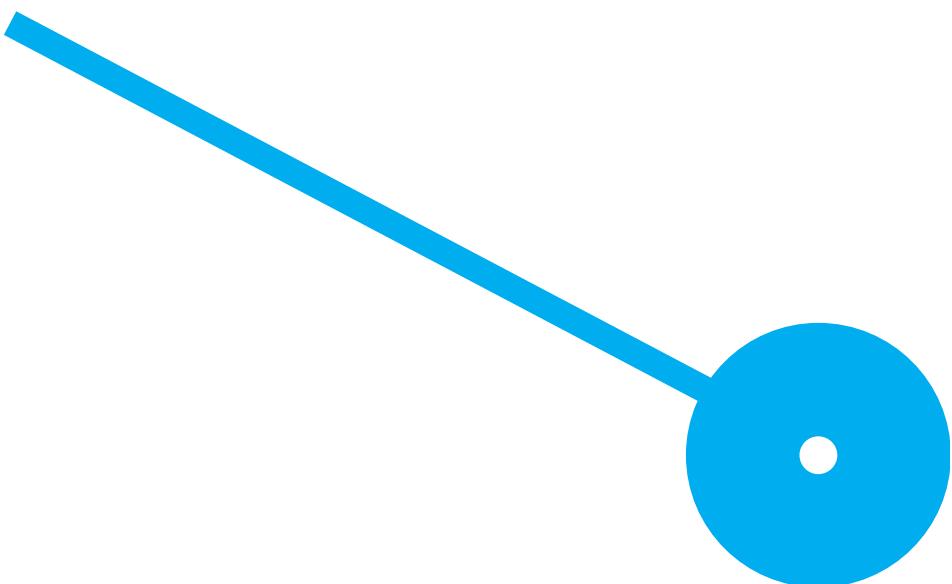


Software de Controlo e Gestão de Obra

Hugo Ricardo Almeida Guimarães

09/2025



Software de Controlo e Gestão de Obra

Hugo Ricardo Almeida Guimarães

8220337

Orientador

Professor Doutor Bruno Moisés Teixeira de Oliveira

Relatório de Estágio apresentado para cumprimento dos requisitos necessários à obtenção do grau de Licenciado em Engenharia Informática pela Escola Superior de Tecnologia e Gestão do Instituto Politécnico do Porto.

09/2025

Declaração de Integridade

Eu, Hugo Ricardo Almeida Guimarães, estudante nº 8220337, da Licenciatura de Engenharia Informática da Escola Superior de Tecnologia e Gestão do Instituto Politécnico do Porto, declaro que não fiz plágio nem autoplágio, pelo que o trabalho intitulado “Software de Controlo e Gestão de Obra” é original e da minha autoria, não tendo sido usado previamente para qualquer outro fim. Mais declaro que todas as fontes usadas estão citadas, no texto e na bibliografia final, segundo as regras de referenciamento adotadas na instituição.

Resumo

O presente relatório descreve o trabalho realizado no âmbito da unidade curricular de Projeto Final da Licenciatura em Engenharia Informática, concretizado em contexto de estágio na empresa DiRoots. O principal objetivo foi o desenvolvimento de uma aplicação web de controlo e gestão de obra, com uma arquitetura modular baseada em microserviços. O projeto envolveu a criação de diversos microserviços no *backend*, com base em *.NET*, bem como a implementação de um *frontend* funcional em *React*, seguindo os padrões visuais e arquiteturais definidos pela empresa.

Apesar de ter sido fornecido um *template* pela empresa, tanto para o *backend* como para o *frontend*, foi necessário adaptar, expandir e implementar funcionalidades específicas, garantindo a integração com bibliotecas internas não documentadas. Uma das particularidades da abordagem da DiRoots é a ausência de chaves estrangeiras e de mecanismos tradicionais de mensagens entre microserviços, substituídos por validações a nível da aplicação e eventos internos síncronos geridos pelo *MediatR*. Esta decisão permitiu alcançar simplicidade e consistência, mantendo o controlo da lógica de negócio no interior de cada serviço.

Agradecimentos

Antecipadamente, gostaria de agradecer à empresa DiRoots pela oportunidade de poder aplicar os conhecimentos adquiridos ao longo da minha formação a partir da realização de um projeto. Em particular, expresso a minha gratidão ao supervisor responsável por mim, o engenheiro Hugo Moreira, pelo acompanhamento e orientação durante o estágio, bem como ao Professor Doutor Bruno Oliveira, orientador a mim atribuído pela ESTG, pelo suporte académico e pelos valiosos conselhos ao longo deste percurso.

Gostaria de agradecer especialmente ao José Oliveira, fundador e CEO da DiRoots, a entidade de acolhimento, pela confiança demonstrada ao proporcionar esta experiência profissional, e à Maria Lameiras, gestora financeira e de recursos humanos da empresa, pelo apoio nos processos administrativos iniciais, facilitando a integração e o cumprimento dos procedimentos necessários.

Alargo os meus agradecimentos ao Diogo Vieira e ao Bruno Couto pela disponibilidade apoio durante o estágio, como também a todos os colegas da DiRoots que, diretamente ou indiretamente, contribuíram para o meu crescimento profissional.

Índice

Índice de Figuras.....	8
Índice de Tabelas.....	9
Abreviaturas, Siglas e Acrónimos.....	10
Cap. I – Contextualização e Motivação.....	11
1. Introdução.....	11
1.1. Entidade de Acolhimento.....	11
1.2. Contextualização	11
1.3. Objetivos	12
1.4. Organização do Documento	12
2. Fundamentação Teórica	14
2.1. Fundamentos do Domínio da Aplicação.....	14
2.2. Fundamentos Web e de Desenvolvimento de Aplicações.....	15
2.3. Arquiteturas e Padrões de Desenvolvimento.....	18
2.3.1. Padrão CQRS.....	18
2.3.2. Arquitetura Limpa.....	19
2.3.3. Padrão Mediator.....	21
2.4. Gestão de Bases de Dados Relacionais na Arquitetura Microserviços	21
Cap. II – Conceptualização do problema / projeto.....	24
3. Requisitos.....	24
3.1. Requisitos Funcionais.....	24
3.2. Requisitos Não Funcionais.....	26
4. Casos de Uso.....	28
4.1. Atores do Sistema	28
4.2. Recursos Humanos.....	28
4.3. Negócio.....	29
4.4. Planeamentos	29
4.5. Folhas de Pagamento.....	30
4.6. Custos.....	30
4.7. Alarmes.....	31
4.8. Clientes	31
4.9. Contactos.....	32
4.10. Identidade e Acesso (IAM).....	32
4.11. Utilizadores.....	33
5. Processo de Processamento Salarial	34
6. Arquitetura Conceptual.....	35

6.1.	Arquitetura do Sistema.....	35
6.2.	Ciclo de Vida de um Pedido.....	36
6.3.	Tecnologias Utilizadas.....	37
	Cap. III – Metodologia de operacionalização do trabalho	40
7.	Desenvolvimento da Solução	40
7.1.	Comunicação entre Microserviços	40
7.2.	Implementação do Padrão CQRS.....	42
7.3.	Funcionalidades CRUD no Frontend	43
7.4.	Internacionalização.....	46
7.5.	Exportação de dados para Excel.....	48
7.6.	Processamento Salarial.....	49
7.7.	Testes de Integração.....	50
7.8.	Automatização de Trabalho Repetitivo.....	50
8.	Processo e Metodologia de Trabalho.....	53
8.1.	Convenções e Boas Práticas da Entidade de Acolhimento	53
8.2.	Abordagem ao Desenvolvimento do Projeto.....	55
	Cap. IV – Discussão dos Resultados.....	57
9.	Apresentação e Discussão dos Resultados.....	57
10.	Apresentação e Discussão dos Impedimentos e/ou constrangimentos.....	59
	Cap. V – Conclusão.....	60
11.	Reflexão Crítica dos Resultados.....	60
12.	Conclusões e Trabalho Futuro.....	60
	Referências	62

Índice de Figuras

Figura 1 – Arquitetura Monolítica.....	17
Figura 2 – Arquitetura em Microserviços	17
Figura 3 – Padrão CQRS	19
Figura 4 – Diagrama da Arquitetura Limpa.....	20
Figura 5 – Arquitetura de Comunicação entre Controladores e Base de Dados Tradicional.....	21
Figura 6 – Arquitetura de Comunicação entre Controladores e BD com o Padrão Mediator	21
Figura 7 – Casos de Uso de Recursos Humanos.....	29
Figura 8 – Casos de Uso de Negócio.....	29
Figura 9 – Casos de Uso de Planeamentos	30
Figura 10 – Casos de Uso de Folhas de Pagamento.....	30
Figura 11 – Casos de Uso de Custos.....	31
Figura 12 – Casos de Uso de Alarmes.....	31
Figura 13 – Casos de Uso de Clientes.....	32
Figura 14 – Casos de Uso dos Contactos	32
Figura 15 – Casos de Uso de Identidade e Acesso.....	32
Figura 16 – Casos de Uso de Utilizadores.....	33
Figura 17 – Processo de processamento salarial.....	34
Figura 18 – Arquitetura do Sistema.....	36
Figura 19 – Diagrama do ciclo de vida de um pedido	37
Figura 20 – Fluxo simplificado de remoção de Cliente e contactos associados	42
Figura 21 – Diagrama de Classes do Padrão CQRS para a Entidade Contoso	43
Figura 22 – Exemplo do componente "ContosoManagementList" para a gestão da entidade.....	44
Figura 23 – Exemplo da tabela de gestão da entidade "Contoso" apresentada na interface web....	45
Figura 24 – Estrutura de pastas da entidade "Contoso" no Frontend.....	45
Figura 25 – Exemplo de definição do array de colunas para a entidade "Contoso"	46
Figura 26 – Estrutura de ficheiros JSON utilizados na internacionalização da aplicação.....	47
Figura 27 – Opção de seleção de idiomas no Frontend.....	47
Figura 28 – Modal para a seleção dos campos dos Funcionários para exportar.....	48
Figura 29 – Dados dos Funcionários Exportados para o Excel.....	49
Figura 30 – Função criada para a exportação de dados para Excel.....	49
Figura 31 – Script para a Criação Automática de Módulos e Entidades no Frontend em execução.....	51
Figura 32 – Interface de Utilitários para Strings i18n.....	52

Índice de Tabelas

Tabela 1 - RF1 Gestão de Funcionários	24
Tabela 2 - RF10 Gestão de Custos.....	25
Tabela 3 - RF19 Processamento de Salários.....	26
Tabela 4 - RNF1 Internacionalização.....	26
Tabela 5 - RNF3 Usabilidade.....	27

Abreviaturas, Siglas e Acrónimos

ESTG – Escola Superior de Tecnologia e Gestão

AEC – Architecture / Engineering / Construction

BIM – Building Information Modeling

CQRS – Command Query Responsibility Segregation

HTML – Hypertext Markup Language

CSS – Cascading Style Sheets

SQL – Structured Query Language

API – Application Programming Interface

REST – Representational State Transfer

JSON – Javascript Object Notation

UI – User Interface

ORM – Object–Relational Mapping

SGBD – Sistema de Gestão de Bases de Dados

CRUD – Create; Read; Update e Delete

DTO – Data Transfer Object

XLSX – Excel Open XML Spreadsheet

Cap. I – Contextualização e Motivação

1. Introdução

1.1. Entidade de Acolhimento

A DiRoots é uma empresa fundada em 2017, atualmente sediada em Guimarães, que se especializa no desenvolvimento de software para a indústria AEC (Arquitetura, Engenharia e Construção), reconhecida mundialmente como uma das principais empresas que desenvolve plugins para a plataforma *AutoDesk*, estando mais focada, e por sua vez mais reconhecida na plataforma *AutoDesk Revit*, onde são comercializados os seguintes plugins, que melhoram a produtividade e a gestão de modelos BIM (*Building Information Modeling*): *DiRoots One*; *ProSheets*; *DiStem*; *IDS for Revit*.

Para além dos plugins para a plataforma *AutoDesk*, a DiRoots desenvolve também aplicações *web* e *cloud*, geralmente sob medida, adaptadas às necessidades específicas dos clientes. Além disso, presta serviços de consultoria e suporte em projetos BIM, contribuindo ativamente para a inovação e otimização dos processos na indústria AEC.

1.2. Contextualização

Antes de entrar na empresa, um dos colaboradores da DiRoots trabalhava como freelancer, um profissional que presta serviços de forma independente, e durante esse período desenvolveu um programa de controlo e gestão de obra para uma empresa cuja principal fonte de rendimento não era propriamente a execução direta de obras, mas sim a subcontratação de mão de obra especializada para outras empresas do setor AEC (Arquitectura, Engenharia e Construção), que necessitavam de trabalhadores para realizar os seus próprios projetos.

Visto que que a DiRoots desenvolve software personalizado para o setor AEC, esse projeto enquadrava-se no escopo da empresa, podendo tornar-se num produto que a empresa poderia comercializar a empresas com necessidades semelhantes, no entanto, como o software original foi desenvolvido e vendido em regime freelancer, a DiRoots não o poderia reutilizar, logo surgiu a oportunidade de colocar no *backlog* um novo projeto inspirado nesse trabalho anterior, mas desenvolvido do zero, seguindo os padrões visuais e arquiteturais da empresa.

Como o projeto ainda não tem um cliente final definido, é fundamental que a solução a desenvolver seja flexível e modular o suficiente de modo a permitir futuras adaptações, para que a mesma possa

responder facilmente às exigências específicas de diferentes clientes, minimizando custos e esforços de manutenção a longo prazo.

1.3. Objetivos

O principal objetivo deste projeto é a aplicação dos conhecimentos adquiridos ao longo da Licenciatura em Engenharia Informática em contexto empresarial, através do desenvolvimento de uma aplicação de controlo e gestão de obra para a empresa DiRoots, que integra nas necessidades reais da organização e respeita os padrões de desenvolvimento e convenções internas da mesma. Tendo os seguintes objetivos específicos para o projeto:

- Desenvolver um conjunto de microserviços organizados por domínio de negócio, com bases de dados logicamente independentes, assegurando a separação de responsabilidades e a consistência dos dados;
- Implementar um *frontend* funcional e responsivo em *React*¹, que integre as funcionalidades dos microserviços e com uma interface simples e intuitiva;
- Aplicar os mecanismos de validação e regras de negócio existentes no *template*, adaptando-os às necessidades do projeto, e garantindo a integridade da informação entre entidades, mesmo sem o uso de chaves estrangeiras a nível do SGBD (Sistema de Gestão de Bases de Dados);
- Implementar funcionalidades críticas como a gestão de colaboradores, folhas de pagamento, clientes, contactos e estrutura de custos, de forma a responder às operações reais da empresa;
- Estruturar o sistema de modo a facilitar a testabilidade, manutenção e evolução futura, respeitando as convenções e o ecossistema interno da DiRoots.

1.4. Organização do Documento

Este documento está estruturado em capítulos e subcapítulos de forma a garantir uma leitura fluída e uma compreensão clara do trabalho desenvolvido ao longo do estágio curricular no âmbito da unidade curricular de Projeto Final. Para além deste capítulo introdutório, o documento contém os seguintes capítulos:

- **Contextualização e Motivação** – introdução ao projeto, descreve a entidade de acolhimento, identifica os objetivos a atingir e explica a motivação por detrás da aplicação desenvolvida.

¹<https://react.dev>

- **Fundamentação Teórica** – são explorados os principais conceitos técnicos e metodologias que suportam o desenvolvimento do projeto, com destaque para as arquiteturas: microserviços, CQRS e padrões como *Mediatore* e *Clean Architecture*, bem como as tecnologias utilizadas.
- **Conceptualização do Problema/Projeto** – aprofunda a análise do problema e as necessidades identificadas, descrevendo os requisitos funcionais, não funcionais e a arquitetura conceptual da solução.
- **Metodologia de Operacionalização do Trabalho** – apresenta a abordagem adotada ao longo do estágio, inclui o processo de aprendizagem, as convenções e boas práticas definidas pela empresa, e a forma como o desenvolvimento foi organizado ao longo do tempo.
- **Desenvolvimento da Solução** – detalha o trabalho técnico realizado, desde a implementação dos microserviços, passando pela validação da lógica de negócio e integração entre *frontend* e *backend*, até à criação de funcionalidades específicas como a internacionalização e exportação de dados para excel.
- **Discussão dos Resultados** – análise crítica do trabalho desenvolvido, apresentando os resultados alcançados, os desafios enfrentados, as decisões tomadas e os respectivos impactos no desenvolvimento.
- **Conclusão e Trabalho Futuro** – reflexão crítica sobre os objetivos atingidos, as competências desenvolvidas ao longo do estágio, e sugere melhorias e possíveis evoluções para a aplicação no futuro.

2. Fundamentação Teórica

Esta secção apresenta os principais conceitos e arquiteturas que sustentam o desenvolvimento da aplicação implementada. Serão abordados fundamentos do domínio da aplicação, desenvolvimento web e padrões utilizados na arquitetura da aplicação.

2.1. Fundamentos do Domínio da Aplicação

2.1.1. Subcontratação em Obras

A subcontratação é uma prática bastante comum dentro do setor de construção, que pode acontecer de forma bidirecional, onde: por um lado, uma empresa pode recorrer a entidades externas para fornecer mão-de-obra especializada ou complementar; por outro, pode também disponibilizar os seus próprios recursos humanos a outras empresas, atuando como fornecedora de serviços. Esta flexibilidade é bastante vantajosa, no entanto bastante complexa de gerir.

Por isso é crucial que os sistemas de informação permitam não só a distinção clara entre recursos internos e externos, mas também o rastreio preciso de qual entidade patronal é responsável por cada trabalhador em determinado momento, assegurando a correta afetação de custos e o controlo de prazos em cada projeto.

2.1.2. Projetos e respetivos planeamentos

Uma obra, designada como projeto no contexto do negócio, pode conter um número ilimitado de planeamentos associados. Cada planeamento pode corresponder a uma fase distinta do projeto ou a um conjunto de procedimentos específicos, dependendo da forma como a gestão da obra é organizada.

Em alguns casos, os planeamentos são sequenciais e representam fases do projeto que não podem decorrer em simultâneo (por exemplo: preparação do terreno, construção da estrutura e acabamentos). Noutros casos, os planeamentos podem coexistir de forma paralela, representando procedimentos ou atividades complementares que ocorrem em simultâneo, mas que ainda assim devem ser monitorizados individualmente.

A divisão do projeto em vários planeamentos traz vantagens claras para a gestão e acompanhamento da obra, permitindo que cada etapa possa ser monitorizada e auditada de forma independente. A cada planeamento pode ser associada informação detalhada sobre os recursos humanos envolvidos (funcionários e dias trabalhados), bem como os meios logísticos necessários, como veículos e alojamentos atribuídos durante o decorrer dessa fase.

2.1.3. Processamento Salarial

O processamento de salários representa uma das funcionalidades mais críticas para empresas que atuam no setor de construção, sendo frequentemente o principal motivo para a adoção de um sistema de informação. O cálculo do salário de um colaborador envolve múltiplos fatores e pode tornar-se complexo, especialmente devido à diversidade de obras e fases em que os funcionários se encontram alocados simultaneamente.

Uma das maiores dificuldades neste processo reside no facto de muitos colaboradores trabalharem em diferentes obras em paralelo, distribuídos por várias fases do projeto. Cada fase deve ser considerada de forma independente, uma vez que o método de cálculo varia consoante o tipo de contrato do funcionário. Por exemplo:

- **Pagamento fixo:** é suficiente contabilizar as faltas do colaborador, deduzindo os dias em que não esteve presente.
- **Pagamento à hora:** torna-se necessário registar e calcular o número exato de horas trabalhadas em cada obra ou planeamento, aplicando o valor acordado por hora.

Para além destes aspetos, o sistema deve também contemplar adiantamentos feitos ao colaborador, bem como ajustamentos salariais. É prática comum que os salários sejam arredondados para múltiplos de 5 ou 10, o que obriga a registar ajustamentos positivos ou negativos, que posteriormente devem ser refletidos no processamento salarial do mês seguinte.

Importa ainda salientar que o pagamento efetivo não é realizado diretamente pela aplicação, mas sim pelos contabilistas da empresa. O sistema tem como objetivo gerar relatórios detalhados do processamento salarial, permitindo ao contabilista validar os valores, aplicar as regras fiscais ou legais necessárias, e proceder ao pagamento de forma segura e conforme à legislação em vigor.

2.2. Fundamentos Web e de Desenvolvimento de Aplicações

2.2.1. Backend vs Frontend

No desenvolvimento Web, existem dois componentes fundamentais e responsáveis por diferentes partes de uma aplicação, sendo eles:

- **Frontend** – é a parte visual de uma aplicação, a parte que o utilizador consegue visualizar e interagir com os dados.
- **Backend** – é o lado do servidor, a parte que o utilizador não vê, responsável por processar pedidos, gerir as bases de dados e garantir a lógica de negócio.

2.2.2. Server-Side Rendering vs Client-Side Rendering

Quando se programa para a Web, existem dois paradigmas para fazer o processamento das páginas Web, sendo elas:

- **Processamento do lado do servidor** – uma página HTML é montada no lado do servidor, incluindo a informação dinâmica, e manda-a ao cliente para que este não precise de fazer nenhum tipo de processamento adicional para poder visualizar a página. Este paradigma oferece tempos carregamento iniciais muito curtos, porque o trabalho mais demorado está a ser realizado na parte do servidor, no entanto o servidor pode facilmente ficar sobrecarregado. [1]
- **Processamento do lado do cliente** – o servidor irá enviar inicialmente uma página HTML vazia, e o cliente irá usar JavaScript para poder atualizar dinamicamente a página. Com este paradigma é possível criar páginas mais dinâmicas, onde não será necessário recarregar a página inteira para poder fazer alterações na página, não sobrecarregando tanto o servidor, visto que o cliente faz uma parte do trabalho, no entanto, pode ter tempos carregamento iniciais mais demorados, para além de ser mais difícil anexar as páginas aos motores de pesquisa. [1]

2.2.3. Arquitetura Monolítica vs Microserviços

A arquitetura de uma aplicação tem um impacto crítico na forma como a mesma irá funcionar e evoluir com o tempo, portanto, é preciso ter-se bastante cuidado na escolha da arquitetura da mesma, pois, mesmo que seja algo que o utilizador final não veja, ela influenciará o desempenho e a escalabilidade do sistema. Atualmente as arquiteturas mais comuns são: a monolítica e a de microserviços.

Numa arquitetura monolítica toda a aplicação é construída numa única camada, onde os componentes estão fortemente ligados e partilham os mesmos recursos, como a base de dados e a memória, permitindo simplificar o desenvolvimento inicial e a gestão do sistema em projetos mais pequenos [2], no entanto, à medida que a aplicação cresce essa abordagem pode tornar-se problemática, pois o código fica mais difícil de manter, as alterações impactam várias partes do sistema e torna-se complicado escalar a aplicação horizontalmente. Além disso, é bastante difícil introduzir novas tecnologias, visto que qualquer mudança precisa de ser aplicada no sistema todo [3].

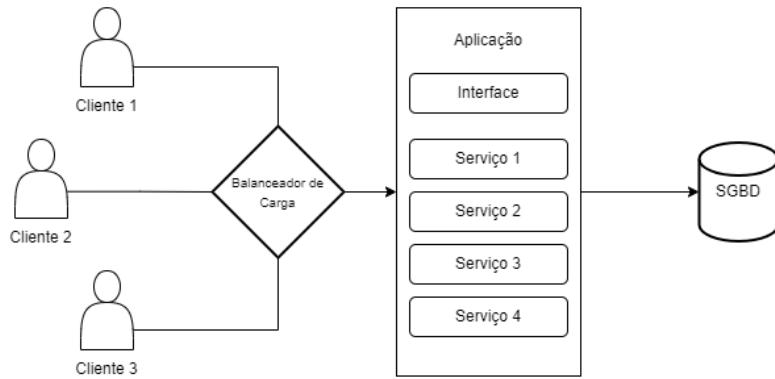


Figura 1 – Arquitetura Monolítica

Adaptado de Mehmet Ozkaya (2021)

Numa arquitetura em microserviços vai-se tentar dividir a aplicação em várias aplicações mais pequenas, tentando isolá-las o máximo possível de modo que consigam operar de forma independente, fragmentando assim o processo de desenvolvimento, e facilitando assim a escalabilidade da aplicação, pois cada serviço consegue escalar de forma independente dos outros.

Outra das vantagens da arquitetura em microserviços é que, ao contrário da arquitetura monolítica, dificilmente fica-se preso a uma linguagem ou ferramenta, pois cada serviço pode ser construído com uma tecnologia diferente. [4]

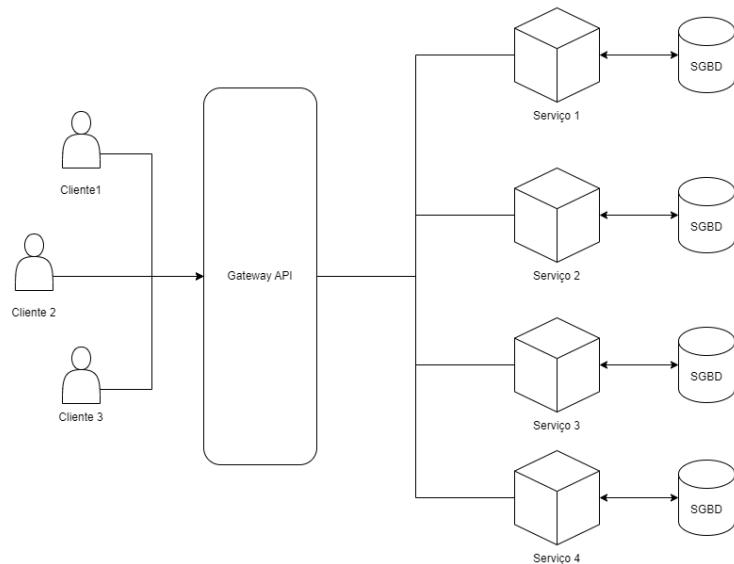


Figura 2 – Arquitetura em Microserviços

Adaptado de vsourz (2022)

No projeto realizado foi adotada a arquitetura em microserviços, visto que o sistema pode escalar de forma indefinida ao longo do tempo, assim cada microserviço poderá ser ajustado ou expandido individualmente conforme as necessidades dos clientes que surgirem, que vão ter necessidades

específicas para poderem integrar a aplicação nos seus negócios. No entanto, algumas características típicas desta arquitetura foram propositadamente descartadas para manter a coerência com o domínio do projeto. Esses pontos serão abordados na Secção 2.4.

2.2.4. REST API

Uma API (*Application Programming Interface*) é um conjunto de regras e protocolos que permitem que diferentes aplicações consigam comunicar e trocar informação entre si, funcionando como se fosse uma ponte entre um provedor (servidor) e um consumidor (cliente), permitindo que exista interação sem expor as complexidades subjacentes.

Quando se quer construir uma aplicação para a Web, onde o processamento acontece no lado do cliente, é quase obrigatório o uso de uma API, sendo atualmente o modelo mais utilizado para a construção de APIs para a Web os modelos REST (*Representational State Transfer*), que surgiram como uma tentativa de simplificar o modelo *SOAP*, que eram conhecidas por ser bastante complexas. No entanto, o uso de APIs REST não é obrigatório, mas sim predominante devido à sua simplicidade e aderência ao protocolo HTTP. [5][6]

No modelo REST, as principais operações são realizadas através dos seguintes métodos HTTP:

- GET → Utilizado para obter os dados de um recurso.
- POST → Utilizado para criar um recurso.
- PUT → Utilizado para atualizar totalmente um recurso já existente.
- DELETE → Utilizado para remover totalmente um recurso.

Com o tempo, além dos métodos mais utilizados (GET, POST, PUT, DELETE), foram adicionados outros métodos ao protocolo HTTP, como o PATCH, que permite a atualização parcial de um recurso, ao contrário do PUT, que substitui todo o recurso. A disponibilidade e suporte a determinados métodos podem variar consoante a *framework* ou tecnologia utilizada. [7]

2.3. Arquiteturas e Padrões de Desenvolvimento

2.3.1. Padrão CQRS

O *CQRS* é um padrão utilizado no desenvolvimento de *software* que tem como principal objetivo a separação das operações de leitura e escrita, de modo a facilitar a manutenibilidade, capacidade de manutenção e o desempenho da aplicação quando a mesma é bastante complexa [8].

O padrão *CQRS* trabalha sobre estes dois conceitos:

- **Comandos:** representam uma ação que irá alterar o estado do sistema (escrita ou alteração)
- **Querys / Consultas:** servem para devolver dados do sistema sem alterar o estado do mesmo

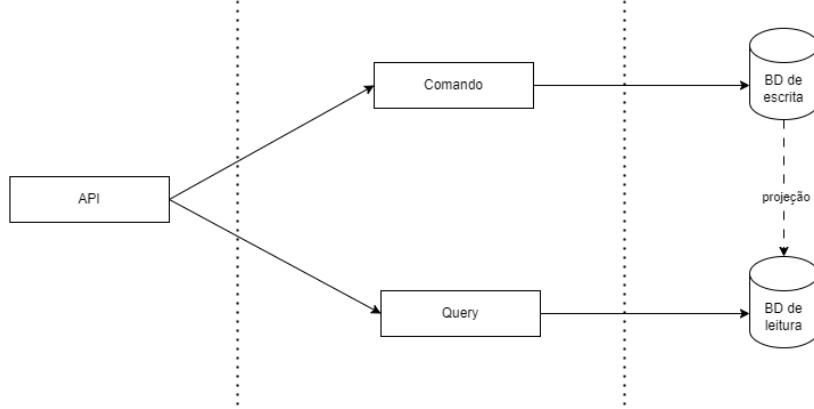


Figura 3 – Padrão CQRS

Adaptado de Mehmet Ozkaya (2021)

Normalmente quando se utiliza o padrão *CQRS* costuma-se ter duas bases de dados, uma relacional para escrita, utilizada pelos comandos, e uma base não relacionar para as operações de leitura, utilizada pelas *querys*, e as duas bases de dados têm um entre diversos mecanismos de sincronização existentes para poder realizar a sincronização entre as duas bases de dados, no entanto, este padrão pode ser utilizado com apenas uma base de dados [9], como foi feito no projeto desenvolvido, onde cada serviço contém apenas uma base de dados relacional, e o padrão serve como uma convenção que facilita a manutenção e legibilidade do código, tornando o projeto mais homogéneo mesmo que existam várias pessoas a trabalhar sobre o mesmo.

2.3.2. Arquitetura Limpa

A arquitetura limpa é um padrão desenvolvido pelo Robert C. Martin (*Uncle Bob*), autor dos livros *Clean Code* e *Clean Architecture*, que tem como principal objetivo o desacoplamento entre as regras de negócio, e os recursos externos como *frameworks* e bases de dados, de forma a produzir código que seja fácil de entender, seja testável e flexível, de modo a conseguir criar sistemas que sejam facilmente modificáveis e escaláveis ao longo do tempo, permitindo que as mudanças no sistema não impactem negativamente as outras partes do código.[10]

A estrutura da arquitetura limpa é a seguinte:

- **Entidades:** Representam o núcleo da lógica de negócio. São objetos genéricos reutilizáveis, totalmente independentes de *frameworks*, interfaces ou bases de dados. Contêm regras de negócio de alto nível e não mudam com facilidade.
- **Aplicação:** Contém os casos de uso do sistema, compondo o fluxo da lógica de negócio. Esta camada coordena as interações entre as entidades e o exterior, sem depender de infraestrutura ou apresentação.
- **Apresentação:** Responsável por lidar com a interface do utilizador ou qualquer entrada/saída externa. Essa camada converte as ações do utilizador em comandos da aplicação e apresenta os resultados.
- **Infraestrutura:** Reúne os detalhes técnicos, como acesso a bases de dados, chamadas a *APIs*, *frameworks* e bibliotecas externas. Esta camada pode ser alterada sem afetar as regras de negócio ou os casos de uso.

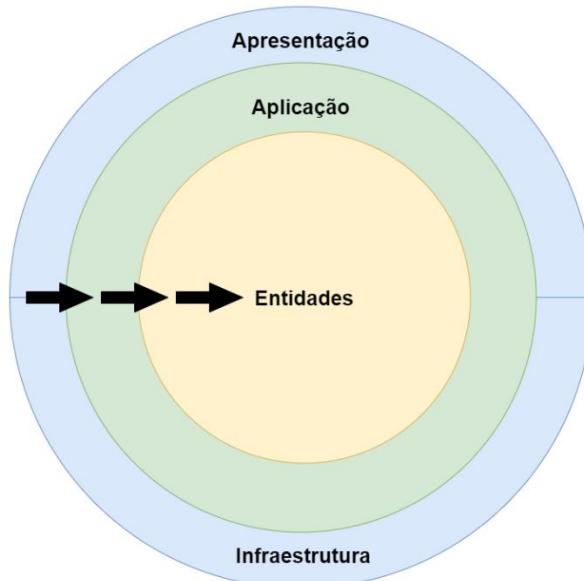


Figura 4 – Diagrama da Arquitetura Limpa.

Adaptado de Milan Jovanovic (2022)

Para que estrutura indicada anteriormente seja realmente eficaz, é necessário que o sistema seja flexível e modular, o que leva à introdução de um dos princípios mais importantes da arquitetura limpa: a regra da dependência. Esta regra diz que as dependências no código devem sempre fluir de fora para dentro, ou seja, das camadas mais externas (Apresentação e Infraestrutura) para as camadas internas (Entidades), e as camadas internas nunca devem depender das camadas externas. Esta separação é o que permite com que a lógica de negócio permaneça protegida das mudanças das tecnologias externas, podendo substituir tecnologias externas, como base de dados ou interface de utilizador, sem impactar a lógica de negócio do sistema, o que facilita tanto a manutenção como a evolução do sistema.

2.3.3. Padrão Mediator

O padrão Mediator tem como principal objetivo a redução do acoplamento entre os objetos através da introdução de um mediador responsável por toda a comunicação entre os objetos, onde, invés dos objetos comunicarem e trocarem informação entre si, formando uma rede de comunicação que irá dificultar o processo de manutenção e correção de falhas, comunicam apenas com um objeto que está responsável por toda a comunicação, fazendo com que os objetos não estejam fortemente acoplados entre si [11], simplificando a manutenção e a escalabilidade do sistema.

Quando este padrão é implementado na aplicação desenvolvida traz uma melhoria significativa na arquitetura da aplicação. A Figura 5 e Figura 6 mostram como seria a comunicação entre controladores e base de dados com e sem o mediador de comunicações.

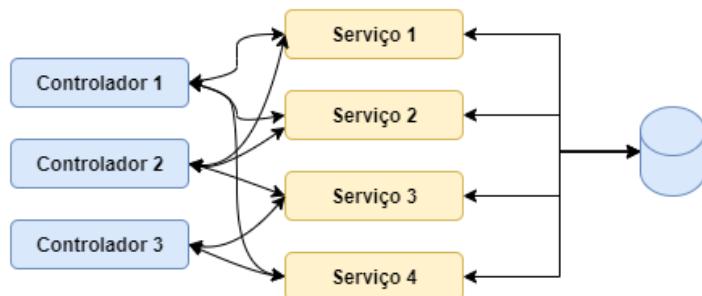


Figura 5 – Arquitetura de Comunicação entre Controladores e Base de Dados Tradicional

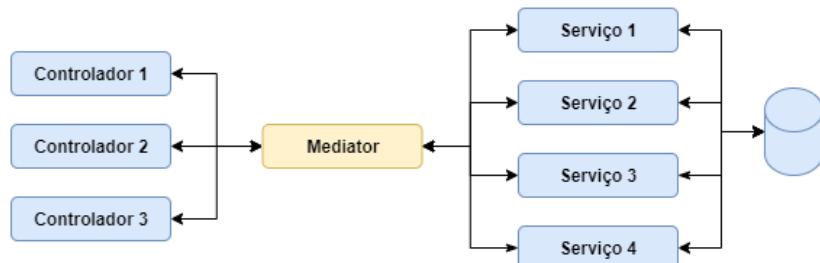


Figura 6 – Arquitetura de Comunicação entre Controladores e BD com o Padrão Mediator

Adaptado de Refactoring Guru

2.4. Gestão de Bases de Dados Relacionais na Arquitetura Microserviços

Uma das maiores dificuldades ao adotar microserviços surge quando os serviços precisam aceder a dados distribuídos entre múltiplas bases de dados relacionais. Como cada serviço tem a sua própria base de dados, então torna-se complexo garantir a integridade referencial (consistência dos dados entre domínios) e partilhar informações entre serviços. Para resolver esses problemas, são comumente utilizadas as seguintes abordagens:

- **Comunicação entre serviços** → Permitir que os serviços comuniquem diretamente entre si, seja por pedidos síncronos (ex.: HTTP) ou por eventos. Esta abordagem facilita a partilha de dados, mas não resolve diretamente a integridade referencial e pode introduzir problemas de latência e acoplamento implícito;
- **Partilha de Bases de Dados** → Permitir que múltiplos serviços partilhem uma base de dados comum, preservando relações diretas e a integridade referencial. Contudo, esta solução compromete a escalabilidade e autonomia dos serviços, aumentando o acoplamento e dificultando a evolução independente;
- **Comunicação com um Serviço Centralizado** → Criar um serviço único responsável pelo acesso e gestão da base de dados, com os restantes serviços a comunicarem apenas com este mediador. Esta abordagem abstrai a complexidade relacional, mas introduz um ponto único de falha e pode tornar-se um *bottleneck*².

A DiRoots optou por uma abordagem intermédia, onde todos os microserviços partilham a mesma base de dados, mas cada serviço tem um *schema* diferente, logicamente isolado dos outros, de forma a manter uma separação lógica clara. Ou seja, embora a infraestrutura seja comum entre todos os serviços, cada serviço apenas interage com os dados dentro do seu contexto, como se estivesse numa base de dados diferente.

Para que os *schemas* estejam isolados, não são utilizadas chaves estrangeiras entre entidades de diferentes domínios, e a integridade referencial entre os dados relacionados é realizada ao nível da aplicação, através de validações e lógica de negócio bem estruturada. Esta decisão está alinhada com o objetivo da empresa de manter um sistema modular, coeso e fácil de manter, mesmo que para isso sejam sacrificados alguns princípios da arquitetura de microserviços, como a total independência de *deploy*, tecnologias e bases de dados.

Resumidamente, a abordagem seguida apresenta as seguintes vantagens:

- Separação lógica entre os domínios dentro da mesma de dados;
- Desacoplamento, permitindo a evolução dos serviços sem afetar os outros;
- Redução da complexidade de infraestrutura, evitando a gestão de múltiplas bases de dados físicas;
- Facilidade de manutenção, visto que os dados permanecem no mesmo SGBD, mas organizados por contexto

² Ponto de um sistema que limita o desempenho, por não conseguir processar dados na mesma velocidade que outros componentes.

No entanto a abordagem seguida apresenta algumas desvantagens que são mitigadas através de boas práticas de desenvolvimento, sendo elas:

- A validação da integridade referencial depende inteiramente do código, exigindo um grande rigor durante o desenvolvimento;
- A ausência de chaves estrangeiras pode permitir inconsistências acidentais caso o domínio não seja devidamente respeitado;
- Maior necessidade de testes nas operações de escrita

Para garantir a consistência de dados, a aplicação recorre a validadores criados com o auxílio da biblioteca *Fluent Validation*³ e a uma estrutura de comandos bem definida. Além disso, operações complexas como eliminação em cascata (normalmente garantidas pelo SGBD via `ON DELETE CASCADE`) são implementadas através da biblioteca *MediatR*⁴ com *Command Handlers*, que encapsulam a lógica necessária para apagar ou atualizar entidades dependentes, assegurando que o estado do sistema permanece consistente.

Apesar da separação lógica e da ausência de relacionamentos diretos no SGBD, continua a existir comunicação entre microserviços através do *MediatR*, onde são publicadas notificações de forma síncrona dentro do mesmo processo, sem a necessidade de dependências adicionais como *message brokers*. Tal como na gestão de bases de dados, a implementação da comunicação através de notificações pelo *MediatR* foi moldada por decisões pragmáticas da empresa, privilegiando soluções simples e controladas, em vez de dependências complexas ou plataformas de mensagens altamente distribuídas.

³ <https://docs.fluentvalidation.net/en/latest/>

⁴ <https://mediatr.io>

Cap. II – Conceptualização do problema / projeto

3. Requisitos

Esta secção apresenta uma visão geral dos requisitos do sistema definidos para o projeto, organizados em requisitos funcionais e não funcionais, onde cada requisito é identificado por um código único, um título, uma descrição, e caso se verifique, pode conter restrições, verificações, fluxo principal e fluxos secundários. A versão completa dos requisitos, encontra-se no anexo designado “PF_Lista_Requisitos_8220337”.

Cada requisito tem uma prioridade atribuída que varia entre 1 e 5, que indica o impacto do mesmo sobre o sistema, onde cada dígito é interpretado da seguinte maneira:

- **1 – Opcional:** O requisito pode ser incluído se houver tempo, mas não afeta o funcionamento do sistema se não tiver sido implementado.
- **2 – Pouco Importante:** Útil, mas não tem muito impacto, pode ser implementado numa fase futura.
- **3 – Importante:** Deve ser incluído, melhora a experiência ou funcionalidades principais, mas o sistema consegue funcionar sem o mesmo.
- **4 – Muito Importante:** Necessário para garantir o bom funcionamento do sistema e cumprir os objetivos do projeto.
- **5 – Essencial:** Requisito crítico, onde a ausência do mesmo pode comprometer o funcionamento ou a entrega mínima do sistema.

3.1. Requisitos Funcionais

Abaixo encontram-se alguns requisitos funcionais mais relevantes:

Tabela 1 - RF1 Gestão de Funcionários

RF1	
Nome	Gestão de Funcionários
Descrição	O sistema deve permitir a gestão de funcionários, possibilitando realizar operações de criação, visualização, atualização, eliminação, pesquisa e filtragem de registo. Os dados do funcionário devem ser apresentados em formato de tabela com suporte a paginação.
Prioridade ⁵	5

⁵1 (Opcional) / 2 / 3 / 4 / 5 (Essencial)

Restrições	<ul style="list-style-type: none"> • Apenas utilizadores autenticados podem aceder a esta funcionalidade; • Todos os campos obrigatórios devem estar preenchidos ao criar ou editar um funcionário
Verificações	
Fluxo Principal	<ol style="list-style-type: none"> 1. O utilizador autentica-se no sistema. 2. Acede à secção de gestão de funcionários. 3. Visualiza a lista de funcionários numa tabela. 4. Pode criar, editar, remover ou pesquisar um funcionário. 5. O sistema executa a operação e atualiza a tabela.
Fluxo Secundário	<ul style="list-style-type: none"> • Se não existirem registo, o sistema apresenta a mensagem "Nenhum registo correspondente encontrado". • Em caso de erro de validação (ex: campos obrigatórios em branco), o sistema apresenta uma mensagem não preenchidos.

Tabela 2 - RF10 Gestão de Custos

RF10	
Nome	Gestão de Custos
Descrição	O sistema deve permitir a gestão de custos associados a veículos, alojamentos e projetos, possibilitando realizar operações de criação, visualização, atualização, eliminação, pesquisa e filtragem de registo. Os dados dos custos dos projetos devem ser apresentados em formato de tabela com suporte a paginação.
Prioridade	5
Restrições	<ul style="list-style-type: none"> • Apenas utilizadores com permissões de administrador podem visualizar o lucro de um projeto; • Só pode ser possível associar o custo a um veículo caso o custo seja do tipo "Gasolina" ou "Veículo"; • Só pode ser possível associar o custo a um planeamento de um projeto caso o custo seja do tipo "Custo do Projeto"; • Só pode ser possível associar o custo a um funcionário caso o custo seja do tipo "Custo Extra"; • Só pode ser possível associar o custo a um alojamento caso o custo seja do tipo "Alojamento", "Eletricidade" ou "Gás".

Verificações	
Fluxo Principal	<ol style="list-style-type: none"> 1. O utilizador autentica-se no sistema. 2. Acede à secção de gestão de custos. 3. Pode criar, editar, remover ou pesquisar um custo. 4. O sistema executa a operação e atualiza a tabela.
Fluxo Secundário	<ul style="list-style-type: none"> • Se não existirem registo, o sistema apresenta a mensagem "Nenhum registo correspondente encontrado". • Em caso de erro de validação (ex: campos obrigatórios em branco), o sistema apresenta uma mensagem não preenchidos.

Tabela 3 – RF19 Processamento de Salários

RF19	
Nome	Processamento de Salários
Descrição	O sistema deve permitir o processamento automático dos salários dos funcionários para um determinado mês. Caso o mês anterior ainda não tenha sido processado, o sistema deverá realizar o seu processamento em simultâneo.
Prioridade	5
Restrições	Só é possível processar salários de meses anteriores ao atual.
Verificações	Verificar se o mês anterior está pendente de processamento; confirmar que não existem dados em falta para os funcionários do mês a processar.
Fluxo Principal	<ol style="list-style-type: none"> 1. O utilizador seleciona o mês a processar. 2. O sistema verifica se o mês anterior já foi processado. 3. Caso contrário, processa primeiro o mês anterior. 4. O sistema realiza o processamento dos salários para o mês selecionado. 5. São gerados os valores salariais com base nos dados dos funcionários e respetivos ajustes.

3.2. Requisitos Não Funcionais

Entre os requisitos não funcionais definidos, destacam-se:

Tabela 4 – RNF1 Internacionalização

RNF1	
Nome	Internacionalização

Descrição	A aplicação deve suportar múltiplos idiomas, incluindo pelo menos inglês, francês, espanhol e alemão , permitindo ao utilizador selecionar o idioma desejado, sendo a preferência guardada para sessões futuras.
Prioridade	3
Critérios Mensuráveis	<ul style="list-style-type: none"> • Deve existir uma opção de idioma nas configurações; • A interface deve apresentar pelo menos 85% das strings traduzidas em cada idioma suportado; • A preferência do idioma deve ser mantida após logout/login.

Tabela 5 – RNF3 Usabilidade

RNF3	
Nome	Usabilidade
Descrição	A aplicação deve proporcionar uma experiência simples e eficiente. A navegação deve ser clara, com mensagens de erro descritivas e um fluxo lógico.
Prioridade	5
Critérios Mensuráveis	<ul style="list-style-type: none"> • As mensagens de erro devem conter texto explicativo em pelo menos 90% dos cenários de erro previsíveis

4. Casos de Uso

Para facilitar a compreensão do sistema e obter uma representação visual dos requisitos funcionais, foram construídos vários diagramas de caso de uso, agrupados por áreas funcionais. Cada diagrama mostra os atores envolvidos e as ações que estes podem realizar, permitindo validar os requisitos funcionais e estruturar a solução de forma modular.

4.1. Atores do Sistema

No sistema desenvolvido foram identificados os seguintes atores:

- **Utilizador** – Representa o perfil base da aplicação. O acesso às funcionalidades é determinado pelas permissões atribuídas por um administrador, podendo incluir ações como visualizar, criar, editar ou eliminar registo em módulos como recursos humanos, planeamentos, projetos, custos, entre outros. O utilizador está limitado a interagir apenas com os dados da empresa à qual pertence;
- **Administrador** – Perfil com permissões avançadas, responsável pela gestão de utilizadores e pela definição de permissões personalizadas para cada um, permitindo configurar diferentes níveis de acesso (por exemplo, apenas leitura ou acesso total a determinadas funcionalidades). Pode ainda aceder a funcionalidades mais sensíveis, como a gestão de permissões e a visualização de dashboards financeiros. Tal como o utilizador, o administrador também está limitado aos dados da empresa à qual pertence.

4.2. Recursos Humanos

Este módulo é responsável pela gestão dos colaboradores da empresa, bem como das suas categorias, períodos de ausência e processos de recrutamento. Este módulo é bastante utilizado por outros módulos da aplicação, como o de planeamento e o de folhas de pagamento.

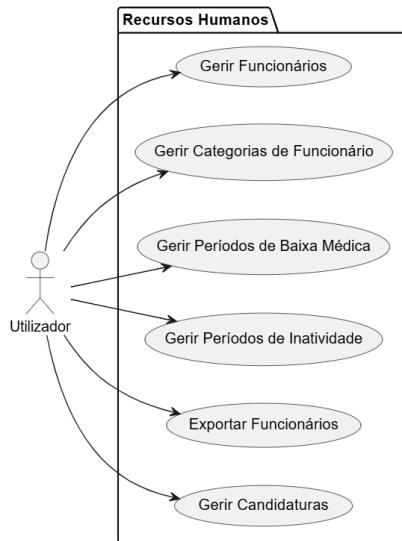


Figura 7 – Casos de Uso de Recursos Humanos

4.3. Negócio

Este módulo lida com a gestão das entidades que participam nos projetos da empresa, como fornecedores, clientes, subcontratados, alojamentos, veículos e equipamentos. Estes elementos estão diretamente ligados aos planeamentos e custos dos projetos.

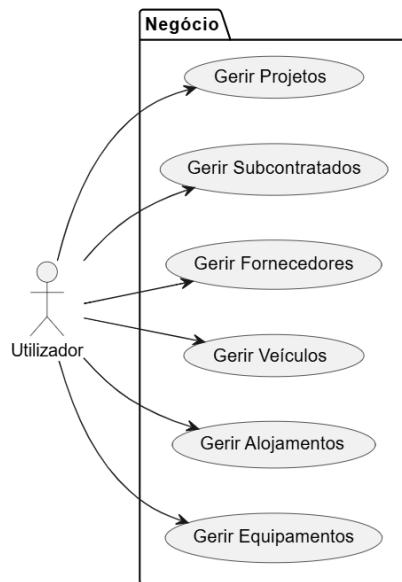


Figura 8 – Casos de Uso de Negócio

4.4. Planeamentos

O módulo de planeamentos centraliza o planeamento de recursos, permitindo a alocação eficiente de funcionários, veículos e alojamentos aos projetos da empresa. Este módulo será futuramente integrado com o sistema de custos e de processamento de salários, para que estes possam operar corretamente.

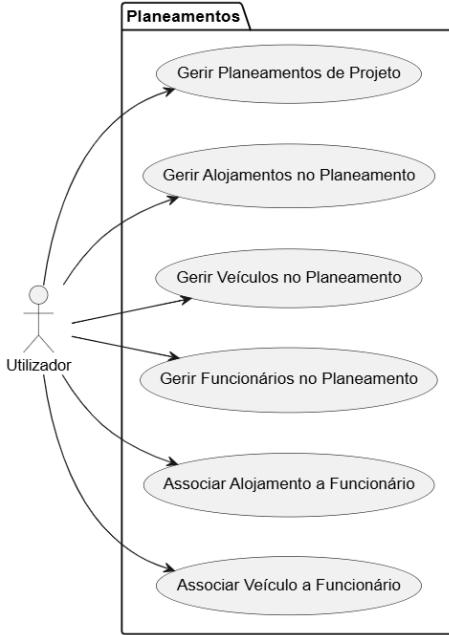


Figura 9 – Casos de Uso de Planeamentos

4.5. Folhas de Pagamento

Módulo dedicado ao processamento de salários. Abrange ajustes salariais, adiantamentos, cálculo de salários e exportação de relatórios para posteriormente enviar aos contabilistas das respetivas empresas. Tendo em conta a sensibilidade deste módulo, o desenvolvimento do mesmo exige especial atenção às regras de negócio, validação, para além de ser necessário testagem intensiva.

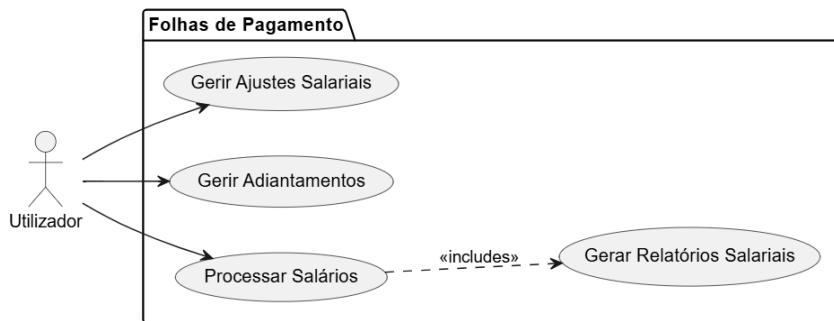


Figura 10 – Casos de Uso de Folhas de Pagamento

4.6. Custos

Este módulo agrupa a informação financeira relevante aos projetos. Permite gerir custos por domínio (veículos, alojamentos, projetos) e calcular o custo de cada funcionário por obra, sendo informação crítica para o controlo financeiro da empresa.

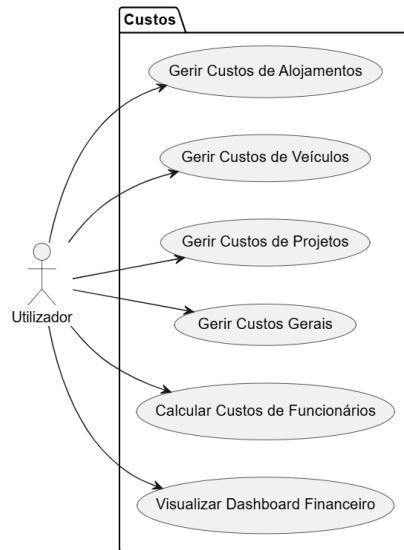


Figura 11 – Casos de Uso de Custos

4.7. Alarmes

O sistema de alertas irá permitir automatizar a notificação de prazos críticos, desde equipamentos de proteção individual até documentação de veículos, garantindo o cumprimento atempado de obrigações empresariais.

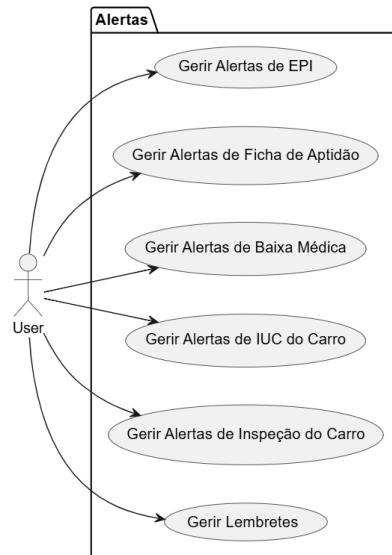


Figura 12 – Casos de Uso de Alarmes

4.8. Clientes

Este módulo permite a gestão dos dados dos clientes da empresa, sendo essencial para manter um registo organizado e acessível de todas as entidades com quem a empresa colabora.

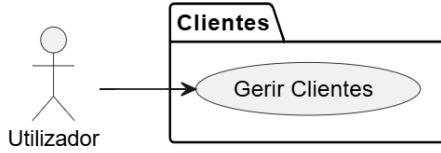


Figura 13 – Casos de Uso de Clientes

4.9. Contactos

O módulo de contactos permite um utilizador guardar contactos que sejam importantes para o bom funcionamento da empresa. Esses contactos podem ser independentes ou diretamente vinculados a registos de outras entidades no sistema. Como por exemplo, uma obra pode ter um conjunto de contactos associados que contribuem diretamente para o bom funcionamento da mesma, logo, esses contactos devem ser estar diretamente ligados a essa obra em específico quando forem registados na aplicação.

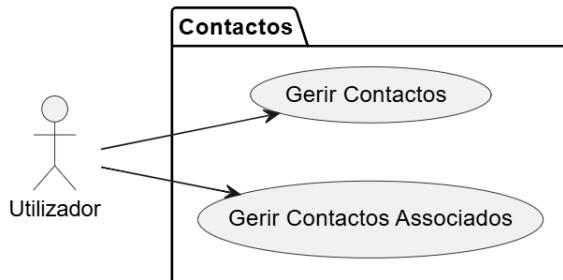


Figura 14 – Casos de Uso dos Contactos

4.10. Identidade e Acesso (IAM)

Este módulo é responsável pela gestão de permissões e empresas. É acessível apenas a utilizadores com permissões de administrador, permitindo adicionar, editar e remover empresas e permissões do sistema. Este módulo já vem parcialmente configurado nos *templates* da entidade de acolhimento, servindo como base para a implementação da lógica de autenticação e controlo de acessos da aplicação.

O sistema foi desenvolvido com suporte a multiempresa (*multitenant*), garantindo que os dados de cada empresa estejam logicamente isolados. Cada utilizador interage apenas com a informação da empresa a que pertence, sendo que apenas o administrador tem acesso global.

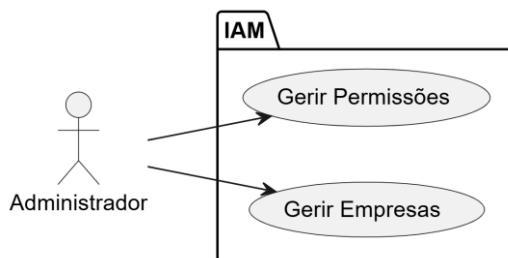


Figura 15 – Casos de Uso de Identidade e Acesso

4.11. Utilizadores

Este módulo permite a gestão dos utilizadores da plataforma, estando acessível apenas a administradores. Este módulo já vem parcialmente configurado nos *templates* da entidade de acolhimento, servindo como base para a implementação da lógica de autenticação e controlo de acessos da aplicação.

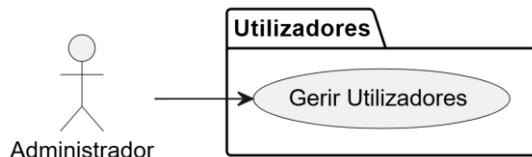


Figura 16 – Casos de Uso de Utilizadores

5. Processo de Processamento Salarial

O processamento salarial constitui um dos processos mais críticos no contexto da gestão de obras, uma vez que envolve a consolidação de informação proveniente de diferentes módulos, como os recursos humanos, os planeamentos e os custos. Para garantir a correção dos salários, é necessário considerar diferentes tipos de remuneração (fixa ou por hora), faltas, adiantamentos, ajustamentos e subsídios.

Na Figura 17 apresenta-se um diagrama de sequência que ilustra o fluxo do processo de processamento salarial, destacando as interações entre os módulos do sistema e a base de dados. Este diagrama reflete de forma clara como a aplicação integra informação dispersa e a transforma em resultados consistentes, permitindo a posterior validação por parte do contabilista da empresa.

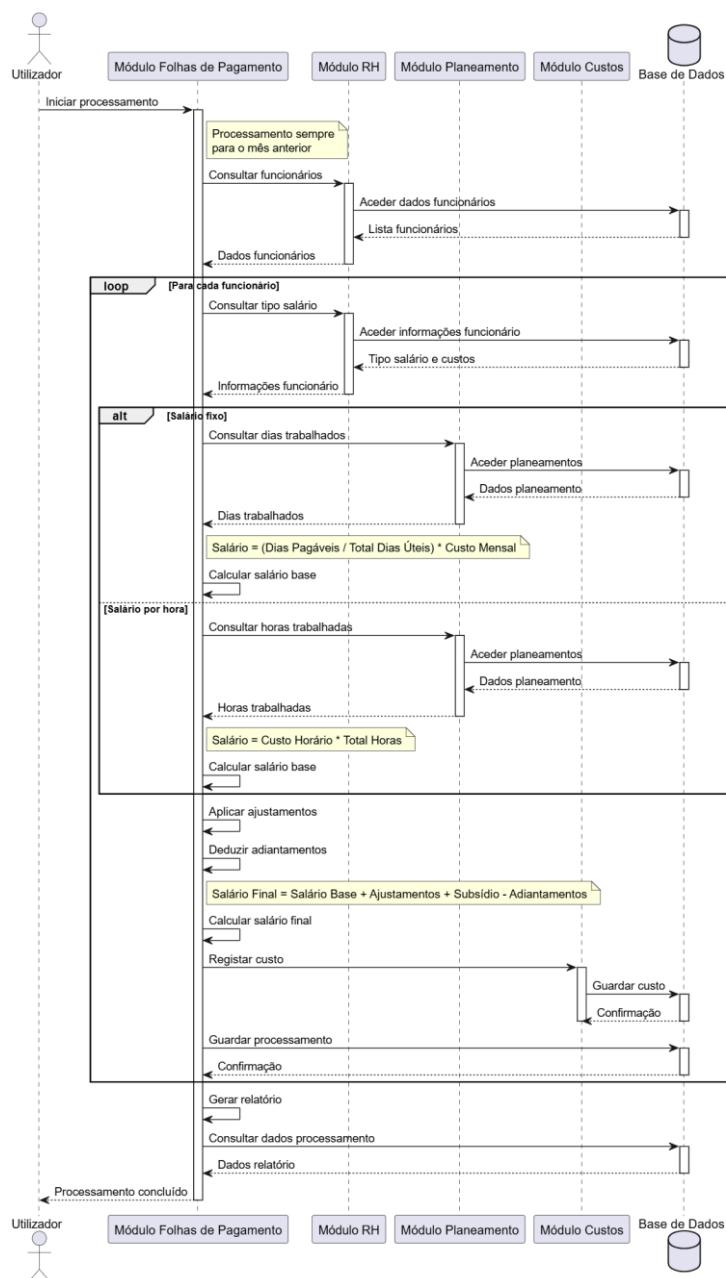


Figura 17 – Processo de processamento salarial

6. Arquitetura Conceptual

6.1. Arquitetura do Sistema

A Figura 18 representa a arquitetura do sistema da aplicação desenvolvida, composta por um conjunto de microserviços organizados por domínio de negócio, cada um comunicando com uma mesma base de dados em *SQL Server*. Apesar dos microserviços partilharem a mesma de base de dados, cada um está logicamente isolado através da utilização de *schemas* independentes, garantindo a separação dos dados e responsabilidades. Assim, permitindo manter os princípios de modularidade e encapsulamento, essenciais numa arquitetura baseada em microserviços.

O cliente através da aplicação web construída em *React* comunica com cada um dos microserviços através de uma *API Gateway*, que é uma aglomeração de todos os *endpoints* de cada um dos microserviços, criando assim uma camada de abstração que irá ocultar a complexidade da arquitetura, tornando-a transparente para o utilizador final. Tanto a *API Gateway* como os microserviços foram construídos com a linguagem de programação *C#* com a *framework* *.Net Core*, utilizando como bibliotecas comuns: *MediatR*, *FluentValidations* e *AutoMapper*. Cada microserviço segue o padrão arquitetural *Clean Architecture*, estando internamente dividido em camadas bem definidas: *API*, que atua como a camada de aplicação; *Domain* e *Storage*, que atua como camada de infraestrutura.

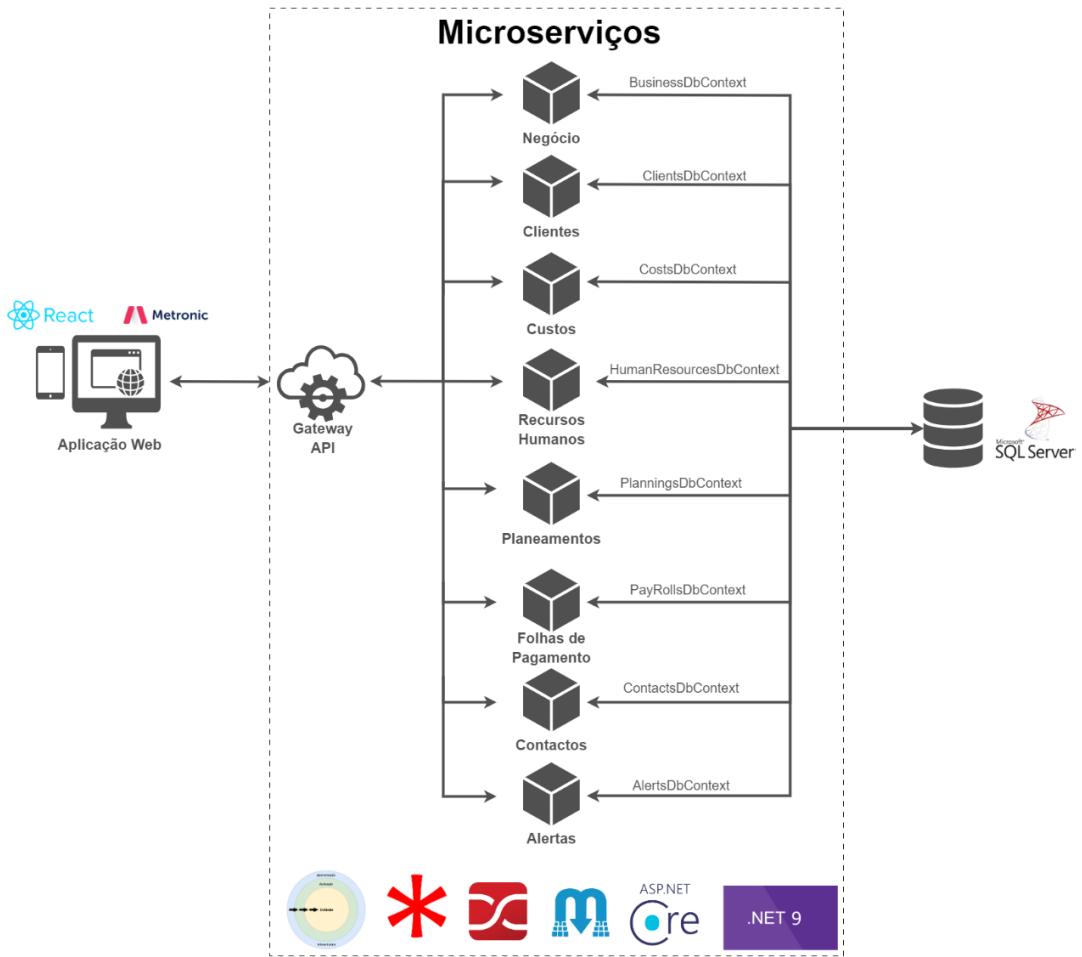


Figura 18 – Arquitetura do Sistema

6.2. Ciclo de Vida de um Pedido

Na Figura 19 está representado o ciclo de vida de um pedido realizado ao *backend*, onde, quando um cliente realiza um pedido à API através da aplicação Web, o controlador do ASP.NET Core responsável por esse *endpoint* irá chamar *MediatR*, que atua como um mediador, e reencaminhar o pedido para o respetivo *CommandHandler*, cada *handler* executa a lógica associada ao pedido e comunica com o repositório, abstraindo o acesso à base de dados, e após o processamento, o resultado é devolvido ao cliente como resposta, promovendo dessa forma, uma arquitetura mais escalável e fácil de manter.

Este fluxo representa a aplicação do padrão CQRS, onde as operações de leitura (queries) e de escrita (commands) são separadas em *handlers* distintos, permitindo separar e isolar responsabilidades.

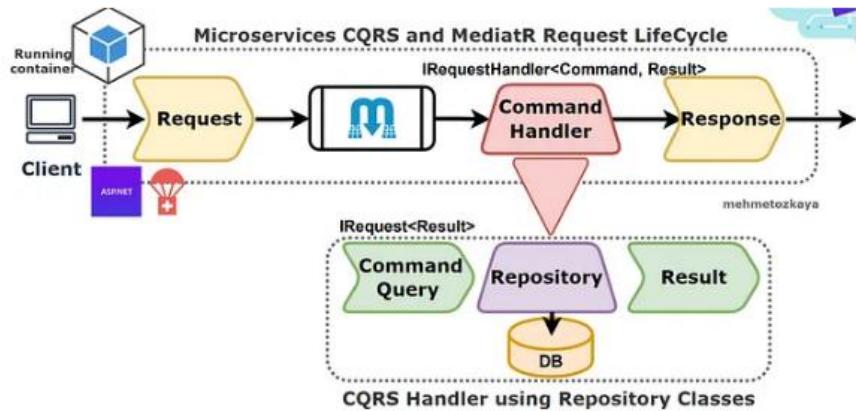


Figura 19 – Diagrama do ciclo de vida de um pedido

6.3. Tecnologias Utilizadas

6.3.1. Tecnologias Fundamentais

O desenvolvimento da aplicação recorreu a um conjunto de tecnologias amplamente utilizadas no mercado, que foram escolhidas pela entidade de acolhimento por integrarem o seu ecossistema de desenvolvimento.

No backend, foi utilizado o .NET Core⁶, em conjunto com o ASP.NET⁷ Core para a construção dos microserviços. Estas tecnologias foram selecionadas pela sua robustez, e pela facilidade de integração dos padrões CQRS e Clean Architecture. Para a camada de persistência, recorreu-se ao SQL Server⁸ como sistema de gestão de base de dados relacional, em conjunto com o Entity Framework⁹, que facilitou a comunicação com a base de dados através de mapeamento objeto-relacional (ORM).

No frontend, foi utilizada a biblioteca React.js, escolhida pela sua ampla utilização e fiabilidade no desenvolvimento de interfaces modernas. A maturidade desta tecnologia e o vasto ecossistema de bibliotecas de apoio tornam-na uma solução adequada para a criação da aplicação de controlo e gestão de obra, que precisa de ser escalável e fácil de manter.

6.3.2. Frameworks, Bibliotecas e Ferramentas de Apoio

O projeto também contou com ferramentas, bibliotecas e frameworks auxiliares que assistem as tecnologias principais, ajudando a reduzir o esforço de desenvolvimento e a manter boas práticas

⁶ <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>

⁷ <https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet>

⁸ <https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver17>

⁹ <https://learn.microsoft.com/en-us/aspnet/entity-framework>

de qualidade e organização. Tendo um papel fundamental para a produtividade e qualidade do software, sendo elas:

- **Jira**¹⁰ – utilizado para gestão de tarefas e acompanhamento das issues, garantindo rastreabilidade no ciclo de desenvolvimento;
- **Metronic**¹¹ – um template de UI que acelerou a construção do frontend, oferecendo uma base consistente para a identidade visual da aplicação;
- **AutoFixture**¹² – uma biblioteca de geração automática de dados de teste, que simplificou a implementação de testes ao reduzir a necessidade de criação manual de objetos complexos;
- **MediatR** – implementa o padrão Mediator, centralizando a comunicação entre componentes e suportando a aplicação do padrão CQRS
- **FluentValidation** – fornece uma forma declarativa e consistente de aplicar validações de negócio;
- **AutoMapper**¹³ – Facilita o mapeamento entre entidades e DTOs, reduzindo a duplicação de código e erros associados à transformação manual de objetos.

6.3.3. Biblioteca Interna: DiRootie Core

Para o desenvolvimento do projeto, uma parte fundamental da infraestrutura técnica foi a utilização da biblioteca DiRootieCore, que é uma biblioteca interna desenvolvida pela própria empresa para acelerar e padronizar o desenvolvimento de aplicações *backend*, sendo inicialmente criada por um ex-colaborador da DiRoots, sendo posteriormente restruturada, servindo agora de base para todos os projetos *backend* da empresa.

O DiRootieCore funciona como uma camada de abstração por cima do *Entity Framework*, *MediatR*, *AutoMapper* e outras tecnologias, oferecendo um conjunto de extensões e convenções que facilitam significativamente o desenvolvimento de microserviços em *.NET*. A utilização desta biblioteca é obrigatória em todos os projetos da DiRoots, pois garante uniformidade.

Entre as funcionalidades mais relevantes da biblioteca destacam-se:

- **Geração automática de Views de leitura:** após a restruturação da biblioteca, tornou-se possível gerar *views* aplicacionais de forma automática, sem necessidade de escrever SQL

¹⁰ <https://www.atlassian.com/software/jira/guides/getting-started/introduction>

¹¹ <https://keenthemes.com/metronic>

¹² <https://github.com/AutoFixture/AutoFixture>

¹³ <https://automapper.io>

manualmente ou configurar mapeamentos extensos no *Entity Framework*. Estas *views* funcionam como projeções da base de dados, utilizadas para leitura eficiente, sendo fundamentais para a aplicação da arquitetura em microserviços.

- **Facilidade na criação de operações CRUD:** o DiRootieCore oferece uma infraestrutura base que elimina a necessidade de escrever manualmente os *handlers* de comandos mais comuns (criação, leitura, atualização e eliminação) para cada entidade. Esses comportamentos são definidos com base em classes genéricas que podem ser reutilizados, permitindo quem programa focar-se apenas nas regras de negócio específicas.

Durante o projeto, foram utilizados três módulos principais da biblioteca:

- **DiRootie.Extensions.EntityFramework** – estende o comportamento do *Entity Framework* com convenções próprias da empresa, como suporte a soft delete, repositórios assíncronos genéricos, entre outros;
- **DiRootie.Extensions.MediatR** – define a estrutura de *handlers*, notificações e comandos utilizados ao longo da aplicação, incluindo integração com as pipelines de validação;
- **DiRootie.Extensions.AutoMapper** – fornece suporte ao mapeamento entre objetos de domínio, *DTOs* (denominados de recursos neste projeto) e comandos, facilitando a conversão entre camadas de forma transparente e configurável.

Para além destes, a biblioteca contém muitos outros módulos e funcionalidades não exploradas diretamente neste projeto, como o suporte a alguns serviços do *azure*, utilitários para facilitar a manipulação de ficheiros, pagamentos, entre outros. Todo o *template* do *backend* é suportado pelo DiRootieCore, sendo esta biblioteca o pilar da arquitetura utilizada pela empresa, e um dos principais responsáveis pela organização, robustez e consistência entre as aplicações desenvolvidas.

Cap. III – Metodologia de operacionalização do trabalho

7. Desenvolvimento da Solução

7.1. Comunicação entre Microserviços

Como uma arquitetura em microserviços decompõe uma aplicação monolítica em um conjunto de serviços independentes, então a principal preocupação para a integração da arquitetura é a forma como diferentes serviços vão comunicar entre si. Numa abordagem tradicional, a comunicação é pensada de modo a maximizar o desacoplamento entre os serviços, garantir resiliência e permitir a escalabilidade dos serviços de forma independente.

Numa abordagem tradicional de comunicação entre microserviços, os serviços comunicam entre si de duas formas:

- **Assíncrona** – através de *message brokers* como o *RabbitMq*, *Kafka* ou *Azure Service Bus*, cada serviço publica eventos num canal, e os serviços interessados reagem a esses eventos de forma independente.
- **Síncrona** – através de APIs REST ou outras formas de comunicação síncrona (ex: *gRPC*, *SOAP*), cada serviço chama diretamente outro e espera por uma resposta. É uma abordagem simples, no entanto cria dependências e aumenta o acoplamento, que é algo que se quer evitar com a adoção de uma arquitetura em microserviços.

Embora os microserviços partilhem a mesma base de dados e estejam a executar dentro do mesmo processo, os mesmos não podem partilhar informação diretamente entre si, comportando-se como se estivessem fisicamente separados. Numa abordagem em microserviços tradicional, onde os microserviços estão fisicamente separados, os mesmos tendem a comunicar de forma assíncrona, através da utilização de *message brokers* como o *kafka* ou o *RabbitMQ*, mas como no projeto desenvolvido eles coexistem dentro do mesmo processo, a DiRoots optou por uma abordagem mais simplificada, recorrendo a um modelo de eventos internos geridos pela biblioteca *MediatR*.

Sempre que uma operação num microserviço precisa de desencadear ações noutros serviços, é criado um evento de domínio que é publicado via *IMediator*. Este evento é imediatamente tratado por um ou mais *handlers* registados, no mesmo fluxo de execução, garantindo que toda a lógica necessária é executada de forma coordenada e consistente, sem dependência de mecanismos externos de troca de mensagens.

Por exemplo, no caso da eliminação de um Cliente (*Customer*), é publicado um evento que ativa a lógica para a eliminação dos contactos associados, sem recurso a *ON DELETE CASCADE* no SGBD. Toda esta lógica é tratada de forma coordenada no código, assegurando a consistência dos dados entre módulos sem depender de integrações externas nem de mecanismos de troca de mensagens assíncronas.

Apesar de se tratar de uma abordagem síncrona, acaba por não ter muito impacto em termos de acoplamento, porque toda a comunicação acontece entre módulos que partilham o mesmo processo de execução, não existindo assim chamadas na rede, nem dependências externas, mantendo assim baixo acoplamento estrutural, promovendo uma separação clara de responsabilidades sem introduzir complexidade desnecessária. Embora esta abordagem não siga à risca algumas práticas comuns em arquiteturas de microserviços, como a comunicação assíncrona e a independência total de *deploy*, alinha-se com os princípios fundamentais de modularidade, escalabilidade, permitindo também manter o código mais limpo e organizado. Esta decisão, tal como foi referido na Secção 2.4, reflete uma escolha consciente da empresa, que favorece soluções mais simples e controladas, adaptadas ao seu contexto e recursos.

Para ilustrar esta abordagem, pode-se considerar o caso da eliminação de um cliente, onde, sempre que um cliente é apagado, é necessário garantir que todos os contactos associados sejam também removidos para manter a consistência dos dados. Para isso, foi criado um evento de domínio chamado *DeleteContactsBatchData*, que encapsula o identificador da entidade associada (neste caso, o cliente). Este evento é publicado via *IMediator* no final do processo de eliminação do cliente.

O evento é tratado por um *handler* registado no módulo dos contactos, que mapeia o evento para um comando específico (*DeleteContactsBatchDataCommand*) responsável por eliminar os registo no repositório. Esta estrutura permite que outros módulos, como fornecedores, alojamentos ou subcontratantes, reutilizem o mesmo evento para acionar a eliminação dos contactos associados à sua entidade principal, sem duplicar lógica nem acoplar diretamente os serviços.

A Figura 20 representa esse fluxo de forma resumida, destacando os passos envolvidos na eliminação de um cliente e respetivos contactos. Embora os microserviços (Clientes e Contactos) coexistam no mesmo processo, cada um mantém a sua lógica de negócio encapsulada, tratando eventos de forma independente. Desta forma consegue-se recriar o princípio da modularidade típica das arquiteturas em microserviços, mesmo não existindo uma separação física entre os serviços.

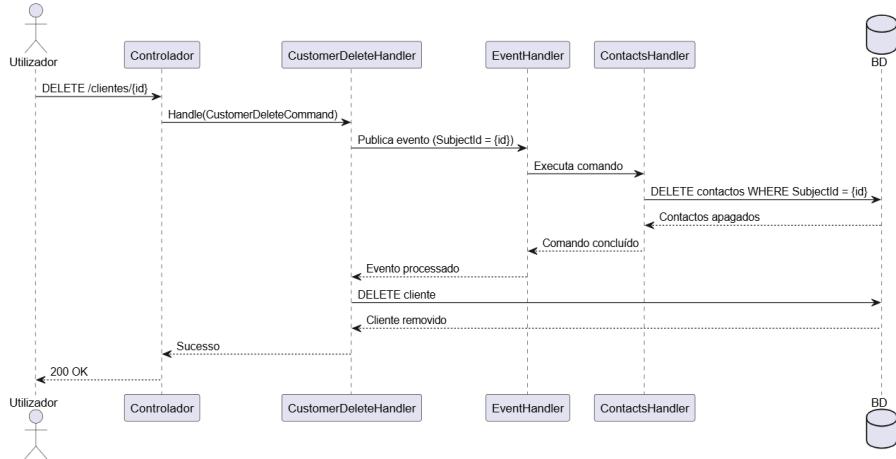


Figura 20 – Fluxo simplificado de remoção de Cliente e contactos associados

7.2. Implementação do Padrão CQRS

A implementação do padrão CQRS no backend depende fortemente das bibliotecas *AutoMapper*, e sobretudo da biblioteca *MediatR*, sendo esta última a espinha dorsal de todas as operações CRUD do Backend. O *MediatR* define interfaces que serão implementadas tanto pelos comandos como pelas *queries* permitindo com que estes sejam tratadas como *handlers* distintos, onde dentro dos mesmos será utilizada a biblioteca *AutoMapper* para realizar o mapeamento dos atributos dos comandos para os atributos da respetiva entidade presente na camada de domínio.

Como serão implementadas diversas entidades com comportamentos muito semelhantes, então serão implementadas sobre os comandos um conjunto de classes e interfaces genéricas que para além de definirem regras sobre aquilo que um comando deve ter, permitem generalizar e automatizar comportamentos genéricos a todas as entidades, permitindo com que seja necessário apenas detalhar comportamentos específicos únicos a cada entidade, reduzindo o tamanho do código e a complexidade do código, ao criar generalizar as operações CRUD.

Na Figura 21 está representado um exemplo de comando para a criação de um registo da entidade *Contoso*. As interfaces da biblioteca *MediatR*, nomeadamente *IRequest* e *IRequestHandler*, servem de base para a definição dos comandos e dos respetivos *handlers*. Sobre estas interfaces, foram criadas interfaces e classes genéricas adicionais, que estabelecem regras e contratos que todos os comandos devem cumprir. No exemplo apresentado, observa-se que todos os comandos que herdam da classe *CreateCommand* devem possuir obrigatoriamente um atributo denominado *EntityId*, garantindo assim a consistência e a uniformidade na implementação das operações de criação para diferentes entidades.

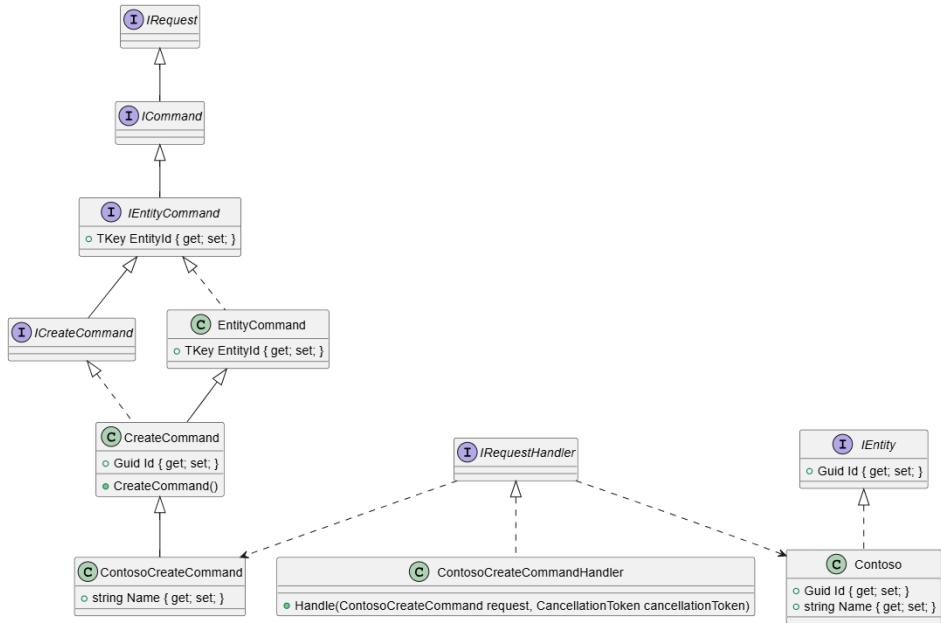


Figura 21 – Diagrama de Classes do Padrão CQRS para a Entidade Contoso

Um dos elementos centrais da Figura 21 é a interface **IRequestHandler**, que representa o componente responsável por processar os comandos e *queries* da aplicação. Neste exemplo, o **ContosoCreateCommand** tem um *handler* específico, no entanto, a operação de criação é uma operação genérica, portanto, na aplicação desenvolvida, esta classe é um *handler* genérico que pode ser utilizado por qualquer classe para definir o comportamento de criação de uma entidade.

Este modelo, onde são criados um conjunto de utilitários por cima dos utilitários já existentes, permite uma elevada reutilização de código, permitindo um rápido desenvolvimento e facilidade na manutenção da aplicação, uma vez que comportamentos comuns são centralizados nas classes e interfaces genéricas, enquanto as particularidades de cada entidade são implementadas apenas quando as mesmas mostram-se necessárias.

7.3. Funcionalidades CRUD no Frontend

As principais funcionalidades da interface Web desenvolvida em *React* consistem na gestão de entidades através de operações CRUD (Criar, Ler, Atualizar e Apagar), elas estão presentes em praticamente todos os módulos do sistema, portanto, de modo a facilitar o processo de desenvolvimento e manutenção, permitindo com que a aplicação cresça rapidamente de forma consistente e rápida, a interface segue uma abordagem modular e reutilizável, onde, sempre que existe uma funcionalidade comum a todos ou a um conjunto específico de entidades, é criado um componente genérico que irá abstrair os detalhes do funcionamento interno dos mesmos e possibilitar a sua reutilização para qualquer entidade.

Quase todas as entidades seguem a seguinte estrutura, podendo conter mais ou menos funcionalidades às apresentadas conforme for necessário:

- **Tabela de listagem:** Apresenta os registos existentes, com paginação, pesquisa e seleção múltipla;
- **Botões de ação:** Permitem criar, editar ou remover registos diretamente a partir da tabela;
- **Modais de formulário:** Utilizados para criar ou editar entidades, com validações em todos os campos através da biblioteca *formik*;
- **Modais de confirmação:** Garantem que ações destrutivas (remoção), são confirmadas pelo utilizador, para que os registos não sejam removidos de forma acidental.

Todas as entidades, cuja informação na base de dados precise de ser manipulada na aplicação desenvolvida, começam com um componente *ManagementList*; esse componente, não genérico, é um padrão utilizado em todas entidades que irá conter toda a informação da mesma. Na Figura 22 encontra-se representado um componente exemplificativo do tipo *ManagementList*, onde é possível notar a modularidade e a reutilização presente na aplicação, pois todas as principais funcionalidades descritas de uma entidade encontram-se nos poucos componentes ali presentes.

```
const ContosoManagementList = () => {
  const {itemIdForUpdate, modalType} = useListView()
  return (
    <>
      <KTCard>
        <ContososListHeader />
        <TableBuilder
          | items={useQueryResponseData()}
          | columnsArr={contososColumns}
          | tableId='#kt_table_contosos'
          | hasColumnVisibility={true}
        />
      </KTCard>
      {itemIdForUpdate !== undefined && modalType === '' && <ContosoEditModal />}
    </>
  )
}
```

Figura 22 – Exemplo do componente "ContosoManagementList" para a gestão da entidade.

Figura 23 – Exemplo da tabela de gestão da entidade "Contoso" apresentada na interface web

A organização modular do *frontend* pode ser observada na estrutura de pastas na Figura 24, onde cada entidade possui uma pasta própria, onde encontram-se subpastas para os diferentes tipos de componentes, não reutilizáveis, como: cabeçalhos, modais de edição e um array de definição de colunas. Esta estrutura facilita a separação de responsabilidades e a reutilização de componentes genéricos em diferentes contextos.

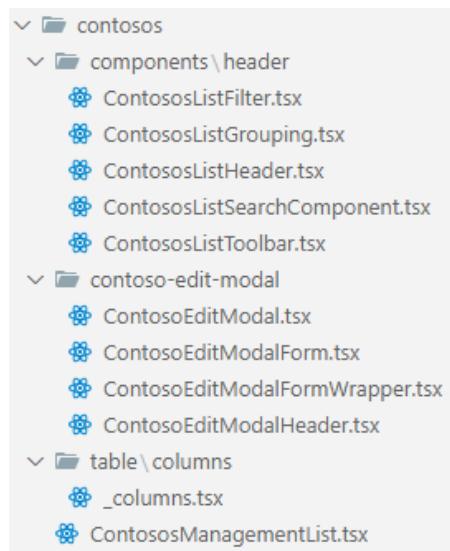


Figura 24 – Estrutura de pastas da entidade "Contoso" no Frontend

Todos os componentes genéricos que envolvem uma entidade precisam de estar cientes de todos os atributos da mesma, e precisam de ter uma forma de comunicar aquilo que pretendem ou não mostrar ou modificar para os outros componentes da aplicação. Para isso, é adotada a definição de um *array de colunas* para cada entidade. Este *array* funciona como um contrato entre os componentes, especificando de forma clara quais os dados relevantes, como devem ser apresentados e que operações estão disponíveis para cada registo.

Desta forma, todas as entidades precisam de ter definido um *array* de colunas, que descreve quais os campos a apresentar, os respetivos títulos, formatações e as ações disponíveis (como editar ou remover). Um exemplo de utilização deste *array* pode ser visualizado na Figura 25, onde o componente genérico da tabela utiliza o mesmo para construir dinamicamente a interface, tornando possível adaptar rapidamente a apresentação e funcionalidades de cada tabela apenas com a configuração deste *array*, sem necessidade de alterar o código do componente base.

```
export const contososColumns: ReadonlyArray<Column<Family>> = [
  {
    id: 'selection',
    headerName: '',
    formatter: ({...props}) => (
      <SelectionCell
        | id={props.row.original.id}
        | listView={useListView()}
        | target='#kt_table_contosos'
      />
    ),
  },
  {
    id: 'name',
    headerName: 'TABLE.NAME',
    field: 'name',
    className: 'min-w-125px',
  },
  {
    id: 'description',
    headerName: 'TABLE.DESCRIPTION',
    field: 'description',
    className: 'min-w-125px',
  },
  {
    id: 'actions',
    headerName: 'TABLE.ACTIONS',
    className: 'min-w-150px text-center',
    formatter: (props) => (
      <ActionsCell
        | {...}
        | buttons: [
          {
            text: 'ACTION.EDIT',
            display: true,
            onClick: openModalForm(props.row.original),
          },
          {
            text: 'ACTION.REMOVE',
            display: true,
            onClick: onRemoveEntity(props.row.original),
          },
        ],
      />
    ),
  },
]
```

Figura 25 – Exemplo de definição do array de colunas para a entidade "Contoso"

7.4. Internacionalização

No desenvolvimento de software existe um processo denominado de *i18n*, abreviação de *internationalization* ("i" é a primeira letra e "n" a última, contendo 18 letras no meio), que se refere ao processo de internacionalização de uma aplicação, normalmente estando mais focado no aspecto linguístico, como foi o caso no projeto desenvolvido.

No *React* existem diversas bibliotecas criadas pela comunidade que auxiliam o processo de internacionalização de uma aplicação, cada uma tem as suas vantagens e desvantagens, mas de modo abrangente fazem todas o mesmo, portanto, para não acrescentar redundância nas dependências do projeto, utilizou-se o módulo *i18n* da biblioteca *metronic*, que tem como base a biblioteca *react-intl*, que já estava a ser utilizada para construir a parte visual da aplicação. O *metronic* tem uma pasta separada só com os ficheiros destinados para a parte de internacionalização (*i18n*), contendo lá dentro ficheiros JSON com todas as *strings* utilizadas durante o desenvolvimento da aplicação, cada elemento contém uma chave, que é comum a todos os ficheiros, sendo que cada ficheiro representa um idioma diferente, mudando apenas o corpo do elemento, que é a *string* traduzida para o idioma do ficheiro.

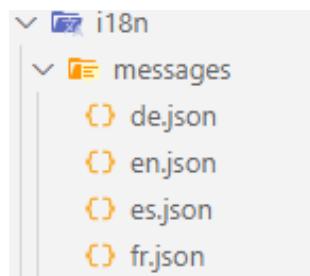


Figura 26 – Estrutura de ficheiros JSON utilizados na internacionalização da aplicação

Na aplicação Web desenvolvida, o utilizador pode mudar o idioma através da barra superior da aplicação. Ao clicar no seu perfil, pode escolher um dos seguintes idiomas:

- Inglês;
- Francês;
- Espanhol;
- Alemão

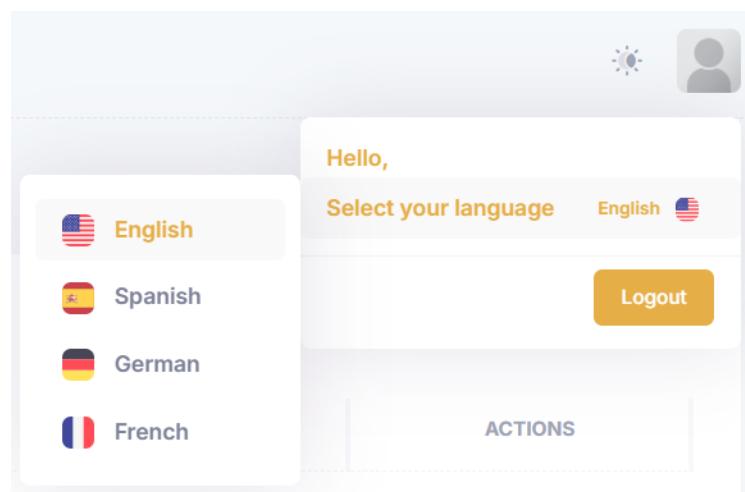


Figura 27 – Opção de seleção de idiomas no Frontend

7.5. Exportação de dados para Excel

Uma das funcionalidades implementadas na aplicação foi a exportação dos dados para um ficheiro *Excel*, permitindo os utilizadores gerar relatórios personalizados numa ferramenta que já conhecem e com a qual estão provavelmente habituados a trabalhar, facilitando assim o processo de tratamento, procura e análise de informação.

A exportação dos dados é realizada diretamente através da interface da aplicação, basta o utilizador clicar no botão para exportar (caso essa entidade suporte a funcionalidade), localizado acima da tabela com os dados da respetiva entidade, e escolher os campos que deseja exportar na caixa de diálogo (modal) que irá abrir.

De modo a melhorar a experiência do utilizador, assim que o mesmo clica no botão de exportar os dados, os dados começam a ser carregados automaticamente assim que o modal é aberto, permitindo que, enquanto o utilizador escolhe os campos que deseja exportar, internamente o sistema já esteja a carregar os dados para a exportação, reduzindo o tempo de espera para gerar o ficheiro *Excel*.

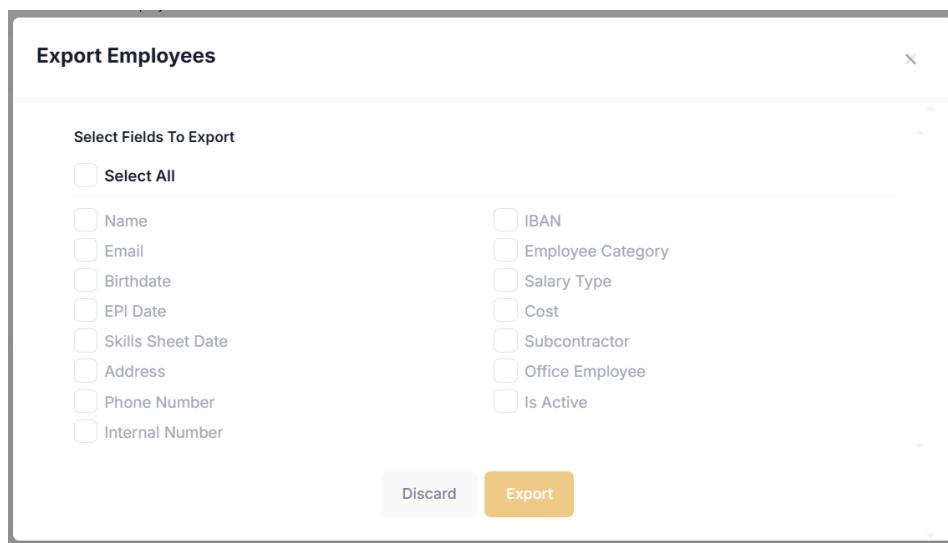


Figura 28 – Modal para a seleção dos campos dos Funcionários para exportar

Para realizar a exportação foi utilizada a biblioteca *ExcelJS*, que é uma biblioteca escrita em *JavaScript* e *TypeScript* que permite a leitura, manipulação e escrita de dados para ficheiros *xlsx*. Sobre esta biblioteca foram construídos um conjunto de utilitários genéricos, completamente independentes de qualquer entidade específica da aplicação, com o objetivo de facilitar e padronizar o processo de exportação. Desta forma, tornou-se possível invocar apenas uma única função, que, de forma encapsulada, fica encarregue de carregar os dados, formatar corretamente a folha de cálculo de acordo com os campos selecionados pelo utilizador, formatá-la esteticamente e realizar

automaticamente a transferência do ficheiro Excel para o dispositivo do utilizador, garantindo uma maior reutilização de código.

Name	Email	Birthdate	Address	Phone Number	Employee Category	Salary Type	Cost	Office Employee	Is Active
Bilbo	bilbo@baggons.com	1981-09-22	Shire - Bag End	962514653	Elrond Guest	Fixed	1022	No	Yes
Frodo Baggins	frodo@baggins.com	2001-09-22	Shire - Bag End	913256849	Fellowship of the ring	Hour	1022	No	Yes
Gandalf, the Grey		1922-05-11	Middle Earth	915468978	Fellowship of the ring	Hour	0	No	Yes
Galadriel		1899-02-01	Lórien	912365555	Lórian Habitante	Fixed	10000	No	Yes

Figura 29 – Dados dos Funcionários Exportados para o Excel

```
export const createExcelFile = async (columns: ExcelExportData[], data: any, intl: IntlShape, fileName: string) => {
  const workbook = new Excel.Workbook()
  const worksheet = workbook.addWorksheet(fileName)

  // Set column headers
  worksheet.columns = createWorksheetColumns(columns, intl)

  // Map employee data to rows
  const rows = buildRows(columns, data)
  worksheet.addRows(rows)

  addStylesToWorksheet(worksheet)

  // Create Excel file
  const buffer = await workbook.xlsx.writeBuffer()
  downloadExcelFile(buffer, fileName)
}
```

Figura 30 – Função criada para a exportação de dados para Excel

7.6. Processamento Salarial

Segundo o que foi planeado inicialmente, a aplicação deveria ter um módulo de folhas de pagamento, onde, segundo as informações colocadas nela como em outros módulos da aplicação, deveria ser possível calcular o salário de todos os funcionários tendo em conta as informações do mês anterior, e posteriormente gerar um relatório a ser entregue aos contabilistas das empresas que pagaram pela aplicação, para que estes possam retificar as informações e proceder ao pagamento dos salários.

Infelizmente não foi possível implementar todas as funcionalidades propostas inicialmente em tempo útil para a entrega do projeto, tendo sido implementado somente a gestão de adiantamentos, ajustes salariais e o cálculo do salário para um funcionário em específico tendo como base os dados associados do mês anterior, faltando implementar:

- O cálculo em lote dos salários para todos os funcionários;
- Integração com o módulo de custos;
- Exportação dos resultados do processamento salarial para um ficheiro *pdf*, que serviria como base para que os contabilistas das respetivas empresas pudessem executar os pagamentos dos salários.

Esta funcionalidade, pela sua natureza, representa uma parte sensível do sistema, uma vez que pode impactar diretamente os valores a pagar aos colaboradores ou causar prejuízos às empresas, logo, o desenvolvimento da mesma torna-se bastante exigente, pois é necessário interagir com muitas tabelas, e aplicar bastantes regras de negócio, que fazem com que o número de casos a testar aumente, obrigando com que exista um equilíbrio constante entre o desenvolvimento e a validação, para que não existam erros que possam afetar os salários ou os custos da empresa.

Apesar de não ter sido possível concluir totalmente o processamento salarial, o trabalho desenvolvido até à entrega constitui uma base sólida para a continuação do desenvolvimento desta funcionalidade, visto que as entidades já se encontram criadas, restando somente integrá-las com o restante do sistema.

7.7. Testes de Integração

Durante o desenvolvimento das funcionalidades CRUD, foram realizados testes de integração com o apoio da biblioteca interna *DiRootie Core*, que disponibiliza uma infraestrutura preparada para este tipo de testes. Os testes de integração de uma entidade eram sempre realizados após a implementação da mesma, sendo que o *pull request* correspondente não era aceite pelo supervisor caso os testes não existissem ou não estivessem a passar.

Os testes de integração foram utilizados principalmente para:

- Garantir que os controladores estavam corretamente ligados aos seus respetivos handlers;
- Verificar que os dados eram corretamente persistidos nas tabelas

Embora o foco tenha sido nas operações básicas de leitura e escrita, os testes de integração foram bastante úteis, pois garantiram que as funcionalidades implementadas estavam a funcionar corretamente. Sempre que uma alteração comprometia o comportamento esperado das operações CRUD, os testes falhavam, servindo assim como um indicador fiável de que algo havia deixado de funcionar com o novo código introduzido.

7.8. Automatização de Trabalho Repetitivo

7.8.1. Script para a Criação de Módulos e Entidades no Frontend

Para que o código seja fácil de ler e manter, é necessário que todo o projeto tenha um conjunto de regras sobre o código e como os ficheiros são organizados nas respetivas pastas. Essas regras fazem com que o código entre módulos e entidades sejam bastante semelhantes entre si, logo,

sempre que era preciso criar algo, copiava-se um módulo ou entidade já existente, e gastava-se entre meia a uma hora somente a alterar nomes de ficheiros, componentes, atributos e importações.

Essa forma é mais eficiente do que estar a escrever todo o código necessário do começo, no entanto, como é um processo repetitivo e demoroso, distrações aconteciam e acabava-se por gerar erros que demoravam muito para ser identificados, como um erro ao renomear algo, ou a não renomeação do mesmo, esta situação acontecia mais do que o esperado devido aos ficheiros que contêm informação de todos os módulos ou entidades, onde muitas vezes os mesmos não eram atualizados.

Para otimizar o processo de desenvolvimento no *frontend* foi criado um *script*, inspirado num já existente para o *backend*, em *Python*, que a partir do nome do módulo, entidade, caminho do projeto e dos atributos e os respetivos tipos da respetiva entidade, gera automaticamente todos os ficheiros necessários. Além disso, o script atualiza ficheiros existentes que requerem informações sobre o novo módulo, reduzindo a possibilidade de erros e poupando tempo a quem desenvolve.

O script desenvolvido é responsável por criar e modificar automaticamente um total de 18 ficheiros distintos. Entre esses ficheiros incluem-se componentes de apresentação, chamadas à API para operações *CRUD*, definições de permissões, rotas e traduções para todas as *strings* utilizadas. O número de linhas de código varia consoante os atributos desejados, mas em média são geradas quase 1000 novas linhas de código, o que torna o processo significativamente mais rápido e menos suscetível a erros manuais.

Note: If the specified module does not exist, it will be created automatically.

```
[?] Enter the project path (to /src folder): C:\Users\User\GitHub\DiProjectManager.Web\src\diprojectmanager-web-www\src
[?] Enter the module name (separate words with spaces): samples
[?] Enter the entity name (separate words with spaces): contoso
[?] Enter the plural form of the entity name (separate words with spaces): contosos
[?] Enter properties ('name type; ...'). Empty if none: name string; description string?
Module samples does not exist.
v Creating Module completed successfully
v Creating Entity completed successfully
```

Figura 31 – Script para a Criação Automática de Módulos e Entidades no Frontend em execução

7.8.2. Interface para traduções i18n

A abordagem de encaixar as *strings* em diferentes ficheiros JSON é um processo bastante repetitivo, mas não muito demoroso a curto prazo, no entanto, consoante o programa cresce, a quantidade de *strings* também irá aumentar, fazendo com que um processo pouco propenso a erros e pouco demorado torne-se cada vez mais suscetível a erros.

É bastante comum que esses ficheiros conterem milhares de linhas, e sempre que é preciso adicionar uma nova *string*, primeiramente é preciso encontrar no ficheiro o melhor lugar para a encaixar. De modo a minimizar esses problemas, decidiu-se adaptar a parte de traduções do código do script para criação de módulos e entidades, e transformar num programa em *Python* onde o utilizador pode inserir uma *string* que precise inserir, a respetiva chave que irá futuramente identificar a mesma, e o programa fará a tradução dessa *string* para os restantes idiomas através da biblioteca do google tradutor para o *Python*, e encontrará a melhor posição dentro de cada ficheiro para a encaixar, facilitando um processo que antes poderia ser lento e demorado, tornando-o menos suscetível a erros.

Para a interface gráfica em *Python*, decidiu-se utilizar a biblioteca *Tkinter* uma vez que se trata de uma solução simples, fácil de implementar e que já vem pré-instalada com o *Python*. Como o programa é destinado a uso interno, a estética não é uma prioridade, e o *Tkinter* permite uma implementação rápida e funcional. Além disso, fazia pouco sentido investir mais tempo no desenvolvimento da ferramenta do que na própria tarefa que ela visa automatizar.

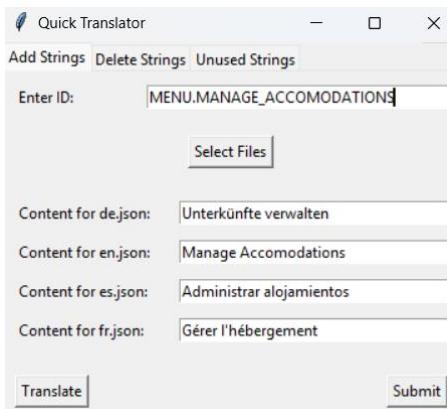


Figura 32 – Interface de Utilitários para Strings i18n

Consoante as necessidades identificadas ao longo do projeto, foram adicionadas novas funcionalidades à aplicação auxiliar. No fim da realização do projeto principal, a aplicação auxiliar para além de traduzir e inserir novas *strings*, passou também a permitir a remoção de uma *string* específica, bem como a identificação e eliminação de *strings* que foram criadas, mas nunca utilizadas no projeto.

8. Processo e Metodologia de Trabalho

8.1. Convenções e Boas Práticas da Entidade de Acolhimento

Quando se trabalha em conjunto com outras pessoas, é preciso com que todos os envolvidos estejam em sintonia, para que isso aconteça, é preciso estabelecer um conjunto de boas práticas para que todos consigam "falar o mesmo idioma", no universo de desenvolvimento de software esse fator mostra-se essencial, pois basta um funcionário não cumprir as normas estabelecidas para que existam dificuldades por parte dos outros trabalhadores a entender o trabalho realizado, e a soma de trabalho heterogéneo ao longo do tempo fará com que o projeto torne-se confuso, difícil de manter e mais propenso a erros, impactando negativamente a produtividade e a qualidade do software desenvolvido.

Para evitar esses problemas, a DiRoots colocou sobre o trabalho realizado as seguintes convenções:

8.1.1. Regras para a Criação de Branches

Uma *branch* no Git é uma ramificação do repositório, permite trabalhar em novas funcionalidades ou correção de erros sem interferir no código principal, facilitando muito o desenvolvimento de código em paralelo. A definição de convenções para a nomenclatura das *branches* é essencial para manter o repositório homogéneo e fácil de identificar propósitos por detrás das alterações.

Como a DiRoots utiliza o Jira¹⁴ para organizar o trabalho a ser desenvolvido, então as *branches* devem ser nomeadas com o código da *issue* que estão ligadas, como por exemplo, se uma *issue* no Jira se chamar [WEB] add employee module, e o código gerado pelo Jira como identificador for DPM-3, então a *branch* deve-se chamar DPM-3.

8.1.2. Regras para Commits

Um *commit* no git é um registo de uma alteração que foi realizada no código do projeto, funciona como um ponto de restauração caso algum erro aconteça e tenha sido identificado quando o mesmo surgiu. Cada *commit* deve representar uma mudança clara e bem definida, de modo a facilitar o rastreio durante o desenvolvimento.

A convenção para *commits* é a seguinte: (nome da branch) # (prefixo): o que foi realizado.

¹⁴ <https://www.atlassian.com/software/jira>

Os prefixos utilizados para *commits* são:

- **fix:** para resolução de erros
- **add:** para adição de algo
- **remove:** para a remoção de algo
- **refactor:** para mudanças de comportamento que não afetam outputs
- **style:** para mudanças visuais
- **docs:** para documentação
- **test:** para mudança em ficheiros de teste

8.1.3. Regras para Pull Requests

Um *pull request* é uma solicitação no GitHub para fundir as alterações realizadas em uma *branch* para outra *branch*.

Todos os *pull requests* realizados devem ser aprovados por um funcionário que não tenha colaborado no desenvolvimento da mudança da *branch*, e o nome do *pull request* deve seguir a seguinte estrutura: (nome da branch) #(prefixo): nome da issue.

Os prefixos utilizados para os nomes dos *pull requests* são os mesmos que dos *commits*, no entanto, para o *pull requests* pode ser utilizado o prefixo "feat" que representa a adição de uma nova funcionalidade.

8.1.4. Regras para a Criação de Migrações no Entity Framework

A utilização do *Entity Framework* para a construção de uma aplicação apresenta como principal vantagem a capacidade de escrever migrações automaticamente, oferecendo uma forma simples e segura de gerir e alterar o esquema de uma base de dados sem a necessidade de escrever SQL.

Como o número de migrações de um projeto é proporcional ao tamanho do mesmo, então, caso não tenham sido estabelecidas convenções para o nome das mesmas, uma ferramenta criada para facilitar a monitorização das alterações de uma base de dados, tornar-se-á desnecessariamente confusa e demorada, visto que, quando for necessário encontrar uma alteração em específico, vai-se perder e imenso tempo desnecessariamente.

Para evitar que isso aconteça, todas as migrações começam com um verbo para identificar a existência de uma adição, remoção ou alteração na migração (Ex: *added* ou *removed*), à frente do verbo é descrito sucintamente o que foi realizado, sendo tudo separado por *underscores*. Como é utilizada uma arquitetura em micro serviços, então cada serviço tem o seu próprio contexto, e as

suas próprias migrações, logo não é necessário especificar o módulo alterado, pois a pasta onde a migração encontra-se revela essa informação.

Exemplo de uma migração: *Added_Employee_PhoneNumber_attribute*

8.2. Abordagem ao Desenvolvimento do Projeto

Durante o desenvolvimento do projeto, não foi adotada uma metodologia ágil formal, como o *Scrum* ou o *Kanban*, no entanto, o trabalho foi realizado de forma iterativa e incremental, seguindo uma abordagem bastante parecida dos princípios ágeis, onde os acontecimentos ocorreram da seguinte maneira:

Inicialmente foi realizada uma reunião de introdução ao projeto, onde me foi apresentado um exemplo de uma aplicação semelhante, desenvolvida anteriormente por um dos colaboradores da empresa durante o seu período como freelancer. Esta reunião serviu como ponto de partida para perceber o tipo de funcionalidades esperadas e o propósito geral do novo sistema.

Cerca de uma semana depois foi realizada uma segunda reunião, onde me foram indicadas as tecnologias e ferramentas que deveria estudar para desenvolver tanto o *frontend* como o *backend* da aplicação. Duas semanas depois, numa nova reunião, foi partilhado o *template* que a empresa exige para todos os projetos *backend*, e uma breve explicação da estrutura do mesmo. Algo importante a destacar, é que este *template* foi restruturado recentemente, sendo este projeto, um dos primeiros da empresa a utilizar a nova versão do *template*.

Numa fase inicial, concentrei-me exclusivamente na implementação das entidades e operações básicas (CRUD) de cada microserviço do *backend*. Após essa etapa, tive outra reunião onde foi partilhado um *template* do *frontend*, bem como discutido o funcionamento pretendido da aplicação e o tipo de experiência que se desejava oferecer ao utilizador.

O *template* fornecido para o *frontend* serviu como base visual e estrutural, permitindo acelerar o início do desenvolvimento da interface. Este *template* continha funcionalidades essenciais como a autenticação via *Azure*, bem como a estrutura para a gestão de utilizadores e empresas, o que foi bastante útil tendo em conta que a aplicação é *multi-tenant*, no entanto, o principal foco deste era estético, que cores utilizar e onde a informação deveria ser apresentada, não sendo tão restrito sobre como as funcionalidades deveriam ser implementadas. Ao longo do desenvolvimento foi necessário adaptar e expandir consideravelmente o *template*, integrando a lógica de negócios, e

várias funcionalidades e componentes específicas do projeto, mas que foram construídas de modo a serem facilmente reaproveitadas para outros projetos da empresa.

A partir desse ponto, o trabalho passou a ser organizado em ciclos de desenvolvimento contínuos, onde, sempre que um conjunto de tarefas era concluída, era realizada uma nova reunião com o orientador de estágio, e um novo conjunto de tarefas era discutido. Frequentemente, estas envolviam o desenvolvimento do *frontend*, mas muitas vezes implicavam também alterações no *backend*, sobretudo ao nível da lógica de negócio quando a mesma mostrava-se mal implementada.

Esta abordagem flexível permitiu que o projeto evoluísse gradualmente, com base em feedback contínuo, garantindo maior alinhamento com os padrões e práticas da empresa.

Cap. IV – Discussão dos Resultados

9. Apresentação e Discussão dos Resultados

Este projeto teve como principal objetivo o desenvolvimento de uma aplicação para a web de controlo e gestão de obra, com uma arquitetura baseada em microserviços, desenvolvida com tecnologias modernas e padrões exigentes, como o CQRS, *Clean Architecture* e uma biblioteca interna da empresa, o *DiRootieCore*, onde, foram implementadas diversas funcionalidades, tanto para o *backend* como para o *frontend*, e, embora nem todos os objetivos iniciais tenham sido totalmente alcançados, os resultados obtidos são sólidos e revelam um progresso significativo.

Entre os módulos concluídos destacam-se a gestão de recursos humanos, folhas de pagamento, contactos, clientes, e a gestão de projetos, o mais importante. A lógica de negócio de cada um destes módulos foi desenvolvida respeitando as convenções da empresa, incluindo a separação entre comandos e *queries*, a utilização de eventos internos com a biblioteca *MediatR*, e a criação de validadores com a biblioteca *FluentValidation*. O *frontend* foi construído a partir de um *template* da empresa, que por sua vez é uma personalização de um *templated* do *Metronic React*, e adaptado para integrar os módulos desenvolvidos, incluindo componentes reutilizáveis, formulários e suporte para múltiplos idiomas.

No entanto, nem todas as funcionalidades propostas foram implementadas, o módulo de custos ficou por concluir, o que impossibilitou a criação dos *dashboards* de análise financeira, já que dependem diretamente desses dados, como também não foi possível implementar o sistema de alarmes, que serviria para alertar os utilizadores em tempo real sobre situações críticas na gestão dos recursos. Apesar disso, as fundações técnicas para estas funcionalidades já se encontram parcialmente estruturadas, o que facilitará a implementação destas no futuro. Outro ponto a destacar é a possibilidade de expansão da funcionalidade de folhas de pagamento, que, apesar de operacional, ainda pode ser melhorada com a exportação dos resultados em formato PDF, funcionalidade relevante para comunicação com os contabilistas das empresas que vierem a adotar o sistema.

Tendo em conta o nível de complexidade da arquitetura, a utilização de bibliotecas internas com pouca documentação e a necessidade de compreender bem o funcionamento do *template* fornecido, os resultados obtidos são bastante positivos. O processo de aprendizagem foi intenso e prolongado, exigindo tempo não só para programar, mas também para estudar os padrões, explorar

os módulos da biblioteca *DiRootieCore*, e compreender como as decisões técnicas da empresa moldam o desenvolvimento.

Embora o tempo disponível não tenha sido suficiente para finalizar todas as funcionalidades propostas inicialmente, a solução construída é sólida, escalável e bem integrada, cumprindo a maior parte dos objetivos definidos no começo do projeto, permitindo uma aplicação prática dos conhecimentos adquiridos ao longo da licenciatura, como também uma exposição realista às exigências do desenvolvimento profissional num ambiente empresarial.

10. Apresentação e Discussão dos Impedimentos e/ou constrangimentos

Durante o desenvolvimento do projeto, surgiram alguns desafios e constrangimentos técnicos que impactaram diretamente o processo e o ritmo de trabalho.

Um dos principais impedimentos iniciais foi a ausência de documentação pública sobre as bibliotecas internas da empresa. Estas bibliotecas, fundamentais para a estrutura do projeto, exigiram um maior esforço na leitura do código, experimentação e esclarecimentos com o supervisor responsável da entidade acolhedora. Esse fator, aliado à curva de aprendizagem do *template*, e ao facto que este seria o primeiro projeto a utilizar o *template* após um longo período de restruturação, fez com que houvesse um impacto considerável na velocidade de desenvolvimento, sobretudo nos estágios iniciais do projeto.

Outro desafio importante foi compreender como a empresa implementa comunicação entre módulos de forma síncrona com *MediatR*, sem recurso a *message brokers*. Apesar de parecer contraintuitivo no contexto de uma arquitetura de microserviços, foi necessário perceber as vantagens desta abordagem num cenário controlado, sem dependências externas e com *deploy* unificado. Esta compreensão foi gradual, mas extremamente enriquecedora para consolidar os conhecimentos sobre arquitetura distribuída e como a mesma pode ser implementada de forma diferente de modo a encaixar-se melhor no domínio do projeto.

Apesar destes desafios, todos os impedimentos foram ultrapassados com sucesso, resultando numa aplicação, onde, mesmo que incompleta, está funcional e com qualidade, fornecendo uma experiência de estágio extremamente rica do ponto de vista técnico e profissional.

Cap. V – Conclusão

11. Reflexão Crítica dos Resultados

O desenvolvimento deste projeto permitiu consolidar conhecimentos técnicos adquiridos ao longo da licenciatura, aplicando-os num ambiente real de trabalho através da construção de uma aplicação modular, baseada em microserviços, sendo possível explorar tecnologias modernas e padrões como o CQRS, *Clean Architecture*, *MediatR* e sobretudo a biblioteca interna da empresa, o DiRootieCore, que sustentou toda a estrutura do projeto.

Mesmo que o trabalho não tenha sido concluído na sua totalidade, os objetivos centrais foram cumpridos, visto que o sistema está funcional, modular, e pronto para evoluir. As entidades base estão criadas e integradas no *backend*, o *frontend* encontra-se operacional, e a lógica de negócio foi implementada de forma robusta e consistente, a ausência de funcionalidades como *dashboards* e alarmes não se deve à falta de planeamento, mas sim à complexidade real do sistema e à necessidade de garantir a integridade e fiabilidade da aplicação. A testagem, embora morosa, revelou-se essencial para assegurar a estabilidade do sistema, obrigando a um equilíbrio constante entre velocidade de desenvolvimento e qualidade final.

A utilização de um *template* inicial padronizado foi fundamental para compreender as convenções e o ecossistema da empresa, no entanto, a implementação das regras de negócio, dos comandos, dos modelos e a adaptação do projeto às necessidades reais exigiu envolvimento ativo, capacidade de investigação e tomada de decisões técnicas fundamentadas.

Este projeto não só consolidou competências como o uso de repositórios genéricos, injeção de dependências, mapeamento automático e organização modular, como também proporcionou um contacto direto com práticas reais do mercado de trabalho, como bibliotecas proprietárias, convenções internas e fluxos colaborativos, elevando significativamente a maturidade profissional e técnica obtida.

12. Conclusões e Trabalho Futuro

O estágio representou uma oportunidade enriquecedora para aplicar na prática o que foi aprendido academicamente, enquanto proporcionou um contacto direto com metodologias, desafios e realidades do desenvolvimento de software empresarial. A experiência de trabalhar com um sistema complexo, com várias camadas, interações entre microserviços, e a adaptação a um

ecossistema técnico fechado, permitiu um grande crescimento a nível técnico, organizacional e pessoal.

Apesar dos resultados atingidos, há ainda trabalho a desenvolver, onde a conclusão do módulo de custos abrirá caminho para a criação de *dashboards*, relatórios e mecanismos de alarme, que são funcionalidades de grande valor para a gestão operacional da aplicação. Também seria benéfico reforçar a cobertura de testes unitários e de integração, e expandir a comunicação entre serviços com mais eventos.

Independentemente das funcionalidades que não foram terminadas a tempo da entrega do relatório, o trabalho realizado foi fundamental, o sistema tem uma base sólida, e tem agora condições para crescer de forma sustentável e controlada. O conhecimento adquirido sobre padrões de arquitetura, práticas modernas de desenvolvimento e integração com bibliotecas internas foi profundo e exigente, mas recompensador.

Referências

- [1] D. Verma e P. Aland, «A Comparative Review of Server Rendering and Client Side Rendering in Web Development», [Em linha]. Disponível em: <https://benevity.com/why-benevity>
- [2] F. Hassan, «Software architecture between monolithic and microservices approach; Feb 25 2024; Software architecture between monolithic and microservices approach», Fev. 2024. [Em linha]. Disponível em: <https://ssrn.com/abstract=4753649>
- [3] M. Kaloudis, «Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends», Frankfurt, 2024. [Em linha]. Disponível em: www.ijacsa.thesai.org
- [4] O. Amaral e M. Carvalho, «Arquitetura de Micro Serviços: uma Comparação com Sistemas Monolíticos».
- [5] K. Wagh e R. Thool, «A Comparative Study of SOAP Vs REST Web Services Provisioning Techniques for Mobile Host», *Journal of Information Engineering and Applications*, vol. 2, 2012, [Em linha]. Disponível em: www.iiste.org
- [6] T. Fielding, «Architectural Styles and the Design of Network-based Software Architectures», University of California, 2000.
- [7] L. Campos Jorge, «Projeto e arquitetura de API REST para sistema de monitoramento de redes ópticas», Brasília, Dez. 2020.
- [8] «CQRS Pattern – Azure Architecture Center | Microsoft Learn». Acedido: 11 de Julho de 2025. [Em linha]. Disponível em: <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>
- [9] K. & Nilsson, «Practitioners' view on command query responsibility segregation», Ago. 2014.
- [10] M. Robert, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*.
- [11] «Mediator». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://refactoring.guru/design-patterns/mediator>
- [12] «Overview of Entity Framework Core – EF Core | Microsoft Learn». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://learn.microsoft.com/ro-ro/ef/core/>
- [13] «Code First Approach vs. Database First in Entity Framework | Built In». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://builtin.com/articles/code-first-vs-database-first-approach>
- [14] «What is .NET Framework? A software development framework | .NET». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework>

- [15] «Announcing ASP.NET Core in .NET 8 – .NET Blog». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://devblogs.microsoft.com/dotnet/announcing-asp-net-core-in-dotnet-8/>
- [16] «ASP.NET vs. ASP.NET Core: The Ultimate Showdown». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://wirefuture.com/post/aspnet-vs-aspnet-core#p-1>
- [17] «Simplifying complexity with MediatR and Minimal APIs – Q Agency». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://q.agency/blog/simplifying-complexity-with-mediator-and-minimal-apis/>
- [18] «AutoMapper – AutoMapper documentation». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://docs.automapper.io/en/stable/>
- [19] «Test Data Generation With AutoFixture in .NET – Code Maze». Acedido: 8 de Julho de 2025. [Em linha]. Disponível em: <https://code-maze.com/csharp-test-data-generation-with-autofixture/>
- [20] «1. What Is React? – What React Is and Why It Matters [Book]». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html>
- [21] «Overview | TanStack Query React Docs». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://tanstack.com/query/v4/docs/framework/react/overview>
- [22] «Introduction – Metronic React». Acedido: 7 de Julho de 2025. [Em linha]. Disponível em: <https://docs.keenthemes.com/metronic-react>