-se obter melhorias em alguns aspetos como na autonomia das equipas, *deployability* da solução e manutenibilidade. No capítulo 6 estas particularidades estão contidas com mais detalhe.

1.5 Metodologia de Investigação

Com base no problema e no objetivo, tem-se a seguinte questão de pesquisa: “Quão benéfica é a adoção de uma arquitetura de *micro frontend* comparada a um *frontend* monolítico para este caso?”

A pesquisa partiu de um problema existente num projeto real, seguida da recolha de informações sobre os principais problemas e como a empresa deseja enfrentá-los, o que resultou no termo chave deste documento: “*micro frontend*”. Depois, tendo uma boa base de conhecimento sobre *micro frontends* como possível solução para o problema da empresa, será desenvolvida uma prova de conceito que contempla a melhor abordagem escolhida para o

assunto. Esta prova de conceito é comparada com o projeto monolítico de *frontend*, permitindo que se atinja o objetivo deste trabalho e se conclua se o problema pode ser solucionado com essa abordagem.

1.6 Estrutura do Documento

A estrutura deste documento contém 9 capítulos: Introdução, Estado da Arte, Contexto da Empresa e Problema, Análise de Valor, Projeto Base, Requisitos, Solução, Experimentação e Avaliação e Conclusões.

- O Capítulo 1 contém o contexto do tema, e contexto empresarial, o problema, o objetivo, a metodologia de pesquisa e a estrutura do documento.
- O Capítulo 2 tem o estado da arte. O seu conteúdo aborda conceitos relativos à tecnologia relevante.
- O Capítulo 3 apresenta informação necessária para entendimento da situação atual da empresa.
- O Capítulo 4 tem a análise de valor, onde se encontra informação sobre o *Fuzzy frontend*, o *New Concept Development Model* e o Processo Analítico Hierárquico.
- O Capítulo 5 contém uma pequena introdução ao projeto do *frontend* monolítico. Apresenta-se o *design*, a *stack* tecnológica e como foi feita a sua implementação.
- O Capítulo 6 contém os requisitos.
- O Capítulo 7 apresenta a solução desde o seu *design* até à sua implementação, incluindo a descrição da *stack tecnológica*.
- O Capítulo 8 procura validar e analisar a solução desenvolvida, usando o GQM para definição de métricas que são consideradas para apresentação dos resultados. No fim, encontra-se um resumo dos resultados.
- O Capítulo 9 apresenta as conclusões, bem como dificuldades, trabalho futuro e a apreciação final.

2 Estado da Arte

2.1 Metodologia

Na revisão bibliográfica realizada foram utilizadas várias fontes de informação. Como o conceito de *micro frontends* é relativamente recente, ao tentar aplicar uma pesquisa avançada em diferentes bases de dados (*B-on*, *ACM Digital Library* e *Google Scholar*) com a *query string*: *AllField: ("micro frontend" OR "micro-frontend" OR "microfrontend")*, foram obtidos poucos resultados e nem todos eles se focavam nesse mesmo conceito, sendo, portanto, descartados.

Para obter mais detalhes sobre o conceito, as principais fontes utilizadas foram:

- livro “Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers” de Luca Mezzalana (Mezzalana, 2021).
- livro “The Art of Micro Frontends” de Florian Rappl (Rappl, 2021).
- livro “Enterprise Angular: Micro Frontends and Moduliths with Angular” de Manfred Steyer (Steyer, 2022b)
- Artigos científicos focados no estudo dessa arquitetura, contendo alguns dos artigos encontrados com a sequência de pesquisa mencionada acima.
- Conferências, questionários, pesquisas, notícias, apresentações, publicações em *blogs* e *sites* escritos por autores interessados e especializados no campo, encontrados após análise das referências e citações dos livros e artigos acima.

Os resultados da pesquisa serão sintetizados e explicados para consolidar o conhecimento necessário para o desenvolvimento da prova de conceito.

2.2 Arquitetura Monolítica do *Frontend*

A maioria dos projetos existentes iniciam, invariavelmente, do mesmo modo, e é por isso que a maioria dos projetos são construídos com uma arquitetura monolítica de *frontend*.

Geralmente, tudo começa com o conceito de criar um *MVP* (Produto Mínimo Viável) para obtenção de *feedback* do cliente rapidamente. Esse conceito é circular tendo três palavras-chave: construir, medir e aprender. Assim, para o *MVP*, tem-se uma base de código única para minimizar custos em *pipelines*, as ferramentas de observabilidade necessárias para obter o estado dos servidores da aplicação, uma única estratégia de entrega e, finalmente, o uso duma *framework* já conhecida pelos programadores. Estes construiriam um projeto simples (*MVP*) que cobrisse a ideia geral do negócio. Em vez de escolher a melhor abordagem ou a tecnologia mais adequada, os programadores usariam o seu conhecimento atual para escolher que tecnologias utilizar, dado que o foco seria a lógica de negócio do problema que se está a tentar resolver com o *MVP*. Se o projeto tiver sucesso, a seguinte estrutura representa-o (Figura 2).

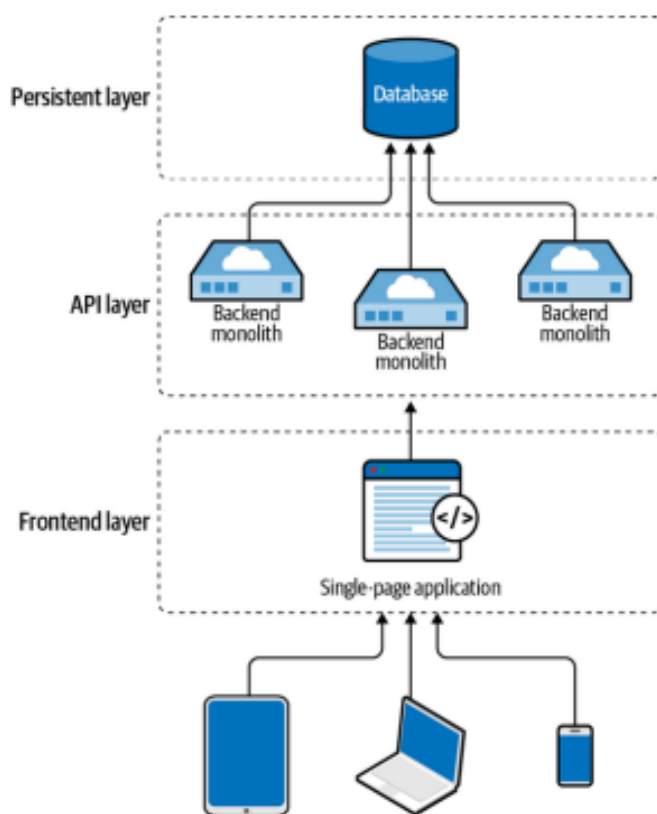


Figura 2 - Aplicação Monolítica
Fonte: (Mezzalira, 2021)

À medida que o projeto cresce e a empresa também cresce, percebe-se que se está a enfrentar alguns problemas de escalabilidade, com uma grande quantidade de pedidos ao *backend* várias vezes por segundo. De forma resumida, este problema fez surgir a arquitetura de microsserviços como uma possível solução, uma vez que escalar toda a solução não seria suficiente para acompanhar o crescimento do projeto.

A arquitetura de microsserviços permite trabalhar de forma independente, implantar de forma autónoma e escalar serviços individualmente. Ao migrar de uma para outra, pode-se precisar de rever algumas estratégias na camada de persistência para que as equipas possam escolher a base de dados mais adequada para o microsserviço a elas atribuída, e após essas decisões, tem-se a seguinte solução (Figura 3).

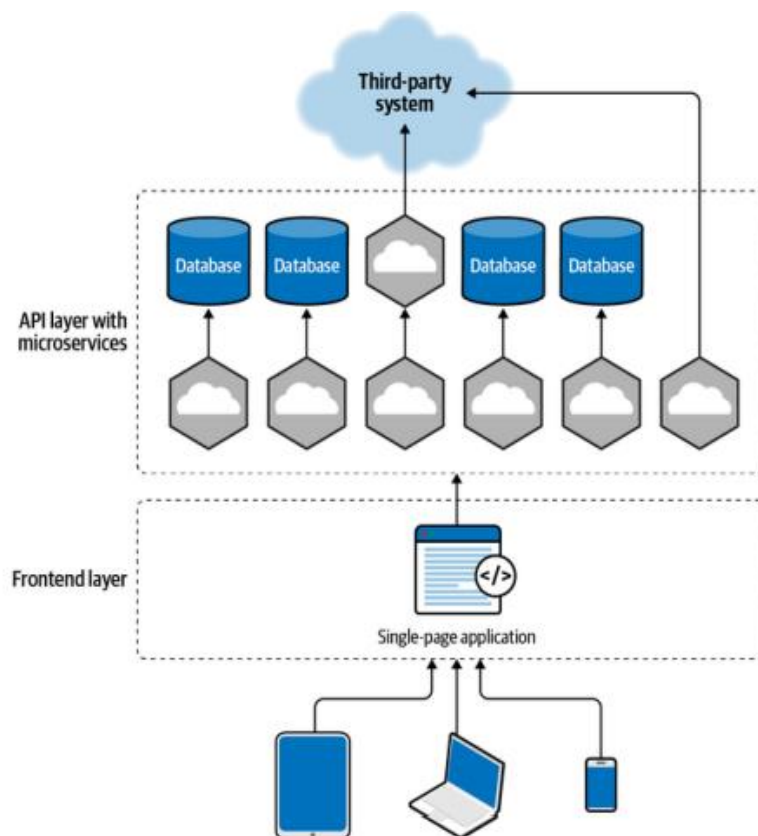


Figura 3 - Microsserviços com *single-page application*

Fonte: (Mezzalira, 2021)

Assim, o produto já está capacitado para escalar a *API* e as camadas de persistência, portanto, tornou-se mais fácil de gerir o lado do *backend*. No entanto, à medida que mais recursos surgem, os programadores de *frontend* estão a ter uma baixa produtividade, devido ao *frontend* conter toda a lógica de negócios numa única aplicação, o que pode conduzir a atrasos na entrega de

novas funcionalidades ao utilizador. Apesar da rápida implementação no *backend*, na maioria dos casos, não se pode fazer uma entrega sem a interface do utilizador atualizada de acordo, tornando o *frontend* monolítico um fator limitante.

Conforme se verifica, os problemas do *backend* agora estão no lado do *frontend*, portanto, a mesma abordagem de migração poderá adequar-se também a este. Assim, surge a arquitetura de *micro frontends* (Figura 4) (Mezzalira, 2021).

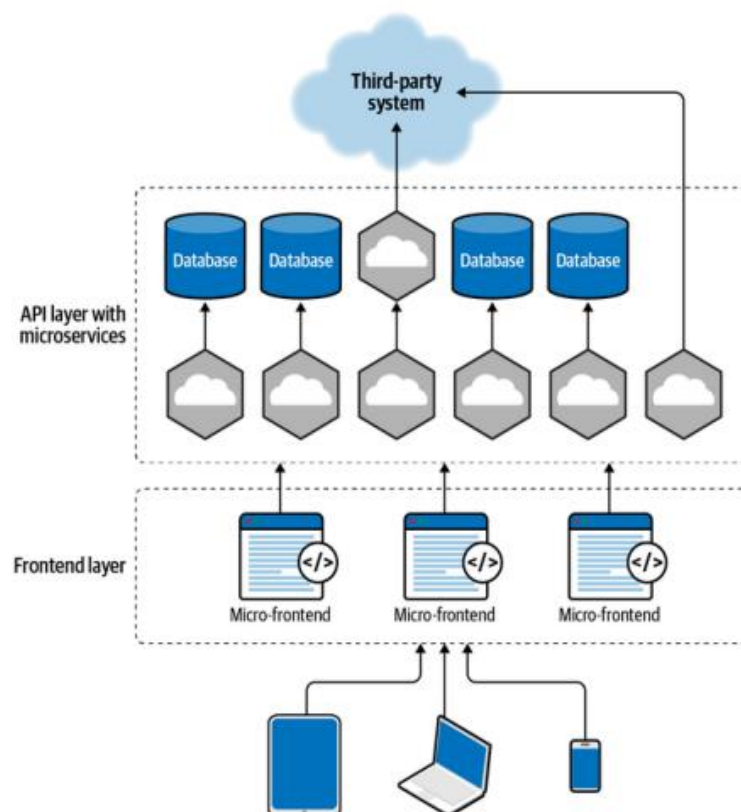


Figura 4 - Microserviços com *micro frontends*

Fonte: (Mezzalira, 2021)

2.3 Arquitetura de *Micro Frontends*

A primeira vez que se usou o termo de *micro frontends* foi no final de 2016 por uma empresa chamada *ThoughtWorks*. Eles perceberam que, em relação aos benefícios que a arquitetura de microserviços estava a proporcionar - equipas independentes capazes de escalar a entrega e implantação de seus serviços - muitas vezes enfrentavam dificuldades com a parte do *frontend*, que não conseguia acompanhar o desenvolvimento dos microserviços, além de ser difícil de

manter. Assim, algumas das equipas começaram a usar o termo *micro frontends* para se referir a partes de uma aplicação divididas por páginas e funcionalidades de propriedade de uma única equipa. Com isso, permitiam que cada equipa desenvolvesse, testasse e implantasse de forma independente (Geers, no date; *Thoughtworks*, no date).

A arquitetura de *micro frontends* é baseada na de microsserviços, portanto, alguns princípios também foram adotados, como:

- Cultura de automação: a entrega e implantação contínuas devem ser sólidas e seguras.
- Governança descentralizada: como uma equipa específica é especialista num domínio de negócio, os seus integrantes são os mais adequados para escolher a abordagem certa para o seu projeto, embora algumas práticas possam ser partilhadas e aplicadas em todas as equipas.
- Implantação independente e rápida: se a arquitetura for bem aplicada, as equipas não precisam depender de outras e esperar que estas implantem a sua parte para o utilizador final. Com equipas multidisciplinares é possível realizar lançamentos diários de microsserviços e de *micro frontends*, promovendo assim uma entrega rápida.
- Isolamento de falhas: fornecer conteúdo alternativo quando uma parte específica da interface do utilizador não estiver disponível.
- Alta observabilidade: o nível de complexidade aumenta ao adotar esta arquitetura, portanto, a observabilidade é um ponto essencial.

Dependendo do problema, também existem muitas abordagens diferentes para escolher, quando se trata de uma arquitetura de *micro frontends*. Escolher qual abordagem arquitetural seguir moldará as decisões futuras no projeto (Mezzalira, 2021; Peltonen, Mezzalira and Taibi, 2021). No entanto, a escolha depende de pontos cruciais, como os requisitos do projeto, a estrutura da organização (organização das equipas, os seus limites e o que lhes pertence) e a experiência dos programadores. Portanto, existem quatro pilares definidos na *framework* de decisões de *micro frontends* (Figura 5), que ajudam os programadores a escolher uma abordagem de “arquitetura menos pior”, no sentido em que não existe abordagem perfeita:

- Identificar o que seria considerado um *micro frontend* na arquitetura atual (o que deve ser dividido/separado);
- Compor *micro frontends*;
- Routing de *micro frontends*;
- Comunicação entre *micro frontends*.

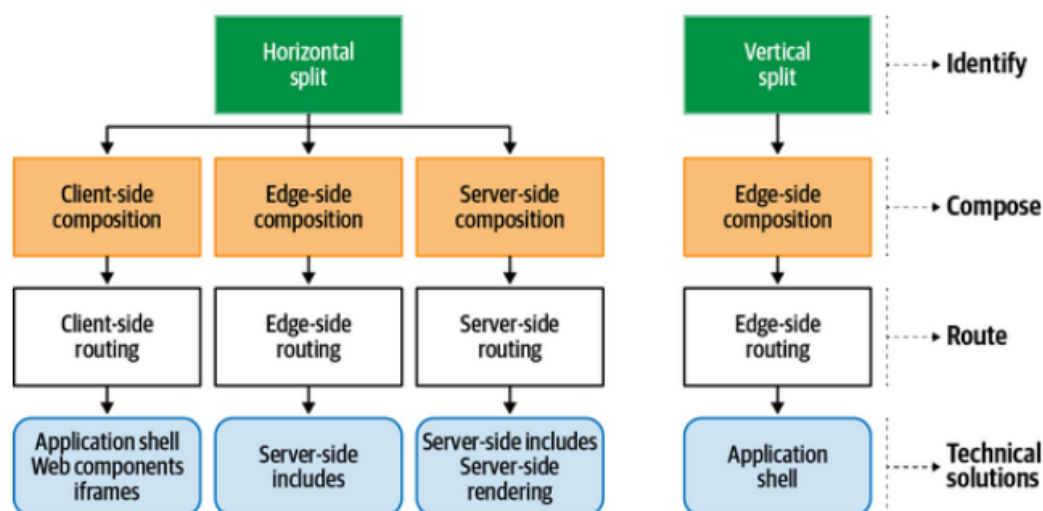


Figura 5 - *Framework* de decisão de *micro frontends*
 Fonte: (Mezzalira, 2021)

2.3.1 Identificação

Numa **divisão horizontal** (Figura 6), há vários *micro frontends* por página, o que significa que aplicações de pequenas dimensões são carregadas e cada uma delas geralmente é atribuída a uma equipa diferente. Ao dividir o projeto dessa forma, obtém-se uma grande flexibilidade ao considerar a reutilização da mesma parte do *frontend* em diferentes páginas. Um exemplo comum existente nas aplicações convencionais seria o rodapé ou cabeçalho que aparece repetido em todas as páginas dum website.

De outro modo, dividir o conteúdo atual para ter um *micro frontend* por página é uma das alternativas à divisão horizontal, mas é necessário entender como a informação deve ser partilhada entre os *micro frontends*. Assim, a **divisão vertical** (Figura 6) é bastante semelhante a uma *SPA* tradicional, sem grandes fragmentações por página e como cada equipa possui um domínio de negócios, optar por esta opção pode simplificar as decisões futuras e também a adaptação da maioria dos programadores (Mezzalira, 2020b).

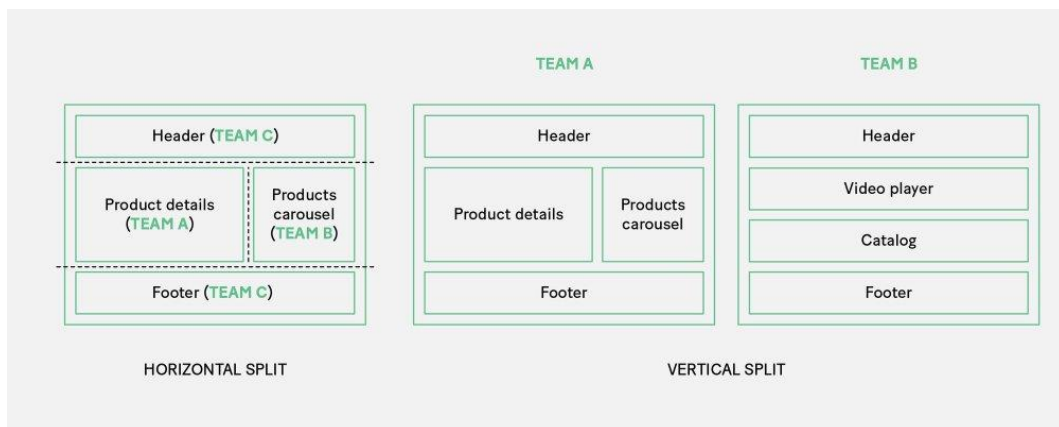


Figura 6 – Divisão Horizontal e Vertical
Fonte: (Mezzalira, 2020)

2.3.2 Composição

Os tipos de composição definem onde se quer construir a aplicação final com todos os micro *frontends* que o compõem. E para isso existem três níveis onde se pode compor a aplicação:

- Servidor (*backend*)
- *CDN (content delivery network)* ou *reverse proxy (aggregation layer)*
- Cliente ou Navegador (*frontend*)

Para a **composição *server-side***, o servidor de origem constrói a página e retorna os micro *frontends* prontos, armazenando em *cache* conteúdo possível na camada de agregação (Figura 7). Usar o *CDN* para o armazenamento de conteúdo da página em *cache* melhoraria a velocidade de carregamento da página uma vez que reduziria o número de chamadas ao servidor. No entanto, no caso de páginas altamente personalizadas, não seria muito útil, levando a um grande número de pedidos por parte de todos os utilizadores (Mezzalira, 2020b, 2021).

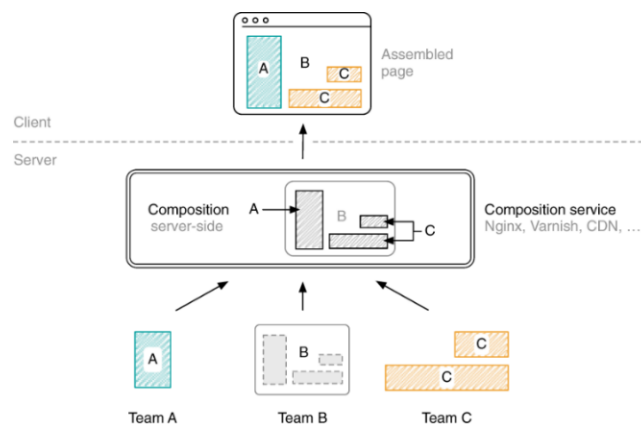


Figura 7 - Composição *server-side*

Fonte: (Geers, 2020)

A principal diferença entre a **composição *edge-side*** e a ***server-side*** é como se usa a camada de agregação (Figura 8). O *CDN* (camada de agregação) é responsável por juntar os fragmentos de *frontend* além de os armazenar em cache. No *CDN* pode usar-se o *Edge Side Includes (ESI)*, que é uma linguagem de *markup* baseada em *XML*, geralmente usada para suportar a capacidade de dimensionar a infraestrutura *web* e servir o utilizador a partir do ponto de presença mais próximo (*CDN PoP*) (Mezzalira, 2020b; Rappl, 2021).

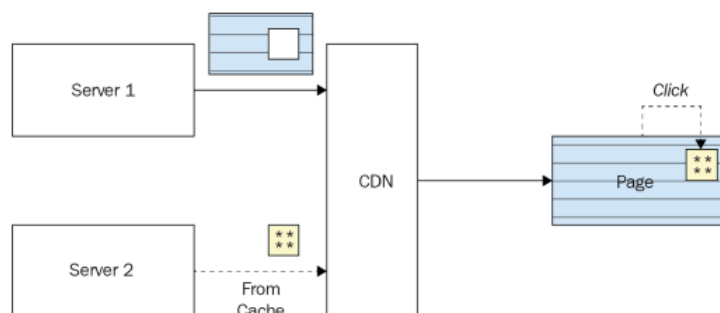


Figura 8 - Composição *edge-side*

Fonte: (Rappl, 2021)

A **composição *client-side*** implica que a parte da construção e ligação está no lado do cliente, ou seja, no navegador (Figura 9). Esta composição não depende de nenhum trabalho do componente de *backend* e também não requer uma camada de agregação.

Em vez disso, existe um *HTML principal* geralmente chamado de *app shell* que carrega cada fragmento de código dentro de si. Cada fragmento que precisa de ser exibido na página deve ter um ponto de entrada (normalmente, um ficheiro *JavaScript* ou *HTML*) para que a "*app shell*" possa inicializar cada fragmento de aplicação (Mezzalana, 2021; Rappl, 2021).

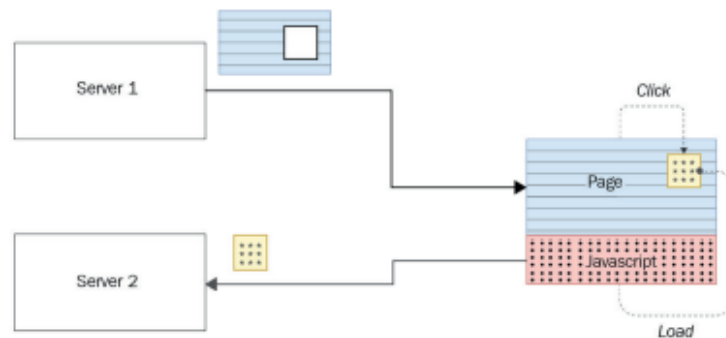


Figura 9 - Composição *client-side*
Fonte: (Rappl, 2021)

2.3.3 Routing

Tendo em mente as três opções de composição mencionadas, são consideradas as mesmas, quando se trata de *routing* de *micro frontends* (Figura 10).

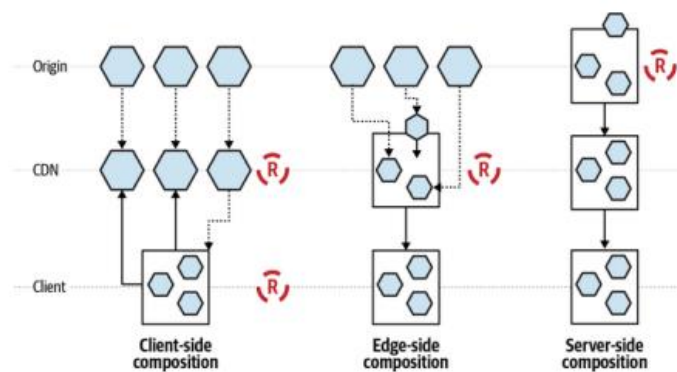


Figura 10 - *Routing* de *micro frontends*
Fonte: (Mezzalana, 2021)

Portanto, quando se trata de uma composição feita no nível de origem, a opção é estabelecer essas rotas no servidor *web*. Uma vez que os micro *frontends* para montar a página inteira são criados nesse nível, o servidor sabe também qual *template* é correspondente a essa rota específica.

O *routing* no *edge-side* utiliza a *CDN* para compor e servir a página montada com base no seu URL, por meio de transclusão. Assim, a parte de escalabilidade está assegurada pelos *CDN providers*.

E, finalmente, tem-se o *routing no client-side*, pode ser feito tanto por transclusão no lado do cliente quanto pela *shell* da aplicação que compõe as páginas. Quando se tem um micro *frontend* como uma aplicação *single-page*, como no caso de uma divisão vertical, a *shell* da aplicação será responsável pela configuração das *routes* para carregar um micro *frontend* de cada vez. Caso contrário, pode-se ter de usar uma abordagem de transclusão que será explicada no Apêndice A. Como se tem a responsabilidade toda no lado do cliente, a escalabilidade não será um problema, uma vez que ambas as abordagens dependem do URL escrito pelo utilizador.

Isto não é uma decisão linear de escolher apenas um de acordo com a composição que se escolheu anteriormente. Em vez disso, também é possível combinar diferentes tipos de *routing*, sendo possível a combinação de origem com a *CDN* e a *CDN* com o cliente (Mezzalira, 2020b, 2021; Stenberg, 2020).

2.3.4 Comunicação

Este é o último pilar a ser abordado na tomada de decisões. Num *frontend* dividido horizontalmente, encontram-se vários micro *frontends* na mesma página e esta situação pode conduzir aos limites do princípio *DRY* (*Don't Repeat Yourself*). Atualmente, os programadores tendem a levar este princípio literalmente e geralmente entendem mal o conceito, o que os leva a partilhar estados entre micro *frontends*. Isso leva a um anti padrão, que é um acoplamento de *design-time*, que faz com que as equipas se bloqueiem mutuamente para a implantação, porque uma alteração feita no estado para resolver um problema numa equipa específica que possui um dado micro *frontend* de uma página, faria esta esperar até que todas as outras equipas alinhasssem os seus micro *frontends* de acordo (Mezzalira, 2022).

Portanto, deve haver um meio-termo para criar entidades com baixo acoplamento e alta coesão. Um padrão de *design* conhecido é o padrão *observer* (padrão *publish/subscribe*) que define relações de um-para-muitos entre objetos. O padrão descreve quando um *publisher* muda os seus estados, irá despoletar um evento que notifica os *observers*, que reagirão em conformidade e se atualizarão.

Existem várias opções disponíveis que fazem uso deste padrão como *event emitter*, *custom events* e *reactive streams*.

Com isso, a comunicação entre micro *frontends* na mesma página está explicada, embora haja a necessidade de comunicar entre páginas diferentes. Existem algumas soluções como fazer uso de parâmetros de *query string*, *URL* - via protocolo *HTTPS* quando existem dados sensíveis - armazenamento web temporário chamado *session storage* ou até mesmo o permanente, o *local storage*. As duas primeiras soluções dadas precisam de recorrer ao servidor para obter a informação, ao contrário do armazenamento web, que permite armazenar dados em pares de chave-valor no navegador (Mezzalana, 2020b, 2021; Stenberg, 2020).

Em conclusão, esta *framework* de decisões para micro *frontends* ajuda os programadores e arquitetos a conhecerem melhor as diferentes possibilidades disponíveis quando se trata de identificar o que pode ser um micro *frontend*, como pode ser composto, o seu *routing* e o que pode ser usado para estabelecer comunicação entre si. Após dividir um *frontend* monolítico em fragmentos e aplicar o conhecimento sabiamente considerado contexto em questão, esta visão, isto é, o diagrama de decisões apresentado, ajuda na escolha certa para cada nível de decisão.

2.4 Anti Padrões

Esta secção menciona alguns padrões que podem conduzir a uma má implementação de uma arquitetura de micro *frontends*. Alguns deles podem-se adequar numa fase transicional de migração do monólito, contudo é importante ter em mente possíveis problemas que o seu uso pode trazer para o projeto e complicações futuras para os programadores.

2.4.1 Micro Frontends e Componentes

Um micro *frontend* é uma representação técnica de um domínio de negócio gerido por uma única equipa, com uma implementação independente que pode ou não ser realizada com diferentes tecnologias. Quando há necessidade de implantar constantemente dois ou mais micro *frontends* juntos, é importante compreender se eles devem ser dois micro *frontends* separados ou se se deve rever essa mesma divisão.

Os micro *frontends* devem ser independentes, representativos dum domínio, com entradas e saídas bem definidas. Assim, ao optar por uma arquitetura de micro *frontends*, é necessário considerar que a divisão em subdomínios é aceitável, mas dividir em excesso pode transformar aquilo que se nomeia de micro *frontends* em componentes que estão simplesmente ligados num contentor que controla o seu comportamento.

2.4.2 Múltiplas Frameworks

Apesar dos *micro frontends* possibilitarem a utilização de várias *frameworks*, isso não significa que seja obrigatório fazer uso dessa oportunidade. Quando se pensa em construir uma *single-page application*, normalmente escolhe-se uma única *framework* de interface do utilizador, e isso também não deve diferir nesta arquitetura.

A utilização de múltiplos *frameworks* pode ser vantajosa em casos específicos, como:

- Quando se pretende que um *micro frontend* coexista com um sistema *legacy*. Uma vez que é um sistema *legacy*, uma abordagem de múltiplas *frameworks* funcionaria como uma solução temporária para fornecer novo conteúdo ao utilizador de forma mais rápida. A arquitetura de *micro frontends* ajuda de certa forma as equipas a migrarem gradualmente código *legacy* para pequenos *micro frontends* e continuarem com a entrega contínua de novas funcionalidades, enquanto atualizam em simultâneo, o produto no seu todo. Sem fazer uso dessa abordagem, a alternativa seria parar as entregas ao cliente durante um longo período, para depois fazer uma implantação enorme que substituiria na sua totalidade o antigo sistema.
- Migrar para uma nova *framework*/biblioteca de interface do utilizador ou atualizar para uma nova versão, não necessita de alinhamento entre equipas em termos de tempo para execução dessa ação. Portanto, novamente tem-se uma solução temporária que permite que cada equipa faça a sua migração de forma iterativa e independente.
- Aquisição de empresas por outras, o que conduz muitas vezes à fusão das suas soluções. Todavia, as empresas precisam continuamente de entregar valor aos clientes. Portanto, usam uma abordagem de *micro frontends* com diferentes *frameworks* usadas pelas diferentes empresas que foram agregadas para fornecer rapidamente novo conteúdo aos utilizadores, enquanto reescrevem todo o sistema por trás.

No entanto, apesar dos benefícios que o uso de múltiplas *frameworks* traz, ainda é considerada uma prática contraproducente, porque algumas empresas romantizam a ideia ao defender que estão a proporcionar liberdade aos programadores. É bom que as empresas se preocupem com os seus programadores, no entanto, eles não são os utilizadores finais nem os clientes que estão a usar a solução.

2.4.3 Dependências Excessivas

Dependências externas existem nos projetos de *frontend*. No caso de termos múltiplos *micro frontends*, eles partilham as mesmas dependências que normalmente pode-se agrupar numa biblioteca central. Assim, cada *micro frontend* herdaria essa biblioteca central cuja responsabilidade é de uma equipa específica. Portanto, cada equipa já tem essa dependência, mas ainda é considerada independente, porque mesmo que precisem de recursos adicionais, além daqueles que a biblioteca central está a fornecer, podem sempre estendê-la para obter também o que o seu *micro frontend* atualmente precisa para continuar o seu desenvolvimento.

No entanto, com esta herança entre bibliotecas, quando a biblioteca central é atualizada para uma versão mais recente, leva mais tempo para que a biblioteca estendida seja atualizada de acordo, o que se refletiria rapidamente em todas as equipas. Os números de bibliotecas partilhadas também aumentariam e dependeriam umas das outras, e assim por diante.

Portanto, uma biblioteca externa partilhada pode ser viável, mas quando estas começam a aumentar e a ser estendidas pelas diferentes equipas, há uma necessidade de repensar toda a hierarquia de dependências criada.

2.4.4 Acoplamento *Design-time*

Quando se trata de agrupar partes de código, é necessário ponderar esta decisão cuidadosamente nesta arquitetura. Ter um estado global que abrange os micro *frontends* que coexistem na mesma página pode ser uma abordagem tentadora.

O acoplamento *design-time* é também um anti padrão para a arquitetura de microsserviços. Este descreve quando forçadamente se altera um serviço devido a uma alteração noutro serviço. Isso geralmente ocorre quando um serviço depende direta ou indiretamente de conceitos específicos que pertencem a outro. Se esses microsserviços são de propriedade de equipas diferentes, isso exigiria que as equipas colaborassem e até atrasassem a implementação do primeiro microsserviço que fosse alterado, o que leva a uma diminuição na produtividade e atrasos na entrega de conteúdo aos clientes.

Isso também acontece no *frontend* quando as equipas optam por um estado global centralizado. Felizmente, pode-se optar por uma abordagem baseada em eventos para comunicação e partilha de informação, como mencionado na subsecção 2.3.4 (Richardson, 2021; *Micro frontends anti patterns*, 2022; Mezzalana, 2023).

2.5 Tecnologias e *Frameworks*

Esta secção abrange algumas das tecnologias e *frameworks* de micro *frontends* exploradas.

2.5.1 Single-SPA

Single-SPA é uma biblioteca que começou a ser utilizada para executar vários micro *frontends* dentro de uma única aplicação. Além da liberdade de uso de diferentes *frameworks* na mesma página, ela proporciona a possibilidade de *lazy loading*, ou seja, carregar os micro *frontends*

apenas quando são necessários, e possui uma configuração de routing para os associar a um caminho específico da aplicação.

Portanto, para implementar uma arquitetura de micro *frontends*, esta biblioteca é usada como um motor de routing no lado do cliente. Ela possui um *root config* que renderiza o HTML e o JavaScript que registra as aplicações (*registerApplication()*) e mantém o controle sobre quais aplicações estão carregadas, ainda em carregamento ou devem ser carregadas. Para registrar uma aplicação, é necessário fornecer o seu nome, a função para carregar o código dessa aplicação e uma função que determinará se a aplicação deve estar ativa ou inativa. Com isso, cada aplicação deve saber como e quando se inicializar, montar e desmontar(*single-spa*, no date).

2.5.2 Mooa

Mooa é uma *framework* de micro *frontends* para *Angular* baseada na *framework Single-SPA*. Ele é otimizado para soluções no *IE10* e utiliza *iframes*. O *Mooa* usa uma arquitetura *Master-Slave*, onde há um projeto principal responsável por carregar todas as outras aplicações *Angular*, podendo carregar mais do que uma ao mesmo tempo.

Quando o projeto principal está em execução, ele obtém a configuração de cada aplicação, criando-as de acordo com os seus ciclos de vida. Quando a rota muda, o projeto principal verifica se há correspondência com a nova rota e, se houver, carrega a aplicação correspondente. Se houver subaplicações, a comunicação será feita por meio de eventos (Figura 11) (Yang, Liu and Su, 2019; Prajwal, Parekh and Shettar, 2021; Huang, 2023).

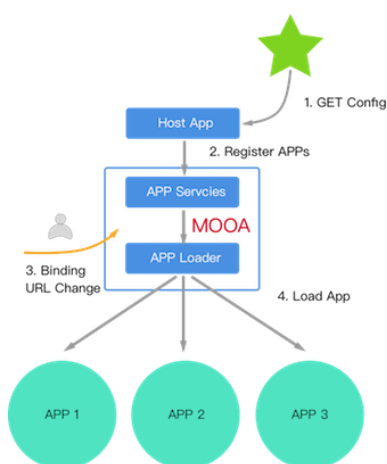


Figura 11 - Conceito de MOOA
Fonte: (Huang, 2023)

É importante observar que esta *framework* não recebe atualizações desde dezembro de 2019 e é específico apenas para projetos Angular. Usá-lo para migrar para uma *framework* diferente não seria possível (Huang, 2023).

2.5.3 Module Federation

Com o lançamento do *webpack 5*, foi lançado um novo *plugin* nativo chamado *Module Federation*. Esse *plugin* forneceu uma funcionalidade que permite o carregamento de partes de código JavaScript síncronos e assíncronos. O *Module Federation* é composto por dois elementos-chave:

- O *host*, responsável por carregar cada pacote *JavaScript* em tempo de execução.
- O *remote*, carregado pelo *host*.

Ele facilita a partilha de código e dependências entre diferentes módulos, o que ajuda a melhorar o desempenho.

Portanto, este *plugin* cuida basicamente de tudo relacionado à abordagem de *micro frontends*, desde a composição até ao *routing*, e lida também com diferentes versões da mesma biblioteca em tempo de execução. Se o programador já está acostumado a trabalhar com *single-page applications*, as mudanças serão tranquilas, porque cada *micro frontend* (módulo), após ser importado, funcionará como um componente numa *UI framework* (Mezzalana, 2021).

2.5.3.1 Dynamic Module Federation

No *Webpack Module Federation*, o *host* também conhecido como *shell* tem na sua configuração (*webpack.config.js*) os *remotes* pré-definidos (*remoteEntry.js*). Contudo, numa situação mais dinâmica como é o caso do uso de diferentes ambientes, um ficheiro de configuração pode tornar-se útil. Assim, com o *Dynamic Module Federation* tem-se a flexibilidade da *shell* carregar os *micro frontends* que não conhece durante a compilação, fazendo uso dum ficheiro *JSON* (*manifest*) que providencia o *URLs* necessários em *runtime*.

O conceito pode ser aplicado independente da *framework* usada, um exemplo de um pacote *npm* que facilita o desenvolvimento para o Angular é *@angular-architects/module-federation*, que na sua versão 14.3 permite a possibilidade da geração do *dynamic host* com o comando da Figura 12.

```
ng g @angular-architects/module-federation --project shell --port 4200 --type dynamic-host
```

Figura 12 -Comando para gerar a *Shell* dinâmica

Este comando gera:

- O ficheiro *manifest.json*
- O ficheiro *main.ts* para carregar o manifest
- O ficheiro *webpack.config.js*

Manfred Steyer, *Google Development Expert* especializado na *framework*, e a sua equipa (*ANGULARarchitects*) têm mantido os pacotes de *npm* bastante atualizados, e providenciam também informação sobre vários temas na sua página, incluindo o conceito de micro *frontends* (Steyer, 2020; ‘ANGULARarchitects Team’, no date).

2.5.4 Import Maps

O *importmap* é um valor que se pode colocar no atributo *type* do elemento *HTML* `<script>`.

Consiste num objeto *JSON*, onde cada chave corresponde a um *ES module* e o seu valor ao caminho que lhe corresponde (Código 1). Essas chaves são reconhecidas nos ficheiros *JavaScript* (Código 2).

ES module também conhecido por *ECMAScript module* é o formato padrão oficial de empacotar código *JavaScript* para reutilização (*ESM Node.js*, no date).

```
<script type="importmap">
{
  "imports": {
    "square": "./module/shapes/square.js",
    "circle": "https://example.com/shapes/circle.js"
  }
}
</script>
```

Código 1 – Uso do *importmap* no *HTML*

Fonte: (*importmap* | Mozilla, 2023)

```
import { name as squareName, draw } from "square";  
import { name as circleName } from "circle";
```

Código 2 – Uso do *importmap* no JS

Fonte: (*importmap* | Mozilla, 2023)

Como isto é nativo da web, este mapeamento é tratado diretamente no navegador sem necessidade de agrupar e traduzir o código de forma que este o entenda. Assim, retira-se essa responsabilidade ao *bundler*, e consegue-se carregar do mesmo modo o módulo pretendido.

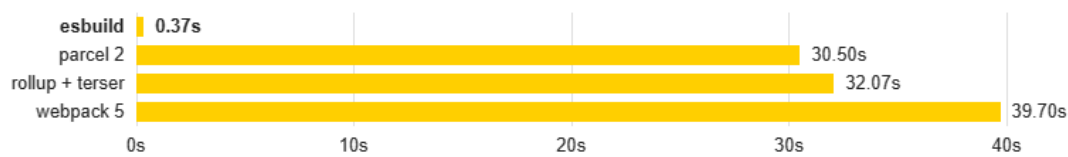
O *module federation* está dependente do *webpack*, ao passo que o *import maps* não depende de nenhum específico (*esbuild*, *parcel 2*, *rollup + terser*). Contudo, ao optar pelo *import maps*, é necessário criar estruturas de código adicionais para tratar de tarefas como resolução de conflitos de versões, remoção de dependências que não estão a ser usadas e/ou permitir a sua partilha. Além disso, o carregamento dinâmico de *remotes* também requer uma abordagem específica.

Por outro lado, o *module federation* oferece configurações simplificadas para essas tarefas, facilitando a configuração e a gestão de micro *frontends* (Steyer, 2022c; Zaikin, 2023; *importmap* | Mozilla, 2023).

2.5.5 Native Federation

Usando o mesmo conceito do *webpack module federation*, foi criado o *native federation*. É uma implementação “nativa do browser” - baseada em padrões web atualizados e tecnologias existente dos navegadores, como o *import maps*- que pode ser usada independentemente de qualquer ferramenta e ou *framework*.

Existe uma biblioteca chamada de *Native Federation Core* que é *framework agnostic* - *@softarc/native-federation*. Mas já existem integrações adaptadas com *framework* e *bundlers* específicos, como é o exemplo do *@angular-architects/native-federation* que está a ser desenvolvida por uma equipa do *Angular*. Ele disponibiliza uma integração com o *Angular CLI* e faz uso do *bundler esbuild*, que promete uma melhoria significativa em termos de desempenho comparativamente a outros *bundlers* (Figura 13).



Above: the time to do a production bundle of 10 copies of the [three.js](#) library from scratch using default settings, including minification and source maps. More info [here](#).

Figura 13 – Desempenho de diferentes bundlers

Fonte: (*esbuild*, no date)

Porém, o pacote *npm* acima referido está ainda na sua versão *beta*. Mas, já é possível verificar que com a evolução dos padrões *web*, têm surgido os primeiros passos no que diz respeito a soluções mais abstraídas e independentes de *frameworks* e ferramentas de construção (Steyer, 2022a, 2022c; *native-federation*, 2023).

2.6 Estrutura das Equipas

A arquitetura de micro *frontends* pressupõe o desenvolvimento de funcionalidades de ponta a ponta, portanto a equipa tem de ser independente e ter uma estrutura que proporcione suporte a cada funcionalidade que lhes pertence.

Em *Agile* considera-se dois tipos de estrutura:

- **Equipa de Funcionalidades**
 - **Multidisciplinaridade:** Contêm as capacidades necessárias (*design*, desenvolvimento, QA, ...) para concluir uma funcionalidade.
 - **Entregas rápidas e Independência:** Sendo colaboração entre elementos da mesma equipa, a entrega torna-se mais rápida, pois não há dependências entre equipas
 - **Orientada para o Cliente:** Procuram compreender as necessidades do cliente e entregar-lhes as funcionalidades de acordo com estas.
 - **Flexibilidade:** Adaptam-se mais facilmente a alterações das necessidades do cliente.
- **Equipa de Componentes**
 - **Conhecimento especializado:** Possuem conhecimento especializado sobre uma área particular do produto.

- **Dependência:** Recorrem as outras equipas para ter acesso a outras capacidades técnicas.
- **Orientada para a parte técnica:** Conseguem concentrar-se em pormenores técnicos, dado serem especializados na área.
- **Estabilidade:** São mais estáveis e previsíveis (*GoRetro*, no date).

A estrutura escolhida pode variar tendo em conta fatores, como o tamanho da companhia e o estado do projeto atual, contudo numa arquitetura de micro *frontends*, cada equipa tem uma área de negócio específica e desenvolve funcionalidades completas, desde a base de dados até à interface do utilizador, sendo mais propício organizar a empresa em equipas de funcionalidades, também conhecidas como multidisciplinares (Mezzalira, 2021; Peltonen, Mezzalira and Taibi, 2021; Geers, no date).

2.7 Exemplos Empresariais

Algumas empresas adotaram a arquitetura de micro *frontends* nos seus projetos de diferentes maneiras de acordo com as suas necessidades. Alguns exemplos encontrados serão explicados para entender a diversidade de opções ao optar por esta arquitetura.

2.7.1 IKEA

A IKEA é um exemplo de uma combinação de *edge-side includes (ESI)* com *client-side includes*. Eles usam uma composição *edge-side* para gerir o site estático onde várias equipas trabalham e utilizam alguma forma de transclusão para incluir o que precisam para aquela parte da solução.

A *ESI* é uma técnica fundamental para o conceito de micro *frontends* adotado pela IKEA. Eles têm conhecimento das páginas e fragmentos que existem, e cada equipa é responsável por um conjunto de páginas e/ou fragmentos. Assim, uma página pode fazer referências a fragmentos que pertencem a outras equipas. Por exemplo, a miniatura do produto é considerada um fragmento que pertence à equipa de produtos e é usada em vários lugares do site; outras equipas podem fazer referência a esse fragmento e incluí-lo na sua parte da solução final. Além disso, as equipas são independentes nesse processo. Como todos os fragmentos precisam de parecer fazer parte da mesma solução, a IKEA decidiu usar *ESI* para estilos e scripts, ou seja, duas inclusões a nível do *CDN*. Para os utilizadores finais, essa técnica de *microcaching* é benéfica, pois cada utilizador recebe uma cópia da página com conteúdo estático em cache, especialmente em situações de alta carga. No entanto, o arquiteto da IKEA concorda que é uma

técnica difícil de implementar, portanto não é amplamente utilizada por muitas empresas (Stenberg, 2018; Mezzalira, 2021).

Embora essa arquitetura funcione para o projeto da IKEA, isso não significa que seja a solução certa para todos os projetos. Em vez disso, foi apenas uma das soluções possíveis que funcionou ao tomar algumas decisões arquitetônicas específicas com base no problema em questão.

2.7.2 DAZN

DAZN é uma plataforma de *streaming* de desporto que usa um agente *client-side* chamado *Bootstrap*. Portanto, a abordagem escolhida é totalmente do lado do cliente, sendo o *bootstrap* a primeira biblioteca carregada na página. Consiste num único ficheiro HTML que contém alguma lógica para carregar os outros micro *frontends*. Este *bootstrap* aciona algumas chamadas de retorno que notificam um dado micro *frontend* para se montar ou desmontar. O seu desmonte remove tudo da memória, o que elimina as dependências que o micro *frontend* possa ter, portanto, para a comunicação entre micro *frontends*, é utilizado o armazenamento local (*local storage*).

Assim, a DAZN possui esse *bootstrap* como um orquestrador para carregar diferentes SPAs em tempo de execução, sempre que necessário, quando o utilizador navega pela plataforma.

O projeto da DAZN também tinha uma biblioteca compartilhada que continha parte do sistema de pagamento da plataforma. Isso criou uma dependência entre todos os micro *frontends* que consumiam essa biblioteca, o que implicava uma implantação simultânea e obrigatória sempre que essa biblioteca fosse atualizada.

Além disso, como os seus micro *frontends* eram como SPAs individuais, isso implicava que certas partes fossem duplicadas na maioria destes, como o rodapé e o cabeçalho da página. Acreditava-se que juntar essas partes poderia conduzir também a um acoplamento de micro *frontends* que se pretendia evitar (Mezzalira, 2020a).

2.7.3 Spotify

O Spotify é uma plataforma de *streaming* de música que usou *iframes* como micro *frontends* na sua aplicação *desktop*, e cuja comunicação era estabelecida por meio de eventos.

Apesar de o termo micro *frontend* ter surgido em 2016, a maioria das empresas já estava a adotar o conceito, e o Spotify foi uma delas. Em 2012, lançaram o *web player* com uma abordagem baseada e composta por vários *iframes*.

Os *iframes* podem ser considerados uma abordagem de composição do lado do cliente que reforça a independência das equipas para evitar dependências de código e conflitos de código, isolando diferentes domínios. Os artefactos presentes na aplicação *desktop* continham ficheiros *.spa*, sendo compostos por ficheiros *HTML*, alguns ficheiros *CSS*, juntamente com um *manifest.json* e um ficheiro de *bundle* minificado. Tudo isso era carregado dentro de um único *iframe*, portanto, no *web player* o desempenho era mau, pois tinha muitos recursos para carregar todas as vezes que um utilizador navegava entre as páginas. Essa é uma das razões pelas quais eles abandonaram a abordagem de *iframes* para a *web player* e voltaram para uma abordagem de *single-page application* (Mezzalana, 2019; Pérez, 2019).

2.8 Sumário

Esta secção contém um resumo daquilo que foi abordado no presente capítulo. Explorou-se e documentou-se como os micro *frontends* surgiram, que passos seguir e que anti padrões evitar ao tentar implementar esta arquitetura. Algumas *frameworks* também foram apresentados de forma breve, bem como alguns casos empresariais.

3 Contexto da Empresa e Problema

Este capítulo descreve a empresa e área onde atua, que problemas foram surgindo ao longo do desenvolvimento do produto, como se organizam as equipas, o estado atual do produto e planos já definidos para o seu futuro.

3.1 Empresa

A Basecone é uma empresa que desenvolve produtos *web* para automação de processos de pré-contabilidade, como o preenchimento de dados de faturas, sua respetiva aprovação e submissão numa determinada ferramenta de contabilidade (Figura 14)(*Basecone*, no date a).

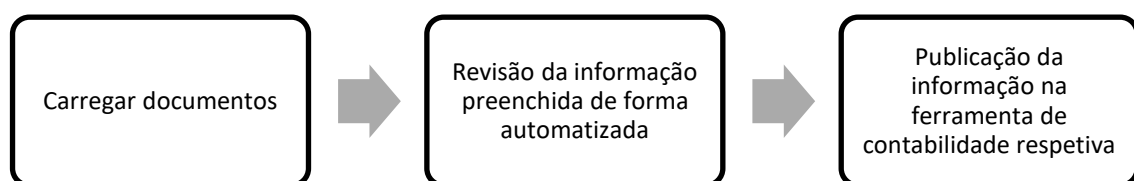


Figura 14 - Produto de pré-contabilidade

A empresa foi adquirida pela Wolters Kluwer que tem um leque de soluções informáticas que suportam profissionais como médicos, contabilistas, advogados e setores tributários, financeiros, de auditoria, de risco, de conformidade e regulatórios (*Wolters Kluwer*, no date a).

Foi necessário integrar as regras de estilos e ferramentas usadas pela Wolters Kluwer. Desta forma, a Basecone começa a pertencer à Divisão Europeia fiscal e de contabilidade. O conceito

do seu produto, que também se intitula de Basecone, consiste em suportar, numa plataforma única, a parte de pré-contabilidade, integrando várias ferramentas de contabilidade. O produto encontra-se atualmente disponível na Bélgica, nos Países Baixos e no Reino Unido (*Wolters Kluwer*, no date b).

A Basecone tem como *slogan* “Devolvemos tempo ao contabilista”, uma vez que a empresa foi criada por contabilistas e pretende com o seu produto facilitar a vida deles e de outros empresários, com o processamento digital de documentos. As faturas podem ser entregues de forma automatizada, rápida e fácil e a sua informação enviada às ferramentas vinculadas à Basecone. Como principais vantagens da plataforma temos:

- O processamento em tempo real na ferramenta de contabilidade.
- A entrega dos documentos pode ser feita de várias formas, carregada diretamente na plataforma, enviada por email ou pela aplicação de telemóvel.
- É permitido o controlo do fluxo dos documentos e faturas, sendo possível acompanhar o seu estado.
- A solução é 100% *cloud*, permitindo enviar, analisar e autorizar documentos a qualquer hora em qualquer lugar.
- É possível a seleção múltipla de documentos. Isto permite ao contabilista realizar a mesma ação num lote de documentos de uma só vez.
- Os documentos permanecem 10 anos armazenados em segurança na Basecone. Assim está-se capacitado a responder a possíveis pedidos por parte da Autoridade Tributária (*Basecone*, no date b).

3.2 Equipas

A Basecone é constituída atualmente por 16 equipas, todas multidisciplinares, exceto a equipa de *UX* e a equipa de operações. As equipas multidisciplinares são constituídas por engenheiros de *backend*, *frontend* e qualidade, *release manager*, *scrum master* e *product owner*. Cada equipa inclui um membro de *UX* e uma a duas pessoas de operações, encarregadas por fornecer suporte nas respetivas áreas. Assim, cada equipa tem alguns domínios a ela atribuída, e procura tratar deles desde a base de dados até à interface do utilizador. Como trabalhamos com várias ferramentas de contabilidade diferentes, cada uma com as suas especificidades, estas são distribuídos pelas várias equipas.

Além das equipas dedicadas, também existem comunidades que reúnem membros de diferentes equipas para abordar e resolver temas específicos. Estas comunidades realizam, pelo menos, uma reunião a cada *sprint* com esse propósito. Por exemplo, na Comunidade de Segurança, analisam-se relatórios de segurança, avalia-se a autenticidade das vulnerabilidades detetadas, determina-se a qual domínio pertencem e identifica-se a equipa responsável por resolvê-las. Outro exemplo é a Comunidade de *Frontend*, onde são apresentadas *frameworks* e

boas práticas para codificar e analisar código de outros programadores. No fundo, as comunidades facilitam a comunicação e a partilha de informações entre as equipas, proporcionando conhecimento aos interessados naquele tópico. Elas também auxiliam as equipas a alinharem-se quando há uma implementação específica que beneficia a todos. Um exemplo seria a configuração do *Checkmarx* – uma ferramenta de análise de código - para todos os projetos da empresa.

As equipas de desenvolvimento regem-se por uma metodologia *Agile*, o *Scrum*. Os *sprints* realizados são de duas semanas e contém várias cerimónias tais como a Reunião Diária, Sessões de Refinamento, Planeamento de *Sprint*, Revisão e Retrospectiva. Cada equipa mantém um quadro visual para acompanhar o progresso do trabalho em curso. Este quadro possui colunas que representam os diferentes estados de uma *user story*. Embora cada equipa tenha a flexibilidade de adicionar ou remover estados conforme necessário, de uma forma genérica, cada *user story* passa pelos seguintes estados:

- “a fazer”: colocada nesta coluna no planeamento do *sprint*;
- “em desenvolvimento”: o programador está a codificá-la;
- “em revisão de código”: outros programadores analisam o código desenvolvido;
- “em teste”: o engenheiro de qualidade executa os testes necessários sobre um ambiente de testes;
- “em aceitação”: o *product owner* aceita a funcionalidade desenvolvida;
- “feito”: o programador faz *merge* do código aceite.

No final, o *release manager* é responsável por realizar a entrega ao cliente, colocando o que foi desenvolvido em ambiente de produção.

3.3 Estado Atual do Produto

Atualmente, as equipas encontram-se no processo de migração do monólito para a arquitetura de microsserviços, com um *frontend* monolítico a ser desenvolvido em Angular. Neste processo, as equipas fazem uso de *feature toggles* para ativar uma funcionalidade migrada quando concluída, ou reverter facilmente o processo desativando a funcionalidade. Isso permite apresentar o projeto antigo com a mesma funcionalidade.

As implantações são realizadas utilizando o modelo de *blue-green deployment*, que envolve a criação de dois ambientes separados, porém idênticos. Um ambiente (*blue*) executa a versão atual da aplicação, enquanto o outro (*green*) executa a nova versão. Essa abordagem simplifica o processo de *rollback*, reduzindo os riscos durante a implantação. Dessa forma, quando os testes são concluídos no ambiente *green*, o tráfego de produção é direcionado para este ambiente, descontinuando o ambiente *blue* (AWS, no date).

3.3.1 Problemas

No *frontend* monolítico, várias equipas, apesar de responsáveis por diferentes domínios, contribuem com novo código no mesmo projeto. Como resultado, a entrega ao cliente é feita a cada duas semanas, e o tempo necessário para passar por todas as etapas de uma pipeline - *build*, testes unitários, testes *end-to-end* e *deploy* - pode ser demorada em algumas situações.

Dadas as circunstâncias do projeto, as equipas aplicam o princípio *DRY* (*Don't Repeat Yourself*) (Figura 15). Assim, partes de código repetidas em métodos, serviços ou classes devem ser reestruturadas para evitar duplicação no projeto. Isso resulta num aumento da abstração do código e promove a sua reutilização. Como consequência, a duplicação de código é minimizada e as funções criadas tornam-se mais concisas e de fácil compreensão.

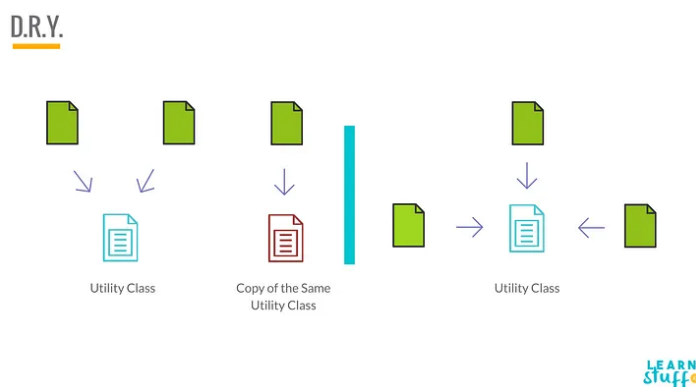


Figura 15 – *Don't Repeat Yourself*

Fonte: (Ungureanu, 2020)

Na empresa existe uma pasta partilhada para serviços e métodos abstraídos que podem ser utilizados noutros domínios. No entanto, ao realizar alterações nesses ficheiros para atender as necessidades específicas de uma equipa e evitar a repetição de código no seu domínio, é necessário corrigir todos os componentes que fazem uso desses. Isso pode resultar em complicações e erros que afetam a funcionalidade de outras equipas. Para evitar esse cenário, a comunicação entre equipas é essencial. No entanto, devido às diferentes prioridades e prazos de entrega para o cliente, aguardar que cada equipa faça as correções pode ser inviável. Muitas vezes, um único programador faz alterações em qualquer domínio.

Um problema recente, envolveu a alteração de uma constante indicando o número de elementos apresentados ao utilizador. Essa constante era compartilhada e utilizada em várias páginas do produto. Uma página específica possuía uma constante com o mesmo nome, mas com um valor diferente. Ao alterar a constante compartilhada, o número de elementos apresentados foi inadvertidamente modificado. Situações como esta geram insegurança nos programadores e tornam desafiador "limpar" o código de variáveis e constantes que possam

estar repetidas, mas com outros nomes. Por vezes, essas constantes são abstraídas o suficiente para atender aos domínios de várias equipas, mas necessitam de ser adaptadas para outras, e assim por diante. Isso aumenta a necessidade de comunicação entre equipas e torna demoroso a resolução de problemas.

Portanto, a empresa percebe a necessidade de reduzir estas dependências e otimizar o tempo de integração de novos colaboradores, bem como o tempo de entrega de valor para o cliente.

3.3.2 Planos

A empresa planeou suporte para novas ferramentas contabilísticas além das já suportadas, distribuindo-as pelas equipas. Existe uma nova equipa que está a desenvolver uma funcionalidade inexistente no monólito, a qual proporcionará um valor significativo para os contabilistas no contexto dos movimentos bancários.

Considerando a importância do *time-to-market* e levando em conta a observação de uma ligeira perda de produtividade, conforme mencionado na subsecção 3.3.1, a empresa pretende realizar uma análise para determinar se a implementação de micro *frontends* pode representar uma vantagem para o produto nesse aspeto sem prejudicar o cliente.

4 Análise de Valor

Este capítulo é dedicado a analisar o valor do processo estudado para solucionar o problema atual do projeto da empresa. Contém uma secção com o processo de inovação, abordando a fase do *fuzzy front end* e o *new concept development model* que o clarifica. Apresenta-se, ainda neste capítulo, o método de análise hierárquica (AHP) usado no processo compreensão e avaliação de um problema.

4.1 Fuzzy front end

O processo de inovação pode ser dividido em três fases diferentes. Começando pelo fuzzy front end (FFE), depois o processo de desenvolvimento do novo produto (em inglês, *new product development* ou NPD), seguido pela execução da sua comercialização.

O processo considerado mais adequado para melhorar o processo de inovação é o FFE também conhecido como o *fuzzy front end*, convencionalmente - um termo dado para definir um processo que é misterioso, incontrolável, pouco estruturado e, portanto, difícil de ser avaliado - mas alguns autores preferem denominá-lo de *frontend of innovation*.

Para que o processo de inovação seja mais eficiente, muitas empresas utilizam algumas abordagens para uma melhor distribuição do tempo no processo de desenvolvimento do novo produto (NPD), como implementar Stage-Gate(TM)(Cooper 1993) ou PACE(R)(McGrath e Akiyama 1996), mas nenhuma das práticas que podem ser aplicadas para a fase do NPD se enquadram da mesma forma para o FFE, uma vez que as fases diferem em vários termos. A data de comercialização, a natureza do trabalho, o financiamento e as expectativas de receita são previsíveis e concretos para o processo NPD com toda a equipa focada em atingir marcos pré-definidos específicos, enquanto, por outro lado, o processo FFE é muito mais incerto e caótico, tendo menos pessoas envolvidas inicialmente no esclarecimento de conceitos (Figura 16).

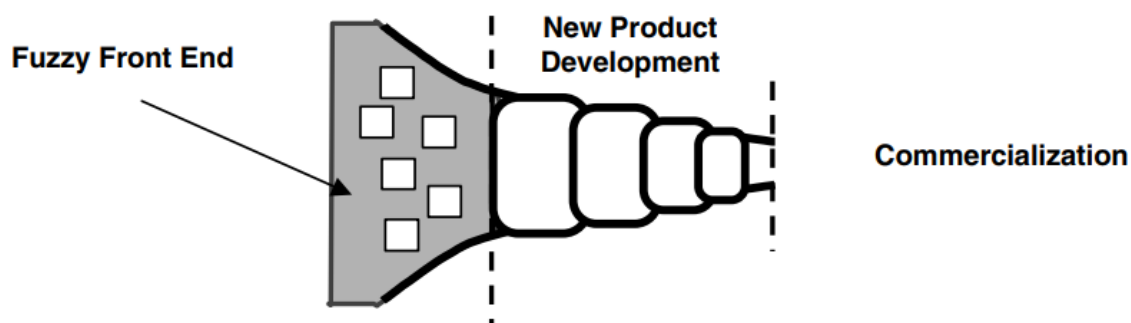


Figura 16 - Processo de Inovação

Fonte: (Koen *et al.*, 2002)

Com o passar dos tempos, surge a necessidade de modelar o processo *FFE* fornecendo uma terminologia generalizada para que as empresas iniciem o processo de inovação de forma orientada, sendo mais eficientes. Assim, o *New Concept Development Model* surge como uma possível resposta para dirimir a incerteza e a confusão do processo acima mencionado (Koen *et al.*, 2002).

4.2 New Concept Development Model

O modelo NCD consiste em três partes chaves e cinco elementos de atividade que representam um modelo relacional ao invés de um processo linear (Figura 17).

As três partes são:

- O motor fornece energia para o *front end* do processo de inovação, que surge da estrutura e cultura da organização e estratégia de negócios, o que significa que esta parte é controlável.
- A roda contém os cinco elementos de atividade controláveis: identificação de oportunidade, análise de oportunidade, geração e enriquecimento de ideias, seleção de ideias e definição de conceito.
- O aro consiste nos fatores ambientais externos que podem influenciar o motor e os elementos da atividade, como clientes, concorrentes, políticas e leis governamentais, situação política e econômica atual e outros, sendo considerada a parte incontrolável do modelo.

As setas que apontam para a roda representam os possíveis pontos de partida para entrar no fluxo iterativo - identificação da oportunidade ou geração e enriquecimento de ideias - e a seta restante representa que o processo de inovação continua para o próximo processo após a

conclusão do *fuzzy front end*. É considerado um fluxo iterativo, pois é preferível ter *loopbacks* e refazer ou combinar atividades na etapa FFE do que nas etapas seguintes (desenvolvimento e comercialização), pois os custos de refazer atividades crescem exponencialmente à medida que se avança no processo de inovação.

Portanto, o termo “cinco elementos de atividade” é dado porque chamá-los de processos, transmite a ideia de que estes têm de ser estruturados, o que "pode não ser aplicável e poderia forçar o uso de um conjunto de controlos mal projetados para gerir atividades de FFE" (Koen *et al.*, 2002).

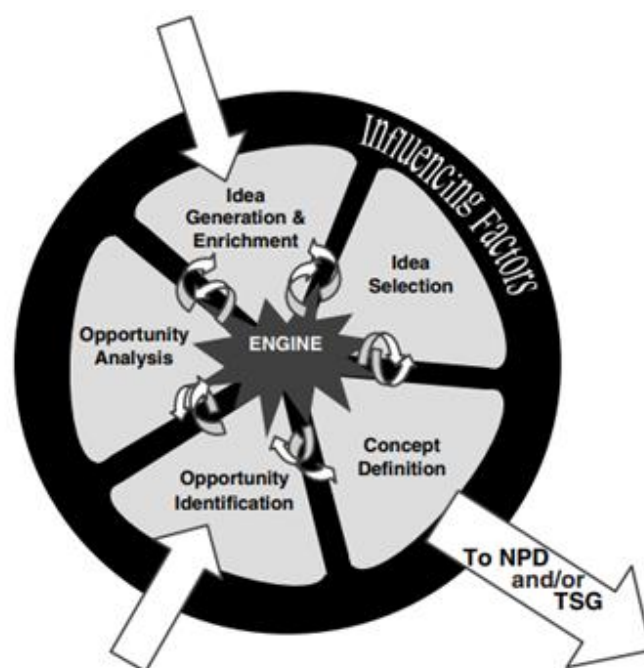


Figura 17 – New Concept Development

Fonte: (Koen *et al.*, 2002)

Assim sendo, vai-se analisar os elementos de atividade que se adequam ao caso atual e como este foram aplicados no processo de inovação.

4.2.1 Identificação da Oportunidade

Uma oportunidade é geralmente uma necessidade comercial ou técnica que a empresa ou diferentes indivíduos reconhecem que existe no produto ou serviço atual e que precisa de ser reavaliada para melhorar a solução em andamento ou até mudar para uma melhor. Estas

necessidades podem surgir para responder a uma ameaça competitiva, encontrando uma possível solução inovadora para acompanhar as novas áreas de crescimento do mercado, simplificando e agilizando processos, e/ou reduzindo os custos das operações realizadas. Normalmente, ter conhecimento destas necessidades ajuda as empresas a saber onde se devem focar, levando-as a definir metas para dar uma resposta a determinada situação.

Assim, uma das metas organizacionais da empresa até ao final do ano é possibilitar implantações no ambiente de produção a qualquer hora do dia, o que significa que as equipas terão mais liberdade no que toca aos processos de entrega, conduzindo ao alcance da metodologia de *CI/CD*. A empresa está a caminhar para isso na parte do *backend*, dividindo o antigo monólito em microsserviços. Portanto, tendo em conta que a empresa trabalha com equipas multifuncionais, os programadores de *frontend* começam a compreender que apenas migrar o *frontend* antigo para uma framework atualizada não é suficiente, pois conforme o projeto do novo *frontend* vai crescendo, os programadores deparam-se com muito mais problemas, conflitos, e o código está a tornar-se confuso e difícil de gerir. Isso leva à lentidão na codificação, o que está a causar menos entregas ao cliente, uma vez que se leva mais tempo para os programadores compreenderem o código-fonte atual e se torna mais difícil para os recém-chegados serem integrados e conhecerem o projeto atual.

Vendo isso como uma ameaça e uma necessidade técnica e arquitetónica para melhorar a manutenibilidade do código e os processos de *release*(entrega), o próximo passo é analisá-la com cuidado. Como os membros de diferentes equipas da empresa já veem o uso de microsserviços como um sucesso para atingir o objetivo mencionado, e como as bases dos micro *frontends* proveem dessa mesma arquitetura, a empresa quer validar se uma abordagem de micro *frontends* pode responder melhor aos problemas mencionados.

4.2.2 Análise da Oportunidade

Neste elemento da roda do modelo NCD faz-se a avaliação da oportunidade, utilizando estudos, questionários e experiências existentes sobre o assunto para reduzir as incertezas da oportunidade.

Assim sendo, verificou-se que não existem muitos questionários disponíveis sobre micro *frontends*, contudo foi encontrado um recente sobre microsserviços que descreve não apenas a arquitetura na qual os micro *frontends* são baseados, mas também um tópico específico sobre UI monolítica.

“*Microservices Adoption in 2020*” é um relatório resultante de um estudo criado pela O’Reilly, que mostra que 92% das organizações obtiveram um grande benefício com a arquitetura de microsserviços. O questionário foi respondido por 1502 pessoas da área de IT, nomeadamente engenheiros de diversas áreas espalhados por todo o mundo. Com este estudo, pode notar-se

que “um terço dos adotantes está a migrar a maioria dos seus sistemas usando microserviços” e outras descobertas notáveis como:

- A maioria (cerca de 61%) usa a arquitetura de microserviços há um ano ou mais e cerca de 23% não faz uso desta.
- Quase três quartos dos respondentes (cerca de 74%) têm todo o ciclo de vida do software (ou seja, desenvolver, testar, implantar e manter) a eles atribuído, e cerca de metade relatou a sua arquitetura de microserviços como “Muito bem sucedida” e 10% como um “Sucesso total”.
- Aproximadamente 45% dos adotantes veem como benefícios a flexibilidade e capacidade de responder mais rapidamente às mudanças nos requisitos, 44% também consideram a escalabilidade e 43% respondem que poder fazer entregas de código com mais frequência é uma grande vantagem (Figura 18).

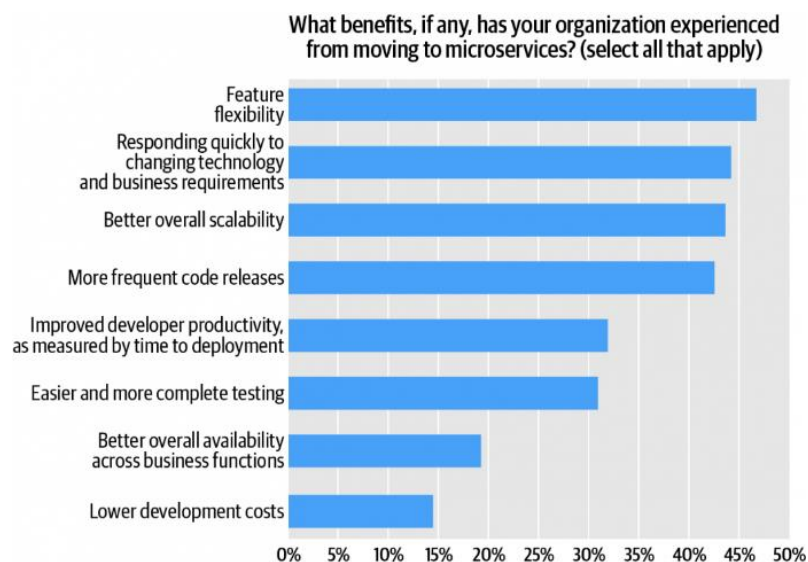


Figura 18 - *Microservices Adoption in 2020*: Benefícios no uso de microserviços

Fonte: (Swoyer, 2020)

- 31% disseram que não constroem uma IU monolítica para seus sistemas implantados como microserviços, mas também são a maioria das pessoas com uma experiência falhada (17% descrevem suas implementações como "Nada bem-sucedidas", 37% como "Na maior parte bem-sucedidas" e apenas 12% como "Sucesso total"). No entanto, no lado oposto, há adotantes que usam UI monolítica em cerca de 50-75% dos seus projetos e cerca de 52% deles disseram que foram “muito bem-sucedidos” nas suas implementações (Figura 19).

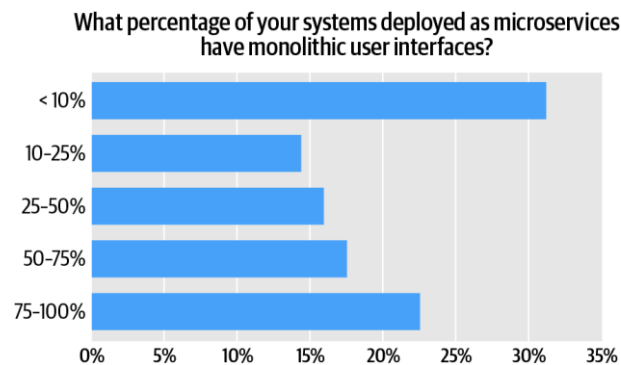


Figura 19 -*Microservices Adoption in 2020*: Sistemas com microsserviços que usam uma UI monolítica
Fonte: (Swoyer, 2020)

Uma análise mais aprofundada dos resultados da pesquisa conclui que, devido à concepção errada do que é uma interface de utilizador monolítica e do que não é, existem respostas em ambas as extremidades do espectro. Se os adotantes apenas considerarem o fator simplicidade, é normal haver uma alta taxa de falha em *UIs* não monolíticas. Portanto, dependendo do projeto e do que é mais necessário para ele, como por exemplo, flexibilidade, facilidade de manutenção e complexidade, fazer uma mudança arquitetural no *frontend* pode proporcionar muitos ganhos e benefícios (Swoyer, 2020; Business Wire, 2020).

Além disso, o “State of Frontend 2020” criado pela The Software House e respondido por cerca de 4500 programadores de *frontend* foi comentado por alguns especialistas em desenvolvimento de software. Por exemplo, alguns resultados interessantes apontados por Witold Ossera (o Chefe de Serviços de *Frontend* na Software House) mostraram que 24,4% dos programadores web já obtiveram alguns benefícios - principalmente melhorias na eficiência de desenvolvimento, redução de erros e bugs e produção acelerada - com uma arquitetura de *micro frontends*, além de que a integração contínua está a tornar-se um padrão no desenvolvimento de *frontend* (Dąbrowska, 2020).

Houve outro questionário desenvolvido pela mesma empresa, chamada “State of Microservices 2020”, contendo 669 respostas de especialistas em microsserviços de todo o mundo, e Luca Mezzalana, autor do livro “Building Micro-Frontends”, partilhou a sua opinião sobre os 24% dos especialistas que utilizaram *micro frontends*. Ele acredita que a percentagem de pessoas que utilizam essa arquitetura não irá crescer muito além desse valor, no entanto, isso não significa que os *micro frontends* não terão lugar como um padrão de *frontend*. Ele geralmente desencoraja o seu uso quando não faz sentido com o projeto. Alguns casos disso podem ser quando se trata de um novo projeto ou quando as pessoas não sabem o que a empresa pretende alcançar no projeto atual. No entanto, se houver necessidade de escalar, a empresa precisa de preparar o projeto abarcando o padrão de *micro frontend*, uma vez que “este

paradigma permite escalar ao dividir a interface em funcionalidades separadas desenvolvidas por equipas distintas” (*State of Microservices 2020 Report*, 2020).

Um exemplo de sucesso é o Agorapulse (*Agorapulse*, no date). Resumidamente, o Agorapulse é uma plataforma de gestão de redes sociais e relacionamento com clientes que permite às empresas gerir todo o seu conteúdo e mensagens de redes sociais num único painel centralizado. Assim, para conseguir escalar facilmente a aplicação, eles utilizaram a arquitetura de micro *frontends*, uma vez que as suas equipas são autónomas - o que significa que conseguem desenvolver uma funcionalidade de ponta a ponta- também melhoraram o processo de entrega, reduzindo o tempo investido nesses processos *multi-threaded*, aumentando também a simplicidade dos mesmos. Além disso, outras empresas de renome adotaram as mesmas abordagens, como Spotify, IKEA, American Express e SoundCloud, entre muitas outras vantagens, devido à sua praticidade em relação a melhorias de UI/UX (Dhaduk, 2021).

4.2.3 Definição de Conceito

Para melhor compreensão, um conceito é uma forma definida que tem características e benefícios para o cliente bem descritos, de acordo com o estudo da solução necessária. Portanto, neste elemento de atividade, existe a etapa final para terminar o NCD. Nesta fase, pode-se abordar os objetivos, a adequação do conceito com as estratégias da empresa e os fatores de risco, juntamente com outras informações relevantes (que podem ser quantitativas e/ou qualitativas) sobre as necessidades e benefícios do mercado ou dos clientes. Além disso, é importante fornecer um plano de projeto com as fases que se pretende seguir.

Consequentemente, foram planeadas algumas etapas:

- Recolher dados e informações científicas sobre o assunto.
- Analisar profundamente as possibilidades dentro do conceito e categorizá-las com base nas necessidades da empresa.
- Desenvolver uma prova de conceito reconstruindo o projeto com a abordagem de micro *frontends* que se adapta ao problema.
- Analisar se o conceito pode resolver o cenário atual.

Alguns riscos também foram identificados para uma implementação futura real:

- Número de programadores com experiência em arquitetura de microsserviços, ferramentas de automação e metodologia *CI/CD*.
- Pelo menos uma equipa dedicada à migração, para não interromper o desenvolvimento atual e a entrega de novas funcionalidades para os clientes.

- Será necessário planejar o tempo para documentar essa migração, para que as outras equipas da empresa possam começar a incluí-la nos seus planeamentos e realizá-la de forma gradual e iterativa.

4.3 AHP

O Processo Analítico Hierárquico é uma teoria de medição desenvolvida por T. L. Saaty entre 1971 e 1975, que organiza uma decisão de forma estruturada através de comparações par a par e, com base no conhecimento atual, constrói escalas de prioridade. A comparação feita nesta teoria considera, para um atributo específico, o quão importante um certo elemento é em relação a outro, ou qual é mais valioso do que o outro, sendo isso decidido pelos especialistas na área (Saaty, 2008).

Como o nome sugere, a chave desse processo é dividir hierarquicamente os diferentes níveis de decisão (problema, critérios e alternativas), facilitando assim a compreensão e a avaliação do problema, obtendo numericamente a melhor opção de decisão (Nicola, no date).

Neste caso específico, quando se trata de abordar a divisão horizontal dos micro *frontends*, há diferentes tipos de composições para decidir qual se adapta melhor às necessidades da empresa.

Na primeira fase do AHP, é criada a árvore de decisão hierárquica (Figura 20) que decompõe o problema, os critérios e as alternativas.

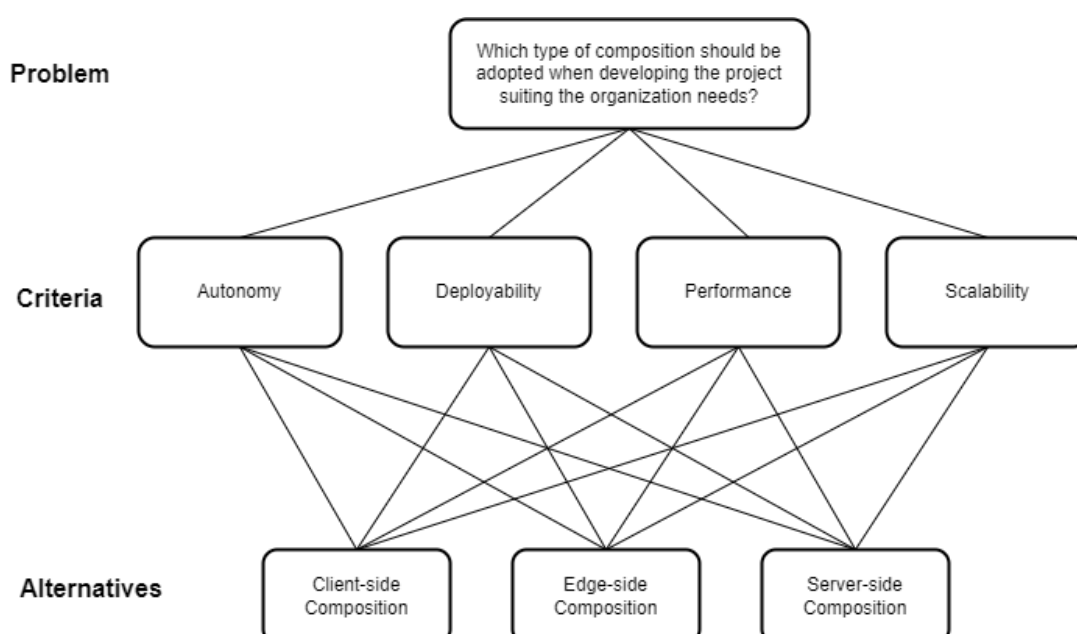


Figura 20 – Árvore de decisão hierárquica

A segunda fase consiste em estabelecer as prioridades entre cada critério, numa matriz de comparação. Os valores preenchidos na matriz baseiam-se na escala fundamental apresentada na Figura 21.

<i>Intensity of Importance</i>	<i>Definition</i>	<i>Explanation</i>
1	Equal Importance	Two activities contribute equally to the objective
2	Weak or slight	
3	Moderate importance	Experience and judgement slightly favour one activity over another
4	Moderate plus	
5	Strong importance	Experience and judgement strongly favour one activity over another
6	Strong plus	
7	Very strong or demonstrated importance	An activity is favoured very strongly over another; its dominance demonstrated in practice
8	Very, very strong	
9	Extreme importance	The evidence favouring one activity over another is of the highest possible order of affirmation
Reciprocals of above	If activity <i>i</i> has one of the above non-zero numbers assigned to it when compared with activity <i>j</i> , then <i>j</i> has the reciprocal value when compared with <i>i</i>	A reasonable assumption
1.1–1.9	If the activities are very close	May be difficult to assign the best value but when compared with other contrasting activities the size of the small numbers would not be too noticeable, yet they can still indicate the relative importance of the activities.

Figura 21 - Escala fundamental

Fonte: (Saaty, 2008)

Portanto, considerando os quatro critérios da árvore hierárquica, obtem-se a matriz de comparação par-a-par da Tabela 1.

Tabela 1 - Comparação de critérios

Criteria	Autonomy	Deployability	Performance	Scalability
Autonomy	1	1/2	3	1/3
Deployability	2	1	3	1/3
Performance	1/3	1/3	1	1/4
Scalability	3	3	4	1

Depois disso, a matriz de comparação é normalizada, dividindo cada valor da matriz pelo total da respectiva coluna (Tabela 2).

Tabela 2 - Comparação de critérios - valores normalizados

Criteria	Autonomy	Deployability	Performance	Scalability
Autonomy	0.158	0.103	0.273	0.174
Deployability	0.316	0.207	0.273	0.174
Performance	0.053	0.069	0.091	0.130
Scalability	0.474	0.621	0.364	0.522

Ao calcular a média aritmética do valor para cada linha da matriz normalizada, obtem-se o vetor de prioridade (Tabela 3).

Tabela 3 - Vetor de Prioridade

Criteria	Priority
Autonomy	0.177
Deployability	0.242
Performance	0.086
Scalability	0.495

Para verificar se o vetor de prioridade é válido, precisa-se de calcular o Rácio de Consistência (RC), que é obtido dividindo o Índice de Consistência (IC) pelo Índice Aleatório (IA).

O IC é calculado dividindo a subtração do maior autovalor da matriz pelo número de critérios, pela subtração do número de critérios com 1.

O IA é dado por valores tabelados, uma vez que temos 4 critérios, o valor correspondente é 0,90.

Tendo isso em conta, pode-se calcular o RC para o problema atual. Se o valor obtido como Rácio de Consistência for inferior a 0,1, as avaliações são confiáveis; caso contrário, os valores dados

acima são considerados aleatórios (Nicola, no date). Assim, com o maior autovalor sendo 4,145 e o IC 0,048, o RC é inferior a 0,1, o que significa que a avaliação é confiável.

Portanto, pode-se passar para a próxima etapa, que é ter a matriz de comparação para cada critério por cada alternativa (Tabela 4).

Tabela 4 - Matriz de comparação para cada critério por cada alternativa

Autonomy			
	Client-side	Edge-side	Server-side
Client-side	1	5	3
Edge-side	1/5	1	1/5
Server-side	1/3	5	1
Deployability			
	Client-side	Edge-side	Server-side
Client-side	1	3	1
Edge-side	1/3	1	1/3
Server-side	1	3	1
Performance			
	Client-side	Edge-side	Server-side
Client-side	1	3	1/5
Edge-side	1/3	1	1/7
Server-side		7	1
Scalability			
	Client-side	Edge-side	Server-side
Client-side	1	3	5
Edge-side	1/3	1	3
Server-side	1/5	1/3	1

Depois das matrizes acima serem normalizadas, pode-se calcular o vetor de prioridade composto por alternativa, obtendo os valores da Tabela 5.

Tabela 5 - Matriz alternativa e vetor final

	Autonomy	Deployability	Performance	Scalability	Priority
Client-side	0.607	0.429	0.193	0.633	0.541
Edge-side	0.090	0.143	0.083	0.260	0.186
Server-side	0.303	0.429	0.724	0.106	0.272

Nesta fase, pode-se prosseguir com a decisão baseada no AHP. De acordo com as prioridades definidas, tem-se para a divisão horizontal, a melhor opção para a abordagem de composição neste caso específico é a *client-side*.

5 Projeto Base

O presente capítulo contém o *design* e implementação do projeto representativo do atual com um *frontend* monolítico e o *backend* constituído por microsserviços.

Primeiramente, introduz-se o *design* do projeto, seguido da sua *stack* tecnológica, onde as tecnologias usadas são apresentadas, finalizando com a descrição desta implementação.

5.1 Design

O projeto do *frontend* monolítico criado apresenta as funcionalidades principais do produto de pré-contabilidade. Estas funcionalidades resumem-se ao registo e análise dos documentos que foram entregues ao contabilista por uma empresa ou pessoa sua cliente. Os contabilistas têm uma variedade de ferramentas contábeis com as quais precisam de trabalhar, podendo ser diferentes dependendo do cliente a quem estão a fornecer este serviço. O projeto com o *frontend* monolítico facilita o trabalho dos contabilistas dado que, é uma plataforma que centraliza várias ferramentas de contabilidade, fornecendo a entrada automatizada de dados com o *OCR (Optical Character Recognition)* para agilizar o processo de validação de faturas, despesas e receitas. De modo, a ser um exemplo mais reduzido, esse projeto apenas simula o suporte de duas ferramentas contabilísticas não havendo uma integração com sistemas reais (Figura 22).

Pre-Accounting SYSTEM [Home](#) [Post Invoices](#)

Post Invoices

Accounting Tools
 @Tokyo System @Seoul System

[Previous Document](#) [Next Document](#)



Beamsweg 10
 3705 AA BARN
 info@paintshop.nl
 NL12RAB00417164300

Invoice to:
 Jorge Afonso B.V.
 Eemweg 8
 3743 LB Baarn

InvoiceNumber: 0011
 InvoiceDate: 03-10-2020
 Invoice DueDate: 01-11-2020

Customer *

1 - Mike Wazowski

[New Customer](#)

Invoice Number

Enter Invoice Number

Amount Currency *

0

Transaction Date *

dd/mm/yyyy

Figura 22 - Página de registo de faturas

Na Figura 23 está apresentado o diagrama de componentes do projeto com o *frontend* monolítico.

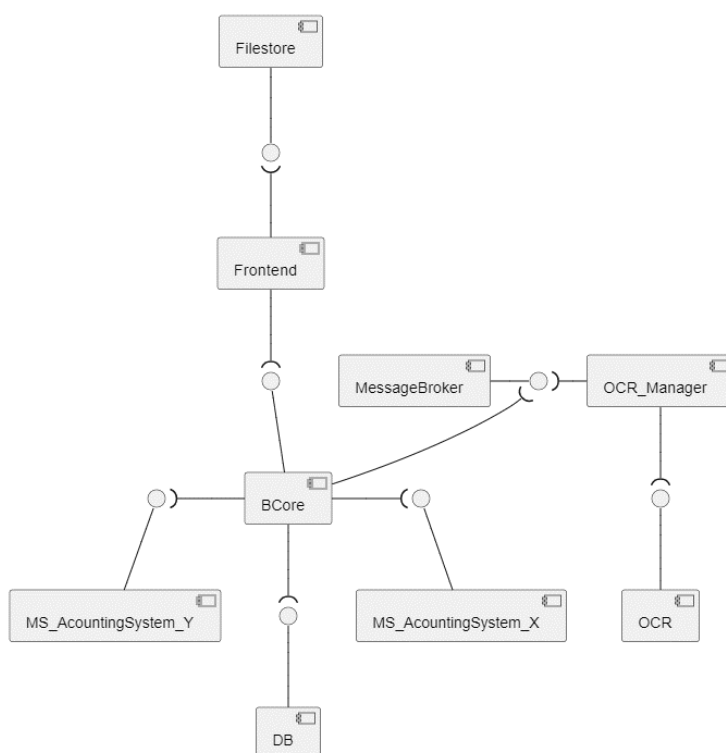


Figura 23 - Diagrama de Componentes: microsserviços com frontend monolítico

O **Filestore** contém um conjunto de ficheiros armazenados na *BlobStorage* de modo a simular o *upload* de ficheiros por parte do utilizador. Esses ficheiros são carregados pelo *frontend*.

O **OCR manager** funciona como um intermediário entre a *API* externa do **OCR** usado e os componentes do sistema. A sua função consiste em mapear os campos detetados, para popular os campos correspondentes dentro do sistema.

Os microsserviços (**MS Accounting System X**, **MS Accounting System Y**) são responsáveis por transformar o *schema* genérico, usado dentro do sistema num *schema* reconhecido pelas diferentes ferramentas de contabilidade suportadas.

O **BCore** engloba a lógica que é independente das ferramentas de contabilidade. E o **Message Broker** promove comunicação assíncrona no upload e análise do documento.

5.2 Descrição da *Stack* Tecnológica

As diferentes tecnologias usadas para construir o projeto base são apresentadas nesta secção.

5.2.1 Microsserviços

Existem quatro microsserviços para dar resposta aos requisitos da aplicação. Estes intitulam-se de *BCore*, *TokyoSystem*, *SeoulSystem* e *OCRManager*.

Cada um deles desenvolvido em *.NETCore* na versão 6, por ser a *framework* usada na maioria dos projetos da empresa.

O *TokyoSystem* e o *SeoulSystem* contêm a lógica relativa a cada ferramenta contabilística. Enquanto o *BCore* tem a lógica que é geral, relativa aquilo que é do domínio Basecone e não depende das ferramentas contabilísticas suportadas.

O *OCRManager* é responsável por estabelecer a comunicação entre o *OCR* e os componentes do sistema, através de *event sourcing*.

5.2.2 *Frontend* Monolítico

O *frontend* monolítico usou a *framework* de *Angular* na versão 14, que é a versão para a qual os programadores da empresa estão a realizar a migração. Este projeto procurou ter semelhanças na sua implementação com o projeto real, para demonstrar algumas dificuldades no desenvolvimento, que estão a prejudicar a produtividade das equipas.

5.3 Implementação

No *frontend* monolítico, embora as equipas usem a mesma fonte de código, foi considerado a distribuição de responsabilidades por três equipas.

Tabela 6 - Responsabilidade de cada equipa

Equipa	Responsabilidade
A	Desenvolver funcionalidades relativas à ferramenta contável <i>SeOul</i>
B	Desenvolver funcionalidades destinadas à ferramenta contável <i>ToKyo</i>
C	Desenvolver funcionalidades do Cliente/Fornecedor e do cabeçalho das faturas. Assegurar que a página inicial, e a barra de navegação estão atualizadas e desenvolver funcionalidades para apresentar o documento.

A página inicial (Figura 24) apresenta um pequeno conteúdo em texto e contém a barra de navegação que permite aceder à página que contém o registo as faturas que foram carregadas previamente.

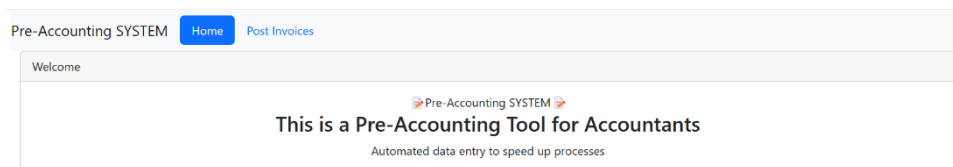


Figura 24 – Página inicial

A página onde todas as equipas trabalham nas suas diferentes funcionalidades é a página de *Post Invoices* (Figura 25).

Figura 25 – Início da página de *Post Invoices*

O componente *Angular Invoice-form* desempenha um papel central na aplicação, abrangendo todas as funcionalidades necessárias. Isto leva as três equipas a colaborarem no mesmo contexto.

O *previewer* apresentado neste projeto é um simples *iframe*, mas pretende representar o presente no projeto real que contém várias funcionalidades requisitadas pelos clientes que o permitem aumentar ou diminuir o ficheiro, ajustar à altura ou ajustar ao comprimento do *previewer*, descarregar o ficheiro, imprimir, copiar texto a partir de uma ferramenta de recorte. Daí a equipa C (Tabela 6) ter também a responsabilidade de apresentação do documento. Contudo, não é o foco principal deste projeto, portanto, simplificou-se num *iframe*.

A página também exibe as linhas da fatura (Figura 26), cuja estrutura pode variar conforme a ferramenta contabilística selecionada. Ao alternar entre diferentes ferramentas ou ao navegar entre documentos anteriores ou posteriores, o formulário é preenchido automaticamente com as informações fornecidas pelo *OCR*, e o documento no visualizador é atualizado de acordo.

Post Invoices

Accounting Tools

⊗Tokyo System ○SeOul System

Previous Document Next Document

InvoiceNumber: 0011
InvoiceDate: 08-10-2020
Invoice DueDate: 01-11-2020

Product	Amount	Price	Total (amount*price)
Orange paint	2.5 litres	€ 40,-	€ 100,-

Amount (excluding VAT)	VAT 23%	Total Amount (including VAT)
€ 100,-	€ 23,-	€ 123,-

PAYMENT WITHIN 30 DAYS!!!

Line 1

Account Code *
Enter accountCode

Cost Center
Enter costCenter

Amount *
0

VAT *
0

Vat Amount
0

Description
Enter description

+

+

POST REJECT SAVE

Figura 26 – Parte inferior da página de *Post Invoices*

Apesar da responsabilidade que cada equipa tem, é necessário que tenham noção das responsabilidades das outras. Um programador da equipa A (Tabela 6) que precise de implementar, o cálculo do VAT (Figura 27) para a ferramenta *SeOul* - para que este se atualize à medida que o contabilista realiza o preenchimento do formulário - e só tenha conhecimento de *SeOul*, vai verificar que já existe um método para esse fim e, portanto, corrigi-lo de acordo com o que lhe foi imposto. Como o engenheiro de qualidade da equipa só testa o que lhe pertence, os testes vão ser um sucesso e a *user story* será aceite e implantada, provocando assim um erro no mesmo cálculo para *ToKyo*. Como são realizados testes de regressão antes da entrega ao cliente o erro é detetado e corrigido por alguém da equipa B (Tabela 6).

```

vatValueCalculation(value: any, id): void {
  let vatValue =
    this.systemForm.controls["accountingSystem"].value === "tokyo"
      ? this.invoiceLines.value[id]["amount"] * value
      : this.invoiceLines.value[id]["quantity"] *
        this.invoiceLines.value[id]["unitPrice"] *
        value;
  this.invoiceLines.at(id).get("vatAmount").setValue(vatValue);
}

```

Figura 27 - Cálculo do VAT

O problema foi resolvido, porém e como consequência disso a equipa B (Tabela 6) não conseguiu entregar ao cliente tudo o que tinha planeado. Neste projeto de *frontend* monolítico este tipo de problemas é bastante comum, e o aparecimento de *bugs* acaba por se tornar constante.

A implementação deste projeto foi realizada de forma a conseguir representar erros que acontecem no projeto real, mas de forma mais simplificada (Silva, 2023).

6 Requisitos

Os requisitos funcionais delineiam as funcionalidades e operações que o sistema deve ser capaz de realizar, enquanto os requisitos não funcionais definem os atributos de qualidade requeridos para o sistema.

Neste capítulo, apresentam-se esses requisitos, garantindo uma compreensão clara e abrangente do que é esperado do sistema.

Assim sendo, os requisitos funcionais estão representados pelos seguintes caso de uso da Figura 28:

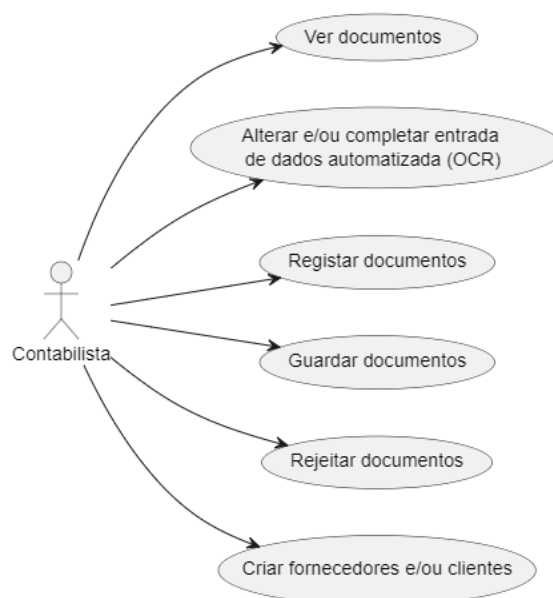


Figura 28 - Diagrama de Casos de Uso

E os não funcionais são:

- A divisão do projeto em micro *frontend*;
- Deve ser mais fácil de suportar mais de uma ferramenta de contabilidade;
- Deve ser mais fácil de adicionar novas funcionalidades ao projeto, de acordo com a necessidade do utilizador;
- O desempenho deve ser pelo menos igual ao da solução anterior;
- O código-fonte deve ser mais fácil de manter e mais claro para que menos tempo seja gasto na compreensão do código atual. Portanto o sistema deve ser dividido em unidades independentes responsáveis por parte da solução, promovendo o desacoplamento, a coesão e consequentemente a manutenibilidade.

Neste capítulo não se quantificou com valores concretos os requisitos não funcionais. Esses valores podem ser consultados no capítulo 8, onde são apresentados os resultados obtidos com o *frontend* monolítico. Com base nesses resultados, verifica-se através de uma comparação, se a implementação de micro *frontends* vai conseguir cumprir os requisitos não funcionais mencionados.

Além disso, tendo os mesmos requisitos funcionais para ambos os projetos, obtém-se uma comparação muito mais justa entre o projeto base e o desenvolvido com micro *frontends*. Assim, as conclusões podem ser tiradas num ambiente controlado.

7 Solução

O presente capítulo tem como intuito a apresentação do *design*, da *stack* tecnológica e implementação da solução de micro *frontends*.

A implementação da solução está descrita com detalhe, incluindo a configuração inicial do projeto, como foi realizado o desenvolvimento e implantação dos micro *frontends*.

7.1 Design

Tal como a divisão já implementada no *backend*, em que cada microsserviço contém a lógica relativa a uma só ferramenta de contabilidade, no *frontend* pretende-se que a divisão principal seja do mesmo modo, isto é, havendo os micro *frontends*: **MF Accounting System X**, **MF Accounting System Y** (Figura 29).

A descrição da responsabilidade dos restantes componentes já foi descrita na secção 5.1.

Apenas se representaram dois micro *frontends* no diagrama de componentes da Figura 29, em jeito de simplificação. Contudo, o *frontend* está dividido nos micro *frontends* representados na Figura 30.

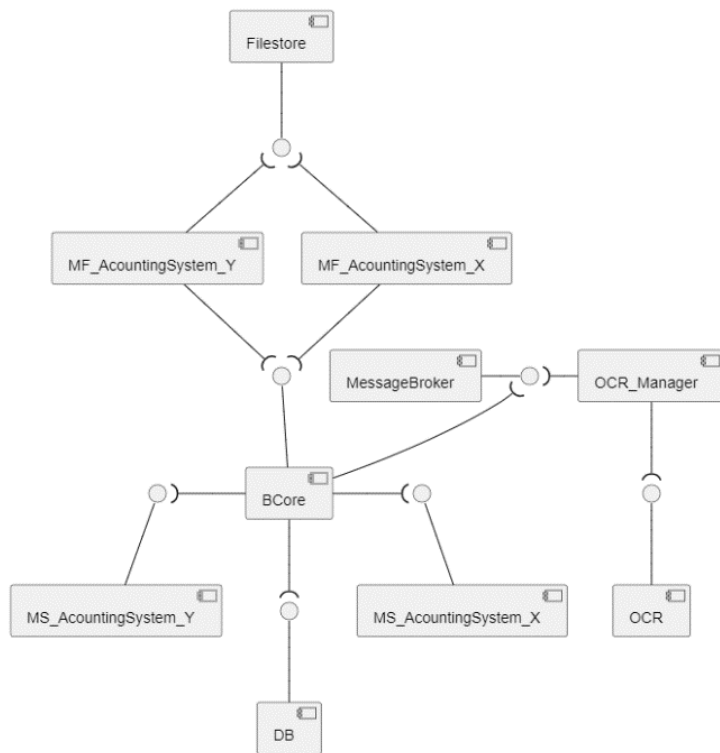


Figura 29 - Diagrama de Componentes: microserviços com micro *frontends*

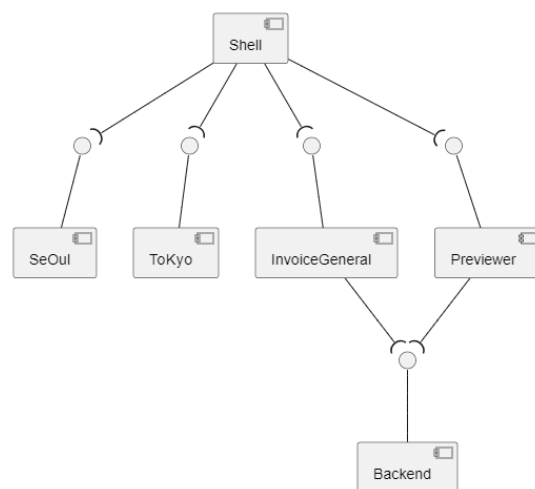


Figura 30 - Diagrama de Componentes: micro *frontends*

A *Shell* contém apenas a página inicial e a barra de navegação, sendo responsável pelo *routing* dos *micro frontends*. Para isso, cada *micro frontend* tem pelo menos um módulo seu exposto com a sua *remoteEntry*, que contém toda a informação necessária que a *Shell* precisa de saber para carregar cada *micro frontend*.

Os *micro frontend SeOul* e *ToKyo* intitulam-se com os nomes das ferramentas de contabilidade suportadas, uma vez que representam aquilo que é específico destas.

O *InvoiceGeneral* contém a lógica dos fornecedores e clientes, bem como a parte geral da fatura que não depende de nenhuma ferramenta de contabilidade.

Por último, o *Previewer* contém a lógica para apresentação do documento ao cliente, para que seja possível comparar os valores do documento com os introduzidos no formulário pelo *OCR*. Este *micro frontend* inclui também a seleção da ferramenta de contabilidade.

SeOul e *ToKyo* não comunicam diretamente com o *backend*. Todavia, a informação chega a estes *micro frontends* por meio de *Custom Events*.

7.2 Descrição da *Stack* Tecnológica

Nesta secção, estão presentes as diferentes tecnologias usadas nesta solução.

A informação relativa aos microsserviços desenvolvidos pode ser encontrada na subsecção 5.2.1, dado que para ambas as abordagens foram usadas os mesmos microsserviços.

7.2.1 Micro Frontends

Para os *micro frontends* decidiu-se manter o desenvolvimento em Angular, pois apesar de haver a possibilidade de diversificar as *frameworks* usadas em cada *micro frontend*, essa abordagem é considerada um anti padrão (secção 2.4) que se poderia justificar, se a empresa optasse por migrar o seu *frontend* para outra *framework*. De momento, nenhuma migração está nos planos da empresa pelo que todos os *micro frontends* se encontram neste projeto na mesma versão que o *frontend* monolítico.

7.3 Implementação

Tendo em conta as mesmas responsabilidades para cada equipa apresentada na Tabela 6 nos *micro frontends* com *microserviços* temos a seguinte distribuição na Tabela 7:

Tabela 7 - Distribuição dos *microserviços* e *micro frontends*

Equipa	Micro frontend	Microserviço
A	SeOul	SeOulSystem
B	ToKyo	ToKyoSystem
C	Invoice General Shell Previewer	BCore OCRManager

Esta divisão tem o intuito de trazer mais independência às equipas evitando que os problemas que ocorrem no *frontend* monolítico aconteçam também com esta abordagem.

Na Figura 31 mostra a parte inicial da página de registo de faturas dividida em *micro frontends*. A *Shell* está rodeada a azul-escuro, o *Previewer* a vermelho e *Invoice General* a azul-claro.

Pre-Accounting SYSTEM Home Post Invoices

Post Invoices

Accounting Tools

Tokyo System SeOul System

Previous Document Next Document

The PAINT SHOP

Baarenweg 10
3700 AA, BAARN
info@baarnshop.nl
NL12RABO0417164300

Invoice to:
Jorge Alonso B.V.
Eemweg 8
3742 LD, Baarn

InvoiceNumber: 0011
InvoiceDate: 03-10-2020
InvoiceDueDate: 01-11-2020

Customer *

1 - Mike Wazowski

New Customer

Invoice Number

Enter Invoice Number

Amount

0

Currency *

Transaction Date *

dd/mm/yyyy

Figura 31 - Página de Registo de Faturas composta por *micro frontends*

7.3.1 Configuração Inicial

Após a investigação realizada sobre algumas tecnologias apresentadas na secção 2.5, a ferramenta escolhida foi a *Dynamic Module Federation*.

Usou-se o comando `ng new <workspaceName> --no-create-application` para se criar um *workspace* sem a aplicação de Angular gerada por defeito.

Recorreu-se ao *package @angular-architects/module-federation* para inicializar os projetos de micro *frontends*. Depois de instalar o *package* globalmente criaram-se os remotes e a Shell:

```
ng g @angular-architects/module-federation
--project <remoteProjectName> --port <port> --type remote
```

Código 3 - Comando para inicializar os projetos *remote*

```
ng g @angular-architects/module-federation
--project <shellProjectName> --port <port> --type dynamic-host
```

Código 4 - Comando para inicializar o projeto da *Shell* dinâmica

O tipo *dynamic-host* faz com que o ficheiro de configuração *manifest* e o ficheiro *main.ts* com código para carregar essas configurações sejam criados na *Shell*.

O *manifest* permite que a definição do *remoteEntry* não esteja diretamente no routing da *Shell*. Apenas é necessário informar onde é que este tem de obter essa informação, mencionando *type: 'manifest'* no *loadRemoteModule()*. A propriedade *remoteName* aponta para a key que consta no *manifest* (Figura 32 e Figura 33).



```
{
  path: '',
  outlet: 'previewer',
  loadChildren: () =>
    loadRemoteModule({
      type: 'manifest',
      remoteName: 'previewer',
      exposedModule: './Module',
    }).then((m) => m.PreviewerModule),
},
```

Figura 32 – Excerto *app.routes.ts* da *Shell*: caminho para o *Previewer*

```

"previewer": {
  "remoteEntry": "http://localhost:4201/remoteEntry.js",
  "exposedModule": "./Module",
  "displayName": "Previewer",
  "routePath": "previewer",
  "ngModuleName": "PreviewerModule"
},

```

Figura 33 - Excerto *mf.manifest.json* da *Shell*: definição do *remoteEntry* do *Previewer*

Como o *remoteEntry.js* está definido no *manifest* deixa de ser necessário definir os *remotes* no *webpack.config.js* da *Shell*. O *remoteEntry.js* é gerado pelo *webpack* durante o *bundling*, e contém informação sobre as dependências partilhadas. No ficheiro de configuração do *webpack* deve-se adicionar essas dependências dentro do *shared* (Figura 34), para que o bundler reconheça que só é necessário carregar determinada dependência uma única vez (Steyer, 2022b).

Relativamente à configuração dos micro *frontends* apenas se tem de expor os módulos que se pretende no seu *webpack.config.js* (Figura 34)(Mezzalira, no date).

```

const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({
  name: 'previewer',

  exposes: {
    './Module': './projects/previewer/src/app/previewer/previewer.module.ts'
  },

  shared: {
    ...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),
  },
});

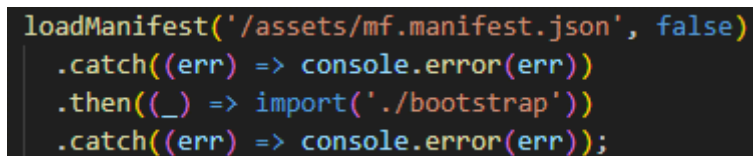
```

Figura 34 - *Webpack.config.js* do *Previewer*

Além disso, o momento em que os *remoteEntry.js* são carregados pode ser facilmente controlado no *main.ts* da *Shell*. A função *loadManifest()* tem um segundo parâmetro opcional do tipo *boolean* que se intitula *skipRemoteEntries*. Se este for *true*, permite realizar o carregamento dos *remoteEntry.js* de cada micro *frontend* apenas quando necessário, isto é, quando se navega para a página que contém um determinado micro *frontend* o *remoteEntry.js* correspondente é carregado. Se for *false*, ou então, por defeito, todos os ficheiros *remoteEntry.js* são carregados no início (Figura 35).

```
export declare function loadManifest(configFile: string,  
skipRemoteEntries?: boolean): Promise<void>;
```

Código 5 – Função *loadManifest* e seus parâmetros

A screenshot of a code editor showing the usage of the *loadManifest* function. The code is written in TypeScript and includes error handling with *catch* and *then* methods. The function is called with the path */assets/mf.manifest.json* and the boolean value *false*.

```
loadManifest('/assets/mf.manifest.json', false)  
  .catch((err) => console.error(err))  
  .then((_) => import('./bootstrap'))  
  .catch((err) => console.error(err));
```

Figura 35 – Uso da função *loadManifest* no *main.ts* da *Shell*

7.3.2 Implementação

Após a configuração inicial, o procedimento para desenvolver os diferentes *micro frontends* envolveu, inicialmente, análise do código do *frontend* monolítico. Em seguida, de forma gradual cada parte do projeto foi separada para garantir o seu funcionamento independente. Usou-se os scripts definidos no *package.json* para correr cada *micro frontend* localmente e acompanhar de forma mais fácil o desenvolvimento de cada um. Alguns estilos foram modificados para que a *Shell* que agrupa os restantes *micro frontends* consiga apresentá-los no mesmo sítio e nas mesmas dimensões. A divisão do código nos *micro frontends*, eliminou muitas condições encontradas no *frontend* monolítico tornando-as em métodos e ficheiros de *html* mais simples e pequenos. Contudo, duplicação de código surge nomeadamente nos *micro frontends* relativos ao que é específico de cada ferramenta de contabilidade.

A duplicação de código não é considerada um problema, é necessário reduzi-la sempre que possível, todavia, o foco com os *micro frontends* é a rapidez da entrega de novas funcionalidades ao cliente, impulsionada pela independência dada às equipas com esta abordagem (Mezzalira, 2023).

Após a distribuição do código entre os diferentes *micro frontends*, foi realizada uma análise das opções disponíveis para estabelecer a comunicação entre esses *micro frontends*, conforme detalhado na subsecção 2.3.4. Para esta aplicação, a escolha recaiu sobre o uso de *Custom Events* (Figura 36) como meio de comunicação. Dado que se trata de um *POC*, os *Custom Events* representaram uma escolha simples e nativa para facilitar a comunicação entre os componentes.

```

setAndDispatchEvent(name, detailValue) {
  const event = new CustomEvent(name, {
    detail: detailValue,
  });
  dispatchEvent(event);
}

```

Figura 36 - Método para espoletar um evento usando *Custom Events*

Esta abordagem, faz com que um micro *frontend* subscreva um evento espoletado por outro, quando precisa de obter determinada informação desse. Além disso, oferece a flexibilidade de subscrever eventos de um ou vários micro *frontends* envolvidos na aplicação.

7.3.3 Implantação

Relativamente à implantação do sistema, cada micro *frontend* tem uma imagem *Docker* associada que é criada aquando da *build* dos micro *frontends*. A cada *commit* é espoletada uma nova *build*, onde a *hash* que identifica esse *commit* é usada como versionamento da imagem de *Docker*. Após a criação dessa imagem, esta é guardada num *Azure Container Registry (ACR)* criado previamente. Na Figura 37 é possível analisar o *Dockerfile* que descreve como deve ser construída a imagem para cada micro *frontend*.

```

FROM node:16.16.0 as build-stage
ARG MICROFRONTEND

WORKDIR /app
COPY package*.json .
RUN npm install

COPY . .
RUN npm run build:${MICROFRONTEND}

FROM nginx:alpine
ARG MICROFRONTEND

COPY --from=build-stage /app/dist/${MICROFRONTEND} /usr/share/nginx/html
EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

```

Figura 37 - *Dockerfile* dos micro *frontends*

Essas imagens de *Docker* são usadas para fazer a implantação de cada micro *frontend* (e também dos componentes responsáveis pela lógica de negócio) no *Azure Kubernetes Service*

(AKS) também criado previamente. Sendo que as imagens estão com versões, é possível a cada *build* forçar a atualização de cada *micro frontend* para a nova versão presente no repositório.

Na Figura 38 está representado o sistema do ponto de vista de implantação, omitindo os componentes que constituem o *backend* por uma questão de simplicidade.

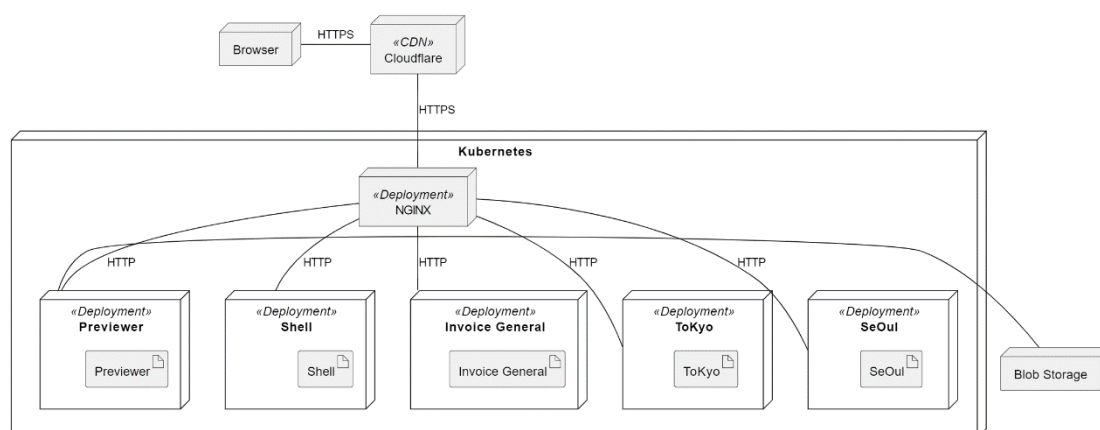


Figura 38 - Diagrama de Implantação dos *micro frontends*

Foram usados nomes de domínio para expor os *micro frontend* publicamente, sendo que estes poderiam ser provisionados em diferentes infraestruturas, garantindo assim que, independentemente do local onde se encontram, não influencie o seu bom funcionamento. Para a gestão do domínio, e criação dos subdomínios associados ao endereço *IPv4* do AKS, foi usada a *Cloudflare*. Além de servir como um sistema de gestão de domínios, também funciona como um *reverse proxy*, protegendo o tráfego entre o browser do utilizador e a infraestrutura.

Os subdomínios apontam para o ponto de entrada do AKS onde um segundo *reverse proxy* (*NGINX*) está também à espera de tráfego. Este é responsável por direcionar o tráfego, em função do *hostname* recebido, para o serviço interno correto e injetar cabeçalhos nas respostas *HTTP* que sejam relevantes para o bom funcionamento de cada componente. As gestões destas regras são possíveis usando o ficheiro *YAML* descrito na Figura 39.

```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: invoice-general
  namespace: microfrontends
  annotations:
    nginx.ingress.kubernetes.io/configuration-snippet: |
      add_header Access-Control-Allow-Origin https://shell.vitorcorreia.me;
spec:
  ingressClassName: nginx
  rules:
  - host: invoice-general.vitorcorreia.me
    http:
      paths:
      - pathType: ImplementationSpecific
        path: "/"
        backend:
          service:
            name: invoice-general
            port:
              number: 80

```

Figura 39 – Ingress no ficheiro YAML do Invoice General

Cada *micro frontend* é provisionado no AKS usando um objeto de *Kubernetes* nativo chamado *Deployment*. Este recurso é responsável por gerir várias réplicas da mesma aplicação, possibilitando a escalabilidade desta. Também é criado um objeto do tipo *Service* que possibilita o balanceamento de tráfego entre as várias réplicas. Estes objetos são representados num ficheiro *YAML* que pode ser consultado na Figura 40 e Figura 41

```

apiVersion: v1
kind: Service
metadata:
  name: invoice-general
  namespace: microfrontends
spec:
  selector:
    app: invoice-general
  ports:
  - port: 80
    targetPort: 80

```

Figura 40 - Service no ficheiro YAML do Invoice General

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: invoice-general
  namespace: microfrontends
spec:
  selector:
    matchLabels:
      app: invoice-general
  template:
    metadata:
      labels:
        app: invoice-general
    spec:
      containers:
        - name: invoice-general
          image: nicolesilva.azurecr.io/invoice-general:#IMAGE_TAG#
          resources: {}
          ports:
            - name: http
              containerPort: 80
          imagePullPolicy: Always
          volumeMounts:
            - mountPath: /etc/nginx/conf.d
              name: nginx-config
      volumes:
        - name: nginx-config
          configMap:
            name: frontend-nginx-config
            items:
              - key: nginx.conf
                path: nginx.conf

```

Figura 41 – *Deployment* no ficheiro *YAML* do *Invoice General*

8 Experimentação e Avaliação

Este capítulo visa descrever o processo usado para avaliar o desenvolvimento do trabalho, recorrendo ao método GQM para estabelecer um objetivo e questões que serão respondidas com métricas definidas.

8.1 GQM

Como o nome indica, o método é composto por metas (*Goals (G)*), perguntas (*Questions(Q)*) e métricas (*Metrics(M)*).

O(s) objetivo(s) deve(m) ser definido(s) com uma estrutura clara, e para isso existem alguns *templates* disponíveis para seguir, que podem ajudar a descrever o objetivo. Este deve conter o propósito do objetivo, o que deve ser levado em consideração, quem está a realizar a medição e o contexto em qual esta ocorre.

Após definir o objetivo, perguntas devem surgir para o refinar num nível mais técnico. Com as respostas para cada pergunta, deve concluir-se se a meta foi atingida. Neste método, cada pergunta deve ter uma resposta que corresponda a uma hipótese.

Por fim, as métricas são dados concretos que podem responder a cada uma das perguntas definidas. Essas métricas devem ser a fonte da verdade para responder a todas as perguntas, no entanto, algumas métricas dependem de fatores externos que podem resultar em resultados errôneos, por isso, devem ser tidos em consideração (Solingen, Berghout and Berghout, 1999).

Assim sendo, temos representado o plano GQM na Figura 42.

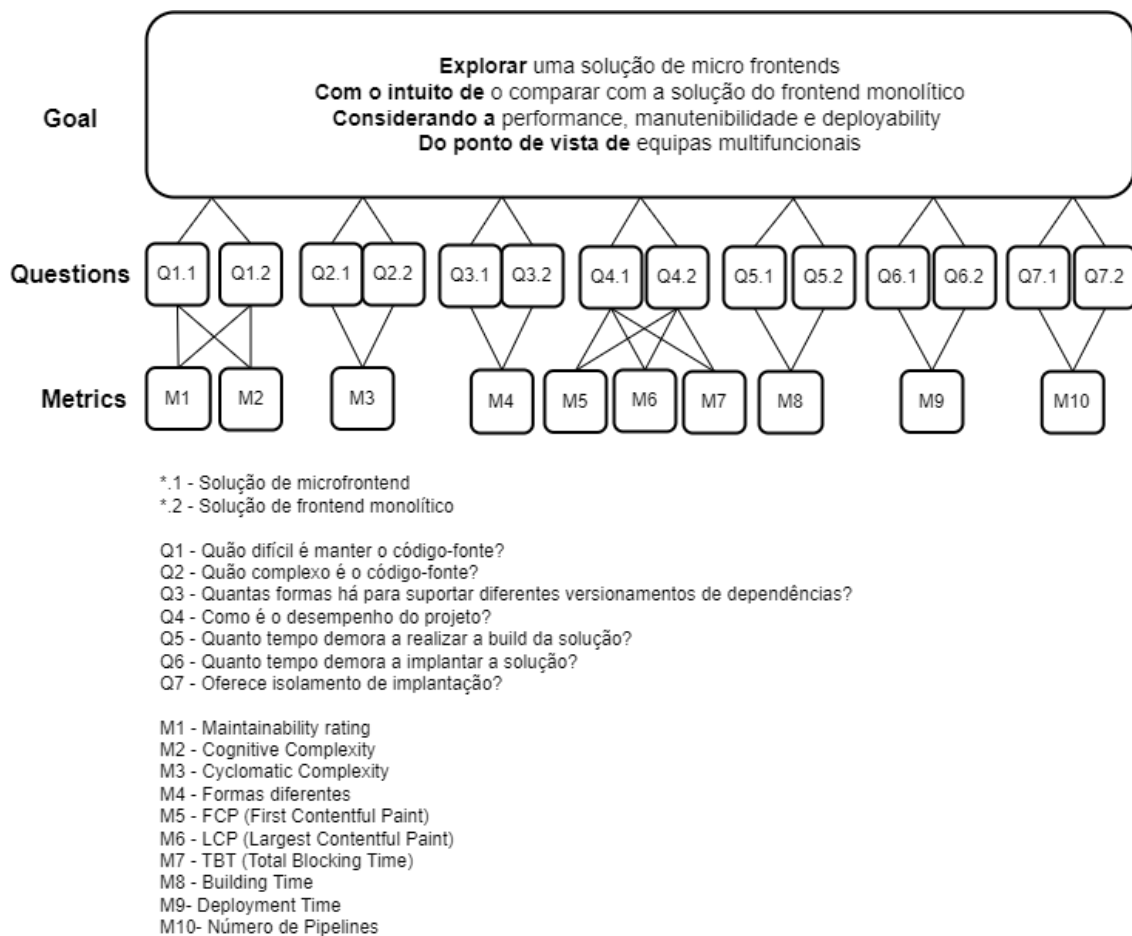


Figura 42 - Plano GQM

As métricas apresentadas surgem da pesquisa e análise de ambas as arquiteturas, tendo por base o objetivo deste trabalho (secção 1.4). A empresa pretende obter melhorias na autonomia das equipas, na facilidade de implantação e manutenibilidade da solução, sem penalizar a sua performance. Assim, as métricas apresentadas na Figura 42 procuram fornecer valores concretos para essas necessidades.

Pretende-se estabelecer uma comparação entre os dois projetos. Portanto, independentemente de se concluir que optar por uma arquitetura de micro *frontends* se adequa ou não ao caso, o objetivo deste trabalho fica concluído, dado que é o que se pretende analisar.

Neste caso, por cada métrica analisada entende-se como resultado aceitável todos os valores do projeto de micro *frontends* que são superiores ou iguais ao projeto do *frontend* monolítico.

Considera-se de igual peso as métricas referidas. Isto significa que se entende os micro *frontends* como uma solução viável, se a maioria dos resultados for positiva.

8.2 Ferramentas de Análise

SonarQube

O *SonarQube* é usado para revisão automática de código e análise de segurança. Ele tem várias métricas que considera ao analisar o código-fonte, como classificação de manutenibilidade, cobertura de código e complexidade cognitiva.

Na plataforma *SonarQube*, é apresentado o número de code smells detetados no projeto, o que ajuda os programadores a encontrar a parte do código que possivelmente pode ser melhorada junto com a medida do débito técnico representado com a estimativa do esforço de tempo para resolver todos os code smells que estão no projeto. Com isso, o *SonarQube* calcula a classificação de manutenibilidade com base no índice de débito técnico. (SonarQube, no date).

Normalmente, quanto mais smells de código um projeto tem, mais difícil é ser um projeto sustentável. Eles não impedirão necessariamente a execução do código, pois não são erros de código, mas a longo prazo, vão transformar o código em algo instável e difícil de manter devido a toda a lógica acumulada. Por exemplo, a complexidade cognitiva (Figura 43), que representa a compreensibilidade do código, tem um máximo permitido. Se um método apresentar complexidade maior que quinze, aparecerá como um smell de código.

A complexidade ciclomática (Figura 43) é outra métrica que também pode ser coberta pelo *SonarQube*. Esta analisa o número de condições que existe numa função e a complexidade mínima para esta é de pelo menos um (SonarQube, no date).

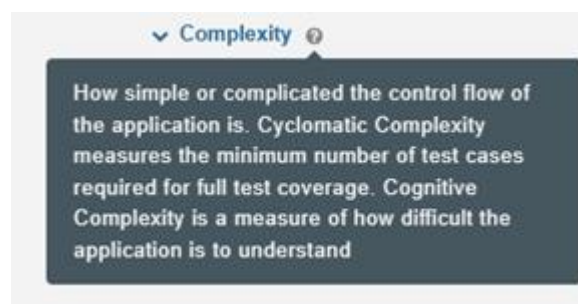


Figura 43 - Definição de Complexidade do *SonarQube*

Google Lighthouse

O *Google Lighthouse* é uma ferramenta que analisa a qualidade das páginas da web, auditando métricas como acessibilidade, desempenho, *SEO* e aplicações da web progressivos.

A pontuação de desempenho fornecida por esta ferramenta é baseada numa média ponderada, o que significa que algumas métricas têm mais peso na pontuação geral do que outras. Por exemplo, na versão mais recente 10, *LCP (Largest Contentful Paint)* pesa mais do que *FCP (First Contentful Paint)*. Esse peso é atribuído com base em pesquisas feitas pela equipa do *Lighthouse*, reunindo feedback sobre o que está a impactar a experiência do utilizador (Chrome, 2023)

GitActions

Deployability é a “Confiabilidade e facilidade de implantação de um micro *frontend* num ambiente” (Mezzalira, 2021).

Então, é um pouco difícil estabelecer um teste para isso. Portanto, considerando que a arquitetura de micro *frontend* terá implantações com uma escala muito menor, os riscos serão reduzidos. Assim sendo, é uma vantagem para esta arquitetura, no entanto, apesar do esforço de implantar um micro *frontend* parecer menor, o tempo de construção e implantação pode ser analisado observando a duração da execução do pipeline para produção e a duração da etapa de construção dentro dele. Assim, usou-se *GitHub Actions* para automatizar as *builds* e implantações (*GitHub Actions*, no date).

8.3 Resultados das Métricas

Os resultados das métricas presentes no plano GQM (Figura 42) são apresentadas nesta secção. Cada métrica apresenta uma análise, tendo em conta os seus valores obtidos nas duas abordagens arquitetónicas. No final de cada subsecção, estas abordagens são comparadas para dar uma resposta ao objetivo definido na Figura 42.

8.3.1 Manutenibilidade

As métricas de *M1* a *M3* (Figura 42), inclusive, foram obtidas com a análise realizada pelo *SonarQube*.

A classificação da manutenibilidade (*maintainability rating*) está relacionada com o índice de débito técnico (Tabela 8).

Tabela 8 - Escala de classificação da manutenibilidade

Fonte: (SonarQube, no date)

Débito técnico	Classificação da Manutenibilidade
<=5%	A
6% - 10%	B
11% - 20%	C
21% - 50%	D
>50%	E

A complexidade cognitiva (*cognitive complexity*) é uma medida que procura quantificar a dificuldade de compreender o código analisado. O *SonarQube* define o valor 15 como o valor que não deve ser ultrapassado por cada função presente no código. Os valores apresentados na Tabela 9 são a soma da complexidade das funções constituintes daquele projeto. Sendo que quanto maior esse valor, maior a sua complexidade cognitiva.

Não existe, também, uma escala específica para a complexidade ciclomática (*cyclomatic complexity*), que procura medir o número mínimo necessário de testes para uma cobertura de código completa. Contudo, por defeito, o *SonarQube* deteta como *smell* as funções que têm complexidade superior a dez. Na Tabela 9 encontra-se o valor total resultante da soma da complexidade ciclomática das funções pertencentes àquele projeto. Quanto maior o valor, maior o número de testes necessários para cobrir todo o código.

Tabela 9 - Análise do *SonarQube*

	Maintainability rating	Cognitive Complexity	Cyclomatic Complexity
<i>Frontend Monolítico</i>	A	23	80
<i>InvoiceGeneral</i>	A	17	69
<i>Previewer</i>	A	4	44
<i>Seoul</i>	A	3	28
<i>Tokyo</i>	A	3	28
<i>Shell</i>	A	1	32

Ao analisar os resultados consegue-se entender que a classificação de manutenibilidade calculada pelo *SonarQube* resulta num A para qualquer projeto.

O débito técnico não ultrapassa os 5%, portanto os projetos mantêm-se na nota A, o que se deve muito ao facto de serem de pequenas dimensões.

É necessário analisar estes valores do ponto de vista das equipas. Estas são responsáveis por micro *frontends* específicos. Observando a Tabela 9, as complexidades ciclomática e cognitiva de cada micro *frontend* são inferiores às apresentadas no *frontend monolítico*.

À medida que a complexidade ciclomática aumenta, cresce a necessidade de realizar um maior número de testes para abranger todo o código. Da mesma forma, à medida que a complexidade cognitiva aumenta, torna-se mais desafiador para os programadores compreender o código. Portanto, verifica-se a dificuldade dos programadores ao desenvolver no *frontend monolítico*.

8.3.2 Gestão de Dependências e Versões

Esta métrica serve, nomeadamente, para verificar qual abordagem proporciona opções de gerir versões de dependências, de modo que, no caso de alguma atualização necessária de versões, as equipas possam fazê-la de forma gradual e independente. Esta métrica surge da necessidade de no *frontend monolítico* ter um único programador responsável por atualizar todo o código quando é preciso atualizar uma dependência. Portanto, pretende-se verificar que alternativas existem para responder a essa necessidade nas duas abordagens.

Relativamente, às versões de dependências no *frontend monolítico* não é possível ter duas versões diferentes do mesmo *package*. Isto impossibilita os desenvolvedores de atualizarem só a parte de código da sua responsabilidade de acordo com uma nova versão do *package*.

Para essa arquitetura, a única forma possível de gerir as dependências é no *package.json* de uma forma bastante limitada. Alguns exemplos são apresentados na Tabela 10.

Tabela 10 - Exemplo de gestão de versões

^1.2.3	É instalado qualquer versão que seja compatível com 1.X.X.(<i>patch</i> e <i>minor versions</i>)
~1.2.3	É instalado qualquer versão que seja compatível com 1.2.X (<i>minor version</i>)
>=1.1.0 <3.0.0	É instalado qualquer versão que respeite o intervalo

No *frontend monolítico*, o *npm* instala uma só versão para o funcionamento do projeto com determinada dependência.

Nos *micro frontends* existe a possibilidade de carregar dependências em *runtime*. E existem várias formas possíveis para o fazer e permitir versões diferentes do mesmo *package* coexistirem e satisfazerem as necessidades dos *micro frontends*.

Algumas possibilidades são:

- Por defeito:
 - É carregada a versão compatível maior.
 - Quando há incompatibilidade, são usadas diferentes versões para os diferentes *micro frontends* (*fallback module*).
- Partilha de dependências do *package.json* configurando o *webpack.config.js*:
 - Método *shareAll()* partilha todas as dependências do *package.json* (Figura 44).

```
shared: {  
  ...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),  
},
```

Figura 44 - Método *shareAll*

- Método *share()* permite o uso de 'auto' na palavra-chave *requiredVersion*, o que remete para a versão daquela dependência no *package.json* (Figura 45).

```
shared: share({  
  "@angular/core": { singleton: true, strictVersion: true, requiredVersion: 'auto' },  
})
```

Figura 45 - Método *share()*

- Com o *Dynamic Module Federation*, a *Shell* apenas conhece o *micro frontend* em tempo de execução, portanto no caso de diferentes versões a *Shell* carrega a versão da dependência que tem definida para si, e se esta for menor que a usada no outro *micro frontend* (carregado dinamicamente), será carregada nova versão, por não encontrar compatibilidade com as existentes (*fallback module*).
Se for ao contrário, quando a *Shell* tem versão superior e compatível à do *micro frontend*, não há necessidade de carregar outra versão.
- Definindo o *singleton*, que faz com que aquela dependência só seja carregada uma única vez.
 - *singleton: true*, apenas carrega a maior versão que conhece na fase inicial.

- Um aviso pode surgir na consola se houver incompatibilidade dos *micro frontends* com versão carregada.
- Essas incompatibilidades podem quebrar a aplicação, e é possível configurar no *webpack* para o aviso se tornar num erro na consola (*fail fast*). Basta definir a dependência com *strictVersion: true*.
 - Removendo o *singleton*, significa que essa dependência não será compartilhada entre os restantes *micro frontends*.
- Aceitar um intervalo de versões no *requiredVersion* (Steyer, 2022b).

Verifica-se uma versatilidade enorme nas possibilidades disponibilizadas pelo *Module Federation*, sendo possível ter versões diferentes de dependências específicas em *runtime*, para cada *micro frontend*.

Além disso, mostra-se mais vantajoso para que, gradualmente, as equipas atualizem uma versão de uma determinada dependência. Contudo, neste caso é necessário realizar mais testes para garantir que as diferentes versões da mesma dependência funcionem e não causem comportamentos inesperados.

8.3.3 Performance

Os testes feitos para a performance foram realizados em ambiente de produção com as opções de *clear cache* e *simulated throttling* ativados.

Os relatórios da Figura 46 e Figura 47 resultaram da simulação da chegada inicial à página de */posting*.

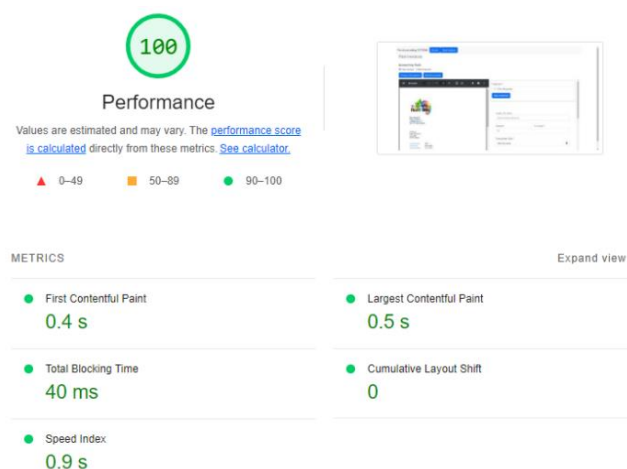


Figura 46 - Resultados do *Lighthouse*: *Frontend* Monolítico

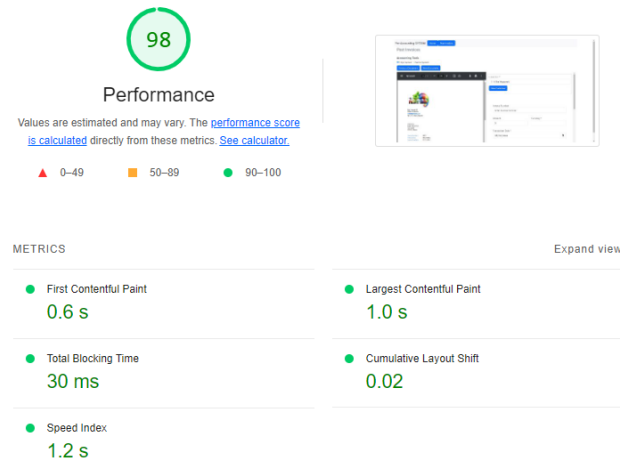


Figura 47 - Resultado do *Lighthouse*: *Micro Frontends*

Tabela 11 – Métricas de Performance

	FCP	LCP	TBT
<i>Frontend</i> Monolítico	0.4s	0.5s	40ms
<i>Micro frontends</i>	0.6s	1.0s	30ms

Verifica-se na Tabela 11 que as métricas *FCP* e *LCP* apresentam valores melhores no *frontend* monolítico e que o *TBT* - que mede o tempo de bloqueio entre FCP e tempo para a página ficar completamente interativa – é ligeiramente melhor nos *micro frontend*.

8.3.4 Deployability

Para analisar a facilidade de implantação recorreu-se ao tempo de *build* e ao tempo de implantação de cada um dos projetos. É possível obter esses tempos a partir do *GitActions*.

Tabela 12 - Duração da *build* e *deploy* de uma execução de pipeline

	Building Time	Deploy Time	Total
<i>Frontend Monolítico</i>	1m 48s	13s	3m 45s
<i>InvoiceGeneral</i>	2m 49s	12s	4m 20s
Previewer	2m 35s	12s	4m 1s
<i>Seoul</i>	2m 10s	7s	3m 41s
<i>Tokyo</i>	2m 14s	9s	3m 50s
<i>Shell</i>	2m 0s	8s	3m 26s

Como se pode verificar a diferenças de tempo não são muito diferenciais, contudo verifica-se que o tempo de *build* é menor no *frontend* monolítico, e o tempo de implantação é menor nos micro *frontends*.

Duração da *Build*

Assim sendo, analisou-se a execução da pipeline do *frontend* monolítico e a de um dos micro *frontends* (*InvoiceGeneral*). Obteve-se a amostra de 10 execuções para cada.

Primeiramente, verificou-se se as amostras seguiam uma distribuição normal com o teste de *Shapiro-Wilk*. O *p-value* resultante foi inferior a 0,05, o que significa que as amostras não seguiam uma distribuição normal, sendo necessário em vez de o *t-test*, realizar um teste não paramétrico. O escolhido foi o *Wilcoxon rank-sum*.

Sendo, μ_1 a média dos tempos de *build* do *frontend* monolítico e μ_2 a média dos tempos de *build* do micro *frontend* considerado, temos:

$$H_0: \mu_1 = \mu_2$$

$$H_1: \mu_1 < \mu_2$$

O *p-value* resultante foi inferior a 5% (*p-value* = 0.00009032) que torna possível rejeitar a hipótese nula (H_0). Portanto, a opção melhor em termos de *build* é a do *frontend* monolítico.

Duração do Deploy

Para os tempos de implantação, como os valores seguiam uma distribuição normal segundo o teste de *Shapiro-Wilk*, foi realizado o *t-test*.

Sendo, μ_1 a média dos tempos de implantação do *frontend* monolítico e μ_2 a média dos tempos de implantação do micro *frontend* considerado, temos:

$$H_0: \mu_1 = \mu_2$$

$$H_1: \mu_1 < \mu_2$$

Neste caso, o *p-value* é de 47,57% (superior a 5%), portanto, a hipótese nula (H_0) não pode ser rejeitada, sendo a média do *frontend* monolítico inferior à do micro *frontend*, mas não é pequena o suficiente para ser considerada significativa em termos estatísticos.

Assim, em termo de implantação, usar uma ou outra abordagem trará praticamente os mesmos resultados.

Implantações independentes

Relativamente ao isolamento da implantação (*deployment isolation*), os micro *frontends* têm vantagem sobre o *frontend* monolítico.

Nos micro *frontends*, o código está separado em projetos de dimensões mais pequenas. O desenvolvimento, os testes e a implantação de cada um é feito e assegurado por uma só equipa. Isto permite à equipa realizar facilmente entregas diárias (*daily releases*) sem condicionar ou atrasar outras equipas.

Por outro lado, o *frontend* monolítico é um projeto único onde as equipas têm de cooperar e comunicar para realizar uma entrega ao cliente. As diferentes equipas trabalham na mesma base de código que apenas tem uma pipeline para *build* e implantação do projeto. Entregas ao cliente são feitas contendo um enorme número de funcionalidades desenvolvidas pelas várias equipas, visto que entregas diárias não se adequam numa base de código partilhada.

Portanto, neste caso, existe uma pipeline para o *frontend* monolítico. Mas, há uma pipeline por cada micro *frontend*, que totaliza cinco pipelines que permitem a implantação isolada.

Assim sendo, o isolamento proporcionado pelos micro *frontends* torna-se uma vantagem que capacita as equipas de realizarem integrações e entregas contínuas (*CI/CD*).

8.4 Resumo dos Resultados

Considerando todas as métricas analisadas, a Tabela 13 mostra o resumo dos resultados.

Tabela 13 - *Frontend* monolítico vs. Micro *Frontends*

Métrica	<i>Frontend</i> Monolítico	Micro <i>Frontends</i>
M1 - <i>Maintainability rating</i>	X	X
M2 - <i>Cognitive complexity</i>		X
M3 - <i>Cyclomatic Complexity</i>		X
M4 - Gestão de dependências e versões		X
M5 - <i>FCP</i>	X	
M6 - <i>LCP</i>	X	
M7 - <i>TBT</i>		X
M8 - Duração da <i>Build</i>	X	
M9 - Duração da Implantação	X	X
M10 - Implantações Independentes		X

A separação em micro *frontends* apresenta melhores resultados na maioria das métricas ponderadas. Além disso, providenciará outros benefícios que não estão considerados diretamente nessas métricas, como é o caso da independência e autonomia de cada equipa. Com esta abordagem, como a solução estará dividida em projetos de dimensões mais pequenas atribuídos a uma só equipa, tornar-se-á mais fácil a realização de *releases* diárias por parte desta.

Esse benefício pode, por si só, solucionar parte dos desafios enfrentados pela empresa e contribuir para atingir o objetivo por ela estabelecido: capacitar as equipas para realizarem *releases* diárias, de modo a caminhar para uma abordagem de integrações e entregas contínuas.

No POC é difícil medir diretamente a autonomia e independência das equipas, porém algumas das métricas analisadas apresentam formas indiretas de entender como a arquitetura de *micro frontends* pode responder positivamente nesses dois aspetos. As métricas M4 e M10 (Figura 42) procuram mostrar que ao permitir as equipas gerirem as suas dependências e a implantação isolada do seu micro *frontend*, se pode providenciar maior autonomia das equipas, dando-lhes uma maior responsabilidade, mas também mais liberdade para realizar a gestão que lhes pertence.

Conclui-se que na maioria dos critérios considerados a arquitetura de micro *frontends* é vantajosa comparativamente ao *frontend* monolítico.

9 Conclusões

O presente capítulo contém as conclusões obtidas com o projeto, incluindo os objetivos alcançados, dificuldades e trabalho futuro e uma apreciação final.

9.1 Objetivos Alcançados

O objetivo consistia em explorar uma solução baseada em micro *frontends* que trouxesse benefícios, sem penalizar o desempenho comparativamente ao projeto do *frontend* monolítico. Alguns dos benefícios procurados eram: uma maior autonomia das equipas, facilidade de build e *deploy* da solução e manutenibilidade.

Portanto, foi construído um *POC* usando *Module Federation* para desenvolvimento dos micro *frontends* que procurou responder a todos os requisitos funcionais e não-funcionais. Foi realizada uma análise com base nos fatores e atributos de qualidade que se procurava ter em consideração nesta arquitetura, tendo sempre como comparação os resultados obtidos com o *frontend* monolítico.

A pesquisa realizada e contida no presente documento servirá para cobrir a necessidade de adquirir conhecimento na empresa em relação ao conceito de micro *frontends*. A investigação do tipo de composição e tecnologias usadas foi direcionada para o caso em estudo, considerando a estrutura das equipas em questão e a distribuição das funcionalidades por estas. Portanto, ouve uma investigação mais aprofundada pela composição do lado do cliente e tecnologias usadas para a mesma.

9.2 Dificuldades e Trabalho Futuro

Durante o desenvolvimento deste projeto, muitas adversidades e desafios foram encontrados. A existência de poucos artigos científicos fez com que a pesquisa se cingisse a livros e *blogposts*, muitas vezes escritos pelos mesmos autores.

A documentação específica sobre o estabelecimento da comunicação entre os micro *frontends*, bem como a implantação e outros aspetos relacionados à implementação, frequentemente apresentavam explicações superficiais. Isso resultou num processo demorado para a execução de determinadas tarefas. Durante o desenvolvimento do projeto, foi necessário recorrer a *issues* encontrados no *GitHub*, que forneciam soluções alternativas e temporárias para depurar alguns erros gerados pelo *webpack*. No entanto, é compreensível que as tecnologias utilizadas sejam relativamente recentes, o que implica que muita documentação ainda está em fase de desenvolvimento. Um exemplo disso, é o livro "Enterprise Angular: Micro Frontends and Moduliths with Angular" de Manfred Steyer.

Por questões empresariais não foi possível usar o contexto real. Foi criado um contexto o mais próximo possível do real para que os resultados obtidos pudessem ser aplicados na própria Basecone. Assim sendo, isto foi uma limitação que permitiu de certa forma uma obtenção mais controlada de resultados, contudo alguns conceitos de contabilidade tiveram de ser simplificados para efeitos da prova de conceito. Além disso, algumas métricas obtidas tiveram resultados similares para ambas as abordagens devido ao tamanho reduzido do projeto.

E para trabalho futuro, espera-se a realização de partilha de informação para com todas as equipas integrantes da empresa. Primeiramente, através de uma *Technology Session* para apresentar o conceito mais geral, seguido de reuniões, mais específicas e focadas em pormenores do assunto.

No início do próximo ano, será delineada a estratégia para implementar a arquitetura de micro *frontends*, aproveitando os planos da empresa mencionados na subsecção 3.3.2. A implementação usando *Module Federation* mostra-se muito conveniente, contudo a empresa vai manter em consideração a possibilidade de não depender do *webpack*, com o uso de *Native Federation* (se este for lançado como versão oficial).

9.3 Apreciação Final

O desenvolvimento deste documento e do projeto serviu para clarificar o conceito de micro *frontends*. O projeto foi uma maneira de aprofundar conhecimentos nesse âmbito e de perceber certos desafios que não se compreendem apenas com a leitura da documentação disponível.

O projeto desenvolvido permitiu expandir o meu conhecimento na área, contribuindo para a minha realização profissional. A Basecone já demonstrou a sua satisfação relativamente ao que foi estudado e analisado, pretendendo incluir no planeamento para o próximo ano a implementação de micro *frontends*.

Espera-se, assim, que esta pesquisa e trabalho tenham valor não só para a empresa em questão, mas também para outras que tenham a intenção de envergar por esta opção arquitetural.

Referências

AccountingTools (2022). Available at: <https://www.accountingtools.com/articles/business-transaction> (Accessed: 18 February 2023).

Agorapulse (no date). Available at: <https://www.agorapulse.com/> (Accessed: 5 February 2023).

'ANGULARarchitects Team' (no date) *ANGULARarchitects*. Available at: <https://www.angulararchitects.io/en/team/> (Accessed: 12 September 2023).

AWS (no date). Available at: <https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html> (Accessed: 13 September 2023).

Basecone (no date a). Available at: <https://www.wolterskluwer.com/en/solutions/basecone> (Accessed: 6 June 2023).

Basecone (no date b). Available at: <https://www.wolterskluwer.com/nl-solutions/basecone/products/basecone-web-application> (Accessed: 13 September 2023).

Bragg, S. (2022) *Accounting system definition*, *AccountingTools*. Available at: <https://www.accountingtools.com/articles/accounting-system> (Accessed: 18 February 2023).

Business Wire (2020). Available at: <https://www.businesswire.com/news/home/20200716005101/en/O%E2%80%99Reilly%E2%80%99s-Microservices-Adoption-in-2020-Report-Finds-that-92-of-Organizations-are-Experiencing-Success-with-Microservices> (Accessed: 23 January 2023).

Dąbrowska, A. (2020) *Frontend technologies in 2021*. Available at: <https://tsh.io/blog/frontend-technologies-in-2020-state-of-frontend-report/> (Accessed: 27 January 2023).

Dhaduk, H. (2021) 'Top Frontend Trends Organizations are Embracing in 2023', *Simform - Product Engineering Company*, 14 June. Available at: <https://www.simform.com/blog/top-frontend-trends/> (Accessed: 27 January 2023).

ESM Node.js (no date). Available at: <https://nodejs.org/api/esm.html> (Accessed: 12 September 2023).

Geers, M. (no date) *Micro Frontends - extending the microservice idea to frontend development*, *Micro Frontends*. Available at: <https://micro-frontends.org/> (Accessed: 5 February 2023).

GitHub Actions (no date) *GitHub Docs*. Available at: <https://ghdocs-prod.azurewebsites.net/en/actions> (Accessed: 25 February 2023).

GoRetro (no date). Available at: <https://www.goretro.ai/post/feature-teams-vs-component-teams> (Accessed: 12 September 2023).

Huang, F. (2023) 'Mooa'. Available at: <https://github.com/phodal/mooa> (Accessed: 12 September 2023).

importmap / Mozilla (2023). Available at: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script/type/importmap> (Accessed: 12 September 2023).

Koen, P.A. *et al.* (2002) 'Effective Methods, Tools, and Techniques', *The PDMA ToolBook for New Product Development* [Preprint].

Kotte, G.N. (no date) *Microservice Websites*. Available at: <https://gustafnk.github.io/microservice-websites/> (Accessed: 5 February 2023).

Mezzalira, L. (2019) 'Adopting a Micro-frontends architecture', 8 April. Available at: <https://lucamezzalira.com/2019/04/08/adopting-a-micro-frontends-architecture/> (Accessed: 5 February 2023).

Mezzalira, L. (2020a) *Lessons from DAZN: Scaling Your Project with Micro-Frontends*, InfoQ. Available at: <https://www.infoq.com/presentations/dazn-microfrontend/> (Accessed: 5 February 2023).

Mezzalira, L. (2020b) *Micro-frontends in context – Increment: Frontend*. Available at: <https://increment.com/frontend/micro-frontends-in-context/> (Accessed: 5 February 2023).

Mezzalira, L. (2021) *Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers*. 1st Edition.

Mezzalira, L. (2022) *The Micro-Frontends future | Better Programming*. Available at: <https://betterprogramming.pub/the-future-of-micro-frontends-2f527f97d506> (Accessed: 5 February 2023).

Mezzalira, L. (2023) *Microfrontends Anti-Patterns: Seven Years in the Trenches*, InfoQ. Available at: <https://www.infoq.com/presentations/microfrontend-antipattern/> (Accessed: 4 February 2023).

Micro frontends anti patterns (2022). Available at: <https://www.youtube.com/watch?v=PF1y8vFTBgQ> (Accessed: 4 February 2023).

native-federation (2023) *npm*. Available at: <https://www.npmjs.com/package/@angular-architects/native-federation> (Accessed: 12 September 2023).

Nicola, S. (no date) 'Análise de Valor'.

Peltonen, S., Mezzalira, L. and Taibi, D. (2021) 'Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review', *Information and Software Technology*, 136, p. 106571. Available at: <https://doi.org/10.1016/j.infsof.2021.106571>.

Pérez, J.M. (2019) *Building Spotify's New Web Player, Spotify Engineering*. Available at: <https://engineering.atspotify.com/2019/03/building-spotifys-new-web-player/> (Accessed: 5 February 2023).

Prajwal, Y.R., Parekh, J.V. and Shettar, D.R. (2021) 'A Brief Review of Micro-frontends', 02(08).

Rappl, F. (2021) 'The Art of Micro Frontends'.

Richardson, C. (2021) *Minimizing Design Time Coupling in a Microservice Architecture*. Available at: <https://www.infoq.com/presentations/microservices-design-time-coupling/> (Accessed: 5 February 2023).

Saaty, T.L. (2008) 'Decision making with the analytic hierarchy process', *International Journal of Services Sciences*, 1(1), p. 83. Available at: <https://doi.org/10.1504/IJSSCI.2008.017590>.

Silva, N. (2023) 'NicoleNicoNi21/TMDEI_microfrontends'. Available at: https://github.com/NicoleNicoNi21/TMDEI_microfrontends (Accessed: 14 October 2023).

single-spa (no date). Available at: <https://single-spa.js.org/docs/getting-started-overview/> (Accessed: 13 February 2023).

Solingen, R. van, Berghout, E. and Berghout, E.W. (1999) *The goal/question/metric method: a practical guide for quality improvement of software development*. London: The McGraw-Hill Companies.

SonarQube (no date). Available at: <https://docs.sonarqube.org/latest/user-guide/metric-definitions/> (Accessed: 12 February 2023).

State of Microservices 2020 Report (2020). Available at: <https://tsh.io/state-of-microservices/> (Accessed: 27 January 2023).

Stenberg, J. (2018) *Experiences Using Micro Frontends at IKEA*, *InfoQ*. Available at: <https://www.infoq.com/news/2018/08/experiences-micro-frontends/> (Accessed: 5 February 2023).

Stenberg, J. (2020) *Decision Strategies for a Micro Frontends Architecture*. Available at: <https://www.infoq.com/news/2020/01/strategies-micro-frontends/> (Accessed: 5 February 2023).

Steyer, M. (2020) 'Dynamic Module Federation with Angular', *ANGULARarchitects*, 26 June. Available at: <https://www.angulararchitects.io/en/aktuelles/dynamic-module-federation-with-angular/> (Accessed: 12 September 2023).

Steyer, M. (2022a) 'Announcing Native Federation 1.0', *ANGULARarchitects*, 9 October. Available at: <https://www.angulararchitects.io/blog/announcing-native-federation-1-0/> (Accessed: 12 September 2023).

Steyer, M. (2022b) 'Enterprise Angular: Micro Frontends and Moduliths with Angular'.

Steyer, M. (2022c) 'Import Maps - The Next Evolution Step for Micro Frontends?', *ANGULARarchitects*, 26 December. Available at: <https://www.angulararchitects.io/blog/import-maps-the-next-evolution-step-for-micro-frontends-article/> (Accessed: 12 September 2023).

Swoyer, M.L., Steve (2020) *Microservices Adoption in 2020*, *O'Reilly Media*. Available at: <https://www.oreilly.com/radar/microservices-adoption-in-2020/> (Accessed: 23 January 2023).

Thoughtworks (no date) *Thoughtworks*. Available at: <https://www.thoughtworks.com/radar/techniques/micro-frontends> (Accessed: 5 February 2023).

Wolters Kluwer (no date a). Available at: <https://www.wolterskluwer.com/en/about-us> (Accessed: 13 September 2023).

Wolters Kluwer (no date b). Available at: <https://www.wolterskluwer.com/en-gb/expert-insights/wolters-kluwer-acquires-basecone> (Accessed: 13 September 2023).

Yang, C., Liu, C. and Su, Z. (2019) 'Research and Application of Micro Frontends', *IOP Conference Series: Materials Science and Engineering*, 490, p. 062082. Available at: <https://doi.org/10.1088/1757-899X/490/6/062082>.

Zaikin, V. (2023) 'You Might Not Need Module Federation: Orchestrate your Microfrontends at Runtime with Import Maps', *Mercedes-Benz.io*, 5 January. Available at: <https://www.mercedes-benz.io/2023/01/05/you-might-not-need-module-federation-orchestrate-your-microfrontends-at-runtime-with-import-maps/> (Accessed: 12 September 2023).

Apêndice A

Mecanismos de Transclusão

É a inclusão de parte ou a totalidade de um documento externo em um ou mais outros documentos por meio de uma referência de hipertexto. Isso resulta em um único documento presente para o usuário final. Existem três técnicas de transclusão:

- SSI é usado para incluir múltiplos fragmentos numa página através de um servidor. No passado, o SSI era usado para separar a parte dinâmica da página do conteúdo estático. Esse tipo de transclusão contém algumas diretrizes que permitem incluir o conteúdo externo e diretrizes de controle que fornecem a adição de alguns atributos, parâmetros ou condições para executar uma ação específica na página. Portanto, antes da página chegar ao cliente, todas as rotas e diferentes fragmentos são reunidos com base nas diretrizes daquela página, só retornando a página para o cliente quando estiver totalmente carregada.
- O ESI é bastante semelhante ao SSI, mas é realizado no nível do CDN, resolvendo alguns problemas de escalabilidade que poderiam existir no anterior. No entanto, isso depende dos recursos disponíveis no CDN escolhido, o que pode causar limitações futuras. No entanto, pode ser uma ótima solução para micro *frontends* que não têm conteúdo dinâmico, como o site da IKEA, que é como um catálogo online. Portanto, algumas características que se pode obter desta transclusão são que ela faz uso de um *template* que substitui o *placeholder tag* pelo micro *frontend* correspondente, suporta o uso de atributos de pedidos HTTP que podem ser usados por declarações ESI, permitindo ter algumas declarações condicionais e também oferece a possibilidade de lidar com erros mostrando conteúdo alternativo ao usuário (Mezzalana, 2021; Rappl, 2021).
- O CSI usa a mesma lógica que o ESI para interpretar as *tags*. Ele usa pequenos excertos de JavaScript no navegador para carregar conteúdo de outros lugares no DOM atual. O CSI faz alguns pedidos AJAX para recursos do lado do servidor e inclui a resposta na página visualizada. Portanto, esta transclusão usa a capacidade do JavaScript de examinar o DOM em busca de *tags* para substituir e carregar pedaços de HTML no lado do cliente (Kotte, no date).