INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Conceção de arquitetura de Micro-Frontend

PEDRO MIGUEL COSTA SOUSA
outubro de 2023

POLITÉCNICO
DO PORTO

Conceção de arquitetura de Micro-Frontend

Pedro Sousa

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Computer Systems

Orientador: Emanuel Silva
Co-Orientador: Pedro Cunha

Júri:
Presidente:

Vogais:

Porto, 14 de outubro de 2023

Dedicatória

"I don’t stop when I’m tired, I stop when I’m done."David Goggins

Resumo

Recentemente, o mundo de desenvolvimento de *software* tem vindo a ser alvo de um crescente aumento de complexidade no que toca às aplicações desenvolvidas. O mercado das aplicações web tem vindo a crescer muito rapidamente tal como as exigências dos utilizadores por uma boa experiência de utilização.

Neste contexto, as equipas de desenvolvimento de *software* começaram a sentir dificuldades em manter e escalar o *software* que desenvolviam dadas as maiores exigências dos utilizadores finais. Habitualmente, as organizações arquitetavam as aplicações numa estrutura monolítica, pois numa primeira instância facilitava todos os processos de desenvolvimento e permitia entregar novas funcionalidades de forma rápida. Porém, com o passar do tempo o projeto tornava-se cada vez mais complexo e, por fim, a velocidade e produtividade de desenvolvimento acabava por cair de forma consistente.

Deste modo, começaram os movimentos de divisão do *software* das aplicações em vários módulos. Em primeiro lugar, começou-se por dividir a parte de interface de utilizador, pela parte de computação e manipulação de dados e deu-se a separação de *Frontend* e *Backend*. Numa segunda instância, o próprio *Backend* começou a ser dividido, dando a origem às arquiteturas orientadas aos Microsserviços. Por fim, mais recentemente começou-se a aplicar a mesma lógica à camada do *Frontend* e assim nasceram as arquiteturas de *Micro-Frontends*.

Estas divisões do *software* em diferentes módulos dá a oportunidade às equipas de separarem a complexidade de grandes aplicações em menores módulos com uma complexidade mais baixa, a até mesmo separar o desenvolvimento de cada módulo por diferentes equipas criando menor entropia dentro do processo de desenvolvimento da aplicação. Ainda assim, devido ao facto do conceito ser recente, a maioria das aplicações que beneficiariam desta arquitetura já foram desenvolvidas durante vários anos numa arquitetura monolítica e a sua migração integral para uma nova arquitetura não é exequível de uma só vez.

Para mitigar este problema, é possível a aplicação de um padrão de *software* chamado *Strangler Pattern* que permite a migração gradual de um sistema, módulo a módulo, fazendo com que a migração seja mais gradual e suave, facilitando a resolução de possíveis problemas e possibilitando fazer esta migração em aplicações *legacy* com maior complexidade.

Esta dissertação desenvolve a prova de conceito da migração de uma aplicação Frontend monolítica para uma arquitetura orientada a *Micro-Frontends*. É feito um estudo sobre as vantagens e desvantagens da migração, padrões comuns a serem usados e o planeamento da divisão do processo por etapas.

A prova de conceito é desenvolvida usando a *framework single-spa*. A validação da solução é feita usando testes de desempenho e aceitação.

Palavras-chave: Micro-Frontend, Migração, Escalabilidade, AngularJS, Domain-Driven-Design

Abstract

Recently, the world of software development has seen a growing increase in the complexity of the applications developed. The web application market has been growing very rapidly, as have user demands for a good user experience.

As a result, software development teams began to find it difficult to maintain and scale the software they were developing, given the greater demands of end users. Therefore, new software architectures and standards have been studied and presented to the community to solve these problems.

Organisations used to architect applications in a monolithic structure, as this initially facilitated all development processes and allowed new features to be delivered quickly. However, as time went by, the project became more and more complex and, in the end, development speed and productivity fell consistently.

This is how the division of the software of applications into various modules began. Firstly, the user interface part was split from the computing and data manipulation part and Frontend and Backend were separated. In a second instance, Backend itself began to be split up, giving rise to microservice-orientated architectures. Finally, more recently, the same logic began to be applied to the Frontend layer and thus the Micro-Frontends architectures were born.

These divisions of the software into different modules give teams the opportunity to separate the complexity of large applications into smaller modules with lower complexity, and even separate the development of each module by different teams, creating less entropy within the application development process. Even so, because the concept is recent, most of the applications that would benefit from this architecture have already been developed for several years in a monolithic architecture and their full migration to a new architecture is not feasible in one go.

To mitigate this problem, it is possible to apply a software pattern called Strangler Pattern that allows the gradual migration of a system, module by module, making the migration more gradual and smooth, facilitating the resolution of possible problems and making it possible to make this migration in legacy applications with greater complexity.

This dissertation develops a proof of concept for the migration of a monolithic Frontend application to a Micro-Frontends orientated architecture. A study is made of the advantages and disadvantages of migration, common patterns to be used and the planning of the division of the process into stages.

The proof of concept is developed using the framework single-spa. The solution is validated using performance and acceptance tests.

Agradecimentos

Esta tese é dedicada à minha família e amigos por todo o apoio e paciência que tiveram durante o meu percurso académico.

Dedico este trabalho em especial à minha tia Ana Maria, que teve um grande papel na minha educação, e à minha avó Maria Antónia. Tenho a certeza que neste momento ambas estão muito orgulhosas.

Aos meus colegas, Bruno Vilar, João Bernardo, Carlos Pinto, João Querido, Vasco Pinto, Marta Silva e Daniela Silva, que me acompanharam durante toda a licenciatura e mestrado e aos quais agradeço por todo o apoio, companheirismo e todos os momentos inesquecíveis que passamos juntos.

Ao meu amigo Henrique que me acompanhou durante este processo e que, indiretamente, mas eficazmente, não me deixou desistir. Ao longo deste tempo mostrou ser um craque dentro e fora de campo.

À minha amiga Beatriz que nunca me deixou deixar de acreditar em mim e sempre foi uma fonte de apoio independentemente de tudo.

Ao meu colega Ricardo Batista, por toda a ajuda e inspiração que me proporcionou que sem dúvida foi fulcral para a realização desta tese.

Quero expressar o meu profundo apreço a toda a proGrow, por todo o apoio e a ajuda a gerir delicadamente o equilíbrio entre a profissão e a vida académica. Um agradecimento especial ao Marco Tschan Carvalho, pela compreensão e apoio durante esta jornada. O compromisso para o desenvolvimento e o bem-estar do colaborador tornou esta jornada bem mais fácil.

Ao meu supervisor Pedro Cunha, a quem agradeço por toda a ajuda na revisão do documento, pela orientação durante o projeto e pelo encorajamento. A sua mentoria foi fundamental para a realização desta tese.

Ao meu orientador Emanuel Silva tanto na qualidade de orientador como de professor, que desempenhou o seu papel exemplarmente e a quem eu agradeço todo o tempo disponibilizado. O seu *feedback* construtivo elevou a qualidade desta tese.

A todos os professores que me acompanharam durante a minha jornada académica, um muito obrigado.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

SPA	Single-Page-Application
SOA	Service-Oriented-Architecture
SSR	Server-Side Rendering
DDD	Domain-Driven-Design
SEO	Search Engine Optimization
FFE	Fuzzy Frontend
NPD	New Product Development
NCD	New Concept Development
QFD	Quality Function Deployment
HOQ	House Of Quality
CI/CD	Continuous Integration/Continuous Deployment
SWOT	Strength Weakness Oportunities Threats

Glossário

Assets	Ficheiros como imagens, vídeos ou audios usados na plataforma.
Backend	Parte de uma aplicação software que opera fora do olhar do utilizador, responsável por gerir toda a lógica e persistência de dados..
Benchmark	Teste padrão usado para aferir o desempenho e eficiência de um *hardware* ou *software*.
Bottleneck	Ponto no sistema ou processo onde o fluxo de dados é restringido, causado uma eficiência reduzida.
Codebase	Conjunto de ficheiros do código fonte, *scripts* e *assets* que constituem um projeto *software* ou uma aplicação, incluindo toda a lógica necessária para o seu funcionamento.
Deployment	Processo de tornar uma aplicação *software* acessível e operação num ambiente específico.
Framework	Base estruturada de componentes de código que podem ser reutilizados que permitem desenvolver aplicações *software*.
Frontend	Parte do *software* orientada ao utilizador num sistema ou *website*, com a qual os utilizadores interagem diretamente.
Legacy	Sistemas de *software* desatualizados ou antigos, que por vezes usam tecnologias e arquiteturas obsoletas.
Plugin	Componente ou módulo de *software* que adiciona uma funcionalidade específica a um sistema já existente.
Query	Pedido ou comando feito a uma base de dados para receber ou manipular dados.
Rollback	Processo de reversão de um *update* de *software*.

Script	Pequeno conjunto de instruções escritas em linguagens como *python*, *Javascript* ou *Bash*, que automatizam tarefas ou executam funções específicas.
Stacks	Uma combinação de ferramentas de *software*, *frameworks*, linguagens de programação e componentes usados para compilar e correr *software*.

Capítulo 1

Introdução

Este capitulo tem o objetivo de introduzir o tema a ser tratado nesta dissertação assim como proporcionar um contexto adequado ao leitor.

1.1 Declaração de Integridade

Declaro que a presente dissertação de mestrado, é o resultado da minha própria pesquisa, realizada de forma ética e honesta. Toda a informação apresentada neste trabalho foi obtida por meios legítimos sem recurso à prática de plágio. Qualquer ideia ou informação baseada em outras fontes está devidamente citada e referenciada de acordo com as normas académicas estabelecidas pela instituição de ensino. Adotei os oito princípios do IEEE-CS/ACM sobre Ética em Engenharia de *Software* e Práticas Profissionais (*Code of Ethics | IEEE Computer Society* 2023).

1.2 Contexto

Atualmente, o desenvolvimento de aplicações *web* tem vindo a ser cada vez mais complexo, sendo necessário adicionar novas funcionalidades continuamente, para satisfazer as necessidades dos utilizadores e manter-se a par da concorrência (Zdravkova e Basnarkov 2022).

O desenvolvimento de aplicações *web*, adotando uma arquitetura monolítica, tem sido um padrão comum devido à necessidade de entregar projetos rapidamente, onde todo o código e *Ficheiros como imagens, vídeos ou audios usados na plataforma (Assets)* (ficheiros que dão suporte gráfico à aplicação) estão todos contidos numa única *Conjunto de ficheiros do código fonte, scripts e assets que constituem um projeto software ou uma aplicação, incluindo toda a lógica necessária para o seu funcionamento (Codebase)* (base de código escrito para uma aplicação). Nesta arquitetura, a aplicação é desenvolvida como uma única peça, fazendo com que todos os seus componentes estejam fortemente acoplados (Mezzalira 2021).

Com o tamanho e complexidade destas aplicações a crescer continuamente, fica cada vez mais difícil de manter, atualizar e escalar estas soluções. Arquiteturas monolíticas não são adequadas quando estamos a lidar com aplicações complexas, pois potencia uma baixa produtividade de desenvolvimento, baixa escalabilidade e fraca flexibilidade de *deployment*.

Um dos maiores desafios em projetos com este tipo de arquiteturas é a dificuldade em múltiplas equipas trabalharem no mesmo. Devido ao facto de todo o código estar centralizado, torna-se mais difícil que diferentes equipas trabalhem em diferentes partes da aplicação sem

se afetarem umas às outras. Isto origina atrasos, conflitos e problemas de integração(Geers 2020).

Outro grande desafio é a enorme dependência à tecnologia ou *Base estruturada de componentes de código que podem ser reutilizados que permitem desenvolver aplicações software (Framework)* usada. Projetos Monolíticos estão totalmente dependentes da tecnologia em que estão implementados, o que dificulta bastante o cenário de ser necessário fazer uma troca, pois irá afetar toda a aplicação.

Para resolver estes problemas, cada vez mais se tem vindo a observar a adoção de uma arquitetura de microsserviços para o *backend* das aplicações. A aplicação desta filosofia na área do *frontend* (Micro-Frontends) não é tão comum, porém tem ganho alguma popularidade, tendo já sido adotada por várias grandes empresas na área (Geers 2020).

Concluindo, arquiteturas monolíticas têm-se tornado num *Ponto no sistema ou processo onde o fluxo de dados é restringido, causado uma eficiência reduzida (Bottleneck)* para o desenvolvimento e escalabilidade de aplicações web. A arquitetura de *micro-frontends* é uma nova alternativa que tenta resolver estes problemas dividindo a aplicação em módulos mais pequenos, independentes e desacoplados. Adicionalmente, uma arquitetura de *micro-frontends* reduz a dificuldade de múltiplas equipas trabalharem no mesmo projeto, assim como a dependência deste a uma tecnologia ou *framework* específica.

1.3 Problema

A proGrow desenvolve e comercializa uma plataforma *web* que permite a qualquer empresa industrial digitalizar as suas operações, capturando e analisando dados operacionais em tempo real. Em termos de *Parte de uma aplicação software que opera fora do olhar do utilizador, responsável por gerir toda a lógica e persistência de dados. (Backend)*, a proGrow apresenta uma arquitetura orientada a microsserviços multi-tecnológica. Contudo, o mesmo já não se verifica no *Parte do software orientada ao utilizador num sistema ou website, com a qual os utilizadores interagem diretamente (Frontend)*, tendo sido desenvolvida uma solução monolítica em *AngularJs*. *AngularJs* é uma *framework* para o desenvolvimento de *Single Page Aplications* (SPA) que foi lançada pela *Google* em 2010. Com o lançamento de uma nova *framework Angular*, a *Google* anunciou que a partir de janeiro de 2022 o *AngularJs* deixaria de receber suporte oficial (*AngularJS: Miscellaneous: Version Support Status* 2023).

Com o avançar do tempo e com o desenvolvimento contínuo de *features*, a solução de *frontend* em *AngularJS* tem ficado cada vez mais complexa, tornando a migração para uma nova *framework* uma tarefa de elevada dificuldade. Por esta razão, a migração deixou de ser uma prioridade. Neste momento, com o aumentar da dívida tecnológica a aplicação deixou de conseguir acompanhar as soluções mais atuais, tanto em termos de *performance*, qualidade de desenvolvimento como de processo de *deployment*.

Para além disto, esta solução apresenta várias dificuldades à equipa de desenvolvimento, tanto pela falta de eficiência da própria *framework* no processo de desenvolvimento como também pelos problemas de integração de trabalho que uma arquitetura monolítica origina.

Reescrever a aplicação numa nova *framework* poderia ser uma solução, porém é normalmente apontado como um erro estratégico (Joel Spolsky 2000). Outra solução passaria por dividir a aplicação em vários pequenos módulos, migrando assim a solução de forma gradual e iterativa.

1.4 Objetivos

A arquitetura de *Micro-Frontends* propõe a decomposição da aplicação *web* em módulos menores, por página ou funcionalidade. Este projeto irá enquadrar-se na conceção de uma arquitetura de *Micro-Frontend* para a plataforma da proGrow.

Pretende-se com este projeto:

- Desenvolver um estudo entre as melhores opções para a migração da solução atual para uma arquitetura de *Micro-Frontends*, indicando as respetivas vantagens e desvantagens;
- Planear a migração da arquitetura em várias etapas;
- Implementar provas de conceito das novas arquiteturas da aplicação;
- Analisar a nova arquitetura em comparação com a atual em termos de escalabilidade e manutenibilidade.

1.5 Estrutura do Documento

Este documento está organizado em sete capítulos:

- Capítulo 1 inicia com a apresentação da declaração de integridade, seguindo-se da contextualização do assunto e a apresentação do problema a abordar e a definição dos objetivos a alcançar. Por fim, é apresentada a estrutura do documento.
- Capítulo 2 explora o estado de arte atual das arquiteturas orientadas a *Micro-Frontends*, discute as vantagens e desvantagens destas e examina padrões de software importantes para a migração de arquiteturas. O capitulo é finalizado com a apresentação da aplicação sobre a qual a migração vai ser efetuada.
- Capítulo 3 é conduzida uma análise de valor detalhada para o projeto em questão. Uma análise *SWOT* é realizada para avaliar os pontos fortes, fracos, oportunidade e ameaças do projeto e utiliza-se a ferramenta *House of Quality* para medir a qualidade do projeto.
- Capítulo 4 apresenta o estudo e planeamento da migração da arquitetura, descreve as etapas a serem executadas durante a migração e aborda estratégias de teste da solução, assim como possíveis desafios que possam surgir.
- Capítulo 5 centra-se na implementação prática da solução. São apresentados detalhes do desenvolvimento, destacando as decisões importantes tomadas ao longo desse processo.
- Capítulo 6 remete para a análise de resultados. Neste capitulo, são apresentados os métodos de testes que foram usados, bem como análise e interpretação dos resultados.
- Capítulo 7 é o capitulo final. São apresentados os objetivos alcançados, os desafios e limitações encontradas, e apontadas algumas direções para trabalho futuro. São também expostas algumas potenciais contribuições deste trabalho e, finalmente, é apresentada uma reflexão final do autor.

Capítulo 2

Estado de Arte

Neste capítulo serão abordados os atuais conhecimentos teóricos acerca de arquiteturas e Micro-Frontends como também de padrões de migração entre tecnologias em Frontend.

2.1 Aplicações Frontend

Nesta secção irá ser abordado o conceito de aplicações Frontend, o seu funcionamento e diferentes formas de implementação.

O desenvolvimento de aplicações Frontend remete para o design e implementação de interfaces gráficas para aplicações web e mobile. O conceito de Frontend começou a ser referido aquando do surgimento do modelo arquitetural Client-Server. Este modelo arquitetural consiste na divisão de responsabilidades da visualização e manipulação da informação, sendo o cliente o solicitante e responsável pela visualização da informação e o servidor o provedor da informação e responsável pela alteração da mesma.

No desenvolvimento web, esta arquitetura é muito usada na divisão entre o Backend e Frontend em que, o Backend assume todas as responsabilidades relativas à persistência, obtenção e manipulação de dados e o Frontend assume a responsabilidade de mostrar a informação ao utilizador final e servir de interface para alteração de dados. É possível ver uma representação desta arquitetura na Figura 2.1 (School of Computing Universiti Utara Malaysia Kedah, Malaysia e Oluwatosin 2014).

Nos inícios da Internet, os web sites existentes eram bastantes simples, apenas consistiam em páginas HTML muito básicas com interatividade muito limitada. Com o amadurecimento da Web, a procura por web sites mais dinâmicos e interativos aumentou, levando ao desenvolvimento de tecnologias como o *Javascript*, que permitiu que os programadores conseguissem tornar os web sites mais interativos.

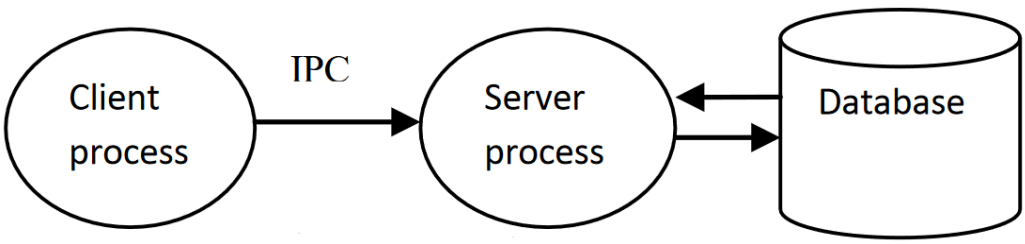

Figura 2.1: Modelo Client-Server (School of Computing Universiti Utara Malaysia Kedah, Malaysia e Oluwatosin 2014).

Com avançar dos tempos e o crescente aumento da complexidade dos *websites*, era necessário standardizar o desenvolvimento web. Com isto nasceram várias *frameworks* de aplicações Frontend como React, Angular e Vue, que oferecem várias ferramentas e componentes para construir interfaces gráficas para a Web. Com o crescimento da adoção dos smartphones, nasceu um novo desafio para os desenvolvedores Frontend de tornar as suas interfaces responsivas e orientadas a interfaces touch que levaram ao design responsivo.

2.1.1 Single-Page Applications (SPAs)

As *Single-Page Applications* (SPAs) têm-se tornado uma opção muito recorrente no que toca ao desenvolvimento web atual devido à sua interface apelativa (aqui é mais *user experience*) e performance melhorada. As SPAs consistem num único ou pequeno conjunto de ficheiros *Javascript* que encapsulam a aplicação por inteiro, descarregados à priori.

O uso de SPAs tem vários benefícios, como o facto de o utilizador descarregar o código todo apenas uma vez, quando acede a aplicação, e toda a lógica relativa à aplicação está presente já no cliente, evitando assim problemas de carregamentos recorrentes. Enquanto o utilizador navega pela página, a apenas faz os pedidos às API's para obter apenas a informação que necessita para a página em questão, diminuindo os tempos de carregamento e fazendo com que as transições entre páginas sejam mais suaves. Outra vantagem das SPAs é a capacidade de, através do uso de *cache* e *services workers*, oferecer uma melhor experiência a utilizadores com uma má conexão ou mesmo sem acesso à internet (Mezzalira 2021).

Porém, este tipo de aplicações apresenta algumas desvantagens. Uma delas é o facto do download da totalidade da aplicação ser feito no momento de carregamento da página e não apenas a página que o utilizador está a aceder. No caso de a aplicação não ter sido desenvolvida com esse ponto em consideração pode fazer com que este problema seja amplificado, piorando a experiência do utilizador. Para combater estes problemas de performance, é possível implementar uma estratégia de Lazy Loading.

O Lazy Loading é na prática o atrasar do carregamento de recursos ou objetos até que estes sejam necessários de modo a melhorar o desempenho e poupar recursos ao sistema. Como por exemplo, carregar os recursos de um website dinamicamente à medida que é feito o *scroll*, como é possível na Figura 2.2 por parte do utilizador. Com esta técnica é possível reduzir o tempo de carregamento inicial e a poupança de recursos tanto na largura de banda como no sistema.

Adicionalmente, as SPA's podem ter dificuldades em serem facilmente encontradas e compreendidas pelos motores de busca, o que pode impactar a sua visibilidade e ranking nos resultados das pesquisas. Este processo é conhecido como *search engine optimization* (SEO). SEO é uma prática de otimização de websites para melhorar a sua visibilidade e ranking nos resultados das pesquisas em motores de busca. As SPAs, como geram o seu conteúdo dinamicamente através do JavaScript, os motores de busca têm dificuldade em indexar o conteúdo (Jadhav, Sawant e Deshmukh 2015).

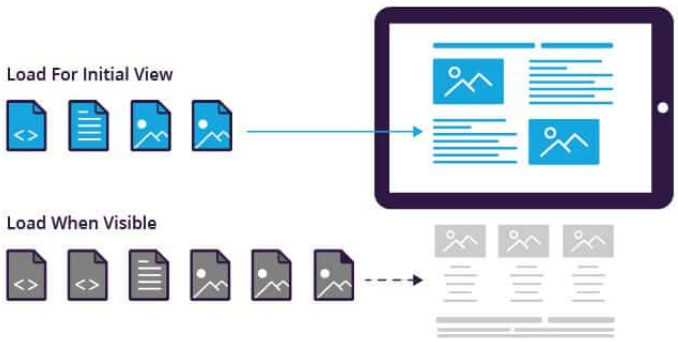

Figura 2.2: Mecanismo de Lazy Load no scroll (*What is Lazy Loading | Lazy vs. Eager Loading | Imperva* 2023).

Porém, existem algumas formas de melhorar o SEO das SPAs:

1. *Server-Side Rendering* (): Uma técnica onde o conteúdo da SPA é renderizado no servidor e enviado para o *browser* como um HTML renderizado, permitindo que os motores de busca indexem o conteúdo mais facilmente;
2. *Dynamic Rendering*: Esta técnica combina a renderização ao nível do cliente e do servidor onde o conteúdo é renderizado no servidor para os motores de busca e no cliente para os utilizadores.

Existem já algumas frameworks que implementam este tipo de técnicas, tais como a Next.js e Nuxt.js. Assim sendo, mesmo existindo esta desvantagem, já existem várias técnicas para a mitigar e otimizar o SEO da aplicação a desenvolver.

2.1.2 Deployment

O *deployment* de uma aplicação Frontend é o processo de tornar acessível a aplicação aos utilizadores através da Internet. Este processo pode ter vários estágios como a compilação, testagem e hospedagem da aplicação.

Compilação

O processo de compilação é um passo importante passo no desenvolvimento de uma aplicação recorrendo a uma *framework* de SPA. O processo de compilação transforma código escrito em *Typescript*, HTML e ainda possivelmente alguns pré-processadores de CSS como *Sass*, num formato que pode ser executado pelo *browser* do utilizador final.

Typescript é um superconjunto sintático estrito de *JavaScript*, que adiciona tipagem estática opcional à linguagem. Quando o *Typescript* é usado numa aplicação Frontend, este código tem que ser traduzido para *JavaScript* antes de ser executado pelo browser. O código de *JavaScript* traduzido inclui todas as funcionalidades definidas no código TypeScript sendo também compatível com um vasto leque de *browsers*.

Sass é um exemplo de pré-processador de CSS que oferece algumas funcionalidades que não existem em no CSS puro, tais como o uso de variáveis, *mixin* e herança, tornando o processo de desenvolvimento mais fácil, rápido e escalável.

O processo de compilação tem como o objetivo a geração de um *bundle*. O *bundle* é um único ficheiro que contém todo o código e recursos necessários para a executar a aplicação. Os *bundles* são normalmente gerados por ferramentas como o *Webpack* ou o *Parcel*, que juntam todas as dependências e código num único ficheiro.

Após a geração do *bundle*, o código é transformado para um formato em que é possível ser executado por todos os browsers. Tipicamente envolve transformar *javascript* moderno como ES6 ou ES7 noutro formato para ser possível ser executado em browsers mais antigos sem perder funcionalidades. Opcionalmente também pode haver um processo de *Minifying* em que é reduzido o tamanho final dos ficheiros removendo espaços em branco e comentários. Outro processo é o *Obfuscating* onde os nomes dos métodos e variáveis são modificados para serem mais difíceis de ler por um humano.

Testes

A execução de Testes automáticos é um passo importante neste processo. Este passo visa ajudar a identificar erros e problemas que estejam na aplicação antes que os utilizadores finais usem a aplicação. Com o crescimento das aplicações e com o aumento da sua complexidade, torna-se incomportável não ter um processo de testes automatizados nas aplicações. Para aplicações Frontend existem ferramentas de testagem automática como *Jest* e Mocha que podem ser usadas para verificar o comportamento expectável.

Após a compilação e testagem da aplicação, é necessário que a aplicação seja hospedada num servidor para ficar acessível aos utilizadores finais. Existem várias formas de hospedar uma aplicação Frontend, incluindo serviços de hospedagem *cloud* como os *Amazon Web Services*, *Google Cloud Platform* e *Microsoft Azure*.

Estes serviços oferecem muitas vantagens como a escalabilidade, confiabilidade e preço.

2.2 Estilos Arquiteturais

Esta secção apresenta o resultado do estudo sobre alguns tipos arquiteturais de software, são descritas as suas particularidades bem como as vantagens e desvantagens da sua aplicação.

2.2.1 Arquitetura Orientada a Microsserviços

Uma arquitetura orientada a microsserviços trata do desenvolvimento de vários serviços, definidos tendo em conta o modelo de domínio, os quais tem um processo de *deploy* independente entre si. É um tipo de arquitetura orientada a serviços (*Service-Oriented Architecture*) a diferença na forma de como as fronteiras entre serviços devem ser definidas e que a independência do processo de *deploy* é uma aspeto chave.

Este tipo de arquitetura oferece várias vantagens. O processo independente de *deployment* abre novas formas de melhorar as escalabilidade e robustez do sistema. Os serviços podem ser desenvolvidos em paralelo e o crescimento natural das equipas de desenvolvimento fica facilitado, pois devido ao facto das aplicações estarem isoladas e de estarem, normalmente,

sobre a alçada de diferentes equipas. Assim, existe muito menos entropia no desenvolvimento multi-equipa.

Esta arquitetura é também tecnologicamente agnóstica, o que significa que permite à equipa de desenvolvimento escolher a tecnologia que mais se adequa ao propósito do serviço em questão. Além da tecnologia, cada serviço pode ter diferentes estilos de programação, plataforma de *deployment* ou mecanismo de persistência. Acima de tudo, uma arquitetura orientada a microsserviços dá a flexibilidade de abrir a porta a várias opções de como resolver possíveis problemas que possam aparecer no futuro.

Ainda assim, este tipo de arquitetura vem com algumas desvantagens também. Uma arquitetura SOA pressupõe a separação dos serviços em diferentes máquinas, o que levanta alguns problemas que anteriormente não existiam. Estando em diferentes máquinas, os serviços agora têm que comunicar através de redes.

A partir do momento que o fator dos serviços comunicarem por rede entre si, temos que trazer uma série de fatores para a equação, tais como: latência, perdas de pacotes ou mesmo falhas rede por fatores externos.

Além disto também é introduzido o fator de que agora não está a ser tratado tudo no mesmo processo, o que indica que podemos estar expostos a problemas de concorrência quando anteriormente não era o caso (Newman s.d.).

2.2.2 Arquitetura Monolítica

Numa arquitetura monolítica toda a *code base* está centralizada numa única aplicação, onde os módulos desta não podem ser executados de forma independente. Este tipo de arquitetura é fortemente acoplada e toda a lógica de negócio está centralizada (Ponce, Marquez e Astudillo 2019).

O exemplo mais comum de uma aplicação monolítica é um sistema cujo *deployment* é feito num único passo. Existem também arquiteturas monolíticas modulares, como é possível observar na Figura 2.3 em que existem vários módulos na aplicação mas, estes continuam fortemente acoplados, de tal maneira que o processo de *deployment* continua a ter que ser feito em conjunto.

Em vários casos, esta arquitetura pode ser uma excelente escolha para o desenvolvimento de uma aplicação, permitindo que haja uma possibilidade de desenvolvimento paralelo com uma divisão de módulos bem definida e, assim, conseguir com que o processo de *deployment* seja significativamente mais simples.

Porém, continuam a haver alguns problemas nesta arquitetura. Para além dos problemas já referidos, de estar mais propenso à existência de um maior acoplamento, tanto na implementação como no *deploy*, existem também problemas que afetam o desenvolvimento de software em equipa. Se diferentes membros da equipa de desenvolvimento decidirem que querem alterar o mesmo excerto de código, ao mesmo tempo, ou alterar a mesma funcionalidade, irá causar entropia. Fazer esta separação e divisão de que equipa ou desenvolvedor é responsável por que módulo é algo bastante mais difícil de fazer numa arquitetura monolítica, onde os limites entre os módulos não são claros.

Ainda assim, esta arquitetura continua a trazer várias vantagens. A principal vantagem que faz com que esta arquitetura seja usada é a facilidade na sua implementação. Por ser simples e direta, todos os *workflows* de desenvolvimento são muito mais simples, facilitando

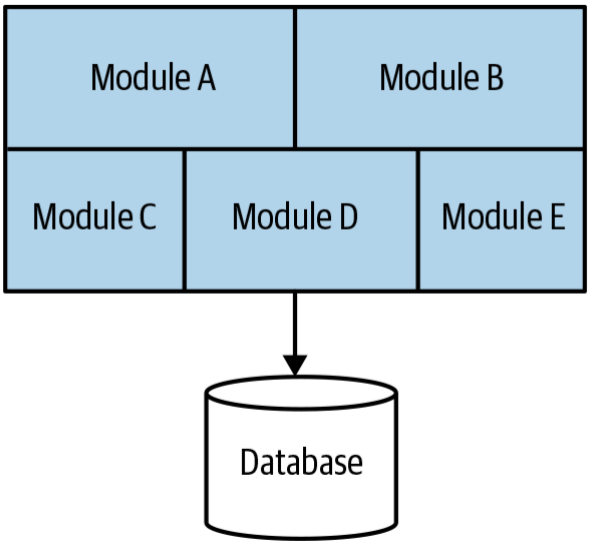

Figura 2.3: Arquitetura Monólitca Modular. Newman s.d.

também a implementação de alguns processos como testes automáticos. Outra vantagem é a facilidade na reutilização de código dentro do próprio monólito. Enquanto num sistema de serviços distribuídos é necessário decidir se a solução consiste em mover o código para uma livraria ou criar um novo serviço, num monólito é muito mais simples, pois todo o código está centralizado.

Nos tempos correntes, este tipo de arquitetura tem sido vista como algo a evitar por estar "ultrapassada"e ser algo "antiquada". Contudo, a adoção de uma arquitetura monolítica, mesmo podendo não ser a melhor opção em algumas circunstâncias, continua a ser uma opção válida. A escolha de uma arquitetura deve ter sempre em conta o contexto do domínio e requisitos da aplicação a desenvolver (Newman s.d.).

2.3 Domain Driven Design ()

No processo de desenvolvimento de software, a captura do domínio e a compreensão dos detalhes e objetivos do negócio são processos críticos para assegurar o sucesso de qualquer projeto. Uma solução que não esteja alinhada com o domínio do negócio acabará por ter que ser redesenhada, o que causa atrasos e aumenta os custos do desenvolvimento, originando descontentamento nos clientes. Assim sendo, no desenvolvimento do software é necessário ter um bom conhecimento do domínio do negócio. No sentido de evitar estes problemas, foi desenvolvido o *Domain Driven Design* (DDD), um processo de desenvolvimento de software que se foca em alinhar as aplicações com o domínio de negócio. Este conceito foi introduzido pela primeira vez por Eric Evans em 2003 no livro *"Domain-Driven Design: Tackling Complexity in the Heart of Software."* (Evans 2014).

O DDD é uma abordagem à arquitetura no desenvolvimento de software que se concentra na criação de uma linguagem comum (*ubiquitous language*) entre os programadores e peritos no domínio do negócio. O objetivo do DDD é reduzir a complexidade dos sistemas de software, melhorar a sua qualidade e capacidade de manutenção e aumentar o seu alinhamento com o domínio de negócio.

Em certos casos, os domínios de negócio são bastante complexos e a quantidade de informação a reter por parte de quem desenha e desenvolve as soluções é demasiada. Nestes casos, para evitar estes problemas opta-se por dividir a aplicação em módulos menores, menos complexos e com a lógica bem definida. Estes módulos idealmente deverão ser o mais independentes possível entre si, para evitar um alto acoplamento.

Bounded Context é um dos conceitos-chave do DDD que remete para a divisão de uma aplicação de software em várias partes distintas com os seus modelos únicos e delimitados. Cada contexto terá restrito a sua parte do negócio, com as suas próprias regras, conceitos e relações. Esta divisão ajuda a diminuir a complexidade do domínio, tornando cada módulo mais simples e mais fácil de gerir. Para decidir onde um contexto começa e acaba é necessário algum conhecimento do domínio e dos seus requisitos. A definição destes contextos é muito importante, pois irá afetar significativamente o design da aplicação (Avram e Marinescu 2006).

Esta abordagem debate também alguns conceitos como os Agregados, Entidades e *Value Objects*. Estes termos referem-se a diferentes tipos de objetos do modelo de domínio e desempenham um papel na definição das relações entre os conceitos.

Entidades são objetos do modelo de domínio com um identificador único que representa um conceito de domínio. *Value Objects* por outro lado, são objetos que não têm uma identidade e apenas existem para representar o seu valor atual. Ao contrário das Entidades, os *Value Objects* uma vez criados não podem ser alterados, pois são tratados como objetos imutáveis.

Por fim, os Agregados correspondem a um conjunto de objetos com associações entre si, que podem ser considerados como uma única unidade quando se executa uma operação. Todos os Agregados são caracterizados por uma única raiz (*Aggregate Root*). A raiz é a única Entidade do agregado acessível fora deste e guarda a referência para qualquer outra entidade do agregado. Estas regras ajudam a assegurar a integridade de dados e o encapsulamento do domínio (Avram e Marinescu 2006).

Historicamente, o DDD tem sido demonstrado sempre no contexto de linguagens mais orientadas ao backend de aplicações, por estas terem uma estrutura melhor definida e cimentada. Porém, alguns trabalhos recentes mostram a aplicação dos princípios de DDD em projetos de Frontend (Le et al. 2022).

2.4 Micro-Frontends

Nesta secção é abordada a arquitetura de Micro-Frontends. É apresentado o resultado do estudo da arquitetura em si bem como diferentes abordagens possíveis para a implementar, associando com os pontos anteriores deste capítulo.

Micro-Frontends é um tema emergente que chama cada vez mais a atenção dos desenvolvedores de *software*. Foi apenas em 2019 que o famoso escritor Martin Fowler escreveu um artigo no seu website sobre exclusivamente Micro-Frontends, demonstrando o quão recente se trata este conceito (Fowler 2019).

Micro-Frontends é uma arquitetura inspirada em arquiteturas orientadas a microsserviços. O aspeto principal desta arquitetura está na divisão da *codebase* monolítica em módulos menores, proporcionando às organizações a separação e divisão do desenvolvimento por diferentes equipas, diminuindo a entropia causada na integração de trabalho. Porém, ao

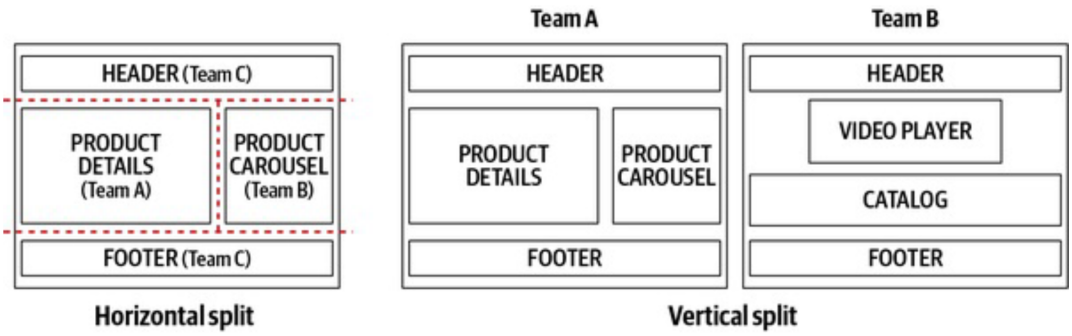

Figura 2.4: Orientação Horizontal VS Orientação Vertical (Mezzalira 2021).

contrário dos microsserviços que dizem respeito ao *backend*, uma arquitetura de Micro-Frontends tenta trazer os benefícios desta para a camada de *Frontend* das aplicações.

Para começar a implementação de uma arquitetura em Micro-Frontends, é necessário em primeiro lugar fazer uma decisão crucial para o projeto. É necessário definir para as equipas de desenvolvimento como vão considerar o conceito de Micro-Frontend. Existem dois tipos de orientação nesta arquitetura, horizontal e vertical. Na orientação horizontal é possível na mesma página ver vários Micro-Frontends, sendo que estes representam componentes da interface.

Na orientação vertical, cada Micro-Frontend corresponde a uma parte do domínio, como a autenticação ou as configurações de utilizador. Neste caso o DDD oferece uma grande ajuda no que toca na definição destes módulos.

É possível observar na Figura 2.4 os dois diferentes tipos de orientações bem como as diferenças nas organizações das equipas que estes implicam.

Como dito anteriormente, DDD é uma abordagem ao design de software que centraliza o desenvolvimento na definição de um modelo domínio que armazena um grande conhecimento das regras de negócio (Mezzalira 2021).

A aplicação do DDD numa aplicação Frontend é bastante diferente com uma de backend. Muitos conceitos não são aplicáveis em frontend, porém outros são úteis para a o design de uma arquitetura de Micro-Frontends.

Um destes conceitos é o *Bounded Context* já anteriormente referido. Este conceito dá ênfase à importância de definir limites entres diferentes módulos dos sistema. Para a design de uma arquitetura orientada a Micro-Frontends isto é crucial pois ajuda a definir que parte do domínio cada Micro-Frontend vai captar.

2.4.1 Benefícios

Trabalhar com microsserviços oferece grandes vantagens no que toca à simplificação da lógica de negócio, porém também é necessário lidar com todos os problemas relacionados com a integração dos sistemas. Como todas as outras arquiteturas, uma arquitetura de Micro-Frontends pode não ser a mais adequada para qualquer projeto, porém esta proporciona novas formas e uma nova estrutura que permite a aplicação escalar que de outra forma não seria possível (Mezzalira 2021).

Uma aplicação cujo *backend* esteja numa arquitetura orientada a microsserviços combinada com um *frontend* com uma arquitetura de Micro-Frontends disponibiliza a possibilidade da organização separar os módulos da aplicação por equipa, ficando uma equipa responsável por toda a *stack* relativa a uma funcionalidade.

No livro "*Micro Frontends in action*"por Michael Geers, o autor referiu algumas das razões para as organizações adotarem a uma arquitetura de Micro-Frontends, das quais se destacam duas (Geers 2020):

1. Otimização de desenvolvimento de funcionalidades – Cada equipa tem as capacidades técnicas necessárias para desenvolver cada funcionalidade.
2. Facilitar a atualização de tecnologias – Cada equipa é responsável para seu próprio módulo e a sua *stack* tecnológica, dependendo apenas de si mesma para fazer mudanças na mesma

O mundo do desenvolvimento *software* está em constante mudança e evolução, fazendo com que seja necessário estar em constante aprendizagem e a par das novas tecnologias que vêm surgindo. Desde modo, é inevitável para uma equipa efetuar uma transição tecnológica durante o seu período de vida. Existe o exemplo da equipa que desenvolve o plataforma *GitHub*, necessitou de fazer um projeto que durou vários anos para remover a dependência em *jQuery* (Engineering 2018).

Posto isto, é cada vez mais importante desenhar sistemas mais preparados para estas transições tecnológicas. Neste ponto uma arquitetura em *Micro-Frontends* destaca-se positivamente. Com a possibilidade de introduzir e criar uma prova de conceito de uma nova tecnologia como um módulo isolado da aplicação sem a necessidade de um grande projeto de migração de toda a aplicação é ativo valioso para uma equipa. Facilmente uma equipa consegue verificar um problema técnico na aplicação, implementar uma prova de conceito numa tecnologia com potencial de servir melhor o propósito da aplicação em apenas um módulo da aplicação e verificar que na prática se comprova a teoria.

Além disto, com a aplicação de *Micro-Frontends* cada equipa é responsável pela sua própria *stack* tecnológica e assim sendo não é necessária nenhum tipo de coordenação para qualquer tipo de migração tecnológica facilitando assim ainda mais este processo. Fazer este género de alterações numa arquitetura monolítica é um projeto muito mais complexo que necessita de uma grande planeamento com reuniões entre várias equipas. Nestas reuniões é possível não se chegar a um consenso pois as diferentes equipas podem ter diferentes necessidades e diferentes *trade-offs* necessários.

O processo de decisão em com várias equipas é muito complexo, demorado e não produtivo o que também leva a que as atualizações fiquem habitualmente para segundo plano. Neste caso uma arquitetura orientada a *Micro-Frontends* resolveria muitas das dores de cabeça permitindo que cada equipa possa gerir a sua *stack* tecnológica de forma independente.

Não estar dependente das outras equipas não se reflete positivamente apenas para manutenção da *stack* tecnológica. Tendo cada equipa a sua aplicação isolada e com o seu processo de *deploy* independente, permite com que os *deploys* de novas versões sejam também independentes. Com isto, deixa de ser necessário coordenar *deploy* e lançamentos de novas *features* com outras equipas, aumentando a produtividade e velocidade de desenvolvimento.

2.4.2 Desvantagens

Como dito anteriormente, uma arquitetura de *Micro-Frontends* nem sempre é a mais adequada para um projeto. Por muitos benefícios que esta arquitetura possa trazer, como tudo também trás algumas desvantagens. Nesta secção irá ser explorada algumas das principais desvantagens inerentes a uma arquitetura em *Micro-Frontends*.

Ter várias equipas de desenvolvimento a desenvolver a sua própria aplicação na sua própria *stack* faz com que haja bastante redundância de código. Redundância é algo que na engenharia de software tenta-se sempre combater, mas nesta arquitetura é inevitável que separando o código por várias aplicações não haja redundância. Esta falta de centralização pode amplificar alguns problemas, como bugs que estejam presentes numa biblioteca comum a vários módulos tenham que ser atualizados individualmente ao invés de atualizar apenas uma vez centralmente. Outro ponto é o facto da otimização de um módulo, à partida não vai beneficiar os outros módulos da aplicação. Otimizar um processo num módulo não faz com que automaticamente todos os módulos fiquem automatizados, sendo necessário que as outras equipas implementem a mesma otimização para que esta tenha efeito em toda a plataforma.

Esta arquitetura exige que os dados de cada módulo estejam isolados ao nível de cada equipa. Por exemplo, numa loja de *e-commerce* apenas equipa responsável pelos produtos da loja tem acesso direto à base de dados dos produtos. Estes dados vão continuar a ser precisos pelos outros módulos e por muitas vantagens que este encapsulamento ofereça, aumenta a complexidade no que toca à obtenção dos dados. Existem várias formas de combater este problema como a implementação de replicação de dados com *Event Bus*, porém além de trazer mais complexidade ao sistema, estes mecanismos trazem outras desvantagens como a latência e possível existência de uma inconsistência de dados.

Como referido anteriormente, uma das grandes vantagens de uma arquitetura em *Micro-Frontends* é o facto de ser tecnologicamente agnóstica e oferecer a possibilidade de usar diferentes tecnologias em diferentes módulos da aplicação. Porém, isto também trás pontos negativos. Se as diferentes equipas usarem diferentes *Uma combinação de ferramentas de software, frameworks, linguagens de programação e componentes usados para compilar e correr software (Stacks)* tecnológicas torna muito mais difícil para desenvolvedores trocarem de equipa dentro da organização ou até mesmo de partilhar conhecimento inter-equipas devido às diferenças nas implementações. Posto isto, não é por ser possível fazer algo que se deva obrigatoriamente de o fazer. Deve ser primeiro feito um estudo com os prós e contras de cada tecnologia e apenas escolher uma tecnologia diferente se esta escolha realmente compensar toda a entropia que vai introduzir na organização (Geers 2020).

2.4.3 Quando usar Micro-Frontends

Micro-Frontends é uma arquitetura a ter em conta quando se projeta uma aplicação de *software* que requer um processo iterativo e de manutenção a longo prazo quando estamos na presença de projetos que requerem a participação de várias equipas na mesma aplicação de *software* ou então quando queremos migrar um projeto *Sistemas de software desatualizados ou antigos, que por vezes usam tecnologias e arquiteturas obsoletas (Legacy)* de forma iterativa. Projetos com uma equipa pequena onde a comunicação não é um problema, a aplicação desta arquitetura não irá trazer muito valor. Segundo o *Michael Geers*, uma arquitetura de *Micro-Frontends* é mais orientada a projetos *Web* Geers 2020. Aplicações que são monolíticas nativamente, como por exemplo, aplicações controladas por *iOS* ou *Android*

que são monolíticas por *design*, podem não valer a pena todo o processo de implementar esta arquitetura.

2.4.4 Micro-Frontends na atualidade

Como dito anteriormente esta arquitetura é um tema bastante de recente na comunidade de engenharia de *software*. Porém, esta já é usada em várias grandes organizações. Muitas destas usaram os *Micro-Frontends* para escalar a sua organização para novos patamares. Algumas empresas *Zalando*, *HelloFresh* e *AllegroTech* implementaram arquiteturas inspiradas em *Micro-Frontends*.

Uma das grandes referências em termos de *Micro-Frontends* é a famosa aplicação do *Spotify*. A aplicação de *desktop* do *Spotify* é montada com múltiplos componentes presentes em diferentes *iFrames*. Esta abordagem foi feita inicialmente para o *player web* porém não passou nos testes de performance. Provando assim que a arquitetura pode nem sempre ser a mais adequada para o projeto em questão.

2.4.5 Frameworks Micro-Frontends

Dar exemplo de algumas *frameworks* usadas para a implementação de *Micro-Frontends*.

Com o aumento da popularidade desta arquitetura, começaram a surgir as primeiras *frameworks* que auxiliam a implementação desta arquitetura. Para uma orientação vertical duas *frameworks* existentes são a *single-spa* e a *qiankun*.

Single-SPA

O conceito por detrás da *single-spa* oferece uma série de ferramentas para a implementação de uma arquitetura *Micro-Frontend*. Esta *framework* é desenhada para ser agnóstica ás *frameworks* de Frontend que estão presentes na aplicação, permitindo que os desenvolvedores usem uma panóplia de *frameworks* diferentes e livrarias para construir ar arquitetura.

A *single-spa* foca-se em oferecer um abordagem consistente na gestão do estado da aplicação do carregamento e descarregamento dos serviços que compõem a aplicação e a comunicação entre os vários módulos da arquitetura.

A principal vantagem desta *framework* é a sua flexibilidade. O facto desta ser agnóstica à tecnológica permite cumprir o ponto forte da arquitetura de *Micro-Frontends*, que é a possibilidade do uso de diferentes *frameworks* para diferentes partes da aplicação. Para além disto, a *single-spa* fornece ferramentas ao desenvolvedor para facilitar a depuração e o processo de desenvolvimento (*single-spa* | *single-spa* 2023).

A *framework qiankun* é desenvolvida por cima da *single-spa*, adicionando algumas funcionalidades das versões mais recentes da *single-spa*.

Module Federation

Module Federation é um *Componente ou módulo de software que adiciona uma funcionalidade específica a um sistema já existente (Plugin)* nativo do *Webpack* 5. Este módulo permite que diferentes partes de *JavaScript* de um projeto possam ser carregados síncrona ou assincronamente. Com isto é possível que vários desenvolvedores ou até mesmo equipas

possam desenvolver código isolado e tratar da composição da aplicação, fazer o *lazy-load* de diferentes ficheiros *JavaScript* como é demonstrado na Figura 2.5 (Mezzalira 2021).

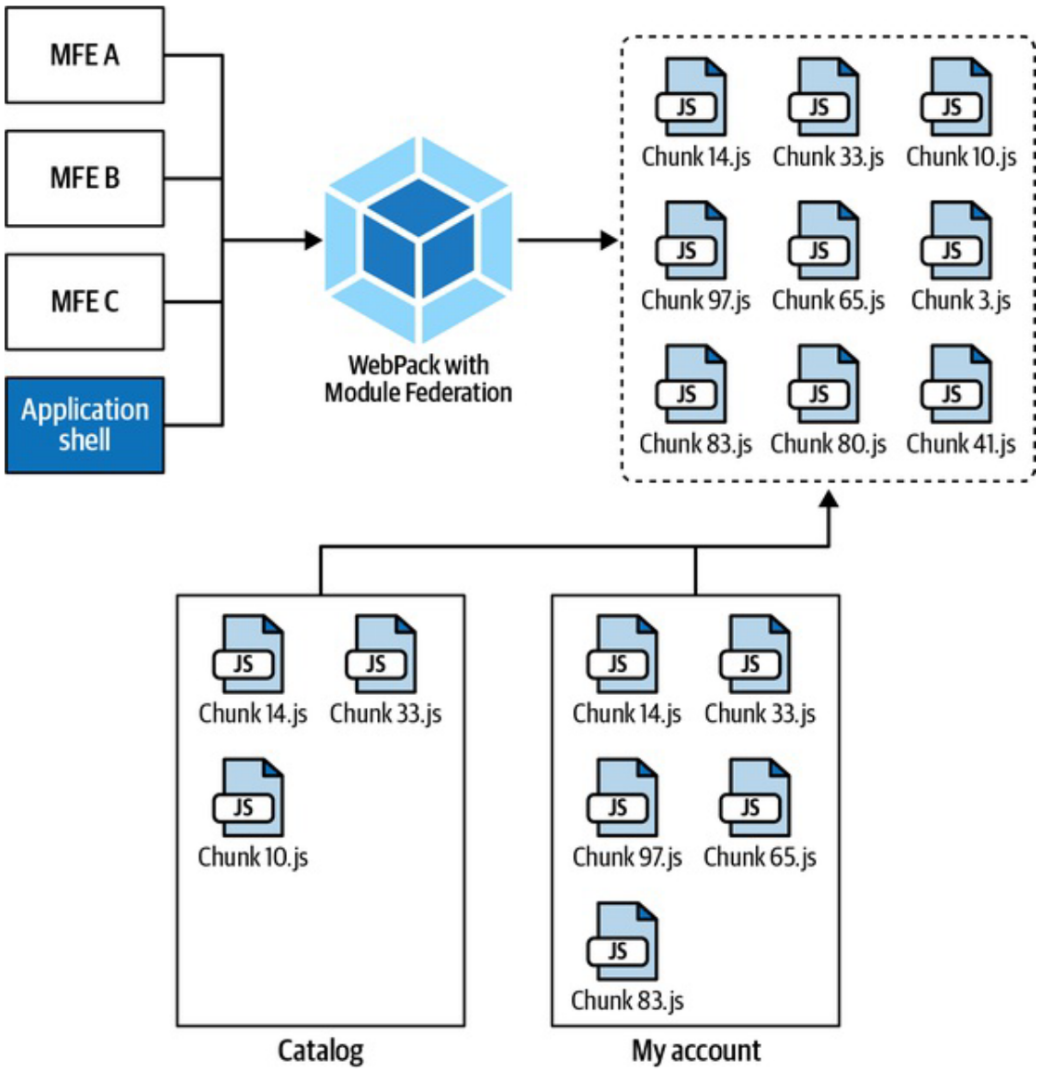

Figura 2.5: O *Module Federation* permite o carregamento de múltiplos *Micro-Frontends* (Mezzalira 2021).

2.5 Strangler Pattern

Nos capítulos anteriores foram estudadas as vantagens de uma arquitetura em *Micro-Frontends*. Porém, os sistemas já existentes que queiram migrar para esta arquitetura têm uma tarefa muito difícil.

A primeira alternativa no que toca a migrar um sistema existente para uma arquitetura em *Micro-Frontends* é a de rescrever a aplicação por completo. Esta opção é muito raramente possível de aplicar devido aos vários problemas que ela apresenta. Em primeiro lugar se existe a vontade de migrar para um sistema de *Micro-Frontends* é porque à partida estamos na presença de uma aplicação complexa para que esta mudança compense.

Reescrever a aplicação toda numa nova arquitetura vai ser um projeto de longa duração e de alta complexidade. Ao mesmo tempo, ainda seria necessário estar a dar suporte ao projeto atual e consequentemente implementar qualquer tipo de desenvolvimento no projeto de migração também.

Como é possível perceber esta alternativa acaba por se tornar demasiado complexa e pesada para as equipas.

A segunda alternativa é a aplicação do *Strangler Pattern* na migração da nossa aplicação *Frontend*. O *Strangler Pattern* traz a ideia de trazer o valor esperado da migração para a aplicação de forma incremental ao invés de tentar fazê-lo todo de uma única vez. A ideia por detrás do padrão é gradualmente "estrangular"o monólito isolando funcionalidades em *Micro-Frontends*. Com o avançar do projeto, mais e mais funcionalidades vão ser migradas do monólito para *Micro-Frontends* até que o monólito é substituído na íntegra.

Este padrão traz várias vantagens no contexto da migração de uma aplicação *frontend*. Em primeiro lugar, permite às organizações fazerem uma transição gradual para a arquitetura de *Micro-Frontends*, isto reduz o risco de alterações não intencionais no funcionamento da aplicação, como por exemplo a possibilidade de existência de bugs nas funcionalidades já existentes. Em segundo lugar, permite que as equipas se foquem em componentes menores e mais fáceis de gerir, reduzindo a complexidade da migração. Desta forma, as equipas podem trabalhar individualmente nos componentes e proceder à sua integração gradualmente reduzindo a complexidade do processo. Por fim, dá a oportunidade da organização ter os benefícios da migração ao longo do projeto fazendo com que a transição seja menos disruptiva.

Na prática o *Strangler Pattern* é implementado substituindo partes do monólito por *Micro-Frontends*.

Este processo deve ser começado pela identificação das componentes a migrar em primeiro lugar. Começar por componentes pequenas que não sejam críticas como por exemplo, algumas componentes da interface ou páginas individuais. Com o avançar do projeto e com maior confiança na nova arquitetura, procede-se à migração de componentes mais críticas da aplicação.

É importante também que ao longo deste processo sejam feitas *releases* frequentes para permitir ver o progresso da migração em ambiente de produção. Desta forma permite que as equipas experimentem e recolham informações diretamente do ambiente de produção de como a nova arquitetura se comporta, ao invés de se basear apenas em projeções (Mezzalira 2021).

Esta abordagem é bastante útil também no que toca a construir uma boa base para a arquitetura. Com o lançamento do primeiro *Micro-Frontend*, é possível analisar e verificar que a abordagem foi correta ou se existem pontos a melhorar. Isto tudo é possível dado que o processo obriga as equipas a pensarem em problemas mais pequenos ao invés da aplicação por inteiro, permitindo a análise do projeto a cada *release*, dando a oportunidade de aprender com os erros ao longo do projeto.

A aplicação deste padrão requer um planeamento cuidadoso e coordenação entre as equipas de desenvolvimento. É importante que se assegure que o monólito e os *Micro-Frontends* estejam a comunicar e a partilhar dados como anteriormente.

Para fazer esta migração gradual é necessário a implementação de um *API Gateway*. Uma *API Gateway* é um servidor que atua como um ponto único de entrada para os vários microsserviços. Esta gere e faz o *routing* dos pedidos para o serviço associado. No contexto da migração do monólito, a *Gateway* tem um papel crucial na comunicação e na partilha de dados entre o monólito e os *Micro-Frontends*. É possível ver um exemplo gráfico deste processo na figura Figura 2.6

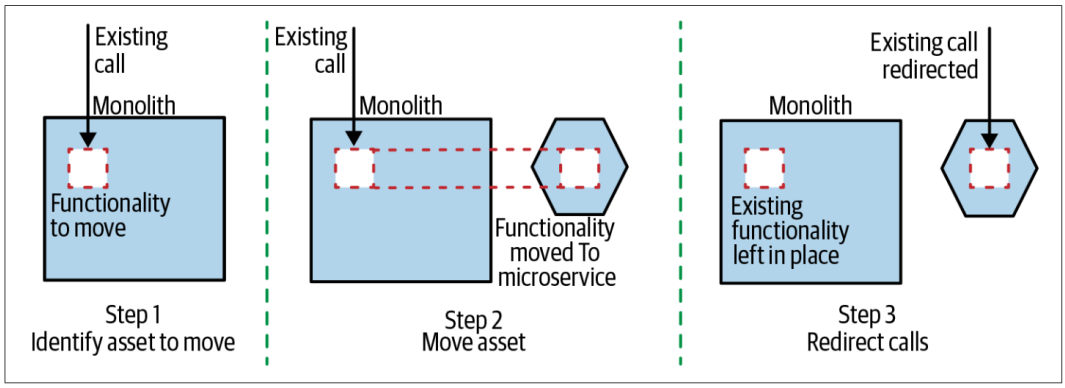

Figura 2.6: Um resumo do processo de aplicação do *Strangler Pattern* (Newman s.d.).

Quando um utilizador faz um pedido à página, a *API Gateway* é responsável por encaminhar o utilizador. Caso o módulo acedido não esteja ainda implementado em *Micro-Frontends*, é feito o encaminhamento para a parte *legacy* da plataforma. Cada vez que é desenvolvido um novo módulo da aplicação, este vai substituir outra parte da aplicação *legacy* até esta ser completamente substituída por *Micro-Frontends*,como é possível observar na Figura 2.7.

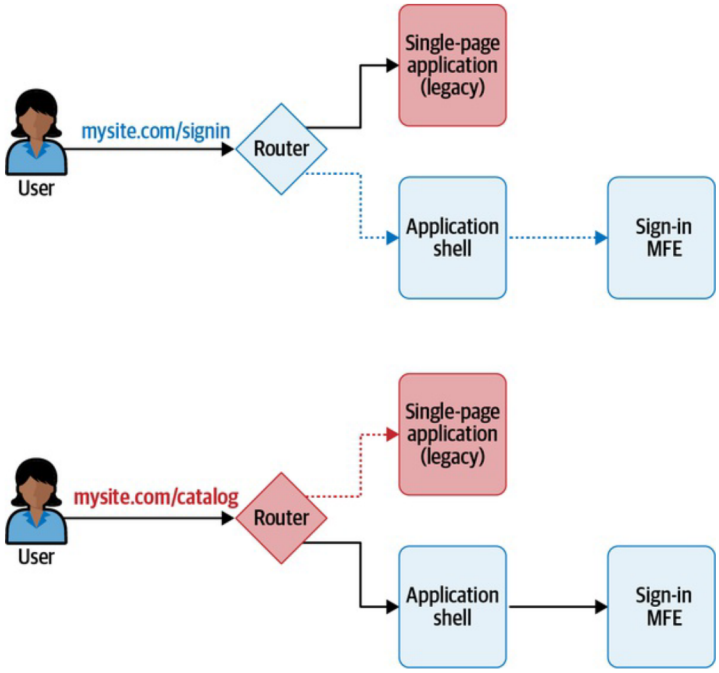

Figura 2.7: Exemplo do uso de uma *API Gateway* para o encaminhamento no processo da aplicação do *Strangler Pattern* (Mezzalira 2021).

2.6 Observabilidade

Outro ponto com grande importância a ter em consideração numa arquitetura em *Micro-Frontends* é a sua observabilidade. Isto é a capacidade de conseguir monitorizar os *Micro-Frontends* em ambiente de produção. Caso contrário não é possível reagir de forma rápida a nenhum possível incidente que possa acontecer durante o processo de migração ou mesmo na fase de maturidade da arquitetura.

Com evolução da migração e com os múltiplos *Micro-Frontends* em funcionamento, fica mais complexo fazer a gestão da performance e monitorizar a estabilidade do sistema como um todo. Para colmatar esta falha, têm sido desenvolvidas algumas ferramentas para dar mais visibilidade do sistema e garantir que o sistema está a funcionar como o esperado.

Nos últimos anos tem sido desenvolvidas vários tipos de ferramentas que ajudam em vários aspetos como:

1. Gestão de *Logs*
2. Monitorização de Métricas
3. Rastreamento de Pedidos
4. Rastreamento de Erros

Algumas das ferramentas mais populares para este efeito são *Sentry*, *LogRocket* ou *Elastic*.

Sentry

Sentry é uma plataforma de monitorização de erros para aplicações de software que fornece relatórios de erros *real-time*. A plataforma fornece aos desenvolvedores a possibilidade de

monitorizar as aplicações tanto no *frontend* como no *backend* fornecendo uma visão da aplicação inteira.

LogRocket

LogRocket é uma ferramenta que grava e analisa as sessões dos utilizadores em aplicações web. Permite aos desenvolvedores ter dados estatísticos das ações dos utilizadores na plataforma. A ferramenta permite visualizar repetições de sessões de utilizadores, mapas de calor de cliques e mais dados que fornecem informação preciosa para melhorar a experiência do utilizador.

Elastic

Elastic é uma plataforma de *logging*, monitorização e analise de dados para aplicações e servidores. A plataforma permite ao desenvolvedor analisar dados *real-time* que dá informações acerca o desempenho da aplicação e dos comportamentos do utilizador no uso da plataforma.

Qualquer uma destas ferramentas dá um maior controlo e visão aos desenvolvedores sobre o estado da plataforma. Com estes dados é possível antecipar problemas e descobrir mais rapidamente a origem de possíveis bugs que possam surgir, melhorando assim a fiabilidade da aplicação de *software*.

2.7 proGrow

A plataforma proGrow tem como principal objetivo a conversão qualquer equipamento ou posto de trabalho *online*, sendo uma solução com instalação *plug & play* e *dashboards* personalizáveis, os retornos são gerados em tempo real. Esta plataforma é composta por vários módulos :

2.7.1 Shopfloor

O módulo *shopfloor* do proGrow agrega toda a informação do chão de fábrica do cliente em questão. Este consegue isto através de diversos *dashboards* divididos por várias áreas imitando a organização da fábrica onde é aplicada.

Existem *dashboards* de dois tipos, monitorização e manuais. Os *dashboards* de monitorização como o próprio nome diz, têm o objetivo de monitorizar e fornecer ao utilizador dados imediatos apresentados de forma concisa. Estes *dashboards* são totalmente configuráveis tanto em termos de cor como a informação que este apresenta. Normalmente as cores destes *dashboards* correspondem ao estado atual de uma máquina ou então o estado dos objetivos estabelecidos.

Estes *dashboards* contêm uma cor que pode ser fixa ou dinâmica, no segundo caso a cor do *dashboard* é definida pelos requisitos do cliente. O *dashboard* é também composto por uma grelha composta por várias células. Em cada célula é possível configurar a informação que esta contém ao nível textual, bem como a sua cor. Na configuração da cor esta pode ter diferentes configurações inclusivamente determinar objetivos. Um exemplo é o caso de uma célula que represente um indicar que pode apresentar cor verde quando atinge o objetivo predefinido.

As configurações podem também ser organizadas por *timeframes*, sendo estes à hora ou ao turno conseguindo assim mostrar dados ao utilizador apenas do *timeframe* em questão.

Usualmente estas configurações são usadas para mostrar ao utilizador projeções de indicadores de performance do turno atual ou turno anterior, como também mostrar indicadores por hora do turno.

Como podemos observar na Figura 2.8, temos um exemplo página *shopfloor* da visão geral de uma fábrica. Todos os cartões são de monitorização, cada um corresponde a uma máquina, a cor da barra superior corresponde ao estado da própria máquina e no interior tem informações que dizem respeito à produção atual da respetiva máquina.

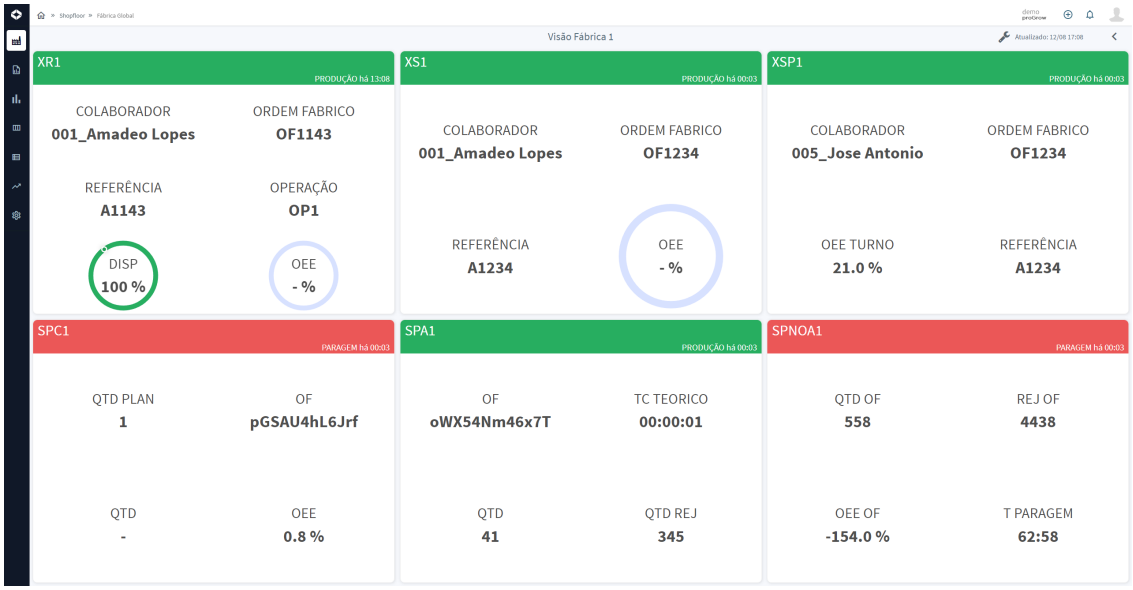

Figura 2.8: Exemplo de uma página do módulo *shopfloor* com a visão geral do Chão de Fábrica.

O segundo tipo de *dashboards* são os de tipo manual. Estes *dashboards* manuais têm como o objetivo o input de dados e contextualização da máquina que representa. Assim sendo, o operador que da máquina interage com o *dashboard* do tipo manual. Estes são similares aos de monitorização em termos de aspeto onde apresenta a mesma grelha, porém a cor do *dashboard* representa sempre o estado da máquina.

O operador consegue contextualizar a produção atual, preenchendo campos configuráveis tendo em conta o domínio do cliente, como também criar e justificar paragens planeadas ou não. É também possível registar outras informações do processo produtivo tais como a quantidade produzida e rejeitada.

Outro ponto de diferenciação destes *dashboards* manuais é a possibilidade de aceder e editar o histórico das diferentes informações recolhidas no *dashboard*, como o contexto da produção, o estado ou outras informações.

Como é possível verificar na Figura 2.9 temos uma página dedicada ao operador da máquina com dois *dashboards* dedicados. O primeiro *dashboard* é do tipo manual onde é controlada a atual produção da máquina como referido anteriormente. O segundo *dashboard* é do tipo de monitorização e apresenta informação que diz respeita à maquina em questão restringindo ao turno atual.

Em suma, o módulo de *shopfloor* tem uma panóplia de *dashboards* informativos e interativos que fornecem ao utilizador uma visão global do chão de fábrica. O módulo também serve

de ponto de interação do operador de máquina para contextualizar as produções de forma a permitir maior detalhe nas análises disponibilizadas.

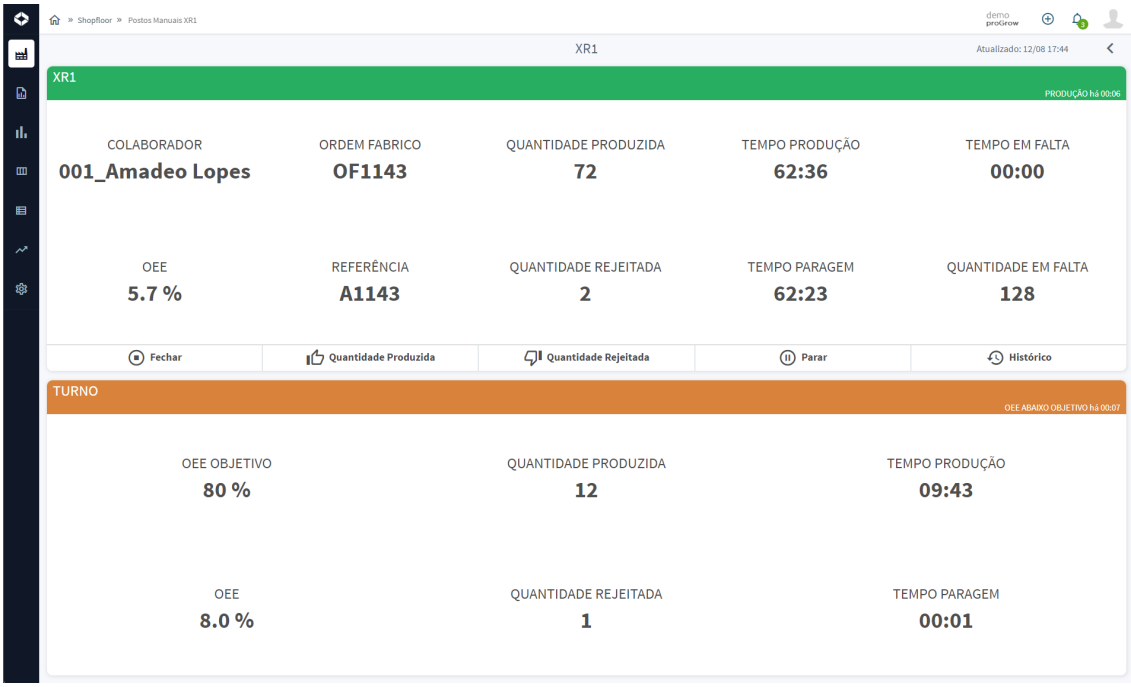

Figura 2.9: Exemplo de uma página do módulo *shopfloor* com a visão da página direcionada para o operador da máquina.

2.7.2 KPI

O módulo KPIs (*Key Performance Indicator*) é onde estão disponíveis todos os indicadores a que o cliente tem acesso. Nesta secção é possível visualizar os vários indicadores separados pelas áreas de negócio do cliente.

Os indicadores podem ser apresentados na forma de gráficos de barras ou de tabela. Em ambos os modos temos a possibilidade de filtrar a informação de forma temporal ou por alguma dimensão de análise. Existe também a possibilidade de exportar esta informação para uma imagem ou uma folha de cálculo.

Os KPIs aos quais o cliente tem acesso são definidos aquando a implementação do produto no cliente.

Na Figura 2.10 temos um exemplo de uma página do módulo de KPI's. Nesta página está presente o KPI Disponibilidade e este está agrupado ao nível da máquina. É possível ver no painel lateral também a possibilidade de aplicação de filtros nos resultados apresentados no gráfico.

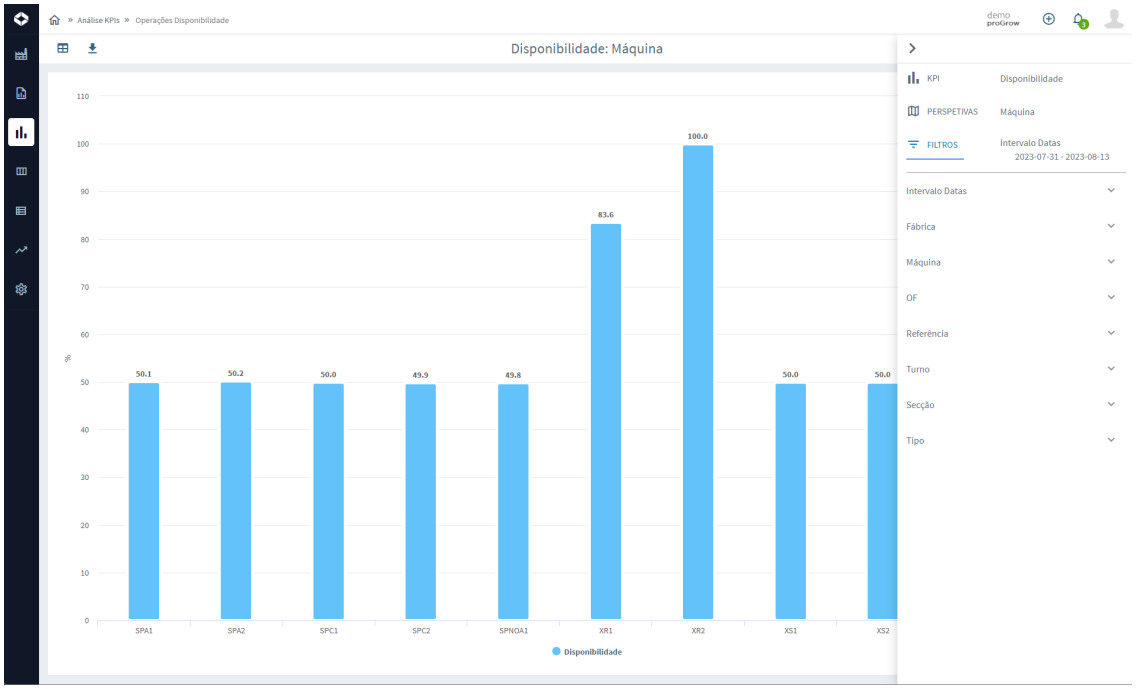

Figura 2.10: Exemplo de uma página do módulo KPI com a visão do KPI disponibilidade por máquina.

2.7.3 Relatórios

O módulo de Relatórios apresenta um menu dividido pelas diversas áreas de negócio do cliente que podem ser definidas em cada caso.

Estes relatórios são compostos por vários *dashboards* que podem conter uma variedade de gráficos configuráveis de forma que os relatórios sejam mais apelativos e fáceis de ler.

Os relatórios estão associados a um intervalo de tempo que determina a periodicidade com que são gerados novos dados. Eles podem ser diários, semanais ou mensais.

Os relatórios contém ainda uma componente cooperativa onde é possível deixar comentários como também a criação de ações de melhoria de associação ao relatório. Na Figura 2.11 temos um exemplo de um relatório de controlo de produção.

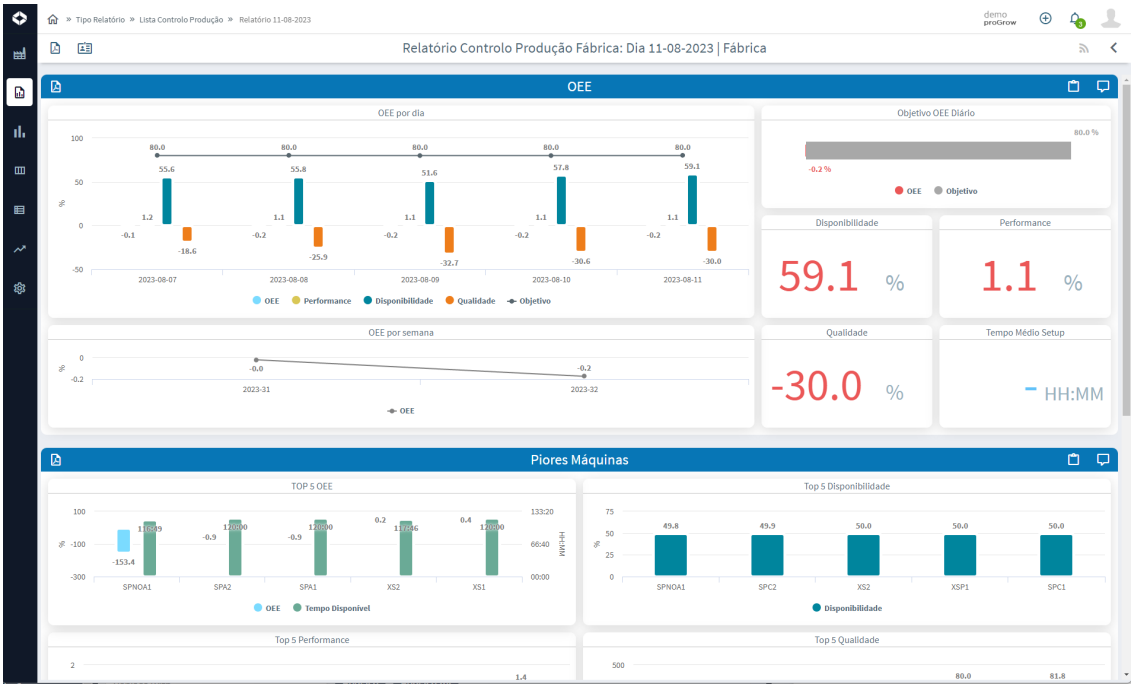

Figura 2.11: Exemplo de uma página de um relatório de controlo de produção.

2.7.4 Melhoria

O módulo de melhoria contínua permite digitalizar o processo manual de melhoria continua no ambiente industrial.

O processo de Melhoria continua, como proposto por Imai (1986), remete para a mudança de paradigma das divisões laborais tradicionais, ao encorajar que todos os membros da organização dediquem parte do seu tempo a resolver problemas e implementar melhorias de forma sistemática usado uma abordagem cientifica. Este processo, por muito sistematizado cientificamente que seja, pode ser desafiante para ser implementado efetivamente nas atividades diárias (Formento et al. 2013).

Através da criação de ações e projetos de melhoria, é possível o fluxo de trabalho de todas as iniciativas criadas para a melhoria de processo.

Posto isto é possível gerir os projetos de melhoria de forma standardizada, centralizada e simples seguindo várias metodologias como PDCA (*Plan, Do, Check, Act*), *Ishikawa*, 5 Porquês, etc... bem como monitorizar os resultados que estas estão a obter.

2.7.5 Conclusão

Com os módulos apresentados é possível perceber que o principal módulo e com uma maior complexidade é o módulo de *shopfloor*. Este mesmo módulo é o que oferece um maior desafio à aplicação *frontend* dado as várias possíveis configurações tanto ao nível de *dashboards* como de formulários. A necessidade dos *dashboards* serem atualizados em tempo real acarreta ainda mais o nível de complexidade e carga sobre a página.

As atualizações são feitas através de subscrições a um *websocket*. Cada célula pode fazer até duas subscrições, uma para a cor e outra para o conteúdo, fazendo com que em páginas de maior dimensão possam existir alguns problemas de desempenho especialmente em dispositivos menos capazes.

Capítulo 3

Análise de Valor

Neste capítulo, irá ser abordado o tema da Análise de Valor(VA). Assim sendo, vai ser apresentada a análise de valor do projeto de forma a se perceber qual a sua importância para a organização.

3.1 Definição de Análise de Valor

A Análise de Valor baseia-se no princípio de que o cliente procura sempre o melhor produto ao menor custo. Desta forma, pode-se afirmar que este método surge à volta do conceito de Valor, representando a ligação entre a satisfação do cliente (contribuição das funções para a satisfação de uma necessidade) e o seu custo.

De forma mais concreta, pode-se dizer que a Análise de Valor é uma metodologia estruturada para aplicação sistémica de um conjunto de técnicas que identificam funções necessárias, estabelecem valores para a mesma e desenvolvem alternativas para desempenhá-las ao mínimo custo, obtendo ainda a estima e a qualidade requerida pelo cliente.

Segundo Rich e Holweg, esta metodologia destina-se a aumentar o valor de um produto ou serviço para o cliente pelo menor custo possível, uma vez que é possível identificar e eliminar funcionalidades que não representam um real valor e cuja implementação traz custos associados, permitindo oferecer ao cliente um produto com qualidade superior e desenvolvido de forma mais eficiente (Rich e Holweg 2000).

Olhando para algumas das causas que podem justificar a aplicação da Análise do Valor, estas podem ser:

- Custos exagerados;
- Conhecimento de novas tecnologias;
- Projetos não suficientemente elaborados;
- Elevado nível de rejeições;
- Forte concorrência;
- Resultados financeiros demasiado baixos;
- Cumprimento de novas normas;
- Elevado número de reclamações;
- Alterações nos hábitos dos consumidores.

3.2 Orientação

Esta secção vai concentrar-se na primeira parte do desenvolvimento do VA. Na Figura 3.1 é possível ver um esquema com as várias etapas do processo do VA.

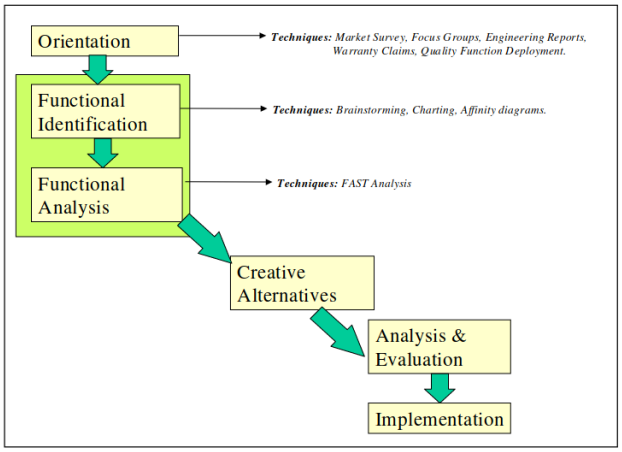

Figura 3.1: Esquema do processo de desenvolvimento de uma análise de valor (Rich e Holweg 2000).

Na fase de orientação é a fase onde é montada a equipa para a análise. Esta fase envolve também a seleção do produto ou processo em qual a análise vai incidir e o estabelecimento de objetivos para a análise. É necessário que nesta etapa seja bem definido o escopo da análise bem como os resultados a obter da mesma.

Nos últimos tempos, a aposta da inovação tem vindo a ser favorável, fazendo com que cada vez mais as organizações desenvolvam processos para que esta seja incentivada.

3.2.1 Fuzzy Frontend

Tendo isto em conta, foi desenvolvido por parte de Koen et al, um processo de inovação dividido em três partes: *fuzzy frontend* (), *new product development* (), e *commercialization* como é possível ver exemplificado na Figura 3.2.

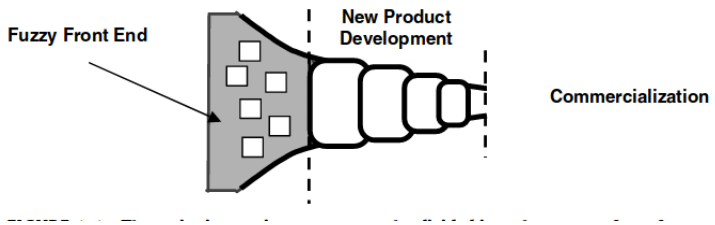

Figura 3.2: Processo de inovação em três fases: *fuzzy frontend* (FFE), *new product development* (NPD), e *commercialization* (Koen et al. s.d.).

A primeira etapa do processo FFE, é a etapa na qual é suposto serem feitas experiências e desenvolvidas ideias sem qualquer tipo de compromisso tais como, data de entrega ou orçamento.

De seguida, sucede-se a seguinte etapa de NPD onde é necessário uma maior disciplina no seu desenvolvimento. Desta forma, é fulcral saber onde se deve estabelecer objetivos e definir não só as datas de comercialização como também os orçamentos. É também importante que já se consiga perceber as expectativas de retorno para que estas se tornem mais claras com o aproximar da data de comercialização.

A terceira e última etapa é a fase de comercialização, que como o nome indica, é a fase na qual o produto é comercializado.

3.2.2 New Concept Development Model

O modelo New Concept Development (), como é possível observar na Figura 3.3 é constituído por três partes-chave:

- O Motor que representa a liderança, cultura e a estratégia de negócio de uma organização que orientam os cinco elementos-chave que são controláveis pela organização;
- O interior da circunferência que define os cinco elementos controláveis da FFE que são: a oportunidade de identificação, oportunidade de análise, geração e enriquecimento de ideias, seleção de ideias e definição de conceitos;
- A circunferência corresponde aos fatores de influência das capacidades da organização, como os canais de distribuição, políticas governamentais, concorrência e clientes. Estes fatores afetam todo o processo de inovação até a comercialização. Estes fatores são relativamente incontroláveis para organização;

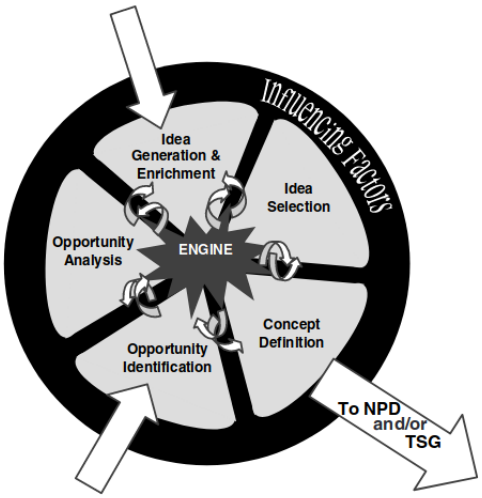

Figura 3.3: Constituição do modelo NCD (Koen et al. s.d.).

O aspeto circular do modelo NCD sugere que as ideias e conceitos são iterados pelos cinco elementos. As setas que apontam para o interior do modelo representam os pontos de partida e indicam que os projetos começam ou na identificação de uma oportunidade, ou no *brainstorm* de ideias e enriquecimento destas. A seta de saída representa como os conceitos saem do processo e entram no NPD.

3.3 Identificação de Oportunidades

Nesta secção irá ser elaborada a identificação de oportunidades, visando recolher as que se enquadram nos valores e visão estratégica da organização.

Ultimamente, no mundo de engenharia de software, tem vindo a ser cada vez mais usual os projetos de escalabilidade. Isto acontece, pois no momento da concessão do produto inicial não é dada tanta importância à escalabilidade do produto, mas sim à entrega de funcionalidades o mais rapidamente possível.

Porém, no ciclo de vida do produto existe um ponto de viragem onde o custo para manter um produto é demasiado elevado e a solução deixa de ser escalável. Para combater isto têm vindo a ser desenvolvidas arquiteturas pela comunidade que permitem tornar as soluções de software mais escaláveis.

Como é possível observar na Figura 3.4, a porção da manutenção nos custos do desenvolvimento de software tem vindo a aumentar para níveis muito altos ao longo do tempo.

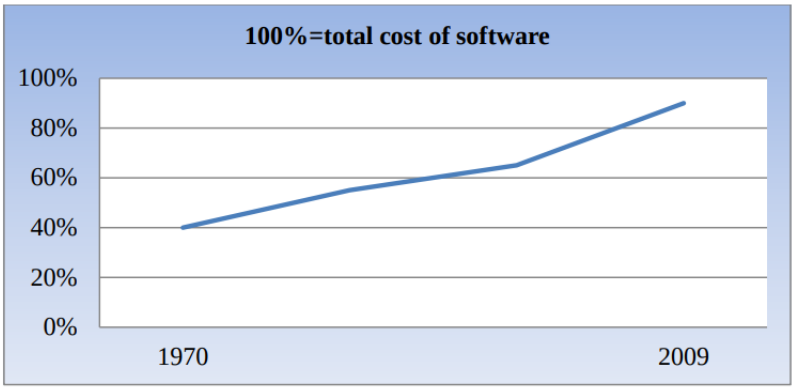

Figura 3.4: Proporção do custo de manutenção de software ao longo do tempo (Bv 2014).

3.4 Análise de Oportunidade

Com a oportunidade definida, é possível ser feita a análise da mesma.

Como foi anteriormente dito, a arquitetura em Micro-Frontends é um tema relativamente recente na comunidade de engenharia de software. Contudo, este deriva da arquitetura orientada a Microsserviços que já tem mais maturidade e provas dadas de ser uma arquitetura eficaz.

Como é possível observar na Figura 3.5, é notável o aumento da popularidade do termo. Isto significa que cada vez mais estão a ser publicados artigos sobre o tema e que estão a ser desenvolvidos mais projetos que adotam esta arquitetura. Isto sugere que esta tem vindo a ter um impacto nas soluções em que é adotada.

Porém, como está descrito na Figura 3.6, comparando o termo "Micro-Frontend"com "Microservice", a popularidade do segundo termo é bastante superior em comparação ao primeiro. Posto isto, é possível entender que efetivamente, o termo Micro-Frontend ainda é um tema recente e em evolução no mundo da engenharia de software.

Figura 3.5: Análise do *Google Trends* da popularidade do termo "Micro-Frontend"desde 2015 até aos dias de hoje.

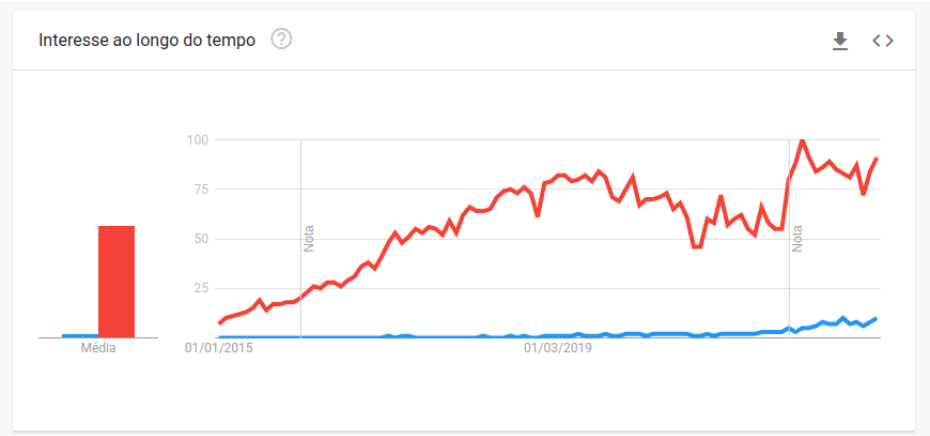

Figura 3.6: Análise do *Google Trends* da comparação da popularidade do termo "*Micro-Frontend*"com o termo "*Microservice*"desde 2015 até aos dias de hoje.

Devido ao recente surgimento do conceito Micro-Frontends, não estão disponíveis muitos estudos de popularidade efetiva da adoção desta arquitetura na comunidade. Assim sendo, vão ser analisados dados relativos à adoção de arquiteturas orientadas a Microsserviços pois os seus princípios são semelhantes tais como os efeitos que traz à solução.

Em Março de 2021 foi realizado um estudo onde foram inquiridos 4294 participantes. No grupo de inquiridos faziam parte desenvolvedores, especialistas de operações e especialistas em segurança. Posto isto, foi perguntado qual o nível de adoção de uma arquitetura orientada a Microsserviços na organização em que trabalham. Os resultados estão representados na Tabela 4.1.

É possível retirar que 71% das organizações adotam pelo menos parcialmente uma arquitetura em Microsservicos. Dessa forma percebe-se que é uma tendência que está a ser cada vez mais adotada devido às vantagens que oferece à organização.

Para concluir a análise de oportunidade foi elaborada uma análise SWOT de forma a identificar os pontos fortes e fracos da oportunidade em questão. É possível ver na Figura 3.7 os fatores internos e externos que influenciam a oportunidade.

Tabela 3.1: Nível de adoção de Microsserviços por parte das Organizações mundialmente em 2021 (*Microservices adoption level worldwide 2021* 2023).

Resposta	**Resultado**
Parcialmente	37%
Sim	34%
Não	28%
Outro	1%

3.5 Proposta de Valor

Nesta secção vai ser apresentado o modelo CANVAS, também conhecido como modelo *Osterwalder*, com a finalidade de mostrar o valor que o projeto irá adicionar às partes interessadas. É possível ver o modelo CANVAS elabora da Figura 3.8.

3.6 Análise Funcional

3.6.1 Quality Function Deployment

Nesta secção, vai ser feita a análise *Quality Function Deployment* () da solução. Para isto, irá ser usado o método QFD para identificar os requisitos dos clientes e a eficácia da solução a cumpri-los.

O primeiro passo no processo do QFD é a identificação da voz do cliente. Isto envolve entendimento das necessidades e requisitos dos clientes. Para a migração da solução atual para uma arquitetura em Micro-Frontends ser bem sucedida, têm que ser cumpridos os seguintes requisitos:

- Uso simultâneo de mais que uma framework de Frontend na solução final
- Maior escalabilidade
- Melhor performance
- Melhor experiência de desenvolvimento
- Maior modularidade
- Menor acoplamento

A ferramenta *House Of Quality* () é usada no processo de QFD para fazer a ligação entre os requisitos do cliente com o design do produto. O HOQ consiste numa matriz com os requisitos do cliente num lado e o design do produto no outro. No meio da matriz estão as relações entre estes.

O primeiro passo para o desenvolvimento do HOQ é a identificação dos requisitos do cliente que já foi feita anteriormente.

O segundo passo é a determinação da importância de cada um destes requisitos. Isto pode ser tipicamente feito com um questionário ao cliente para este avaliar numa escala de 1 a 5 o nível de importância do requisito.

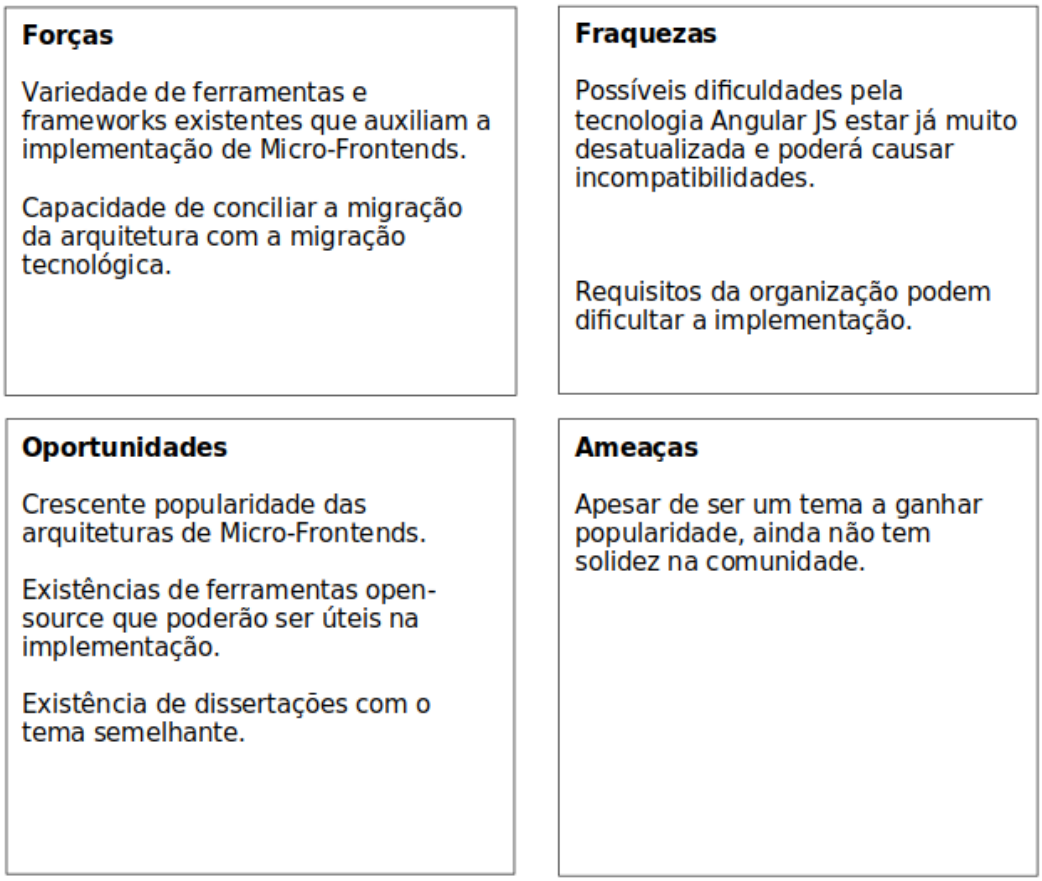

Figura 3.7: Análise SWOT

O passo seguinte é a determinação da forma como a solução apresentada irá cumprir os requisitos. Com a migração da arquitetura, a solução vai atingir os objetivos na seguinte forma:

Com a migração para uma arquitetura Micro-Frontends os diferentes módulos podem ser desenvolvidos em diferentes tecnologias devido a esta arquitetura ser tecnologicamente agnóstica, cumprindo assim o requisito de suportar diferentes *frameworks* de *frontend* em simultâneo.

Com a migração para *frameworks* mais recentes, vai ser possível ter ganhos em bastantes aspetos tais como, uma melhor performance devido às novas tecnologias estarem mais otimizadas nesse aspeto e melhor produtividade ao nível de desenvolvimento devido às *frameworks* estarem em constante desenvolvimento para melhorar esse ponto.

Numa nova arquitetura orientada aos Micro-Frontends será possível a separação de equipas por domínio permitindo que uma equipa conseguia efetuar o desenvolvimento de *feature* de forma independente sem dependências. Isto contribui positivamente para a escalabilidade, produtividade de desenvolvimento e para o baixo acoplamento.

Nesta nova arquitetura também será possível fazer a separação da *codebase* em vários módulos menores e menos complexos, melhorando assim a experiência de desenvolvimento e a modularidade da solução.

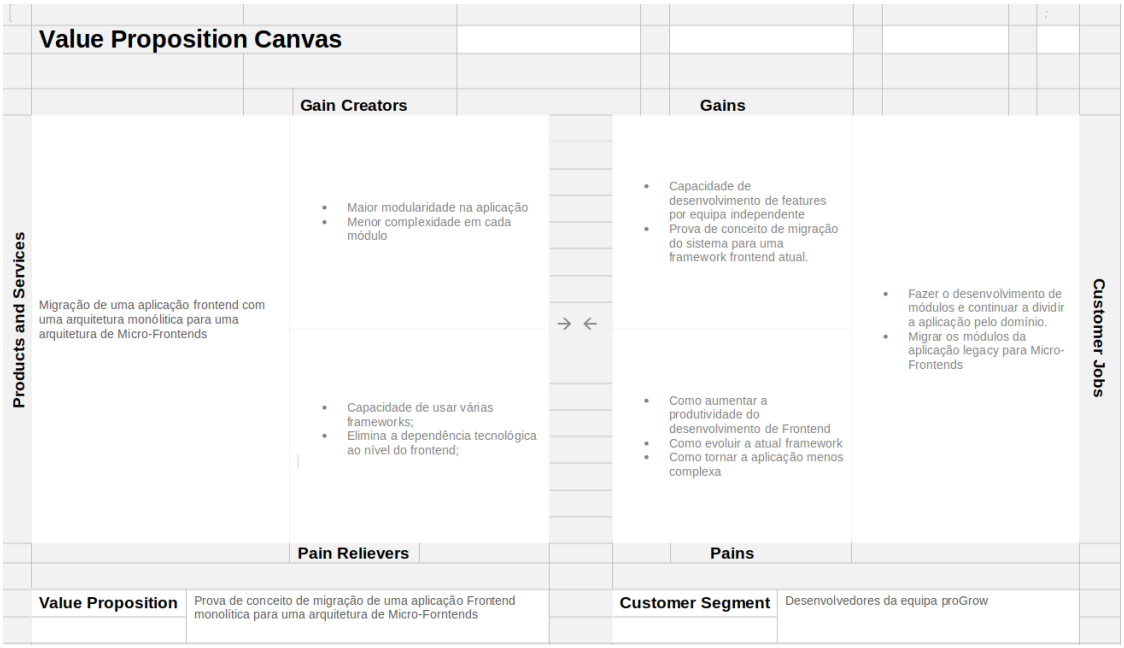

Figura 3.8: Modelo CANVAS

De seguida, é necessário determinar o peso de cada relação entre o requisito do cliente com o design do produto. Tipicamente, isto é feito através da atribuição de uma pontuação a cada relação baseada no quão o design afeta a o requisito. As pontuações são atribuídas entre os valores 1 (relação fraca) a 5 (relação forte).

O ultimo passo é a identificação de áreas de melhoria. Isto pode ser feito procurando por áreas onde as forças das relações entre os requisitos e o design do produto é fraca. Estas áreas podem ser alvo de melhorias.

Na Figura 3.9 podemos ver o diagrama *House Of Quality* desenvolvido.

			+			+	+	
			Functional Requirements					
		Direction of Improvement	□	□	▲	□	□	
Relative Weight	Customer Importance	Customer Requirements	Tecnologicamente Agnóstico	Framework Mais Recente	Seperação de Equipas	Separação de Codebase	Menor Complexidade	
31%	5	Mais que uma Framework	●	●	▽	▽	▽	
19%	3	Maior escalabilidade	▽	○	○	●	●	
6%	1	Melhor Performance	▽	●	▽	▽	▽	
13%	2	Melhor Experiência de Dev	○	●	●	○	●	
19%	3	Maior Modularidade	○	▽	○	●	○	
13%	2	Menor Acoplamento	▽	▽	●	○	○	
0%								
0%								
0%								
0%								
		Importance Rating Sum (Importance x Relationshi	412.5	537.5	375	450	412.5	
		Relative Weight	19%	25%	17%	21%	19%	
		Technical Competitive Assessment						

Figura 3.9: House of Quality

Capítulo 4

Micro-Frontends

4.1 Single SPA

Nesta secção irá ser abordada a *framework single-spa* no ponto de vista de arquitetura e funcionalidades.

4.1.1 Vista Arquitetural

A framework *single-spa* foi criada pela *Canopy* para resolver o problema da transição de uma aplicação *AngularJS* com *ui-router* para uma aplicação *React* com *react-router*. Esta inspira-se nos *lifecycles* dos componentes das *frameworks* modernas e aplica-os a aplicações inteiras (*Getting Started with single-spa | single-spa* 2023).

Ao usar esta *framework*, os programadores podem adotar uma arquitetura em *Micro-Frontends* que oferece inúmeros benefícios, tais como a flexibilidade tecnológica e uma maior modularidade ao sistema.

As aplicações *single-spa* dividem-se em:

- **single-spa root config** - contém a responsabilidade de renderizar o HTML e o *JavaScript* que regista as aplicações/módulos. Cada aplicação registada deve ter: um nome, uma função de carregamento do código da aplicação e uma função que determina se a aplicação está ativa ou não,
- **Aplicações** - estas podem ser assumidas como aplicações *single-page* que são empacotadas em módulos. Cada uma destas deve saber ser carregada e descarregada do DOM (*Document Object Model*).

A grande diferença entre uma SPA tradicional e uma aplicação *single-spa* é o facto de estas serem capazes de coexistir com outras aplicações como se não tivessem a sua própria página HTML.

Dois diferentes módulos da *single-spa*, quando ativos, podem escutar eventos de roteamento e colocar conteúdo no DOM. Quando inativos, não tem o acesso a esses eventos e são totalmente removidos do DOM.

No fundo, a *framework* acaba apenas por ser um pacote *npm* de pequena dimensão que faz a orquestração do carregamento e descarregamento dos módulos/*micro-frontends* do DOM. A *framework* sabe os *timings* de *loading* através de *activity functions* ou então através de *adapter libraries*.

Uma *activity function*, no contexto do *single-spa*, é uma função definida pelo desenvolvedor que determina quando a aplicação se deve manter ativa baseado no URL. Esta função atua como porta de segurança, permitindo ou negando que uma aplicação renderize ou execute código em resposta à navegação ou mudanças de estado no ambiente *single-spa*.

Serve como um componente vital para a decisão de quando um módulo tem que se considerar ativo numa aplicação multi-módulos. Esta função recebe o *window.location* como primeiro argumento e é esperado que retorne um valor verdadeiro no caso da aplicação associada ter que ser considerada ativa.

As *adapters libraries* são bibliotecas de código que ajudam a implementação das *lifecycle functions*, que são as funções de *bootstrap*, carregamento e descarregamento para uma *framework* SPA em específico (*react, vue, angular, AngularJs*...).

A transição de uma aplicação monolítica em *AngularJs* para uma arquitetura *Micro-Frontend* continua a ser um processo complexo. Para podermos ultrapassar esta complexidade, é necessário fazê-lo em várias etapas num processo incremental. As secções seguintes mostram como poderá ser uma potencial estratégia de migração.

4.2 Implementação da solução

Nesta secção irá ser explorado o plano de migração gradual da atual aplicação monolítica para a arquitetura orientada a *Micro-Frontends*.

4.2.1 Etapas de Migração

1. **Entender o sistema atual**: O primeiro passo do processo passará por redefinir a aplicação *AngularJs* monolítica existente como um microsserviço. Isto envolve encapsular a aplicação inteira numa unidade única que funcionará como um *Micro-Frontend*.
2. **Preparar o ambiente de single-spa**: De seguida, um ambiente *single-spa* tem que ser montado. Este passo envolve a criação da aplicação, de raiz, com o uso da *single-spa*. Será necessário encapsular a aplicação *AngularJs* dentro da *framework* com o uso de uma biblioteca auxiliar que irá fornecer as *lifecycle functions* como referido anteriormente.
3. **Registar a atual aplicação AngularJs como um Micro-Frontend**: Após a conversão da aplicação para um *Micro-Frontend* é necessário fazer o registo desta aplicação na *framework*. Este registo envolve a especificação do nome da aplicação, das funções de carregamento e também das condições de roteamento. Será necessário configurar quando é que o dito *Micro-Frontend* deve ser carregado para o HTML onde irá depender do URL da página.
4. **Migrar uma secção da aplicação monolítica**: Nesta fase, irá iniciar a aplicação do *Strangler Pattern*. Irá ser escolhida uma secção do proGrow para fazer a migração para um novo *Micro-Frontend* numa *framework* SPA diferente.
5. **Testes e validação**: A fase final passará por aplicar uma série de testes à arquitetura monolítica e à arquitetura em *Micro-Frontends* para validar se de facto houve melhoria na mudança de arquitetura.

   Neste sentido é necessário estudar a melhor estratégia de testes de forma a validar mais efetivamente o estudo em questão.

4.3 Testes de desempenho em aplicações Web

Nesta secção irão ser exploradas as possibilidades de testes a efetuar para a avaliação da solução.

4.3.1 Introdução

Os testes de desempenho em aplicações web são cruciais no que toca à avaliação da velocidade de resposta, escalabilidade e estabilidade de um aplicação web sobre várias condições (Proko e Ninka s.d.).

Estes envolvem a simulação de cenários que ocorrem em ambiente de produção e a medição de métricas tais como, tempo de carregamento da página e utilização de recursos. Ao introduzir testes de desempenho, as organizações são capazes de identificar potenciais *bottlenecks*, descobrir problemas de desempenho e assegurar que a aplicação consegue gerir a carga esperada e que as funcionalidades tem o comportamento esperado sem qualquer tipo de comprometimento.

Os testes de desempenho também auxiliam na otimização do desempenho da aplicação apontando os principais pontos de melhoria como otimizações de código, latência de rede ou no lado do servidor, como por exemplo, uma *Pedido ou comando feito a uma base de dados para receber ou manipular dados (Query)* de base de dados.

Em suma, avaliar a *performance* de uma aplicação *web* é essencial para providenciar ao utilizador final uma boa experiência de utilização da aplicação, mantendo sempre a robustez e eficiência da aplicação.

Os testes de desempenho são compostos por três de tipos de testes: (1) testes de escalabilidade, (2) testes de carga e (3) testes de stress.

4.3.2 Testes de escalabilidade

Os testes de escalabilidade têm em conta a capacidade da aplicação conseguir gerir diferentes volumes e tipos de atividade. Este tipo de testes deve averiguar a capacidade do sistema assegurar o serviço mesmo sofrendo picos súbitos de utilização. Um sistema robusto é capaz de se ajustar a estas mudanças de forma a assegurar um serviço estável para o utilizador.

Nestes tipos de testes devem ser tidas em conta as seguintes métricas (Proko e Ninka s.d.):

1. **Tempo de resposta**: corresponde ao tempo necessário para fazer o descarregamento das páginas e executar os pedidos necessários ao *backend* para inicializar a página. Esta métrica ajuda a avaliar o tempo que as páginas demoram a ser carregadas mesmo em picos de utilização.
2. **Uso de Memória/CPU**: corresponde ao uso dos componentes de *hardware* do dispositivo do utilizador final. Esta métrica ajuda a avaliar se as funcionalidades implementadas estão suficientemente otimizadas para garantirem que os dispositivos dos utilizadores finais não ficam sobrecarregados e, por consequência, piorar a experiência do cliente.
3. **Taxa de transferência**: corresponde ao número de pedidos que podem ser processados por um certo período de tempo.

4. **Uso de Rede**: corresponde à quantidade de dados transferidos pela aplicação. Esta métrica requer bastante monitorização e otimização. É necessário criar uma grande variedade de testes para garantir que a aplicação irá funcionar eficazmente em qualquer tipo de condições de rede experienciadas pelo utilizador final.

4.3.3 Testes de carga

Esta categoria de testes de desempenho tem o intuito de determinar ou validar o desempenho de certas características do sistema que está a ser testado quando este tiver sobre elevadas cargas de uso.

4.3.4 Testes de stress

Os testes de stress são similares aos testes de carga, porém as cargas usadas neste tipo de testes são muito além das esperadas em ambiente de produção.

4.3.5 Ferramentas de Teste

Como foi referido anteriormente, os testes de *software* são importantes para determinar a qualidade dos sistemas em questão. O principal objetivo da testagem é a deteção de erros e a validação de funcionalidades de forma a encontrar problemas e corrigi-los e maneira a melhorar a qualidade do *software* (Dhiman e Sharma 2016).

A utilização de ferramentas de teste simplificam este processo. Estas também facilitam a criação de ambientes de teste simulados para o sistema que está sobre análise, automatizando o processo de testagem e tornando-o mais eficiente.

A execução dos testes manualmente tem um elevado custo, requer uma grande quantidade de mão de obra, o que requer muito tempo. Por outro lado, a testagem automática com o auxílio de ferramentas reduz o custo, o tempo e o esforço gasto.

Existe uma grande variedade de tipos de ferramentas como as ferramentas de testes funcionais, testes de caixa-negra, testes de caixa-branca, ferramentas de *tracking* de *bugs*, ferramentas de teste de desempenho, entre outras (Arora e Sinha 2012).

As ferramentas de teste de desempenho são desenvolvidas especificamente para os vários tipos de teste de *performance* que foram anteriormente referidos.

Estas ferramentas dão o poder aos *testers* de criar, gerir e executar os testes dentro de um ambiente controlado e fabricado especificamente para a aplicação em questão e os testes necessários.

Algumas das ferramentas mais usadas no desenvolvimento de ambientes de testagem automática de aplicações web são:

- **Apache JMeter**: esta ferramenta é uma aplicação *open-source*, desenvolvida em Java com o objetivo de carregar testes funcionais e medir o desempenho (*Apache JMeter - Apache JMeter™* 2023).

  O objetivo inicial da aplicação era para ser usada para testar aplicações Web mas, entretanto, tem expandido as funcionalidades para outros tipos de testes. O *JMeter* pode ser usado com uma ferramenta de testes unitários para ligações JDBC, FTP, LDAP, JWS, HTTP, ligações TCP genéricas e processos nativos do sistema operativo.

Gerando múltiplas *threads* concorrentemente, o *JMeter* permite a simulação de carga no serviço para garantir a sua robustez e avaliar o desempenho sobre diferentes cargas.

Oferece suporte para a gravação de sessões de *browser* através de um *proxy* e permite o *replay*, oferecendo métricas de desempenho, tais como o tempo de resposta, taxa de transferência, latência e tempo de carregamento. Permite visualizar os resultados em diferentes formatos incluindo tabela e gráficos que podem ser acedidos em simultâneo.

O *JMeter* segue uma arquitetura baseada em *plugins*, onde as funcionalidades são implementadas através de *plugins*. Este design permite a fácil extensão do *JMeter* por desenvolvedores externos que têm a oportunidade de criar *plugins* personalizados.

Os planos de teste podem ser guardados em formato XML, facilitando a reutilização e a gestão dos testes. Fora os teste de carga, o *JMeter* também permite a execução de testes funcionais.

- **Locust**: Locust.io é uma ferramenta *open-source* desenvolvida para correr testes de carga. Os testes são definidos através de código *python* puro, fazendo com que seja facilmente personalizável e extensível para vários casos de uso e também mais fácil de implementar por parte dos programadores (*Locust Documentation — Locust 2.16.1 documentation* 2023).

  Através do *Locust* é possível simular elevadas cargas de utilização nos websites e API's e outros sistemas, para avaliar o desempenho destes sobre grandes cargas. Através do código *python* é possível criar vários cenários complexos de testes para simular realisticamente as interações do utilizador com o sistema.

  Um dos pontos fortes do *Locust* é o facto de ser distribuído e escalável. Com isto quer-se dizer que tem a capacidade de distribuir a carga em múltiplos *workers*, permitindo gerar cargas massivas e eficazmente aferir a escalabilidade do sistema. A distribuição dos testes garante que os testes de carga se assemelhem a casos da vida real, oferecendo um *insight* valioso do desempenho da aplicação e dos seus *bottlenecks*.

  Além disto, esta ferramenta tem uma integração com ferramentas CI/CD, permitindo incorporar os testes de carga na pipeline de desenvolvimento do *software*.

  Concluindo, o *Locust.io* oferece uma forma fácil e prática de implementar testes de carga em vários tipos de sistemas.

- **K6**: é uma poderosa ferramenta de testes de carga *open-source*, para o teste desempenho de sistemas e aplicações. É desenvolvida e mantida pela *Grafana Labs* e por membros da comunidade (*k6 Documentation* 2023).

  A k6 oferece uma interface *user-friendly* e a capacidade de ser possível desenvolver testes usando *JavaScript*, tornado-a mais acessível e flexível para os programadores. Com o k6 é possível simular grandes cargas nas aplicações Web, *websites* e API's de forma a avaliar o desempenho responsivo e estabilidade.

  Ao contrário de outras ferramentas, a k6 simplifica o processo dos testes de carga, eliminando a curva íngreme de aprendizagem associada com configurações complexas. Oferece uma interface de linha de comandos intuitiva que suporta a integração com *pipelines* CI/CD, dando a oportunidade dos testes de desempenho serem incorporados no ciclo de vida do desenvolvimento do *software*.

Com a k6, é possível escrever e executar os testes de carga com facilidade. Com simples *scripts* é possível definir cenários de testes, simular utilizadores virtuais, monitorizar várias métricas de desempenho como tempos de resposta, taxa de transferência e rácio de erros.

É possível analisar os resultados dos testes com várias funcionalidades de apresentação que a ferramenta fornece, que dão um maior detalhe na *performance* em termos de tendências, e ajudam a identificar potenciais *bottlenecks* do sistema.

Em suma, a k6 é uma ferramenta amigável para os programadores, pois simplifica o processo de teste de desempenho. Potencializa os desenvolvedores a proativamente identificar e colmatar as lacunas na *performance* das suas aplicações, assegurando que conseguem gerir grandes cargas de utilização eficazmente.

Esta ferramenta é capaz de testar um *website*, uma API ou qualquer outro sistema e é capaz de dar a oportunidade de otimizar o desempenho destes e ultimamente entregar uma maior fiabilidade da aplicação ao utilizador final.

Fazendo uma análise comparativa das três ferramentas anteriormente referidas, é possível dividi-las em dois grupos, relacionando as diferentes gerações nas quais as aplicações foram desenvolvidas e as respetivas abordagens na filosofia de utilização.

A primeira versão (1.0) da ferramenta *JMeter* foi lançada em 1998 o que apresenta vantagens e desvantagens. Por um lado, tem robustez e fiabilidade por já ter tido bastantes interações mas, por outro lado, peca que por mais que ainda esteja a ser atualizada, acaba por ter uma base antiga, o que torna a ferramenta um pouco desatualizada para os tempos modernos.

Para além disso, a ferramenta está escrita em Java, uma linguagem não muito otimizada relativamente às alternativas concorrentes.

Relativamente às ferramentas k6 e *Locust* são ferramentas da nova escola lançadas em 2016 e 2011, respetivamente. Ambas as ferramentas já adotam uma estratégia diferente do *JMeter* e permitem a elaboração dos testes através de código ao invés de uma interface UI tornando a interação mais intuitiva, rápida e prática para os programadores desenvolverem testes.

k6 é desenvolvida em *golang* enquanto *Locust* é desenvolvida em *python* o que comparativamente com o Java são linguagens de programação mais otimizadas e capazes de gerar softwares mais eficientes.

Tendo isto em conta é possível fazer a seguinte análise comparativa:

Análise comparativa

No que toca a desempenho e escalabilidade, a *JMeter* sendo uma ferramenta mais antiga, tem sido mais usada para testes de desempenho em larga escala. Esta oferece uma variedade de funcionalidades, tornando-a uma ferramenta capaz de comportar casos de testes complexos.

Contudo, devido ao facto de ser desenvolvida em Java, pode ter dificuldades a atingir o mesmo nível de performance e escalabilidade que as demais ferramentas.

k6 e *Locust* são ambas desenvolvidas para serem leves e altamente escaláveis. Usam o poder das linguagens de programação modernas (*Golang* e *Python*, respetivamente) para entregar testes de desempenho eficientemente.

Estas ferramentas podem gerir facilmente milhares de utilizadores virtuais e gerar grandes cargas no sistema sem consumir muitos recursos. O carácter leve e eficiente da arquitetura fá-las ideais para sistemas cloud-base ou sistemas distribuídos de testes.

Quando ao *Pequeno conjunto de instruções escritas em linguagens como python, Javascript ou Bash, que automatizam tarefas ou executam funções específicas (Script)* e desenvolvimento dos testes, o *JMeter* oferece uma interface gráfica (GUI) que permite aos *testers* montar os seus planos de testes visualmente. Esta característica é importante para utilizadores não técnicos porém, pode-se tornar pesada na criação de cenários de teste mais complexos e apresenta uma curva de aprendizagem mais acentuada.

Por outro lado, a k6 e *Locust* seguem uma abordagem orientada ao código. Os testes são escritos usando *JavaScript* (k6) ou *Python* (Locust), que oferece aos programadores um maior controlo e flexibilidade. Esta abordagem é benéfica para as equipas cujo conhecimento em programação está bem presente, pois dá-lhes a oportunidade de usar os seus protocolos de código e *frameworks*. É possível também integrar um controlo de versões, e reutilização de código, melhorando assim o fluxo de desenvolvimento de testes.

Relativamente à extensibilidade e integração, a *JMeter* tem um vasto ecossistema de *plugins* que permite aos utilizadores estender as funcionalidades e integrar com vários sistemas e protocolos. Esta oferece suporte a uma grande variedade de protocolos e também consegue integrar com ferramentas externas para a análise e visualização de resultados.

k6 e *Locust*, sendo ferramentas relativamente mais recentes, têm um ecossistema de extensões em crescimento. Neste caso a k6 acaba por ter um melhor reportório neste ponto. No entanto, estas não têm o mesmo nível de maturidade e variedade que as da *JMeter*.

Relativamente à comunidade e suporte, a *JMeter* está no panorama de testes de performance há muito tempo e tem uma comunidade grande e ativa por detrás. Isto reflete-se na vasta documentação de suporte à ferramenta, nos recursos online disponíveis e nos fóruns que tornam mais fácil encontrar respostas a quaisquer dúvidas.

A k6 e *Locust* têm ganho popularidade com o passar dos anos e têm comunidades por detrás da ferramenta, porém não será tão extensiva com a do *JMeter*. No entanto, ambas as ferramentas têm as equipas de desenvolvimento ativas.

Em suma, o *JMeter* tem uma grande reputação no que toca à robustez da ferramenta, contudo o facto do seu desenvolvimento ser focado na interface gráfica e ser desenvolvido em Java limita a sua flexibilidade e desempenho. Por outro lado, a k6 e *Locust* oferecem uma abordagem mais amigável para os programadores durante o desenvolvimento dos testes com uma melhor performance e escalabilidade. Estas destacam-se em ambientes *Agile* e CI/CD e, são mais direcionados para equipas que preferem ferramentas orientadas a código.

Google Lighthouse

Ferramentas para testes de desempenho, como as enunciadas anteriormente, são valiosas para avaliar o desempenho em aspetos específicos do desempenho do servidor. Mas no entanto podem não fornecer uma visão abrangente da experiência do utilizador final.

A *Lighthouse* é uma ferramenta *open-source*, desenvolvida para desenvolvedores web para fazer uma auditoria e melhorar a qualidade das suas páginas web. Esta ferramenta oferece informações e recomendações de melhoria de métricas como desempenho, acessibilidade, SEO e melhores práticas.

Isto é conseguido através da simulação de interações do utilizador com a página web, tendo em conta várias condições como a ligação à *internet* e o tipo de dispositivo. Ao analisar o carregamento da página e as interações com o utilizador são geradas um conjunto de métricas.

Estas métricas são quantificadas em pontuações para ajudar os desenvolvedores a perceber onde é que a página se destaca e onde precisa de ser melhorada (*Lighthouse* 2023).

Para além das funcionalidades já referidas, esta ferramenta pode também ser integrada num sistema CI/CD através do *Lighthouse CI*, assegurando que os padrões de performance são mantidos ao longo do processo de desenvolvimento. Extensibilidade é outra das funcionalidades dando a oportunidade de criar auditorias personalizadas para coletar informação para além das configurações padrão da *Lighthouse*.

Para o caso que está a ser desenvolvido este projeto, de todas as ferramentas de teste de desempenho, a *Google Lighthouse* parece ser a mais indicada. Isto deve-se ao facto de esta realmente se focar a estar a páginas web, e não os pedidos ao servidor em si. Dado que o projeto apenas se destina à aplicação *frontend*, faz sentido forcar a isolar os testes na área em que o estudo tem lugar.

Tabela 4.1: Comparação entre as diferentes ferramentas para testes de aplicações web.

Funcionalidades	**JMeter**	**k6**	**Locust**
Lançamento	1998	2016	2011
Codebase	Java	*Golang*	*Python*
Desempenho	Robusta, assegura testes em larga escala, porém fica aquém em termos de performance e escalabilidade	Leve e com uma grande escalabilidade	Leve e com uma grande escalabilidade
Desenvolvimento	Baseada numa interface gráfica	Orientado a código escrito em JavaScript	Orientado a código escrito em *Python*
Extensibilidade	Vasto ecossistema de *plugins* com um suporte a vários protocolos	Crescente ecossistema de *plugins* e extensões	Crescente ecossistema de *plugins* e extensões porém ainda bastante básico
Comunidade e Suporte	Grande e ativa comunidade, documentação extensa e vários recursos online	Crescente comunidade e documentação	Crescente comunidade e documentação

4.3.6 Abordagens para os testes de migração para Micro-Frontends

Esta secção explora abordagens possíveis para os testes da solução final, analisando estratégias, melhores práticas e ferramentas disponíveis para garantir o sucesso do processo de migração.

Testes A/B

Testes A/B são um tipo de procedimento de testes usual para a otimização de *websites* (Kaufmann, Cappé e Garivier 2014). O teste consiste na apresentação de duas versões de uma aplicação que são comparadas empiricamente ao serem apresentadas aos utilizadores. Cada utilizador apenas usa uma das versões e o objetivo é determinar qual das versões é a melhor.

No fim, é feita uma atribuição de uma pontuação à característica a avaliar, como por exemplo a responsividade das aplicações, por parte de cada utilizador. As pontuações são modeladas através de distribuições de probabilidades e respetivas médias.

Para a execução de um teste A/B serão necessários os seguintes passos:

- **Definir os grupos de testes**: começar por separar os utilizadores em dois grupos, um que usa a versão monolítica da aplicação e outro que usa a versão com uma arquitetura orientada aos *Micro-Frontends*. É também necessário assegurar que os grupos são representativos e comparáveis em temos de tamanho e características.
- **Métricas de teste**: definir as métricas mais adequadas para medir o desempenho e experiência de utilizador de ambos os grupos. Estas métricas podem incluir o carregamento de página, a retenção do utilizador e outros KPI's relevantes.

- **Análise estatística**: Elaborar uma análise estatística que determina o valor estatístico dos resultados para aferir se as diferenças observadas entre os grupos são estatisticamente significativas e com valor.
- Baseado nos resultados e no *feedback* dos utilizadores, iterativamente é refinada a implementação da arquitetura e melhorado o desempenho e experiência de utilizador. Os testes A/B permitem uma otimização contínua durante o processo de migração.

Compatibilidade e testes Cross-Browser

Existem atualmente vários *web browsers* no mercado no qual cinco *browsers* dominam. Estes são *Chrome, Microsoft Edge, Firefox, Safari* e *Opera* (*Most Popular Web Browsers in 2023 [Jun '23 Update] | Oberlo* 2023).

Assim sendo, nos dias correntes é um desafio para os desenvolvedores implementar aplicações que sejam suportadas por múltiplos *browsers* e várias versões do mesmo *browser*, o que levanta a necessidade de serem feitos testes de *cross-browser*.

Neste tipo de testes é necessário testar em múltiplas versões de vários *browsers* de forma a verificar que em todos existe um comportamento e aparência consistente. É também necessário ter atenção às funcionalidades específicas de cada browser, compatibilidades de CSS e *JavaScript* («International Journal of advanced studies in Computer Science and Engineering» 2014).

Outro ponto a ter em conta é a validação da responsividade da aplicação em vários tamanhos e resoluções de monitor, efetuar testes em vários dispositivos, incluindo telemóveis e *tablets* para garantir que o utilizador tem uma experiência consistente de utilização.

Em termos de acessibilidade, é necessário verificar que a aplicação é usável pelos padrões estabelecidos. Dado que a aplicação é usada em ambiente de chão de fábrica, é necessário confirmar que continua usável nas condições presentes este ambiente.

4.3.7 Gestão de Registo e estratégias de Rollback

Ao longo do processo de migração, é importante ter uma plano de gestão de risco e estratégias de *Processo de reversão de um update de software (Rollback)* preparado. Esta secção discute a importância da avaliação do risco, mitigação deste e potenciais estratégias de *rollback*.

Em primeiro lugar, é necessário identificar os potenciais riscos associados à migração tais como problemas de compatibilidades, problemas com a integração de dados, degradação do desempenho e problemas com a experiência do utilizador. Avaliar o impacto e a probabilidade dos problemas acontecerem.

De seguida, é necessário desenvolver estratégias de mitigação dos riscos identificados. Isto pode incluir a inclusão de testes contínuos, implementações graduais dos *rollouts*, monitorização de métricas de desempenho e envolver utilizadores no processo de testagem.

Por último, é necessário estabelecer um plano de *rollback* que indica os passos para reverter a aplicação para o estado anterior monolítico no caso de haverem problemas críticos não previstos ou complicações aquando da migração. Isto permite ter margem de manobra para manter a estabilidade do sistema e a experiência do utilizador.

Concluindo, neste capítulo foram exploradas várias alternativas de testes para a migração para uma arquitetura de *Micro-Frontends*. Foi sublinhada a importância da testagem incremental, testes A/B, testes de performance, testes de compatibilidade e de *cross-browsing*, tal como a gestão de risco e estratégias de *rollback*. Aplicando estas abordagens, as organizações podem assegurar uma migração suave e válida para uma arquitetura *Micro-Frontends* enquanto também são minimizados os riscos assegurando uma experiência positiva para o utilizador final.

4.4 Desafios e estratégias de mitigação

A migração de uma aplicação monolítica *AngularJs* para uma arquitetura *Micro-Frontend* apresenta vários desafios que são necessários serem tidos em conta no planeamento. Este capítulo discute os principais desafios que podem surgir durante o processo de migração e propõe estratégias de mitigação destes.

4.4.1 Gestão de Dependências e versionamento

Ao longo que a aplicação faz a transição para uma arquitetura *Micro-Frontend*, gerir as dependências e o versionamento fica cada vez mais complexo. Diferentes *Micro-Frontends* podem variar nas suas dependências e consequentes versões e levar a um potencial aumento de conflitos e problemas de compatibilidade.

Assim sendo, é necessário encontrar estratégias para mitigar este problema.

Uma estratégia é o isolamento das dependências, ao encorajar a que cada *Micro-Frontend* faça a gestão das suas dependências independentemente dos outros. É possível através do uso de gestores de pacotes como o *npm* e o *yarn* que evitam que as dependências sejam partilhadas entre os *Micro-Frontends*, reduzindo assim o risco de conflitos.

Na Figura 4.1, podemos ver a estrutura de pastas de um projeto que segue uma arquitetura de Mono-repositório (*MonoRepo*). Esta arquitetura tem como filosofia dividir a aplicação em vários pacotes, todos num único repositório. Desta forma os projetos relacionados e os seus componentes são guardados num único repositório de controlo de versões.

Com esta filosofia, a colaboração entre desenvolvedores é facilitada devido à simplicidade na partilha de componentes. A consistência de dependências é maior entre todos os projetos, reduzindo os problemas de compatibilidade. É também encorajada a reutilização de código, facilitando o *code refactoring* e assim promovendo a eficiência e qualidade do código.

É também possível recorrer ao uso de versionamento semântico para assegurar a compatibilidade ao fazer uma atualização nas dependências de um *Micro-Frontend*.

Outro ponto poderá ser a implementação de uma pipeline de testes automáticos para assegurar a compatibilidade quando são feitas atualizações de dependências.

4.4.2 Comunicação e Coordenação de Equipas

Com o aumentar de múltiplas equipas a trabalhar em diferentes *Micro-Frontends* em simultâneo, a comunicação efetiva e a coordenação entre as equipas torna-se crucial. Mudanças num *Micro-Frontend* podem inadvertidamente afetar outro, criando assim problemas de integração e problemas com as funcionalidades.

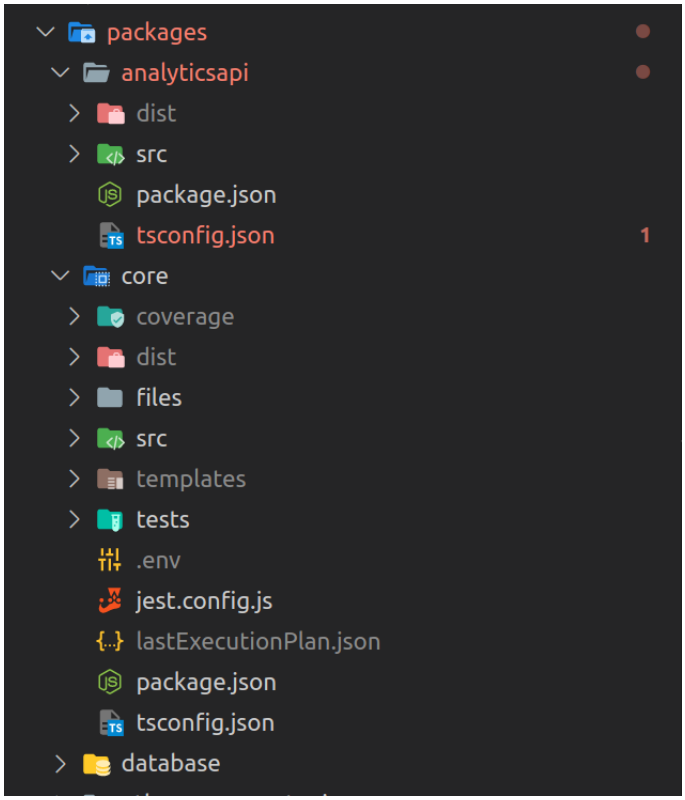

Figura 4.1: Exemplo de estrutura de diretórios de um projeto *MonoRepo*

Este tipo de problemas ocorre quando se faz uma mudança de arquitetura, porém a equipa continua a desenvolver com a mentalidade de *workflow* de uma arquitetura monolítica.

Um sistema desenvolvido numa arquitetura orientada a *Micro-Frontends*, promove a modularidade e a divisão do sistema, assim como a divisão da *codebase* por áreas de domínio dando a possibilidade de haver equipas dedicadas a apenas uma área. Porém, é ainda mais importante haver um planeamento entre equipas para a organização dos componentes da arquitetura de forma a manter a coerência do sistema.

As equipas precisam de planear a integração dos componentes para estes serem o mais desacoplados possível de forma a diminuírem a entropia e a dependência entre equipas.

Consequentemente, a implementação de testes de integração para validar as interações entre os diferentes *Micro-Frontends* poderia ser também ajudar a colmatar esta fraqueza.

4.4.3 Testes de Infraestrutura e Ambientes

Testar uma arquitetura *Micro-Frontend* pode pedir uma infraestrutura mais complexa com múltiplos ambientes. Gerir e configurar estes ambientes pode ser uma tarefa desafiante e levar a que haja inconsistências nos resultados dos testes.

O uso de ferramentas de código como infraestrutura (*Infrastructure as Code* IaC) tais como *Terraform* e *Ansible* podem ajudar a definir e gerir a infraestrutura de testes. Isto permite haver consistência e ambientes reproduzíveis de testes ao longo dos vários estágios de teste.

4.4.4 Monitorização e tratamento de erros

Numa arquitetura orientada aos *Micro-Frontends*, identificar e diagnosticar os problemas que podem ser originados por múltiplos vários *Micro-Frontends* pode ser uma tarefa desafiante. Como foi referido anteriormente, uma fraca observabilidade pode atrasar a deteção de erros e consequente resolução.

A implementação de ferramentas de monitorização e agregação de *logs* dá a oportunidade de ter uma maior visibilidade e torna mais rápido a identificação de potenciais problemas.

Em conclusão, esta secção deu conta dos desafios que podem surgir durante a migração para uma equipa orientada aos *Micro-Frontends* e foram abordadas algumas estratégias que podem ajudar a ultrapassar estes desafios.

Ao gerir dependências, fomentar a organização interequipas, implementar uma estrutura de testes de integração e implementar um sistema de agregação de *logs* e monitorização de erros, torna possível assegurar uma migração segura enquanto mantemos a qualidade e estabilidade do sistema.

Capítulo 5

Implementação da Arquitetura Orientada a Micro-Frontends

Nesta secção irá ser abordada a implementação da migração do sistema atual para uma arquitetura orientada a Micro-Frontends.

5.1 Bower

A primeiro obstáculo encontrado foi o facto da arquitetura atual usar *Bower* como sistema de gestão de dependências.

Esta ferramenta, apesar de não ter deixado de receber suporte oficialmente, a própria equipa de desenvolvimento recomenda a migração para novas tecnologias como *yarn*, *npm* ou *vite*.

Assim sendo, o primeiro passo para esta migração passa por migrar a solução para um novo sistema de gestão de dependências.

Visitando o site da ferramenta *Bower*, é recomendado fazer a migração para *yarn* dando ainda a sugestão de um pacote (*bower-away*), para facilitar a migração dos sistemas de controlo de dependências (*Bower* 2023).

O *yarn* não não é capaz de resolver as dependências do componente do *bower* definidos no ficheiro *bower.json*, como também não é capaz de traduzir os nomes dos componentes para os URLs dos repositórios. O pacote *bower-away* resolve todas as dependências do *bower* e adiciona todas elas, individualmente, ao ficheiro *package.json*.

Para efetuar a migração começou-se por instalar a biblioteca *bower-away*.

```
yarn global add bower-away
```

Após a instalação, aplica-se a operação do *bower-away* no projeto a migrar. Esta operação tem que ser efetuada na raiz do projeto, onde se encontra o ficheiro *bower.json*.

```
bower-away
```

Neste momento, a ferramenta examina o ficheiro *bower.json*, move as dependências do *bower* para o *package.json* e mostra uma pré-visualização das mudanças ao utilizador. A pré-visualização é obtida através do comando:

```
bower-away --diff
```

No terminal são mostradas as adições ao *package.json* em texto verde. Todas as dependências são adicionadas usado o prefixo *@bower_components*, o *yarn* é adicionado aos *"engines"*

e é adicionado o *script "postinstall"*. Para aplicar estas alterações é executado o seguinte comando:

```
bower-away --apply
```

Após aplicar as mudanças, é necessário mudar no código da aplicação de onde as dependências estão a ser carregadas. No caso desta aplicação, terão que ser feitas alterações nos ficheiros *index.html* e *lazyloadConfig.js*.

As mudanças passaram de mudar o caminho que está destinado para carregar os ficheiros das dependências e trocar de "*bower_components*"para "*node_modules/@bower_components*".

Após isto, pode-se apagar a pasta *bower_compoments* e todos os ficheiros associados ao *bower*, uma vez que estes deixaram de ser usados e a migração pode-se dar como concluída.

```
{
  "dependencies": {
    "@bower_components/almond" : "jrburke/almond#~0.2.9",
    "@bower_components/angular" : "angular/bower-angular#^1.0.8",
    "@bower_components/d3" : "mbostock-bower/d3-bower#~3.3.10"
  }
}
```

Listing 5.1: Exemplo de dependências do *Bower* no *package.json*

Deste modo, ao instalar o *package.json* com o *Yarn*, a pasta que contém os módulos irá conter todos os componentes exatamente da mesma forma que eles estariam no caso de serem instalados através do *Bower*.

5.2 Single-SPA

Após feita a migração do sistema de gestão de dependências, é possível começar a implementar a framework *single-spa*.

Seguindo a documentação do *single-spa* podemos encontrar tutoriais para a implementação da framework no nosso projeto.

O projeto em si contém duas particularidades que interessam para esta implementação, o facto de ser uma aplicação *angularjs* monolítica e o facto de não se recorrer a nenhum tipo de *bundler* para a construção da aplicação.

A documentação indica a utilização da *helper library single-spa-angularjs*. No que toca a utilização deste pacote auxiliar, oferece várias opções para proceder à sua implementação dependendo das características do nosso projeto. Existe uma secção para aplicações que usam um *bundler*, para aplicações que não usem e também secção para a migração de aplicações *AngularJS* já existentes.

Acerca da migração de uma aplicação já existente, a documentação começa por alertar da possibilidade de alta complexidade da migração, e acaba por indicar uma abordagem composta por três passos:

1. Converter a aplicação *AngularJS* para uma aplicação *single-spa* através de variáveis globais;
2. Converter a aplicação *AngularJS* de ser uma variável global para ser um módulo *SystemJS* integrado no browser.

3. Adicionar uma nova aplicação *single-spa* com outra framework

Para executar o primeiro passo começou-se por carregar a framework *single-spa* e a *helper library single-spa-angularjs* como variáveis globais, adicionando os *scripts* destas ao *index.html* da aplicação já existente em *AngularJS*.

```
<script src="https://cdn.jsdelivr.net/npm/single-spa@5.9.1/lib/umd/
    single-spa.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/single-spa-angularjs@4.2.1/lib
    /single-spa-angularjs.min.js"></script>
```

Listing 5.2: Exemplo das *tags* de *script* para carregar as *frameworks*

Após esta alteração, é necessário ajustar a aplicação de *AngularJS* para esta não ser montada no DOM, retirando o atributo *ng-app* do ficheiro *index.html*.

Posto isto, é necessário carregar a aplicação *AngularJS* através da framework *single-spa*, registando como uma aplicação *single-spa* numa variável global.

```
<script>
  window.singleSpa.registerApplication({
    name: "legacyAngularjsApp",
    app: window.legacyAngularjsApp,
    activeWhen: ['/']
  })
  window.singleSpa.start();
</script>
```

Listing 5.3: Exemplo do carregamento da aplicação *legacy* pela framework *single-spa*

Após este passo, é possível adicionar um *log* de forma a verificar se a aplicação está a ser montada corretamente.

Concluindo o primeiro passo, segue-se a conversão da aplicação *AngularJS* para um módulo *SystemJS* integrado no browser. Este passo é relevante caso haja interesse em efetuar importações cruzadas entre Micro-Frontends, ou seja, *imports* entre o Micro-Frontend *AngularJS* e os outros. Com as importações é possível partilhar funções, componentes, lógica, informação, *event emitters* e variáveis de ambiente entre os Micro-Frontends.

Para isto é necessário incluir o *system.js* no ficheiro *index.html* e remover a *tag script* com o *single-spa* colocada no passo anterior. Após isto, é necessário registar como um módulo *system.js* no *javascript* da aplicação, neste caso no ficheiro *app.js*.

```
<script type="systemjs-importmap">
  {
    "imports": {
      "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.1/lib/
    system/single-spa.min.js",
      "@org/legacyAngularjsApp": "/main-angular-app.js"
    }
  }
</script>
<script src="https://cdn.jsdelivr.net/npm/systemjs@6.10.2/dist/system.
    min.js"></script>
```

Listing 5.4: Carregamento do *system.js* no ficheiro index.html

```
// app.js - create a systemjs module
System.register('@org/legacyAngularjsApp', [], function(exports, context
    ) {

});
```

Listing 5.5: Registo de um módulo system js

Prontamente substituí-se a forma de importação da aplicação para usar o *systemjs*.

```
<script>
  System.import('@org/legacyAngularjsApp');
</script>
```

Listing 5.6: Nova forma de importação da aplicação

A partir deste momento é possível começar a fazer a migração.

O terceiro e ultimo passo passa por adicionar um novo Micro-Frontend. Na framework *single-spa* é encorajada a divisão dos *Micro-Frontends* pelo roteamento. Ao longo do período de migração/transição será necessário ter a aplicação *legacy* sempre ativa para ser possível mostrar os menus de navegação.

Para adicionar um novo *Micro-Frontend* apenas é necessário registar a aplicação no ficheiro *index.html* (Listing 5.7) e adicionar o *Micro-Frontend* ao *import map* (Listing 5.8).

```
window.singleSpa.registerApplication({
  name: "new-microfrontend",
  app: function () { return System.import("new-microfrontend"); },
  activeWen: ["/route-for-new-microfrontend"]
})
```

Listing 5.7: Registo do *Micro-Frontend*

```
<script type="systemjs-importmap">
  {
    "imports": {
      "single-spa": "https://cdn.jsdelivr.net/npm/single-spa@5.9.1/lib/
    system/single-spa.min.js",
      "@org/legacyAngularjsApp": "/main-angular-app.js",
      "@org/new-microfrontend": "http://localhost:8080/new-microfrontend
    .js"
    }
  }
</script>
```

Listing 5.8: Registo do *import map*

5.3 Desenvolvimento do Novo Micro-Frontend

5.3.1 Escolha do Módulo a Migrar

Para dar início à migração da plataforma para o seu primeiro *Micro-Frontend*, é necessário em primeiro lugar escolher o primeiro módulo a migrar.

Para a escolha do módulo, era necessário que este fosse de baixa complexidade e com um bom grau de isolamento.

Escolheu-se um módulo de baixa complexidade pois é importante evitar módulos complexos e críticos para o modelo de negócio para o primeiro passo, pois estes podem ser mais desafiantes e morosos de migrar.

Além disso, teve-se em conta se o módulo estava relativamente isolado do resto da aplicação, pois não havendo dependências com outras partes do monólito, torna mais fácil a extração e conversão num *Micro-Frontend*.

Tendo todos este fatores em conta escolheu-se o módulo do perfil do utilizador para ser o primeiro *Micro-Frontend*.

O módulo de perfil de utilizador contém uma página com três separadores.

No separador de "Dados Profissionais", são apresentados os dados do utilizador como o *username*, nome, email e a foto do perfil. Além destes dados é possível escolher o idioma do utilizador, o perfil do mesmo que impacta as permissões que este tem e os módulos aos quais tem acesso, e também é possível configurar a que equipas o utilizador pertence, como demonstrado na Figura 5.1.

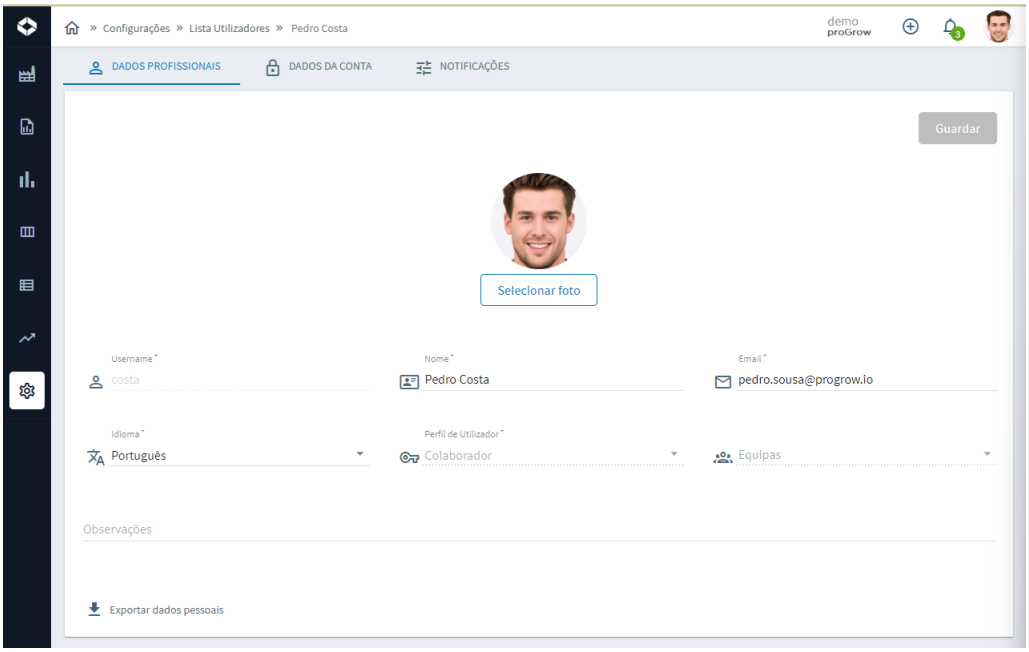

Figura 5.1: Exemplo do separador "Dados Profissionais"

No separador "Dados da Conta"permite o utilizador alterar a palavra-passe, como demonstra a Figura 5.2.

O último separador de "Notificações"permite ao utilizador configurar as notificações que o utilizador pretende receber na plataforma e também quais pretende receber por email, como é possível observar na Figura 5.3.

Relativamente ao URL, este módulo acaba apenas por conter um único URL com o formato de *"/app/profile/{USERNAME}"*, o que facilita o mapeamento do lado do *single-spa* para o carregamento do novo *Micro-Frontend*.

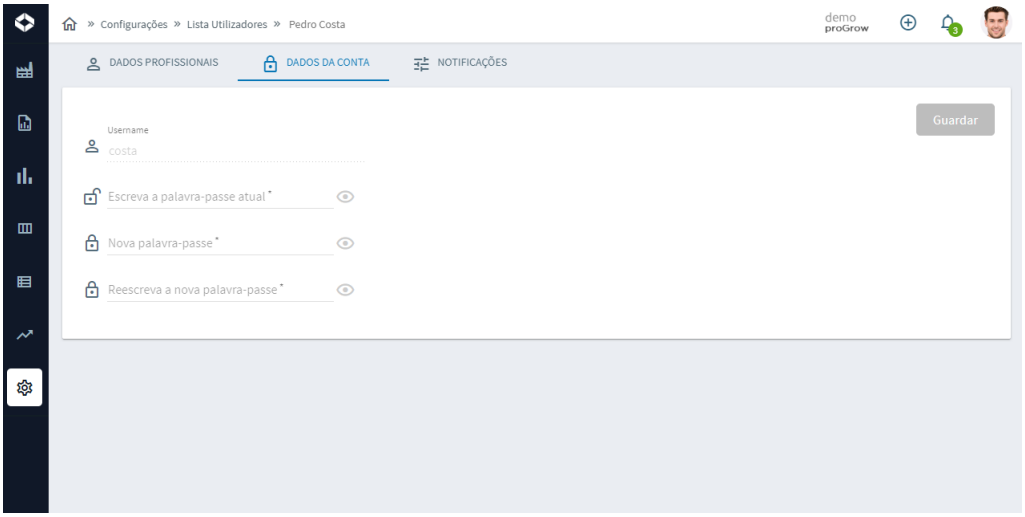

Figura 5.2: Exemplo do separador "Dados da Conta"

Assim sendo, é necessário estabelecer os casos de uso a suportar com o novo *Micro-Frontend* a desenvolver. Estes estão enumerados na Figura 5.4.

5.3.2 Escolha da Framework

A escolha da *framework* certa para o primeiro *Micro-Frontend* é crucial para assegurar a integração de forma suave, mantendo a produtividade dos desenvolvedores e a performance da aplicação. Nesta secção é desenvolvido o raciocínio por detrás da seleção da *framework Angular* como a *framework* do *Micro-Frontend* inicial.

Familiaridade

Fora a *framework* de *frontend* já existente na aplicação, o único conhecimento existente na equipa de outras *frameworks* de desenvolvimento *frontend* residem no *Angular*.

Além disto, a equipa trabalha regularmente com *Typescript*, linguagem usada nesta *framework*, o que dá uma maior familiaridade à ferramenta. Dado que a adoção humana tem um papel muito importante na adoção tecnológica, ter em conta a familiaridade na tecnologia tem um papel importante.

Uma *framework* conhecida reduz as incertezas e riscos associados com a migração e adaptação.

Ganhos de Produtividade

Familiaridade dá origem à eficiência. O conhecimento da equipa em Angular reduz o tempo necessário a ver documentação, na resolução de problemas e no conhecimento das melhores práticas.

Esta familiaridade traduz-se diretamente em ciclos de desenvolvimento mais rápidos e *debugging* acelerado. Ao longo do tempo, estas pequenas poupanças de tempo, irão traduzir-se em reduções significativas do tempo necessário para desenvolver funcionalidades.

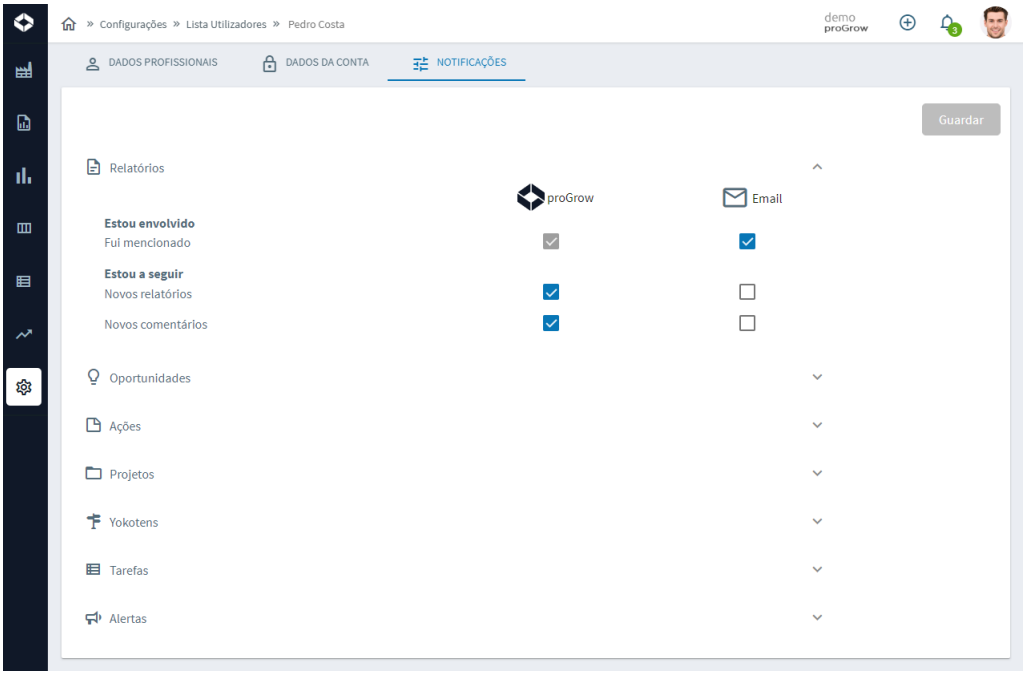

Figura 5.3: Exemplo do separador "Notificações"

Curva de Aprendizagem Reduzida

Enquanto existem múltiplas ferramentas com o mesmo fim do *Angular* disponíveis, cada uma com as suas particularidades, o *Angular* oferece uma curva de aprendizagem modesta, especialmente quando comparada com algumas alternativas como o *React*.

Para equipas familiarizadas com o Angular, esta curva é ainda mais reduzida. Isto leva a *onboardings* mais rápidos para novos membros da equipa e uma mais rápida adaptação dos membros existentes para a arquitetura orientada aos *Micro-Frontends*.

Ferramentas e Ecossistema

O *Angular* é conhecido pela maturidade e robustez do ecossistema. A *Angular CLI*, ferramenta de linha de comandos para a inicialização, desenvolvimento e manutenção de sistema *Angular*, é uma prova desta maturidade.

O ecossistema da *framework* inclui também uma documentação extensa, um grande leque de bibliotecas auxiliares e uma comunidade ativa.

Todos estes elementos combinam numa oferta de um processo de desenvolvimento mais standardizado e melhora a experiência geral do desenvolvedor.

Compatibilidade com os Princípios Micro-Frontend

A arquitetura modular do *Angular*, alinha-se bem com os princípios de uma arquitetura orientada aos *Micro-Frontends*. O foco da *framework* está nos seus componentes e desenvolvimento modular e assegura que as peças do *frontend* podem ser isoladas, desenvolvidas e testadas independentemente. Além disso, as capacidades de *lazy loading* do Angular permitem tempos de carregamento mais curtos, um dos principais fatores para os *Micro-Frontends*.

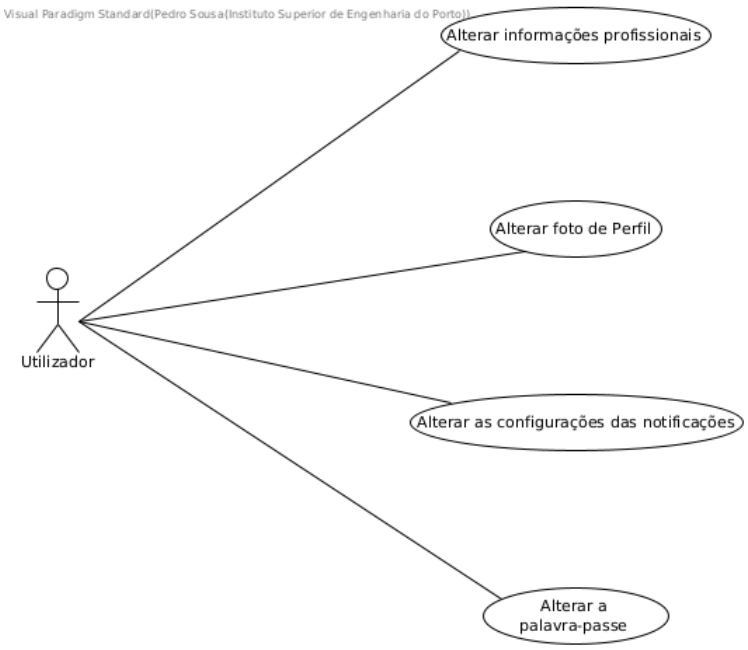

Figura 5.4: Diagrama de caso de uso para as funcionalidades do módulo de Perfil de utilizador

Comparação com Frameworks Alternativas

Na escolha da *framework*, não se pode deixar de contemplar as alternativas possíveis para a migração inicial, daí ser relevante comparar a *Angular, React* e *Vue*, os três *players* mais dominantes no que toca ao espaço no mercado de *frameworks* de aplicações *frontend*. Como é possível observar na Figura 5.5, temos a comparação de popularidade das *frameworks*.

React

1. **Flexibilidade**: O *React*, sendo esta uma biblioteca e não uma *framework* completa, oferece aos seus desenvolvedores uma grande flexibilidade. Isto significa que as equipas podem adaptar o *setup* de desenvolvimento para servir melhor as suas necessidades específicas e potencialmente permitir desenvolver estruturas de *Micro-Frontends* mais refinadas.
2. **Comunidade e Ecossistema**: Esta ferramenta contém uma das maiores comunidades em termos de tecnologias de *frontend*. Isto traduz-se numa panóplia de bibliotecas de terceiros, recursos online em grande quantidade, e uma comunidade ativa e preparada para a resolução de erros e quaisquer problemas, como demonstra a Figura 5.5 a popularidade desta *framework* é bem maior que as restantes.
3. **JSX (JavaScript Syntax Extension) e Composição**: O JSX permite uma forma de construir os componentes UI bastante intuitiva, tornando mais fácil de partir as aplicação em componentes mais pequenos e independentes.
4. **Desvantagens**: Contudo, a grande flexibilidade que o React traz também significa que normalmente não existe uma forma standardizada de completar uma tarefa. Para equipas grandes ou projetos de grande dimensão onde convenções ríspidas são benéficas, podem por vezes ser um obstáculo.

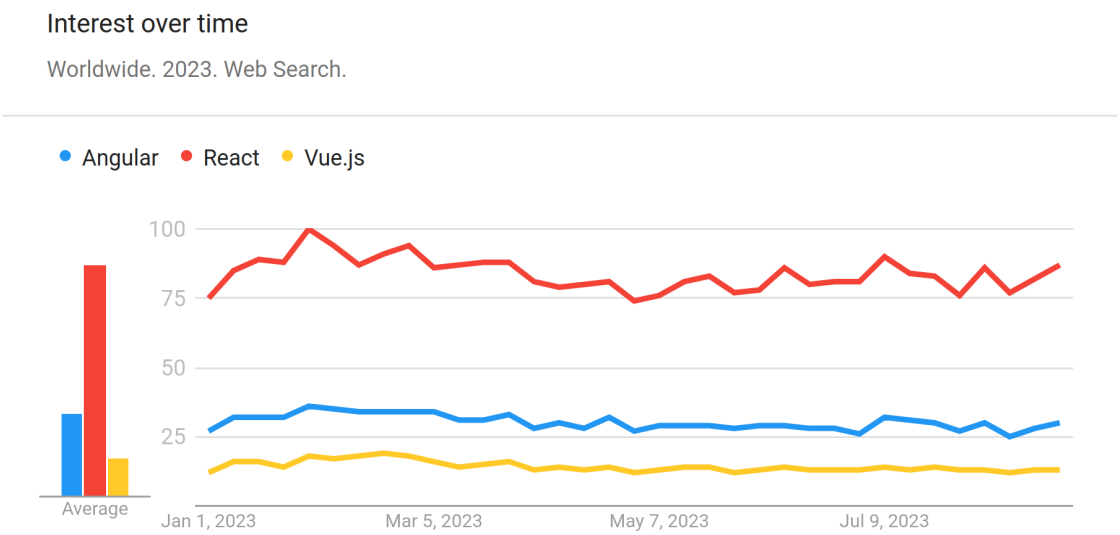

Figura 5.5: Gráfico de comparação de popularidade em termos de pesquisa, com recurso à ferramenta *Google Trends*

Vue

1. **Simplicidade e Integração**: O *Vue* é conhecido pela sua suave curva de aprendizagem e simplicidade. As bibliotecas principais focam-se na camada de visualização apenas, fazendo com que seja mais fácil de aprender e integrar com outras bibliotecas e projetos existentes.
2. **Reatividade**: O sistema de reatividade do *Vue* pode ser visto como um benefício para os *Micro-Frontends*, assegurando que os componentes estão sem em sincronia com as suas fontes de dados.
3. **Single File Components**: O sistema de componentes de ficheiro único pode ser vantajoso para os *Micro-Frontends*, encapsulando o *template*, a lógica e os estilos para um único componente num único ficheiro.
4. **Desvantagens**: O ecossistema do *Vue*, não é tão extenso como o do React. Dependendo das necessidades do projeto, as equipas poderão encontrar menos soluções prontas a usar e iram necessitar de desenvolver uma solução customizada mais regularmente.

Angular (Reiterado)

1. **Solução out-of-the-box**: Ao contrário das ferramentas anteriores *React*, e até certo ponto, *Vue*, o *Angular* é uma *framework* completa que oferece uma variedade de ferramentas e funcionalidades prontas a usar. Isto traduz-se em menos tempo usado a juntar bibliotecas diferentes e ajuda a tomar decisões arquiteturais com uma maior consistência.
2. **Maturidade**: O Angular, por estar há consideravelmente mais tempo no mercado, especialmente considerando as raízes no *AngularJS*, traz um nível de maturidade e estabilidade para cima da mesa. Para um projeto detalhado e complexo como o projeto em questão, a estabilidade é um ponto chave.

Considerando as diferentes características das *frameworks*, optou-se pela escolha do *Angular* pela sua natureza abrangente combinada com a familiaridade da equipa com a sua linguagem e princípios.

5.3.3 Desenvolvimento

Nesta secção irá abordar-se o desenvolvimento do *Micro-Frontend*. O principal objetivo para a migração é assegurar que o *Micro-Frontend* continua consistente com o módulo original. Isto é, procura-se imitar em termos de estilo e comportamento o *Micro-Frontend* já existente.

Preparação do Ambiente do Micro-Frontend

O passo inicial passa pela instalação do *Angular CLI* e da inicialização de um novo projeto *Angular*.

```
npm install -g @angular/cli
```

```
ng new user-profile-microfrontend --strict
```

Com este comando, o *Angular CLI* instala as bibliotecas *npm* necessárias para o Angular e outras dependências. Além disso, a CLI cria um novo *workspace* com uma "*Welcome App*"de exemplo pronta a correr.

Ao correr o comando, são também feitas algumas perguntas no terminal como se o utilizador quer adicionar *angular routing* que tipo formato de *stylesheet* se queria usar. Escolheu-se adicionar *routing* pois era recomendado e escolheu-se o formato de CSS, pois assim facilitaria a migração, pois é o formato já usado na aplicação anterior.

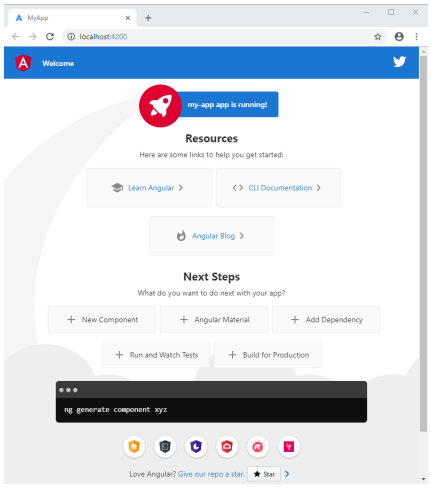

Figura 5.6: Página de boas-vindas da "*Welcome App*"do Angular

Para correr a aplicação basta executar o comando *ng serve*. Este comando cria um processo que levanta o servidor que serve a aplicação na porta configurada nas configurações. Além disto, para facilitar o processo de desenvolvimento, ficheiros da aplicação são observados e caso haja uma alteração é despoletado o processo de compilação da aplicação.

A *Angular CLI* cria uma estrutura de pastas junto com os ficheiros iniciais como é possível ver demonstrado na Figura 5.7. Além desta estrutura inicial, foram adicionadas mais pastas para melhorar a estrutura.

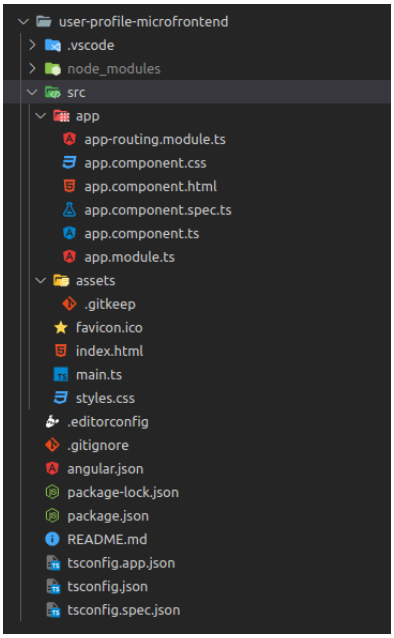

Figura 5.7: Estrutura inicial das pastas do projeto *Angular*

Estrutura de Diretórios

A estrutura de diretórios desempenha um papel importante na manutenção da modularidade da aplicação e para garantir que aplicações de larga escala continuam de fácil leitura para o desenvolvedor.

Nesta secção irá ser abordar a estrutura de diretórios escolhida para este projeto para ser entendido o uso e significância de cada um.

1. "src/app/"
    - "core/": Serviços *singleton* e funcionalidades estruturais
        - "services/": Serviços para serem usados por toda a aplicação
        - "guards/": Route Guards para controlar o acesso a certas partes da aplicação
        - "interceptors/": Interceptores HTTP para transformar os pedidos em respostas
        - "models/": Modelos e interfaces que definem as estruturas de dados de negócio
        - "enums/": Enumerados para serem usados por toda a aplicação
    - "shared/": Contém os componentes e diretivas que são usadas em mais que uma vez na plataforma
        - "components/": Componentes de interface reutilizáveis, como botões e formulários
        - "directives/": Diretivas que adicionam comportamento aos elementos
        - "utils/": Funções utilitárias ou constantes

- "modules/": Módulos das funcionalidades
  - "professional-details/": Módulo para as informações profissionais do utilizador
    - "components/": Componentes da interface para os detalhes profissionais
    - "services/": Serviços específicos para este módulo
  - "change-password/": Módulo para a mudança de palavra-passe do utilizador
  - "notifications/": Módulo para a configuração das notificações do utilizador

  Cada módulo de uma funcionalidade (como "professional-details/") poderá ter de forma similar os seus próprios diretórios de "components/", "services/"e potencialmente os seus próprios "directives/"e "models/".

2. "src/assets/": Neste diretório contém *assets* estáticos como imagens
3. "src/environments/": Contém os ficheiros com as variáveis de ambiente
4. "src/styles/": Contém os ficheiros com os estilos e temas da aplicação

Em suma, uma árvore de diretórios bem estruturada não só ajuda a equipa de desenvolvimento a navegar e manter a base de código, mas também reforça o nível de modularidade de separação de conceitos que é vital para a escalabilidade das aplicações.

Esta estrutura poderá sempre ser ajustada tendo em conta as preferências da equipa.

Arquitetura de Módulos

Estabelecer a arquitetura certa assegura que o código tenha uma alta modularidade e manutenibilidade. Assim sendo, segue-se a descrição de como foi feita a arquitetura do novo *Micro-Frontend*.

Como foi possível já observar pela estrutura de diretórios, foi tomada a decisão de dividir o módulo do Perfil de Utilizador em mais três módulos, referentes às funcionalidades referidas anteriormente.

Vão assim ser criados os componentes: *ProfessionalDetailsModule*, *ChangePasswordModule* e *NotificationsConfigurationModule*; Cada um destes irá encapsular os seus componentes, serviços e roteamentos.

Relativamente ao roteamento, com o uso da componente *RouterModule* do Angular é possível efetuar o roteamento. Será usada também uma funcionalidade chamada *Nested Routes*, que permite criar roteamentos relativos a outros componentes.

```
const routes: Routes = [
  {
    path: 'profile',
    children: [
      { path: 'professional', loadChildren: () => import('./professional-details/professional-details.module').then(m => m.ProfessionalDetailsModule) },
      { path: 'password', loadChildren: () => import('./change-password/change-password.module').then(m => m.ChangePasswordModule) },
      { path: 'notificationsconfiguration', loadChildren: () => import('./notifications/notifications-configuration.module').then(m => m.NotificationsConfigurationModule) }
    ]
  }
];
```

Listing 5.9: Implementação do roteamento

Assim, desta forma, como podemos observar no excerto de código em Listing 5.9, temos um componente pai com o caminho "profile/"que contém três componentes filhos que acrescentam a sua propriedade *path* ao URL e são assim carregados.

Nesta transição, a migração de *AngularJs* com uma abordagem orientada a diretivas para uma tecnologia cuja arquitetura é concentrada em componentes traz uma maior modularidade e encapsulamento de responsabilidades.

As diretivas do *AngularJs* têm a flexibilidade criar um escopo isolado ou então herdar o escopo da diretiva pai, ou seja, é possível isolar as variáveis de cada diretiva ou então partilhá-las com as diretivas filhas.

Os componentes do Angular têm sempre um escopo isolado e, desta forma, é garantido o encapsulamento das variáveis reduzindo os efeitos secundários indesejados. Assim sendo, é adotado um fluxo de dados com tendência a ser mais unidirecional ao invés de bidirecional.

Outra mudança no paradigma é o encorajamento do uso de *templates* HTML externo para tornar o código mais claro.

No que toca à transição em si, examinou-se as diretivas do módulo de Perfil de Utilizador na aplicação existente para aferir quais destes se enquadravam na filosofia de componente do *Angular*. Partindo destes componentes foi possível fazer a transição de forma mais direta.

Para cada componente do *Angular* que fosse necessário ser criado, este era gerado através do *Angular CLI*, executando o seguinte comando:

```
ng generate component [nome-do-componente]
```

Após a geração deste componente, é feita a migração do HTML para o ficheiro de HTML do novo componente bem como os estilos.

No caso da transição para um escopo isolado, o Angular oferece a funcionalidade de *bindings* através dos atributos *@Input()* e *@Output()* do componente do *Angular*, assegurando assim o fluxo de dados para dentro e fora do componente.

Para os casos das diretivas que tinham a função adicionar comportamentos, como por exemplo, validações *custom*, é possível suportar estes criando *directives* com o Angular CLI através do comando:

```
ng generate directive [nome-da-directiva]
```

Na migração do módulo Perfil de Utilizador, não existiam casos em que fizessem sentido migrar para uma *directive*.

Capítulo 6

Avaliação e Comparação

6.1 Introdução

Em qualquer processo de migração de software, especialmente num tão complexo como o retratado neste projeto, é necessário ter em máxima conta a experiência do utilizador final. O utilizador, ao final das contas, é o recetor final de todas as decisões tecnológicas feitas durante a migração.

Este capítulo tem como objetivo explorar as várias abordagens de testes que envolvem a prova de conceito desenvolvida na aplicação com o *Micro-Frontend* migrado. Isto envolve principalmente a execução de testes de aceitação, como também testes de performance e de escalabilidade.

O foco principal será na abordagem de testes de aceitação controlados, usando um pequeno grupo de utilizadores familiarizados com a plataforma. A experiência anterior destes utilizadores torna-se um *Teste padrão usado para aferir o desempenho e eficiência de um hardware ou software (Benchmark)* para comparar com a nova iteração em termos de usabilidade e funcionamento, trazendo valiosas opiniões para o sucesso da migração, num ponto de vista orientado ao utilizador.

6.2 Testes de Aceitação

6.2.1 Valor do Grupo de Utilizadores

A escolha de um grupo controlado de utilizadores, especialmente aqueles já familiarizados com a plataforma, não é apenas uma escolha estratégica, mas também uma escolha pragmática.

Montar um ambiente para testes em larga escala, particularmente numa fase de migração, apresenta desafios complexos e demorados. As complexidades de assegurar que ambos os sistemas correm lado a lado para um vasto número de utilizadores é uma tarefa bastante complexa e requer muito tempo de preparação para evitar qualquer tipo de problemas.

A escolha de um pequeno grupo familiarizado com a plataforma alivia estes desafios e oferece várias vantagens únicas.

Este utilizadores, com a sua experiência na plataforma, têm uma expectativa estabelecida de como as funcionalidades devem funcionar. O seu *feedback* não é apenas baseado na sua primeira impressão e assim torna-se uma análise comparativa com um maior valor.

Além disto, com um pequeno grupo é possível haver uma linha direta de comunicação. Todas as questões, preocupações e observações podem ser expostas em tempo-real, criando uma dinâmica de *feedback loop* que não seria possível com um grande grupo disperso de utilizadores.

6.2.2 Montagem do Ambiente de Testes

Dada a complexidade do projeto, é necessário assegurar que o ambiente de testes é montado corretamente.

O primeiro passo é assegurar que ambas as variáveis estão acessíveis e estáveis.

- Variável A (Controlo): Aplicação *AngularJs* com o modulo Perfil de Utilizador.
- Variável B (Teste): Aplicação *single-spa* com o Micro-Frontend migrado.

É necessário assegurar que as duas variáveis estão em ambientes isolados, e que todas as funcionalidades como atualizar os detalhes profissionais, atualizar a *password* e ajustar as configurações de notificações estão funcionais em ambos os ambientes.

O objetivo é a comparação da nova solução com a antiga de modo a aferir qual das duas oferece uma melhor experiência de utilizador e desempenho.

Relativamente aos utilizadores que irão fazer parte dos testes, terão de ser providos com instruções claras. Existem duas formas de abordar a execução dos testes:

- O grupo ser divido em dois, em que cada subgrupo apenas interage com uma versão. Isto é uma versão mais tradicional dos testes A/B, mesmo para uma escala mais pequena.
- Usar as duas versões sequencialmente, interagindo um tempo específico com uma versão antes de avançar para a próxima. Isto assegura uma perspetiva recente numa variável antes de avançar para a seguinte.

O tempo que os utilizadores vão usar para testar é crucial. É necessário alocar tempo suficiente para haver uma iteração cuidada com as funcionalidades e os módulos, mas não demasiado para o teste não ser demasiado complexo e tedioso.

Dependendo da complexidade e familiaridade dos utilizadores, a sessão deverá demorar entre 15 e 20 minutos.

6.2.3 Definição de Métricas

Antes de entrar no processo de teste, é essencial clarificar quais os aspetos que estão sobre escrutínio. As métricas escolhidas vão guiar os utilizadores sobre em que se deverão focar e dar um *feedback* estruturado.

- **Funcionalidade**: A questão principal para este ponto é "Funciona tudo como é esperado?". Todas as funcionalidades, desde atualizar detalhes a atualizar configurações, devem funcionar corretamente. Quaisquer tipos de bugs devem ser anotados;
- **Experiência de Utilizador**: Para além da funcionalidade, a experiência de utilizador na plataforma é de elevada importância. Esta métrica explora os atributos como consistência de *layout*, responsividade e suavidade na navegação em geral. Dado que

os utilizadores estão familiarizados com a plataforma, estarão na melhor posição para reparar algum tipo de desvios ou melhorias relativamente ao sistema *legacy*;

- **Comparação Direta**: Dado que os utilizadores estão familiarizados com a plataforma, estão numa posição única para comparar o sistema *legacy* com o novo sistema diretamente. Estes devem apontar os pontos de melhoria, que funcionalidades estão consistentes e pontos que se podem ter deteriorado com a migração;
- **Feedback**: Estabelecer as formas pelas quais os utilizadores irão dar o seu *feedback*. Estes podem ser sob a forma de questionários estruturados ou *feedback* em tempo real durante sessões guiadas.

6.2.4 Execução do Teste

Com o ambiente preparado e as métricas definidas, o próximo passo é a execução dos testes. Dado o tamanho do grupo e os objetivos de obter *feedback* detalhado, uma abordagem estruturada, porém também flexível, é crucial.

Sessões Guiadas

Considerando o tamanho do grupo a participar no teste e a importância da migração, as sessões guiadas em tempo-real podem ter um alto valor. Isto envolve a existência de um facilitador que guia o utilizador durante o processo, observando-o enquanto ele navega pelo sistema.

Através da partilha de ecrã, o facilitador pode observar o utilizador enquanto este interage com a aplicação, tomando notas dos padrões de navegação e quaisquer tipos de diferenças que o utilizador sinta.

Os utilizadores são encorajados a pensar em voz alta e a narrar as suas linhas de pensamento e observações. Com este feedback em tempo-real é possível capturar reações imediatas e detalhes subtis que poderiam ser esquecidos no feedback pós-teste.

Qualquer tipo de questões ou incertezas que os utilizadores poderão ter, podem ser resolvidas imediatamente, assegurando que não ficaram presos ou confusos, suavizando assim o processo de teste.

Sessões Independentes

Enquanto as sessões guiadas oferecem *feedback* direto, é igualmente importante deixar os utilizadores navegar pelas funcionalidades ao seu ritmo, sem a possível influência de um facilitador.

Ao deixar os utilizadores interagir com o sistema de forma independente, estes poderiam descobrir caminhos, padrões ou problemas únicos que não surgiriam numa sessão guiada.

O principal objetivo deste tipo de sessões é replicar uma experiência de interação genuína. Ao deixar os utilizadores com os seus dispositivos, vai ser possível ver como estes interagem com a plataforma num cenário de mundo real, sem supervisão.

Ao combinar ambos os tipos de sessões, é feita uma mistura de *feedback* estruturado com utilização orgânica do utilizador. O objetivo continua consistente: entender a experiência do utilizador, comparando o sistema *legacy* com o novo sistema, assegurando que o sistema com uma arquitetura orientada a *Micro-Frontends*, alinha-se com as expectativas e necessidades.

6.2.5 Análise do Feedback

Após as sessões, é necessário trabalhar os resultados. Recolhido o *feedback*, sendo este através das sessões em tempo-real ou por documentos escritos após as sessões independentes, esta documentação reflete a experiência que o utilizador teve ao interagir com as aplicações. Agora será feito o processo de análise.

Organizar o Feedback

O primeiro passo consiste em organizar o *feedback* meticulosamente. Organizar os comentários deixados pelas métricas definidas como funcionalidade, experiência de utilizador e comparação direta. Esta abordagem estruturada, ajuda-nos a categorizar cada comentário dos utilizadores para métricas a analisar.

Segmentação de Funcionalidades Separar o *feedback* relacionado com as *tabs* ou funcionalidades, como os Detalhes Profissionais, Atualizar *Password*, e Configuração de Notificações, que digam respeito a algum problema ou comentário a um módulo específico.

Prioridades Algum do *feedback* terá mais relevância que outro. Posto isto, é crucial que haja uma ordenação baseada em urgência ou impacto, destingindo entre pequenos problemas de usabilidade e grande problemas funcionais.

Análise Comparativa

Com o *feedback* organizado, o próximo passo é a fazer uma análise comparativa com a linha de base, que neste caso é a aplicação *AngularJs Legacy*.

Nesta análise, é fulcral sublinhar quaisquer diferenças na funcionalidade, *layout* e experiência de utilizador em geral entre as duas soluções.

Além disto, por muito que o foco esteja nas melhorias encontradas, é igualmente de elevada importância reconhecer qualquer área onde a nova solução possa ter deteriorado em relação à solução *legacy*. Isto ajuda-nos a perceber onde é possível fazer melhorias como também evitar problemas inesperados na implementação da solução.

Feita a análise, é importante converter o *feedback* em ações de melhoria concretas.

Qualquer tipo de bug ou problema funcional detetado durante os testes deve ser corrigido de imediato. Baseado no *feedback*, deve ser criado um plano de melhoria para refinar a experiência de utilizador e adicionar valor onde possível.

6.2.6 Resultados

Foram efetuadas as sessões a dez utilizadores familiarizados com a plataforma. Para recolher o *feedback* dos utilizadores para aferir as métricas, foi desenhado um formulário para estruturar os inputs de forma a organizar melhor os dados e obter as melhores informações dos utilizadores.

Abaixo estão descriminadas algumas das métricas que foram incluídas no formulário.

Fidelidade de Interface

- **Definição de Métrica**: A métrica a definir é a precisão com que as interfaces foram replicadas do sistema original.
- **Exemplo de Questão**: O exemplo de questão seria, definir numa escala de 1 a 10 se a interface do novo sistema replica a aparência do sistema original.
- **Interpretação**: Em termos de interpretação, pontuações altas indicam que os utilizadores entendem que a nova interface é quase ou até mesmo idêntica à original, enquanto pontuações baixas indicam que há diferenças notáveis entre as interfaces.

Satisfação de Funcionalidade

- **Definição de Métrica**: A métrica definida é o grau de como a funcionalidade que foi migrada do sistema, replica a original.
- **Exemplo de Questão**: Um exemplo de questão seria: "Foi possível executar todas as tarefas com a mesma facilidade do sistema original?"(Opções: Sim, Maioritariamente, Algumas tarefas foram mais difíceis, Não).
- **Interpretação**: Respostas positivas indicam que existe paridade funcional, enquanto respostas negativas serão necessárias ser analisadas para identificar tarefas que podem ser problemáticas.

Satisfação no Desempenho

- **Definição de Métrica**: A métrica a analisar aqui é o desempenho do sistema migrado.
- **Exemplo de Questão**: Um exemplo de pergunta seria: "Numa escala de 1 a 10, como classificaria o grau de satisfação com o desempenho da aplicação (velocidade e responsividade)?".
- **Interpretação**: Em termos de interpretação, pontuações altas indicam uma satisfação na performance do sistema, enquanto pontuações baixas indicam problemas de desempenho e áreas a melhorar.

Alinhamentos e Consistência

- **Definição de Métrica**: Avaliação do alinhamento e consistente dos diferentes componentes da interface.
- **Exemplo de Questão**: Exemplo: "Reparou em algum desalinhamento ou inconsistência na interface como botões, texto ou a barra de navegação?"(Opções: Sim – Especificar, Não).
- **Interpretação**: Relativamente a interpretação, quaisquer afirmações e especificações de instâncias aqui iriam indicar áreas necessárias a refinar os alinhamentos e consistências.

Experiência Geral

- **Definição de Métrica**: A experiência geral do utilizador em nível de satisfação com o sistema migrado.

- **Exemplo de Questão**: "Como avaliaria a experiência geral com o novo sistema comparado com o original? (Opções: Muito melhor, Ligeiramente Melhor, Igual, Um pouco pior, Muito Pior).
- **Interpretação**: Isto daria uma vista holística da satisfação do utilizador e da preferência entre o sistema original e o migrado.

Análise e Interpretação das Respostas

No ponto de vista de uma análise quantitativa, diz respeito às respostas de valor numérico ou baseadas numa escala, será utilizado o valor médio ou então analisadas estatisticamente de forma a obter informação com valor.

Relativamente a uma análise qualitativa, comentários ou feedback dado pelos utilizadores, serão revistos para perceber a real opinião dos utilizadores, bem como problemas encontrados e áreas de melhoria.

As análises das respostas podem conduzir à identificação de ações de melhoria necessárias para melhorar a fidelidade, desempenho e experiência geral do utilizador no sistema migrado.

Interface

- **Pontuação média**: 8.5 de 10
- Comentários comuns:
  - "Maioria dos componentes estão idênticos, fora a barra de navegação e alguns alinhamentos."
  - "O tamanho da fonte parece maior no novo sistema."
- Interpretação: Os utilizadores em geral acham que a interface da nova aplicação está fiel à aplicação *legacy* com pequenas discrepâncias.

Funcionalidade

-
  - Sim: 70%
  - Maioritariamente: 20%
  - Algumas tarefas foram mais difíceis: 10%
  - Não: 0%
- Comentários Comuns:
  - "A funcionalidade é bastante idêntica nas duas aplicações."

Desempenho

- **Pontuação média**: 9.2 de 10
- Comentários Comuns:
  - "Ambas as aplicações são equivalentes a nível de desempenho."
  - "Os tempos de carregamento estão ligeiramente mais rápidos."

Alinhamentos e Consistência

- **Respostas**:
  - Sim: 30%
  - Não: 70%
- Comentários Comuns:
  - "Os itens da barra de navegação não estão alinhados."
  - "Alguns elementos de texto estão desalinhados na aba dos Detalhes Profissionais."

Experiência Geral

- **Respostas**:
  - Muito Melhor: 20%
  - Ligeiramente Melhor: 50%
  - Igual: 20%
  - Um pouco pior: 10%
  - Muito Pior: 0%
- Comentários Comuns:
  - "A experiência geral é positiva, mas corrigindo alguns pequenos detalhes ficaria perfeito."

Análise Quantitativa

Através de feedback quantitativo, foi obtida informação mensurável confirmando semelhanças visuais e funcionais entre os dois sistemas.

Ambos os sistemas foram considerados visualmente congruentes, com uma pontuação de fidelidade visual de 85%. Contudo, alguns problemas isolados relativamente a alinhamentos e problemas com a barra de navegação foram identificados no Micro-Frontend Angular.

Relativamente a desempenho, não mostraram algum tipo de diferença a nível de performance entre ambos os sistemas, reforçando o sucesso da migração em termos funcionais.

Análise Qualitativa

Os comentários dos utilizadores trouxeram informações importantes relativamente às semelhanças e discrepâncias entre as duas aplicações.

O grupo de utilizadores expressou uma grande satisfação no uso das duas plataformas, sublinhando não haver diferenças de desempenho percetíveis. No geral, o consenso foi positivo, refletindo uma apreciação na similaridade visual mantendo o nível de desempenho.

Contudo, os comentários também realçaram problemas com o alinhamento de componentes visuais e inconsistências na barra de navegação dentro do Micro-Frontend.

6.2.7 Análise Correlativa

Efetuando as correlações quantitativas e qualitativas obtidas, deu a oportunidade de encontrar padrões e identificar ações corretivas a efetuar.

Os alinhamentos de certos componentes e os problemas com a barra de navegação identificados pela análise quantitativa estão diretamente correlacionados com as preocupações levantadas pelos utilizadores, sublinhando que estas seriam as áreas que necessitam de melhoria imediata.

O estudo correlativo inferiu que enquanto a satisfação geral dos utilizadores foi alta devido à grande paridade visual e de desempenho, corrigir as pequenas discrepâncias é crucial para melhorar a solução implementada e consequentemente a experiência do utilizador.

6.2.8 Feedback da Aplicação

Os comentários obtidos têm sido um instrumento de delineamento das estratégias de melhoria da prova de conceito.

Ações imediatas devem ser tomadas para colmatar as inconsistências nos alinhamentos e problemas com a barra de navegação no *Micro-Frontend Angular*. Apontar para uma experiência visual congruente com o sistema *legacy*.

A análise dos resultados revela satisfazer os utilizadores com o visual e funcionamento congruente do *Micro-Frontend Angular*. A nova solução manteve os níveis de desempenho com a aplicação *legacy* tornando a prova de conceito numa migração bem sucedida.

Contudo, foram identificadas algumas discrepâncias ao nível de alinhamentos e comportamento da barra de navegação, sublinhando a importância da perfeição visual e atenção ao detalhe na migração de sistemas.

O *feedback* detalhado e os padrões identificados são algumas das luzes que guiam para melhorias possíveis e futuros trabalhos, assegurando o alinhamento com o sistema *legacy* e as expectativas dos utilizadores.

Casando os comentários dos utilizadores com a precisão do alinhamento visual, o processo de migração irá ser refinado para cumprir a satisfação dos utilizadores e entregar uma experiência de utilização familiar.

6.3 Testes de Desempenho

6.3.1 Propósito e Importância

É necessário com a migração efetuada assegurar que a solução não regride em termos de desempenho. *Google Lighthouse*, uma ferramenta bem estabelecida na área de desempenho Web, permite monitorizar a performance, acessibilidade, melhores práticas, *SEO* e muito mais. Ao aplicar o *Lighthouse*, é garantido que o Micro-Frontend não piora em termos de performance e cumpre as expectativas do utilizador.

	Legacy	Micro-Frontend
Performance	53	55
First Contentful Paint	5.9s	4.1s
Speed Index	6.1s	4.9s
Total Blocking Time	0ms	0 ms
Comulative Layout Shift	0.09	0.08

Tabela 6.1: Resultados dos testes efetuados com a ferramenta Google Lighthouse

6.3.2 Definição de Cenários de Teste e Métricas

O foco principal é executar a auditoria do *Lighthouse* na página de perfil de utilizador em ambas as soluções. A comparação direta dá a entender o quão bem otimizada a solução com o *Micro-Frontend* está relativamente ao sistema *legacy*.

Nesta abordagem comparativa, as seguinte métricas do *Lighthouse* vão ter um papel importante:

- **First Contentful Paint(FCP)**: Tempo necessário para os utilizadores verem o conteúdo inicial.
- **Speed Index**: Esta métrica revela a rapidez de renderização do conteúdo em ambas as versões.
- **Total Blocking Time**: Métrica que indica quando é que o utilizador pode começar a interagir com os elementos da página.
- **Comulative Layout Shift**: Métrica que revela a estabilidade visual de ambas as versões
- **Performance**: Índice que junta todas as métricas anteriores numa pontuação de 0 a 100

6.3.3 Analise Comparativa

Após a execução da auditoria do *Google Lighthouse* em ambas as soluções, os resultados obtidos estão representados na Tabela 6.1. Esta informação oferece uma imagem clara de como é que as duas versões se comparam em termos de desempenho. Segue-se a análise comparativa:

- **Performance**: O Micro-Frontend tem uma pontuação ligeiramente maior por 2 pontos. Enquanto as duas pontuações estão dentro de um gama similar, indica que a migração manteve um nível de performance comparável. A superioridade do Micro-Frontend sugere que algumas otimizações ou práticas mais atuais melhoram ligeiramente o desempenho geral.
- **First Contentful Paint (FCP)**: FCP, mede o tempo para a primeira peça de conteúdo aparecer, mostra uma melhoria notável no Micro-Frontend. Os utilizadores irão reparar na nova solução a ser carregada mais rapidamente, melhorando a experiência de utilizador.
- **Speed Index**: O *Speed Index* indica quão rápido o conteúdo se torna visível. O Micro-Frontend ultrapassa o sistema *legacy* por 1.2 segundos, sugerindo que os utilizadores vão experienciar uma interação responsiva e mais imediata na nova versão.

- **Total Blocking Time**: Ambas as soluções apontaram 0 milissegundos no *total blocking time*. Isto indica que os utilizadores não vão experienciar qualquer tipo de atraso.
- **Cumulative Layout Shift (CLS)**: CLS mede a estabilidade visual em que pontuações mais baixas são preferíveis. Ambas as versões têm valores baixos, com o *Micro-Frontend* a ter uma pontuação marginalmente melhor.

6.3.4 Conclusão

A transição para uma arquitetura orientada a *Micro-Frontends*, não apenas preservou os padrões de desempenho existentes, mas também, em algumas áreas, superou-os. Enquanto algumas melhorias em métricas como FCP e *Speed Index* foram mais tangíveis e serão percetíveis para os utilizadores finais, é também importante notar que algumas métricas base como *Total Blocking Time* e CLS continuam boas em ambas as versões. Em suma, a migração para o *Micro-Frontend* atingiu o objetivo de replicar a funcionalidade da aplicação *legacy* enquanto beneficia das práticas web modernas para elevar marginalmente o desempenho.

Capítulo 7

Conclusão

Com a conclusão da prova de conceito da migração de uma aplicação com uma arquitetura Monolítica *AngularJs* para uma Arquitetura Orientada a *Micro-Frontends* com a *framework Single SPA*, é imperativo refletir sobre as experiências, novas perspetivas obtidas e desafios enfrentados.

7.1 Objetivos Atingidos

Refletindo sobre o desenvolvimento do projeto, é possível verificar a realização dos objetivos propostos.

Um dos principais objetivos propostos foi fazer um estudo para entender as potenciais alternativas viáveis na migração da aplicação *legacy* para um paradigma de *Micro-Frontends*.

Foi conduzida uma pesquisa, levando em consideração diversas técnicas, tecnologias e as melhores práticas da indústria. Esse estudo não só orientou a seleção da estratégia de migração, mas também proporcionou *insights* sobre possíveis obstáculos e desafios, juntamente com suas respetivas soluções. Isto permitiu entrar na fase de desenvolvimento com uma compreensão mais consolidada.

Entendendo a complexidade da migração, era imperativo a divisão da migração em várias etapas. Esta abordagem assegura que cada fase recebe a atenção necessária ao detalhe, permite que se mantenha continuidade operacional e mitiga riscos. Adotando a metodologia faseada através do *strangler pattern*, é também facilitado o *feedback* iterativo, assegurando que as lições aprendidas no primeiro passo têm efeito nos restantes.

Antes de haver um comprometimento com alguma estratégia, era crucial validar a hipótese e perceber as implicações no mundo real. A prova de conceito foi executada e simulou-se a nova arquitetura, entendendo os pontos fortes e as áreas a melhorar. Esta implementação preliminar serviu como uma comprovação da real exequibilidade da estratégia de migração.

Com a prova de conceito desenvolvida, procedeu-se à fase empírica de avaliação. A nova arquitetura foi comparada com a atual, focando em métricas críticas de *performance* e qualidade. Análises quantitativas revelaram as características de desempenho de ambas as arquiteturas, enquanto as qualitativas mostraram a apreciação do utilizador final da nova solução comparada com a atual.

Entretanto, é importante salientar que os testes de manutenabilidade, conforme inicialmente proposto, não puderam ser conduzidos devido a restrições de tempo. Os testes realizados enfatizaram claramente as melhorias proporcionadas pela nova arquitetura, ao mesmo tempo que identificaram áreas que necessitam de aprimoramento. Essa limitação temporal

apresentou-se como um desafio significativo na condução abrangente da avaliação, e é um aspeto que merece consideração futura em trabalhos subsequentes.

7.2 Desafios e Limitações

7.2.1 Migração do Bower para o npm

Um dos problemas enfrentados que mais impactaram o desenvolvimento do projeto, foi a transição de gestores de dependências de forma a adotar um ecossistema moderno.

Migrar do *Bower* para o *npm* não foi uma tarefa simples, um dos maiores desafios enfrentados foi a migração de todas as dependências do *Bower* para *npm* . Dadas as diferentes estruturas do *bower.json* e o *package.json*, e a não existência de pacotes idênticos para todas as dependências já existentes.

Estas diferenças a nível de pacotes entre as duas ferramentas pediam uma maior reconfiguração.

A resolução deste problema passava pela instalação de novas dependências e a necessidade de garantir que todos os casos com estas dependências funcionavam de forma idêntica à pré-existente. No caso de não serem idênticas seria necessário fazer adaptações, o que adicionaria ainda mais complexidade à migração.

Enfrentadas estas dificuldades na transição para o *npm*, foi efetuada a decisão de efetuar a migração para *Yarn* que acabou por ser uma alternativa mais eficiente. A ferramenta *bower-away* faz com que o processo de migração seja mais simples. A conversão direta do *bower.json* para a estrutura do *Yarn* elimina muitas das dificuldades que existiam com a migração para o *npm*.

Relativamente à decisão de procurar uma alternativa ao *npm*, a execução da alternativa catalisou os desafios que estavam a ser enfrentados. É essencial reconhecer quando um caminho escolhido poderá não ser o mais eficiente, o *Yarn* providenciou uma transição mais direta, assegurando a estabilidade do projeto.

7.2.2 Integração do Webpack com o AngularJs e o ocLazyLoad

O *Webpack* tem surgido como uma ferramenta popular nos projetos de *frontend* modernos, conhecido pelas suas capacidades de compilação. Contudo, a sua integração com ferramentas antigas, neste caso o *AngularJs* trouxe algumas dificuldades.

O grande problema enfrentado foi a tentativa de implementar o *Webpack* com o módulo do *ocLazyload*, que é usado para fazer o *lazy-loading* dos componentes da aplicação.

Para clarificar, o *ocLazyLoad*, é um módulo do *AngularJS* que permite aos desenvolvedores fazerem o *lazy-load* dos componentes e serviços. Isto melhora significativamente os tempos de carregamento da aplicação e assegura que apenas os componentes necessários são carregados, ao invés de tudo ser carregado.

O *Webpack* opera compilando todo o código da aplicação em pacotes consolidados, como já explicado no Capítulo 2. Porém, no ficheiro de configuração de *lazy-load*, o *Webpack* não estava a conseguir carregar os ficheiros, provavelmente por uma questão de incompatibilidade, o que impossibilitou a implementação desta ferramenta.

Esta experiência ressaltou as complexidades inerentes à mistura de duas *frameworks* de eras diferentes. Enquanto o *AngularJS* e o *ocLazyLoad* foram projetados para uma filosofia diferente de desenvolvimento web, o *Webpack* representa o desenvolvimento moderno das arquiteturas *frontend* modulares e escaláveis.

Fazer a ponte entre eles exigiu entender que por vezes as migrações graduais não são possíveis ou exequíveis, o que nos obriga a refatorar a base de código existente para o bem maior da aplicação.

Esta tentativa de integração do *Webpack* com uma aplicação *AngularJs* já existente fez lembrar que, os desenvolvedores enfrentam desafios quando existe o esforço para modernizar os sistemas *legacy*. Estes esforços, embora exigentes, são passos essenciais e necessários para garantir que as aplicações mantenham o desempenho e a manutenabilidade e em sincronia acompanhar a evolução das melhores práticas do domínio do desenvolvimento *web*.

7.3 Trabalho Futuro

A natureza tecnológica do projeto está em constante evolução e, reconhecendo os objetivos cumpridos, é necessário ainda assim olhar o futuro do projeto. Assim sendo, esta secção tem o objetivo de enumerar as áreas de melhoria para continuar a desenvolver a implementação da arquitetura orientada a *Micro-Frontends* e antecipar os desafios futuros.

Enquanto a migração inicial deixou uma base sólida, ainda existe a necessidade de migrar continuamente toda a aplicação para que esta se torne um sistema *Micro-Frontend* completo.

O cenário das ferramentas usadas para a aplicação da arquitetura é bastante dinâmico, com novas soluções a emergir para resolver problemas específicos. É necessário continuar atento a estes desenvolvimentos, avaliando e integrando ferramentas que possam melhorar o processo de desenvolvimento, migração, *Processo de tornar uma aplicação software acessível e operação num ambiente específico (Deployment)* e monitorização de processos.

Dada a natureza modular desta arquitetura, existe um potencial de adotar uma variedade de tecnologias distintas nos diferentes módulos. Isto apresenta uma oportunidade para a equipa de desenvolvimento e permite diversificar as suas aptidões.

Com o crescente número de clientes a usar a plataforma, é necessário tornar como prioridade o foco na escalabilidade e resiliência. Isto assegura o crescimento sustentável do número de utilizadores, permitindo que a aplicação consiga gerir o aumento da carga enquanto mantém uma consistente qualidade na experiência do utilizador.

O processo de *deployment* é crucial para o sucesso da arquitetura. É necessário explorar a possibilidade de integrar novos processos de *deployment* e dar ênfase aos paradigmas CI/CD, para proporcionar um lançamento mais rápido de novas versões.

7.4 Contribuições

O trabalho de migração de uma aplicação *frontend* monolítica para uma arquitetura orientada a *Micro-Frontends* foi uma fonte de conhecimento repleta de aprendizagem e desafios. Refletindo no trabalho desenvolvido, destacam-se algumas contribuições que reformularam a abordagem de desenvolvimento e de entrega do produto.

A migração sublinhou a realidade que a escalabilidade, tradicionalmente considerada predominante no âmbito dos processo de *backend*, exige uma deliberação igual nos processos de *frontend*. Ao migrar para uma orientação a *Micro-Frontends*, foi instituído um ambiente que fomenta a evolução modular. Este ambiente antecipa que os componentes progridam de forma independente, assegurando que a aplicação se mantém ágil e reativa a mudanças de requisitos.

Consequente a esta abordagem modular, espera-se uma melhoria na eficiência do desenvolvedor. A capacidade de se concentrar apenas em segmentos específicos da aplicação irá acelerar os processos de desenvolvimento. As revisões, agora confinadas ao domínio do respetivo módulo, garantem um escrutínio meticuloso, conduzindo a iterações mais curtas e rápidas e à identificação criteriosa de erros.

O desempenho, a métrica crítica para a experiência do utilizador, testemunhou melhorias tangíveis após a migração. As apreensões preliminares foram atenuadas à medida que os dados empíricos revelavam que, quando bem arquiteturados, os *Micro-Frontends* podiam igualar ou até ultrapassar o desempenho do sistema monolítico.

Enquanto o atual *setup* de *Micro-Frontend* usa uma gama de tecnologias consistente, a porta está aberta para o uso de várias tecnologias no futuro, desenvolvidas para cumprir a necessidades de módulos específicos.

Em conclusão, a escolha da abordagem da arquitetura orientada aos *Micro-Frontends* não foi apenas uma escolha técnica, mas foi também uma escolha estratégica. Reflete o compromisso de oferecer um sistema modular, adaptável e em linha com as atuais melhores práticas de desenvolvimento. A aplicação está melhor posicionada para ultrapassar desafios futuros e entregar um valor consistente aos utilizadores.

7.5 Reflexão final

Em suma, esta tese explorou a transição de um sistema monolítico para uma arquitetura de *Micro-Frontends*. Durante a elaboração desta tese, desenvolveu-se um estudo sobre as opções de migração, elaborou-se um planeamento e a eventual implementação.

A principal conclusão a retirar é o valor dos *Micro-Frontends* para o desenvolvimento moderno. Esta abordagem, enquanto prometia escalabilidade e manutenabilidade, apresentava vários desafios, especialmente quando o objetivo é espelhar as funcionalidades *legacy* tendo que conciliar tecnologias modernas com tecnologias antiquadas. Ao longo deste estudo, foi observado um balanço entre estas vantagens e desafios.

O *feedback* do nosso grupo de utilizadores teve um papel importante na avaliação do sucesso da migração. A sua familiaridade com o sistema original trouxe uma perspetiva crítica para a eficácia da migração. Mesmo com os problemas detetados, os pontos de melhoria tiveram um maior peso na balança.

Os testes efetuados à solução, quer de aceitação como os de desempenho, sublinharam que é necessária uma otimização contínua, para assegurar que existe uma evolução tecnológica e que o utilizador continua a ter uma boa experiência.

Esta tese sublinha a importância da adaptabilidade de um panorama em constante evolução como o desenvolvimento *web*. Evidencia a importância da inovação, melhoria contínua, e compromisso na entrega da melhor experiência possível ao utilizador.

Para concluir, para além dos conhecimentos académicos e profissionais, este trabalho foi fundamental para desenvolver outras competências, proporcionando um desafio prático para aprender e crescer. As descobertas e experiências aqui encapsuladas oferecem não apenas uma base de futuros desafios tecnológicos, mas também um testemunho da evolução académica e profissional do estudante.

Bibliografia

AngularJS: Miscellaneous: Version Support Status (2023). url: https://docs.angularjs.org/misc/version-support-status (acedido em 28/01/2023).

Apache JMeter - Apache JMeter™ (2023). url: https://jmeter.apache.org/ (acedido em 03/10/2023).

Arora, A e M Sinha (2012). «Web Application Testing: A Review on Techniques, Tools and State of Art». Em: *ISS N* 3.2.

Avram, Abel e Floyd Marinescu (2006). *Domain-Driven Design Quickly: A Summary of Eric Evans' Domain-Driven Design*. Enterprise Software Development Series. C4Media. 96 pp. isbn: 978-1-4116-0925-9.

Bower (2023). url: https://bower.io/ (acedido em 03/10/2023).

Bv, Omnext (2014). «An Omnext white paper on software quality». Em.

Code of Ethics | IEEE Computer Society (2023). url: https://www.computer.org/education/code-of-ethics (acedido em 25/02/2023).

Dhiman, Shikha e Pratibha Sharma (2016). «Performance Testing: A Comparative Study and Analysis of Web Service Testing Tools». Em.

Engineering, GitHub (6 de set. de 2018). *Removing jQuery from GitHub.com frontend*. The GitHub Blog. url: https://github.blog/2018-09-06-removing-jquery-from-github-frontend/ (acedido em 08/02/2023).

Evans, Eric (2014). *Domain-driven design: tackling complexity in the heart of software*. 20. print. Addison-Wesley. isbn: 0321125215; 9780321125217. url: libgen.li/file.php?md5=04d80385c9e9bccd2e076a183b003766.

Formento, Hector Ricardo et al. (1 de set. de 2013). «KEY FACTORS FOR A CONTINUOUS IMPROVEMENT PROCESS». Em: *Independent Journal of Management & Production* 4.2, pp. 391–415. issn: 2236-269X. doi: 10.14807/ijmp.v4i2.76. url: http://www.ijmp.jor.br/index.php/ijmp/article/view/76 (acedido em 11/09/2023).

Fowler, Martin (19 de jun. de 2019). *Micro Frontends*. martinfowler.com. url: https://martinfowler.com/articles/micro-frontends.html (acedido em 06/02/2023).

Geers, Michael (2020). *Micro Frontends in action*. OCLC: on1143652333. Shelter Island: Manning. 276 pp. isbn: 978-1-61729-687-1.

Getting Started with single-spa | single-spa (2023). url: https://single-spa.js.org/docs/getting-started-overview (acedido em 27/06/2023).

«International Journal of advanced studies in Computer Science and Engineering» (2014). Em: 3.10.

Jadhav, Madhuri A, Balkrishna R Sawant e Anushree Deshmukh (2015). «Single Page Application using AngularJS». Em: 6.

Joel Spolsky (6 de abr. de 2000). *Things You Should Never Do, Part I*. Joel on Software. url: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/ (acedido em 29/01/2023).

k6 Documentation (2023). url: https://k6.io/docs (acedido em 03/10/2023).

Kaufmann, Emilie, Olivier Cappé e Aurélien Garivier (13–15 Jun de 2014). «On the Complexity of A/B Testing». Em: *Proceedings of The 27th Conference on Learning Theory*. Ed. por Maria Florina Balcan, Vitaly Feldman e Csaba Szepesvári. Vol. 35. Proceedings of Machine Learning Research. Barcelona, Spain: PMLR, pp. 461–481. url: https://proceedings.mlr.press/v35/kaufmann14.html.

Koen, Peter A et al. (s.d.). «Effective Methods, Tools, and Techniques». Em: *The PDMA ToolBook for New Product Development* ().

Le, Duc Minh et al. (dez. de 2022). «Generating Multi-platform Single Page Applications: A Hierarchical Domain-Driven Design Approach». Em: *The 11th International Symposium on Information and Communication Technology*. SoICT 2022: The 11th International Symposium on Information and Communication Technology. Hanoi Vietnam: ACM, pp. 344–351. isbn: 978-1-4503-9725-4. doi: 10.1145/3568562.3568566. url: https://dl.acm.org/doi/10.1145/3568562.3568566 (acedido em 31/01/2023).

Lighthouse (7 de out. de 2023). original-date: 2016-03-08T01:03:11Z. url: https://github.com/GoogleChrome/lighthouse (acedido em 07/10/2023).

Locust Documentation — Locust 2.16.1 documentation (2023). url: https://docs.locust.io/en/stable/ (acedido em 03/10/2023).

Mezzalira, Luca (17 de nov. de 2021). *Building Micro-Frontends*. Google-Books-ID: NDpPEAAAQBAJ. "O'Reilly Media, Inc." 337 pp. isbn: 978-1-4920-8296-5.

Microservices adoption level worldwide 2021 (2023). Statista. url: https://www.statista.com/statistics/1233937/microservices-adoption-level-organization/ (acedido em 25/02/2023).

Most Popular Web Browsers in 2023 [Jun '23 Update] | Oberlo (2023). url: https://www.oberlo.com/statistics/browser-market-share (acedido em 19/07/2023).

Newman, Sam (s.d.). «Monolith to Microservices». Em: ().

Ponce, Francisco, Gaston Marquez e Hernan Astudillo (nov. de 2019). «Migrating from monolithic architecture to microservices: A Rapid Review». Em: *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. 2019 38th International Conference of the Chilean Computer Science Society (SCCC). Concepcion, Chile: IEEE, pp. 1–7. isbn: 978-1-72815-613-2. doi: 10.1109/SCCC49216.2019.8966423. url: https://ieeexplore.ieee.org/document/8966423/ (acedido em 05/02/2023).

Proko, Eljona e Ilia Ninka (s.d.). «Analyzing and Testing Web Application Performance». Em: ().

Rich, Nick e Matthias Holweg (2000). «Value Analysis Engineering». Em.

School of Computing Universiti Utara Malaysia Kedah, Malaysia e Haroon Shakirat Oluwatosin (2014). «Client-Server Model». Em: *IOSR Journal of Computer Engineering* 16.1, pp. 57–71. issn: 22788727, 22780661. doi: 10.9790/0661-16195771. url: http://www.iosrjournals.org/iosr-jce/papers/Vol16-issue1/Version-9/J016195771.pdf (acedido em 01/02/2023).

single-spa | single-spa (2023). url: https://single-spa.js.org/ (acedido em 16/02/2023).

What is Lazy Loading | Lazy vs. Eager Loading | Imperva (2023). Learning Center. url: https://www.imperva.com/learn/performance/lazy-loading/ (acedido em 18/02/2023).

Zdravkova, Katerina e Lasko Basnarkov, eds. (2022). *ICT Innovations 2022. Reshaping the Future Towards a New Normal: 14th International Conference, ICT Innovations 2022, Skopje, Macedonia, September 29 – October 1, 2022, Proceedings*. Vol. 1740. Communications in Computer and Information Science. Cham: Springer Nature Switzerland. isbn: 978-3-031-22791-2 978-3-031-22792-9. doi: 10.1007/978-3-031-22792-9. url: https://link.springer.com/10.1007/978-3-031-22792-9 (acedido em 28/01/2023).