REENGENHARIA DE UMA PLATAFORMA WEB MONOLITICA PARA UM SISTEMA SAAS DE MICROSERVIÇOS APLICANDO OS CONCEITOS *DOMAIN DRIVEN DESIGN* E *REPOSITORY PATTERN*

Aluno

David Patrício Luna

Orientadora

Profª. Doutora Mónica Isabel Teixeira da Costa

Dissertação apresentada à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Desenvolvimento de Software e Sistemas Interativos, realizada sob a orientação científica do Professora Doutora Mónica Isabel Teixeira da Costa, do Instituto Politécnico de Castelo Branco.

Abril de 2024

Composição do júri

Presidente do júri

Prof.ª Doutora, Ângela Cristina Marques de Oliveira

Professora Adjunta da UTC de Informática, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Vogais

Prof. Doutor, Luís Filipe Leite Barbosa

Professor Auxiliar do departamento de Engenharias, Universidade de Trás-os-Montes e Alto Douro.

Prof. Doutor, Alexandre José Pereira Dura da Fonte

Professor Adjunto da UTC de Informática, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Profª. Doutora, Mónica Isabel Teixeira da Costa

Professora Adjunta da UTC de Informática, Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco.

Dedicatória

Aos amores da minha vida, esta dedicatória é uma expressão de gratidão e amor. À minha esposa, Sofia Gonçalves, por ser a minha fonte constante de apoio, paciência e inspiração. Aos meus filhos, Júlia Luna e Pedro Luna, cuja presença trouxe alegria aos meus dias e inspiração na minha jornada académica.

Aos meus pais, Maria Oliveira e Júlio Luna, cujo amor e encorajamento moldaram a pessoa que sou hoje. A eles, sou eternamente grato.

À minha orientadora, Prof. Drª Mónica Costa, pela sábia orientação, pelos conselhos valiosos e pela dedicação incansável ao meu crescimento académico. A sua sabedoria e incansável encorajamento foram decisivos ao longo deste percurso.

A mim mesmo, dedico uma parte especial desta obra. Tenho orgulho no meu esforço, empenho e dedicação a este desafio nesta etapa da minha vida. Que esta conquista seja um marco duradouro no meu percurso académico e inspire os meus filhos a procurarem sempre novos desafios na sua vida.

Este trabalho é dedicado a todos, com profunda apreciação e amor. A cada um que desempenhou um papel vital nesta jornada, tornando-a mais significativa e enriquecedora.

Obrigado por fazerem parte e apoiarem este capítulo fundamental da minha vida.

Agradecimentos

Quando olho para o caminho que percorri ao longo da minha jornada académica e profissional, sinto-me imensamente grato pelas experiências enriquecedoras e pelas pessoas incríveis que encontrei. Gostaria de agradecer às instituições de ensino que moldaram o meu percurso, às empresas que contribuíram para o meu crescimento e, em particular, à 4AllSoftware e ArquiConsult, onde adquiri valiosos conhecimentos que se revelaram fundamentais para o meu desenvolvimento.

Quero endereçar um agradecimento especial a todos os colegas de trabalho no CoLAB ForestWISE, cuja colaboração e partilha de conhecimento tornaram cada dia uma oportunidade de enriquecimento pessoal e profissional. Em particular, gostaria de agradecer ao Prof. Dr. Carlos Fonseca, CTO do CoLAB ForestWISE pela confiança depositada na orientação do Gabinete de Informática. Agradeço também ao Engenheiro Rogério Rodrigues, coordenador executivo do CoLAB ForestWISE, pela liderança inspiradora e pelo apoio constante.

Cada etapa desta jornada foi marcada pela dedicação e apoio de pessoas notáveis, e por isso, quero expressar o meu reconhecimento a todos que, de alguma forma, contribuíram para este capítulo importante da minha vida académica e profissional. Muito Obrigado.

Resumo

Na presente dissertação pretende-se explicar o processo utilizado para a reengenharia de uma aplicação em ambiente Web baseado na *Framework* 4.5 e que já tem um ciclo de vida superior a dez anos. Tendo sido descontinuada a presente *Framework* por parte da Microsoft, decidiu-se reconstruir a nova aplicação recorrendo às últimas *Frameworks* e tecnologias. A nova plataforma é assente na filosofia SaaS (*Software as a Service*), ou seja, cada utilizador só paga pelos módulos e tempo que utiliza e a mesma passa a ser disponibilizada como de um serviço se tratasse.

A nova plataforma, intitulada daqui por diante por Web Cloud será desenvolvida baseada na nova *Framework* .NET 7 e posteriormente migrada para a versão 8, dividida em duas partes: uma API (*Application Programming Interface*) em C# responsável por todas as tarefas de *BackEnd* e comunicação com a base de dados. Esta API permite que sistemas externos desenvolvidos por clientes ou parceiros possam interagir com a mesma. Foi desenvolvido um componente *FrontEnd* em HTML (*Hypertext Markup Language*), CSS (*Cascading Style Sheets*) e *JavaScript* responsável por fazer a ponte entre o utilizador e a API, desenvolvida no sentido de efetuar o maior processamento do lado do cliente, de modo a torná-la mais rápida e amigável, ou seja, *Client Side.*

A API está a ser planeada para tirar o maior partido da arquitetura de micro serviços disponibilizados pela *Framework* .net 7 e do ORM *Entity Framework Core* em conjunto com o ORM *Dapper*, estas são responsáveis por efetuar todos os pedidos e interações com a base de dados em SQL Server. Para esta interação foi implementado o padrão de repositório, centralizado e baseado numa unidade de trabalho para cada interação.

O *FrontEnd* será desenvolvido em HTML conjuntamente com CSS para a parte visual e para a parte algorítmica será utilizado o *JavaScript* com o padrão MVVM *(**M**odel – **V**iew–**V**iew-**M**odel*). Para os componentes visuais das janelas serão utilizados os componentes da Syncfusion, licença comunitária e para manipulação das CSS recorreu-se à utilização da *Framework* Bootstrap na sua última versão 5.0.

No modelo a desenvolver pretende-se deixar todo o código com forte desacoplamento, ou seja, poucas dependências entre as classes, assim como uma grande escalabilidade ao nível de desenvolvimento de novos módulos e gestão da equipa de desenvolvimento.

Todo o processo de desenvolvimento e divisão de tarefas será suportado pela metodologia de *Scrum*, recorrendo ao Azure Devops para a organização dos vários *sprints*, assim como a distribuição das várias tarefas.

Palavras-chave

Web Cloud, BackEnd, FrontEnd, .Net 8, JavaScript, CSS, micro serviços.

Abstract

This dissertation aims to explain the process used to re-engineer a web application based on Framework 4.5, which has already had a more than ten-year lifecycle. Since Microsoft discontinued this framework, it rebuilt the new application using the latest frameworks and technologies. The new platform is based on the SaaS (Software as a Service) philosophy, i.e., each user only pays for the modules and time they use, and the service is made available.

The new platform, Web Cloud, will be developed based on the latest .NET 7 Framework and later migrated to version 8, divided into two parts: an API (Application Programming Interface) in C# responsible for all the BackEnd tasks and communication with the database. This API allows external systems developed by clients or partners to interact with it. A FrontEnd component was developed in HTML (Hypertext Markup Language), CSS (Cascading Style Sheets) and JavaScript, responsible for bridging the gap between the user and the API, developed to carry out the most processing on the client side to make it faster and more user-friendly, i.e. Client Side.

The API is planned to take full advantage of the microservices architecture provided by the .net 7 Framework and the Entity Framework Core ORM in conjunction with the Dapper ORM, responsible for making all requests and interactions with the SQL Server database. The repository standard was implemented for this interaction, centralised and based on a unit of work for each interaction.

The FrontEnd will be developed in HTML together with CSS for the visual part, and JavaScript with the MVVM (Model-View-View-Model) standard will be used for the algorithmic part. For the visual components of Windows, Syncfusion components will be used under a community licence, and to manipulate the CSS, the Bootstrap Framework in its latest version, 5.0, will be used.

The model to be developed aims to leave all the code with strong decoupling, i.e. few dependencies between classes, as well as great scalability in terms of developing new modules and managing the development team.

The entire development process and division of tasks will be supported by the Scrum methodology, using Azure Devops to organise the various sprints and distribute the different tasks.

Keywords

Web Cloud, BackEnd, FrontEnd, .Net 8, JavaScript, CSS, microservices

Índice geral

Índice de figuras

Lista de abreviaturas, siglas e acrónimos

ADO – ActiveX Data Objects

AMQP – Advanced Message Queuing Protocol

API – Application Programming Interface

C# - C XXIIIsharp

CD – Continuous Delivery

CI – Continuous Integration

CLR – Common Language Runtime

CPU – Central Processing Unit

CSS – Cascading Style Sheets

DDD – Domain-Driven Design

DIP – Dependency Inversion Principle

DOM – Document Object Model

DTO – Data Transfer Object

gRPC – Remote Procedure Calls

HTML – Hypertext Markup Language

IDE – Integrated Development Environment

IoC – Inversion of Control

ISP – Interface Segregation Principle

JMS – Java Message Service

JSON – JavaScript Object Notation

Linq – Language Integrated Query

LSP – Liskov Substitution Principle

MQTT – Message Queuing Telemetry Transport

MVVM – Model-View-ViewModel

NAT – Network Address Translation

OCP – Open-Closed Principle

ORM – Object Relational Mapping

Pub – Publisher

RAM – Randoom Access Memory

SaaS – Software as a Service

SDK – Software Development Kit

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

SRP – Single Resposability Principle

STOMP – Simple Text Oriented Messageng Protocol

Sub – Subscriber

UI – User Interface

URL – Uniform Resource Locator

W3C – World Wide Web Consortium

WCF – Windows Communication Foundation

WPF – Windows Presentation Foundation

WWW – World Wide Web

Capítulo 1 - INTRODUÇÃO

1.1 - ENQUADRAMENTO

A evolução das tecnologias de informação tem incentivado as empresas a migrarem das arquiteturas monolíticas para arquiteturas de micro serviços, com o objetivo de conseguir maior escalabilidade, flexibilidade e eficiência. Esta dissertação propõe os passos a seguir desde o estudo das tecnologias, padrões de desenvolvimento e metodologias para a reengenharia de uma plataforma web monolítica para um sistema *Software as a Service* (SaaS) em micro serviços. A abordagem adotada combina os princípios do *Domain-Driven Design* (DDD) [1], *Clean Architeture* [2] e *Repository Pattern* [3] para alcançar uma transição suave e bem estruturada.

A aplicação que se pretende "reconstruir" foi inicialmente desenvolvida como um pequeno portal para uma simples gestão de funcionários da empresa. Mas após a sua utilização por parte dos primeiros clientes, estes foram pedindo mais módulos e, ao longo dos últimos anos, esta teve um elevado crescimento. Para poder dar resposta a todos os pedidos dos clientes foi-se tornando difícil a sua migração entre tecnologias, à velocidade que estas o exigiam. O próprio código foi crescendo, de forma um pouco desorganizada e com base no copiar/colar de umas janelas para outras.

Esta situação levou a que fosse necessário criar uma aplicação, mais capacitada para dar resposta às necessidades atuais, nomeadamente, com menos limitações e maior organização e possibilidade de evoluir.

A arquitetura monolítica tradicional tem sido uma escolha comum para o desenvolvimento das aplicações Web. No entanto, à medida que as necessidades de escalabilidade e performance aumentam, surgem limitações significativas neste modelo. A adoção da arquitetura de micro serviços permite uma arquitetura mais modelar e distribuída, promovendo a independência entre os componentes do sistema.

1.2 - MOTIVAÇÃO

As tecnologias de informação desempenham um papel crucial na transformação digital das empresas, impulsionando a inovação, a eficiência operacional e a competitividade. No âmbito das aplicações *Web*, a arquitetura de software desempenha um papel fundamental na capacidade de uma empresa se adaptar às necessidades do mercado e às mudanças tecnológicas.

Atualmente, a transição de arquiteturas monolíticas para sistemas baseados em micro serviços tem sido uma tendência significativa na indústria de produção de software. Esta mudança é motivada pela necessidade de escalabilidade, flexibilidade e agilidade no desenvolvimento e na implantação das aplicações.

Nesse contexto, a presente dissertação de mestrado procura investigar e propor uma abordagem para a reengenharia de uma plataforma web monolítica para um sistema baseado em micro serviços. A motivação para esta pesquisa reside na importância de oferecer às empresas uma metodologia estruturada e eficaz para realizar essa transição de maneira eficiente e bem-sucedida.

Ao aplicar os princípios do *Domain-Driven Design, Clean Arquitecture e Repository Pattern,* pretende-se não apenas abordar os desafios técnicos associados à migração das arquiteturas, mas também fornecer diretrizes claras e práticas para a implementação desses conceitos em contexto real. Espera-se, assim, contribuir para o avanço do conhecimento na área da arquitetura de software e para a capacidade de as empresas adotarem modelos mais modernos e eficazes de desenvolvimento de software.

Além disso, a pesquisa oferecerá *insights* valiosos sobre os benefícios e desafios específicos enfrentados durante o processo de reengenharia, permitindo uma compreensão mais profunda das melhores práticas e estratégias para o sucesso na migração para arquiteturas de micro serviços.

A motivação subjacente a esta dissertação é contribuir significativamente para a literatura académica e para a prática da engenharia de software, capacitando as empresas a modernizar as infraestruturas tecnológicas e a manterem-se competitivas num mercado em constante evolução.

1.3 - OBJECTIVO

A presente dissertação tem por objetivo apresentar todo o processo e tecnologias utilizadas para efetuar a reengenharia de uma aplicação web desenvolvida na antiga Framework 4.5 descontinuada por parte da Microsoft.

A reengenharia de sistemas monolíticos para arquiteturas de micro serviços tem sido uma prática comum para melhorar a escalabilidade, manutenção e a performance no desenvolvimento de software. O uso do DDD ajuda a identificar e definir os domínios do negócio, enquanto o *Clean Architeture* incentiva a separação das preocupações e a manutenção da independência entre as camadas. O *Repository Pattern* é utilizado para abstrair o acesso aos dados, facilitando a substituição de tecnologias de persistência e melhorando os testes do sistema. Esta dissertação descreve os passos e considerações necessárias para a reengenharia, incluindo a identificação do domínio, a definição dos limites do contexto, a arquitetura de micro serviços e a implementação dos padrões mencionados. São discutidos os benefícios esperados dessa transformação, como a capacidade de escalar horizontalmente, a redução da complexidade do código, e a facilidade da integração continua e da entrega continua (CI/CD).

1.4 - CONCEITOS BÁSICOS

O Web Old é uma aplicação monolítica, ou seja, é um tipo de aplicação em que todo o código está integrado num único monólito. Por outras palavras, todas as partes da aplicação, como a *interface* do utilizador, o processamento de negócio e o armazenamento de dados, estão agrupados numa única unidade.

Geralmente, uma aplicação Web monolítica [4] é mais fácil de desenvolver e de implementar do que outras arquiteturas, tais como as aplicações de micro serviços, pois não requer a coordenação entre vários serviços separados. No entanto, esta abordagem pode tornar a aplicação mais difícil de manter e escalar à medida que a complexidade aumenta.

Além disso, numa aplicação monolítica, cada alteração de código pode exigir que a aplicação seja testada novamente, o que pode tornar o processo de desenvolvimento e entrega mais lento e difícil. No entanto, mesmo com essas limitações, muitas empresas ainda usam arquiteturas monolíticas para o desenvolvimento de grandes aplicações Web.

Com o fim do suporte, por parte da Microsoft, da .Net *Framework* 4.5, pretende-se reconstruir o produto de raiz, tentando corrigir os pontos que se pensa serem cruciais para a sua melhoria, de entre os quais, passar a ser um SaaS, ou seja, em vez de um produto adquirido pelo cliente, ser um serviço em que o cliente paga apenas aquilo que utiliza.

O novo produto, de ora avante intitulado de Web Cloud, será desenvolvido na *Framework* .Net 8, segundo as boas práticas e os padrões recomendados, tais como o padrão *Domain Driven Design* (DDD), em conjunto com o padrão de repositórios e toda a programação orientada a objetos tenta cumprir todas as diretivas do *S.O.L.I.D* [5].

O Web Cloud será uma aplicação baseada em micro serviços [6], isto é, um estilo arquitetónico de desenvolvimento de software em que aplicação é composta por pequenos serviços independentes, conhecidos como micro serviços. Cada micro serviço é responsável por uma tarefa especifica dentro da aplicação, o que permite que os programadores trabalhem em cada serviço de forma isolada, sem afetar outros serviços.

Esta abordagem permite uma maior flexibilidade, escalabilidade e confiabilidade na construção de aplicações complexas, pois cada micro serviço pode ser desenvolvido, implantado e escalado separadamente, sem afetar os outros componentes da aplicação. Além disso, os micro serviços podem ser desenvolvidos em diferentes linguagens de programação e tecnologias, o que permite que os programadores escolham a melhor tecnologia para cada serviço.

Em resumo, uma aplicação baseada em micro serviços é composta por vários pequenos serviços independentes, que trabalham em conjunto para fornecerem uma funcionalidade completa e escalável.

No Web Cloud, que tem como objetivo ser sistema SaaS [7] e que tira partido da computação em nuvem, será inevitável que o termo *"Multitenant"* surja. Para perceber o que era o *"Multitenant"* (multi-inquilino), foram estudadas as abordagens padrão existentes no mercado. Das três mais utilizadas, optamos por uma base de dados por cada inquilino, ou seja, cada cliente terá a sua própria base de dados. As restantes duas abordagens são: uma única base de dados onde todas as tabelas têm um campo chamado *"TenantID"*, em que cada um pertence a um cliente diferente que identifica a quem pertence o registo e, por último, a outra abordagem, onde existe uma base de dados com um *"schema"* para cada cliente.

Como gestor de base de dados foi escolhido o SQL Server. Para comunicação entre o repositório e a base de dados foi feito um estudo de todos os ORM (*Object-Relational Mapping*) e abordagens padrão utilizadas e optou-se pela utilização do ORM *EntityFrameworkCore* [8]da Microsoft em conjunto com o Dapper [9].

Para tentar seguir com o maior rigor possível o padrão DDD, os vários módulos da aplicação serão divididos, ficando os seguintes módulos:

- **Domínio**: É responsável por modelar e implementar as entidades, objetos de valor, agregados e regras de negócio que são específicos do domínio do problema que a aplicação está a resolver.
- ***Data***: Responsável por conter o mapeamento entre cada modelo do domínio e respetivos atributos de cada tabela da base de dados, contendo também o contexto que faz a ligação com o SQL Server, depende do módulo de domínio.
- **DTOs**: Responsável por conter os objetos transformados dos dados vindo da base de dados e que irão ser devolvidos pela API, este modulo é responsável pelos dados que são transportados entre a API e o cliente, não depende de nenhum modulo.
- ***CrossCutting***: Responsável por conter todos os métodos e atributos que são transversais a todos os módulos, este módulo não depende dos outros.
- ***Gateway***: Responsável por conter toda a interação e toda a lógica aplicacional, fazer a ponte entre todos os micro serviços.
- ***FrontEnd***: Responsável por conter todos os ecrãs de interação com o utilizador, está desenvolvido em HTML, CSS, JavaScript, todos os pedidos à API são feitos através de *Restfull*, sendo que este módulo depende do *CrossCutting*.

1.5 – ORGANIZAÇÃO DA DISSERTAÇÃO

A presente dissertação encontra-se estruturada em dezassete capítulos.

No Capítulo 1, é feita a introdução ao tema abordado nesta dissertação, indicando qual o enquadramento, a motivação, os objetivos e os conceitos básicos do trabalho desenvolvido.

No Capítulo 2, é abordada a atual aplicação monolítica, identificando qual a Framework que utiliza e qual a estrutura existente no projeto.

No Capítulo 3, é feita a comparação entre uma arquitetura monolítica e uma arquitetura micro serviços, onde é explicada cada uma delas, assim como a sua escalabilidade. Na arquitetura de micro serviços é abordada a sua forma de implementação.

No Capítulo 4, é apresentada a Framework .Net 8 e o porquê de ser uma das recomendadas para o desenvolvimento de software seguindo a arquitetura de micro serviços.

No Capítulo 5, são apresentados os padrões, os modelos pesquisados, estudados e recomendados para o desenvolvimento de aplicações web segundo a arquitetura de micro serviços.

No Capítulo 6, são abordados os dois principais tipos de comunicação entre micro serviços e feita a sua comparação. Neste capítulo são também apresentados tipos de comunicação e livrarias mais utilizados.

No Capítulo 7, são apresentados os dois principais tipos de cache, para otimizar a performance dos micro serviços.

No Capítulo 8, são abordadas as principais livrarias e ferramentas utilizadas no desenvolvimento de aplicações web segundo a arquitetura de micro serviços.

No Capítulo 9, são apresentadas as principais metodologias de gestão de projeto utilizadas e identificadas quais são utilizadas no desenvolvimento da nova aplicação web.

No Capítulo 10, é apresentada a ferramenta de gestão de projeto da Microsoft, o *Devops*. São abordadas as suas principais características.

No Capítulo 11, é apresentada a viabilidade da reengenharia da antiga aplicação para a nova aplicação, passando de uma arquitetura monolítica para uma arquitetura de micro serviços.

No Capítulo 12, é apresentada a proposta de como desenvolver todos os micro serviços e *gateway* para a nova aplicação.

No Capítulo 13, são apresentadas as linguagens de programação para o *BackEnd* e *FrontEnd*, assim como linguagens de marcação e de estilos.

No Capítulo 14, são apresentadas as duas principais *Frameworks* de *FrontEnd* escolhidas para o desenvolvimento das *interfaces*.

No Capítulo 15, são apresentadas as ferramentas que serão utilizadas para o desenvolvimento da aplicação.

No Capítulo 16, é apresentada a organização e configuração básica de um projeto para desenvolvimento da aplicação web segundo a arquitetura de micro serviços.

Neste capítulo demonstra-se que tipo de projeto deve ser escolhido e configurado para cada camada, tentando respeitar sempre o *DDD* e o *Clean Architeture*.

No Capítulo 17, apresentam-se as conclusões, tentando indicar quando se deve utilizar uma arquitetura monolítica e uma arquitetura de micro serviços. São também referidos os benefícios económicos e de infraestruturas para as empresas aquando da utilização de uma arquitetura de micro serviços.

Finalmente, nos anexos são apresentadas algumas capturas de ecrã da nova aplicação *Web* desenvolvida.

Capítulo 2 - APLICAÇÃO WEB MONOLÍTICA EXISTENTE

A aplicação Web Old é uma aplicação MVC (*Model-View-Controller*) com uma estrutura monolítica baseada em Framework 4.5 e com 14 anos de existência. Por conter muito código repetido, enfrenta vários problemas e desafios significativos, nomeadamente complexidade crescente: com o passar do tempo, a complexidade do código aumentou consideravelmente. Isso tornou a aplicação difícil de entender e manter, já que algumas partes do código estão interligadas de uma maneira muito complexa e repetitiva.

- **Manutenção difícil:** À medida que o código se tornou mais complexo, a manutenção tornou-se mais difícil e demorada. Corrigir erros, implementar novos recursos e fazer atualizações passou a ser um processo arriscado e demorado, pois as alterações numa parte do código fazem com que outra parte deixe de funcionar.
- **Duplicação Código:** A presença de muito código repetido é um grande problema: Isso não apenas aumenta o tamanho da aplicação, mas também dificulta a manutenção, pois qualquer mudança precisa ser replicada em várias partes do código.
- **Rigidez na Arquitetura:** Com uma estrutura monolítica, a aplicação tornou-se rígida em termos de arquitetura. Isso significa que é difícil adotar novas tecnologias, integrar serviços externos e adicionar recursos modernos.
- **Desempenho limitado:** *Frameworks* mais antigas e descontinuadas, como o .NET Framework 4.5.1, não são tão eficientes em termos de desempenho como versões mais recentes. Isto acarreta problemas de desempenho à medida que a carga de trabalho da aplicação aumenta.
- **Escalabilidade Comprometida:** A capacidade de escalar horizontalmente (adicionar mais servidores para lidar com o aumento de tráfego) está comprometido numa aplicação monolítica.
- **Dificuldades na integração:** Integrar novas bibliotecas, serviços ou tecnologias também passou a ser um desafio, uma vez que a estrutura existente muitas vezes não é compatível.
- **Baixa Produtividade:** O código repetido e a complexidade geral afetam a produtividade da equipa de desenvolvimento, pois as tarefas tornam-se demoradas.
- **Falta de testes adequados:** Com o tempo, a aplicação não tem testes automatizados adequados, tornando difícil garantir que as alterações não danifiquem funcionalidades existentes.
- **Segurança Vulnerável:** A segurança é uma preocupação, especialmente porque já não existe atualizações para bibliotecas e componentes de terceiros.

Para resolver estes problemas e desafios, muitas organizações consideram modernizar as aplicações. Isto pode envolver a reengenharia do código para reduzir a duplicação, a migração para uma versão mais recente do .NET ou para uma arquitetura moderna, como micro serviços, e a implementação de melhores práticas de desenvolvimento, como DevOps e CI/CD. A modernização é um projeto desafiador que pode trazer benefícios a longo prazo em termos de facilidade de manutenção, melhor desempenho e escalabilidade.

2.1 - FRAMEWORKS UTILIZADAS NO WEB OLD

O Web Old está desenvolvido em .Net Framework 4.5.1, é uma versão madura e amplamente utilizada da Framework .Net da Microsoft. A última versão foi a 4.8, lançada em 2019 e é conhecida pela sua ampla compatibilidade com aplicações desenvolvidas com versões mais antigas do .NET. Isso torna-a numa escolha adequada para manter e dar suporte a aplicações existentes. No entanto, o .Net Framework 4.8 é mais adequado para o desenvolvimento de aplicações Windows tradicionais, como aplicações *desktop Windows Forms* e *Windows Presentation Foundation* (WPF), bem como serviços baseados em Windows *Communication Foundation* (WCF). Não é modelar, o que significa que é necessário instalar todo o pacote da Framework, mesmo que se necessite apenas de uma parte. Além disso, não é facilmente portável para outras plataformas, o que limita a sua utilização para o desenvolvimento de aplicações multiplataforma.

Também tira partido do .NET Standard 2.1, que é uma especificação que visa fornecer um conjunto padrão de APIs comuns a todas as implementações de .NET compatíveis com essa versão. Uma das suas principais vantagens é a modularidade, o que permite incluir apenas as bibliotecas necessárias para um projeto, economizando espaço e recursos. O .Net *Standard* 2.1 também introduz melhorias de desempenho em relação às versões mais antigas do .Net standard, tornando-o uma escolha atraente para novos projetos. Além disso, oferece portabilidade avançada, permitindo o desenvolvimento de bibliotecas e componentes que podem ser usados em várias plataformas, incluindo o .Net Core 3.0 ou posterior e o .Net 5 ou posterior. Com a evolução da .Net 5 para 6, 7 e 8 (atual) a .net Standard 2.1 não teve mais evolução ficando estagnada, o que não permite mais evolução da plataforma.

2.2 - ESTRUTURA DA APLICAÇÃO WEB OLD

O Web Old é um software monólito, ou seja, é uma aplicação Web que possui uma estrutura única e centralizada. Isto significa que todas as funcionalidades e componentes do sistema estão integrados num único código-fonte e são executados

num único processo. Um software monólito é uma estrutura indivisível e coesa, onde todas as partes estão interligadas e funcionam com um todo.

Todo o processamento é feito do lado do servidor, em que o utilizador depende da velocidade da rede e do servidor para fazer o processamento. Um exemplo desse processamento é quando numa lista descendente de valores, a pesquisa de um valor é feita chamando rotinas no servidor, quando poderia ser feita do lado do cliente com JavaScript.

Na Figura 1 poderemos começar por visualizar a organização das pastas, num único projeto.

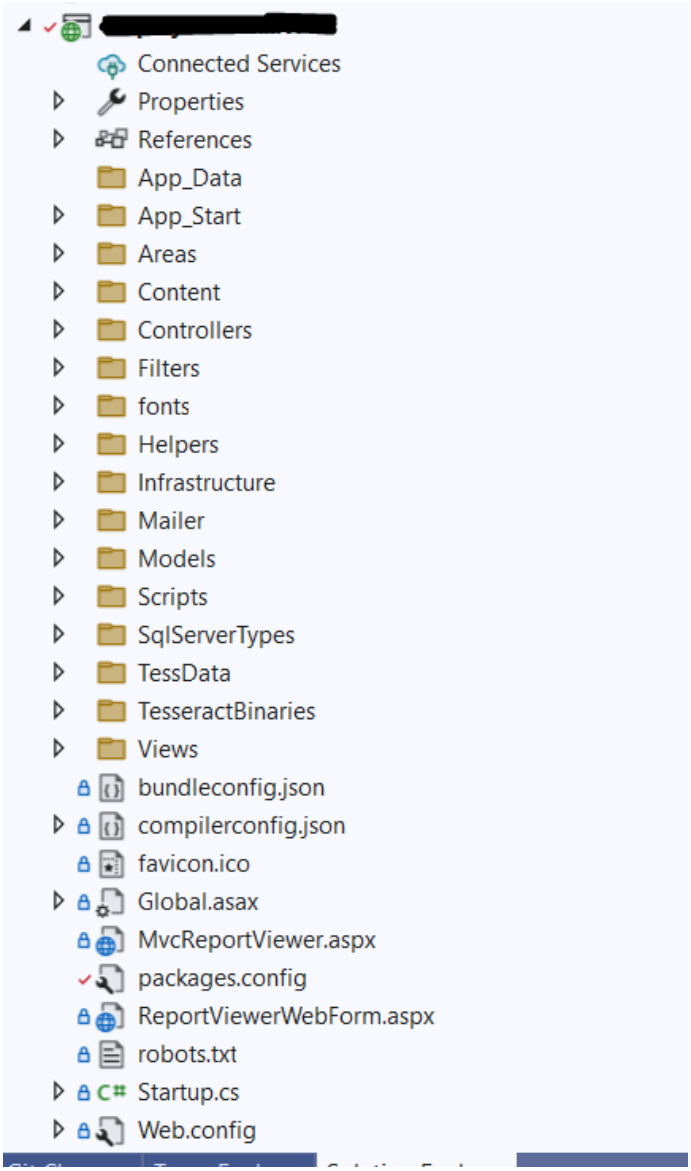


Figura 1 – *Solution Explorer* do Web Old

Na Figura 2 podemos visualizar a grande quantidade de *controller* desdobrados em *sub controllers* em vez de recorrer à herança e aos desdobramentos por módulos.

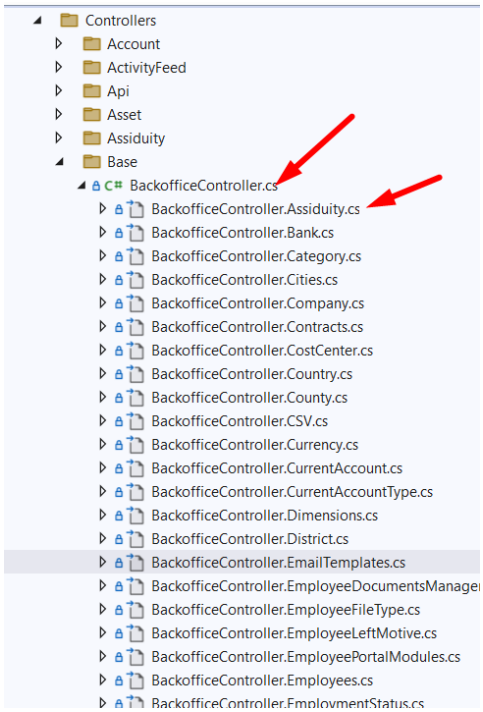


Figura 2 – Apresentação da estrutura de *controller* do Web Old

Existência de muitos *controllers* com mais de 3 mil linhas de código, tal como apresentado na Figura 3, o que torna muito difícil a manutenção e dependência de código.

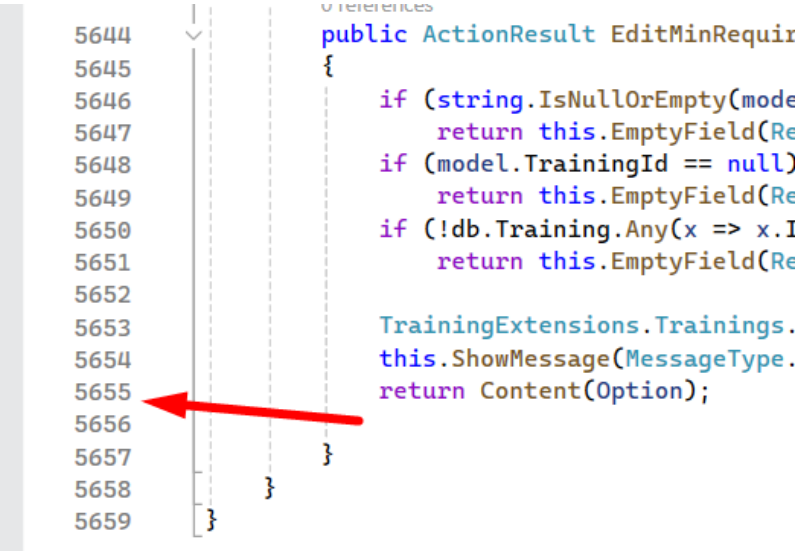


Figura 3 – Total de linhas de alguns *controllers*

Na Figura 4 podemos visualizar a organização das janelas de *interface* com o utilizador, existe o problema de as mesmas estarem misturadas com os ficheiros de JavaScript e o próprio JavaScript estar dividido entre ficheiro JS e no próprio html.

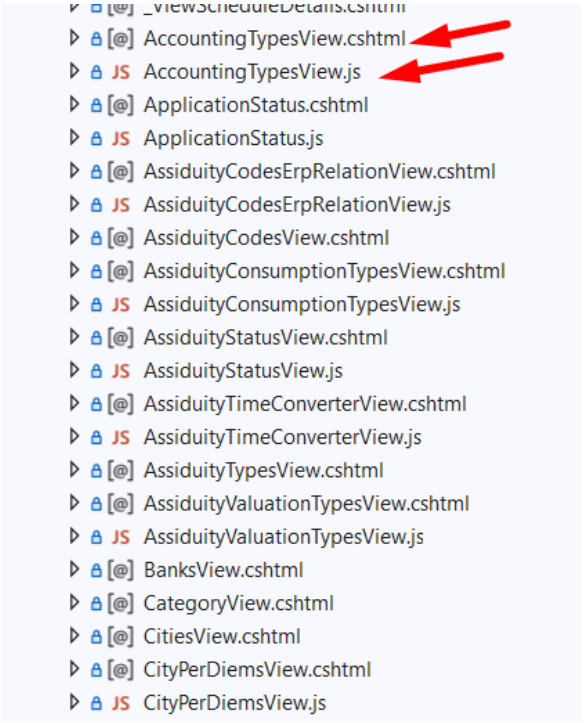


Figura 4 - Organização dos ficheiros de *layout* com JavaScript

Um dos erros frequentes é a dupla definição das variáveis do JavaScript no lado do ficheiro JS e do lado do ficheiro cshtml, como podemos visualizar na Figura 5, o que tem como consequência a incoerência de dados.

```
106  @section Scripts {
107      @Scripts.Render("~/bundles/DataTablesJS")
108      @Scripts.Render("~/bundles/AutosizeJS")
109      @Scripts.Render("~/bundles/TableToolsJS")
110
111      <script>
112          var _rejectToolTip = "@Html.Raw(Resource_General.Reject)";
113          var _approveToolTip = "@Html.Raw(Resource_General.Approve)";
114          var _detailToolTip = "@Html.Raw(Resource_General.Details)";
115          var _noLinesSelected = "@Html.Raw(Resource_Vacations.NoDaysSelected)";
116
117          var _approved = "@_BalanceTransferStatus.Approved";
118          var _approvedErp = "@_BalanceTransferStatus.ApprovedErp";
119          var _notProcessed = "@_BalanceTransferStatus.NotProcessed";
120          var _pending = "@_BalanceTransferStatus.Pending";
121          var _pendingErp = "@_BalanceTransferStatus.PendingErp";
122          var _processed = "@_BalanceTransferStatus.Processed";
123          var _rejected = "@_BalanceTransferStatus.Rejected";
124
125          var _CurrentYear = "@DateTime.Now.Year";
126      </script>
```

Figura 5 - Declaração de variáveis do JavaScript no cshtml

Na Figura 6 demonstra o recarregamento em todas as janelas das mesmas dependências de JavaScript constantemente.

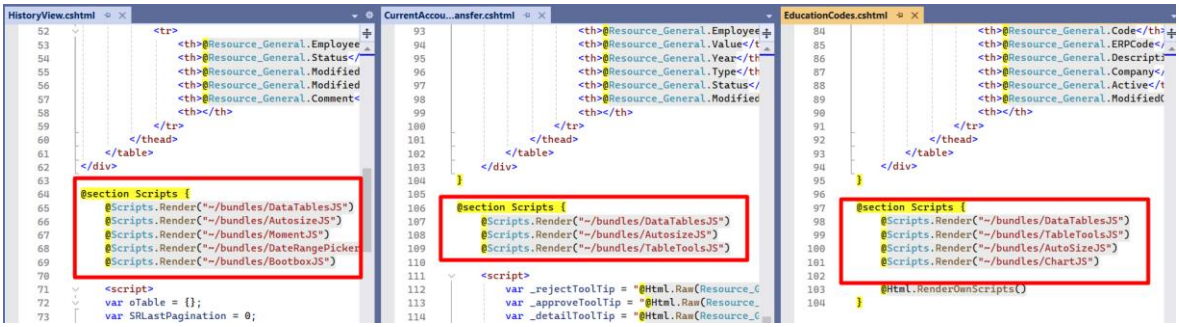


Figura 6 – Redundância do carregamento de ficheiros JavaScript

Utilização incorreta de extensões de classes, tal como visualizado na Figura 7, sendo apenas classes estáticas para acesso direto sem instânciar e alocar memória.

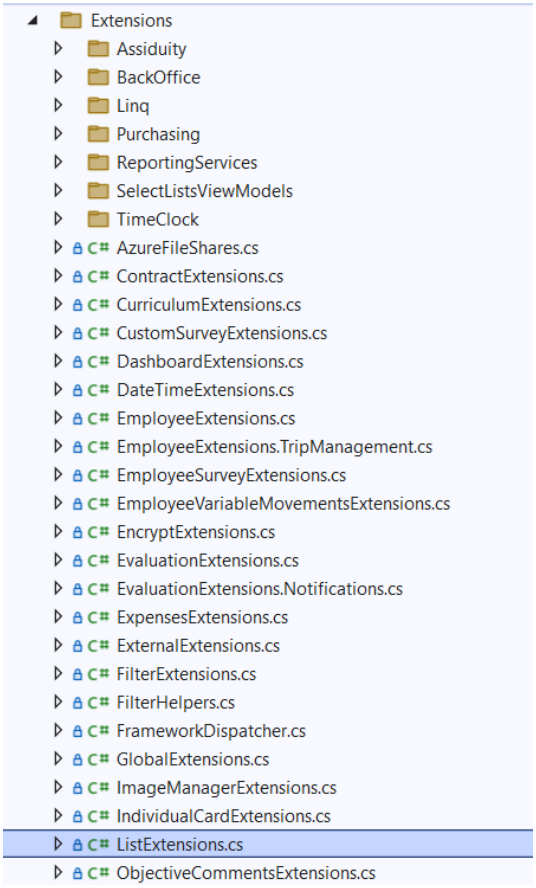


Figura 7 – Declaração excessiva de extensões

Estes ficheiros de extensão são muito complexos e de quase impossível manutenção pois alguns chegam a ter mais de 10 mil linhas de código, como podemos visualizar na

Figura 8. Isto deve-se a estes conterem muitos métodos idênticos, com retorno do mesmo resultado e com o mesmo algoritmo, apenas diferindo os parâmetros de entrada. Contêm ainda múltiplas classes no mesmo ficheiro, não sendo fácil identificar a classe a partir do nome do ficheiro.

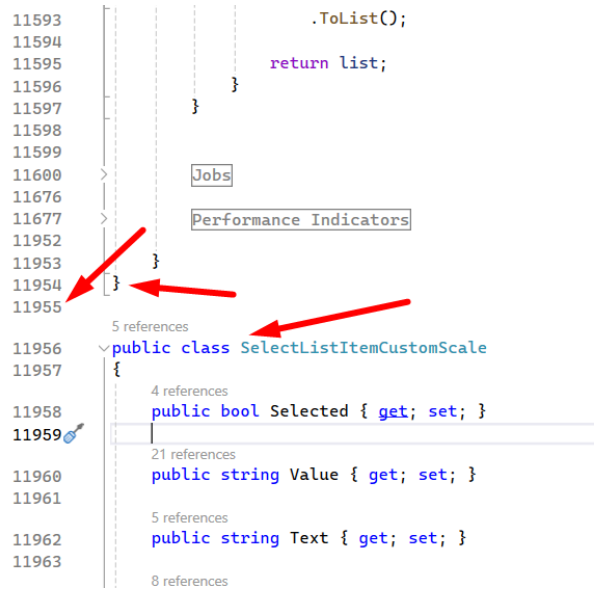


Figura 8 – Tamanho excessivo dos ficheiros de extensão

Estes são alguns dos exemplos de uma organização incorreta do projeto, consequência do crescimento do mesmo sem acompanhar as boas práticas de desenvolvimento de software. Podemos afirmar que o projeto contém muito "*Code Smell*".

"*Code smell*" [10] é uma expressão utilizada na programação de software para descrever padrões de código-fonte que podem indicar a presença de problemas ou áreas que precisam de melhorias. Estes "cheiros" não são erros específicos, mas pistas que sugerem a possibilidade de existir um problema subjacente no design ou na implementação do código.

"*Code smells*" podem ainda ser indicativos de más práticas de programação, complexidade desnecessária, falta de clareza no código, entre outros problemas. Identificar e corrigir "*Code smells*" pode melhorar a qualidade do código, tornando-o mais legível, sustentável e menos propenso a erros.

Existem diversos tipos de "*Code smells*", como duplicação de código, métodos muito longos, classes muito grandes, dependência excessiva entre componentes, entre outros. A deteção precoce e a correção de "*code smells*" são práticas importantes para o desenvolvimento de software de alta qualidade.

Capítulo 3 - ARQUITETURA MONOLÍTICA VERSUS ARQUITETURA MICRO SERVIÇOS

Neste capítulo será efetuada a comparação entre a arquitetura monolítica e a arquitetura micro serviços, Figura 9. Ao longo das várias secções são apresentadas as principais características deste tipo de arquiteturas.

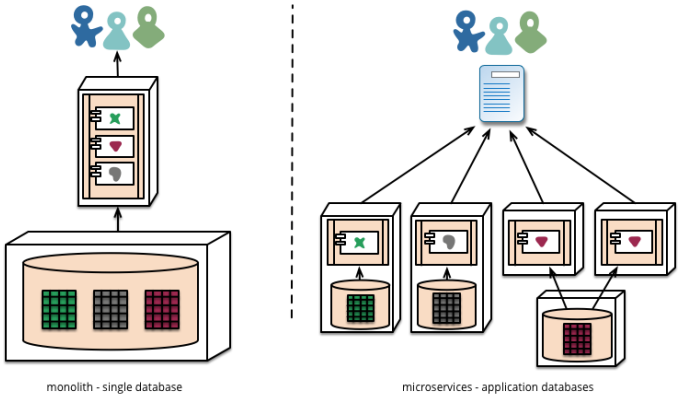


Figura 9 - Sistema monolítico vs Sistema micro serviços[1]

3.1 - ARQUITETURA MONOLÍTICA

A arquitetura monolítica é um modelo tradicional de desenvolvimento de software, Figura 10, onde todo o código da aplicação é agrupado num único sistema. Isto significa que todos os componentes da aplicação, assim como a *interface* do utilizador, a lógica de negócio e o acesso a dados, são agrupados num único pacote de software. Esta arquitetura é comum em aplicações de pequeno e médio porte.

[1] Retirado de: https://medium.com/trainingcenter/microservi%C3%A7os-dos-grandes-mon%C3%B3litos-%C3%A0s-pequenas-rotas-adb70303b6a3

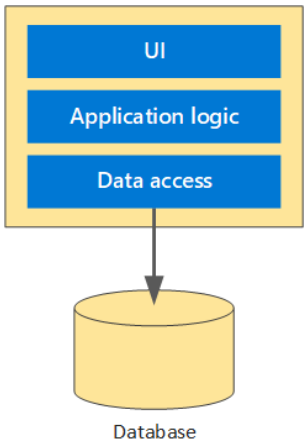


Figura 10 – Estrutura sistema monolítico[2]

Algumas das principais características da arquitetura monolítica incluem:

- **Ligação forte**: Todos os componentes estão interligados e dependem uns dos outros. Isso significa que qualquer alteração num componente/classe pode afetar outros componentes, o que torna o sistema mais difícil de manter e modificar.
- **Escalabilidade vertical**: É projetada para ser escalada verticalmente, o que significa que a capacidade do sistema é aumentada adicionando mais recursos, como CPU, RAM ou disco rígido, num único servidor. No entanto, isso tem limites e pode levar a um aumento significativo nos custos da infraestrutura.
- **Implantação única**: É geralmente implantada num único servidor ou instância. Isso significa que todo o sistema é implementado e executado num único lugar, o que pode levar a problemas de desempenho e disponibilidade.
- **Monólito único**: O sistema é mantido como um monólito único, o que pode tornar o processo de desenvolvimento mais fácil de gerir e implantar. No entanto, isso também pode tornar o sistema mais complexo e difícil de manter à medida que o tamanho e a complexidade do sistema aumentam.
- **Ciclo de vida do software mais longo**: A arquitetura monolítica é geralmente associada a ciclos de vida de software mais longos, já que as correções exigem atualizações e implantações do sistema inteiro.
- **Maior tempo de inicialização:** O tempo de inicialização do sistema é geralmente mais longo, já que todo o sistema deve ser iniciado antes que o utilizador possa começar a utilizá-lo.

A arquitetura monolítica pode ser vantajosa em situações em que:

[2] Retirado de: https://learn.microsoft.com/pt-pt/azure/architecture/microservices/migrate-monolith

- O objetivo do projeto é pequeno e não deve crescer significativamente no futuro.
- Não é necessário separar diferentes áreas funcionais em módulos distintos.
- O sistema é relativamente simples e não requer muitas interações complexas entre os componentes.
- Os recursos da infraestrutura são limitados, o que torna difícil implementar uma arquitetura distribuída ou em nuvem.

Vantagens:

- **Simplicidade:** É relativamente mais simples de desenvolver, testar e implantar, pois todos os componentes estão contidos numa única unidade.
- **Desempenho:** Como não há chamadas de rede entre diferentes componentes, a comunicação interna é mais rápida, resultando num melhor desempenho em comparação com a arquitetura distribuída.
- **Facilidade de escalabilidade vertical:** É mais fácil dimensionar uma aplicação monolítica verticalmente (adicionando mais recursos, como CPU e memória), já que não há a necessidade de gerir vários serviços independentes.

Desvantagens:

- **Dependência:** Os componentes de uma arquitetura monolítica estão fortemente dependentes, o que significa que qualquer mudança ou atualização de um componente pode afetar outros componentes. Isso pode dificultar a manutenção e a evolução do sistema.
- **Escalabilidade Limitada:** A escalabilidade horizontal (adicionando mais instâncias de um serviço) numa arquitetura monolítica pode ser complicada pois o sistema é tratado como um todo. Isso pode limitar a capacidade de lidar com os picos de carga ou distribuir a carga de trabalho eficientemente.

É importante realçar que a escolha da arquitetura deve sempre ser baseada numa análise cuidadosa dos requisitos do projeto e das necessidades do negócio. A arquitetura monolítica pode ser adequada para algumas situações, mas noutras pode ser mais apropriado utilizar uma arquitetura distribuída, baseada em micro serviços ou em nuvem. Assim, é importante avaliar cuidadosamente as opções disponíveis antes de decidir qual arquitetura utilizar.

3.1.1 - ESCALABILIDADE VERTICAL: LIMITAÇÃO NUMA ARQUITETURA MONOLÍTICA

A escalabilidade vertical, Figura 11, refere-se à capacidade de aumentar a capacidade do sistema, adicionando mais recursos (como CPU (*Central Processing Unit*), memória, armazenamento) a uma única instância. Nesse caso, o objetivo é tornar uma única máquina mais poderosa para lidar com uma carga de trabalho maior.

Como exemplo, vamos considerar um servidor de base de dados que está a ter um aumento no volume de transações devido ao crescimento de uma empresa. Para lidar com o aumento da carga de trabalho, a escalabilidade vertical pode ser implementada aumentando os recursos do servidor.

Por exemplo, se o servidor da base de dados possui inicialmente oito *gigabytes* de memória RAM (*Random Access Memory*) e está a enfrentar um problema de desempenho, é possível escalá-lo verticalmente, aumentando a memória para 16GB. Com mais recursos, o servidor pode armazenar mais dados em memoria, reduzir a latência de acesso e lidar com um maior número de transações simultâneas.

Figura 11 – Escalabilidade vertical[3]

3.2 - ARQUITETURA MICRO SERVIÇOS

A arquitetura de micro serviços é uma abordagem de desenvolvimento de software, Figura 12, que visa contruir um sistema como um conjunto de serviços independentes, cada um executa dentro do seu próprio processo e comunica com outros serviços por meio de APIs bem definidas.

[3] Retirado de: https://desorientado.dev/2018/04/09/modelos-armazenamento-hibrido/figura-1-2-escalabilidade-vertical-x-escalabilidade-horizontal-adaptado-de-marquesone-2017

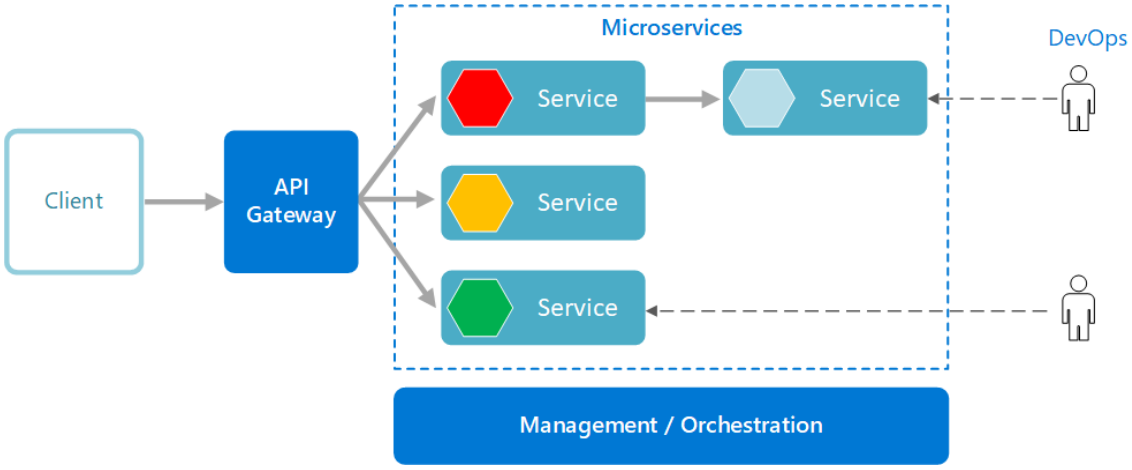


Figura 12 - Estrutura sistema de micro serviços[4]

Algumas das principais características da arquitetura de micro serviços incluem:

- **Decomposição do sistema em serviços independentes:** Em vez de ter um único sistema monolítico, a arquitetura de micro serviços divide o sistema em serviços independentes, cada um responsável por uma única funcionalidade. Isso torna o sistema mais fácil de entender e manter.
- **Escalabilidade independente:** Como cada serviço é executado no seu processo, é possível escalá-los independentemente uns dos outros. Isso permite lidar com picos de carga em áreas especificas do sistema sem ser preciso escalar o sistema todo.
- **Resiliência:** É projetada para ser resiliente a falhas. Se um serviço falhar, isso não afeta todo o sistema. Além disso, é possível implementar mecanismos de recuperação de falhas em cada serviço individual.
- **Flexibilidade:** Permite que novos serviços sejam adicionados ou removidos do sistema sem afetar os restantes serviços existentes. Isso torna o sistema mais flexível e adaptável às mudanças nos requisitos do negócio.
- **APIs bem definidas:** Os serviços comunicam-se por meio de APIs bem estruturadas e especificadas. Isso permite que os serviços sejam desenvolvidos de forma independente uns dos outros e sejam substituídos por outros serviços sem afetar o sistema como um todo.
- **Poliglota:** Como cada serviço é desenvolvido e implantado de forma independente e autónoma, é possível usar diferentes tecnologias para cada serviço, em vez de ter de usar a mesma tecnologia para todo o sistema.

As principais características da arquitetura de micro serviços incluem:

- A decomposição do sistema em serviços independentes.

[4] Retirado de: https://learn.microsoft.com/pt-pt/azure/architecture/guide/architecture-styles/microservices

- A escalabilidade independente.
- A resiliência.
- A flexibilidade.
- APIs bem definidas.
- A possibilidade de utilizar diferentes tecnologias e linguagens para cada serviço.

Vantagens:

- **Independência e escalabilidade:** Os serviços são independentes uns dos outros, o que permite maior flexibilidade e escalabilidade horizontal. É possível escalar apenas os serviços necessários em resposta ao solicitado.
- **Facilidade de evolução:** Com componentes separados, é mais fácil evoluir e atualizar partes especificas do sistema sem afetar todo o sistema. Cada serviço pode ser desenvolvido, testado e implantado de forma autónoma.
- **Resiliência e isolamento:** Se um serviço falhar, outros serviços podem continuar a funcionar normalmente. Além disso, o isolamento entre serviços ajuda a evitar o efeito em cascata de falha.

Desvantagens:

- **Complexidade de gestão:** com vários serviços independentes, há uma maior complexidade na implantação, gestão e monitorização do sistema como um todo. É necessário um planeamento cuidadoso e ferramentas adequadas para lidar com essa complexidade.
- ***Overhead* de comunicação:** devido à natureza distribuída da arquitetura de micro serviços, a comunicação entre serviços geralmente ocorre por meio de chamadas de rede. Isso introduz um *overhead* de comunicação em comparação com a comunicação interna em uma arquitetura monolítica.

Em resumo, a arquitetura de micro serviços é uma abordagem de desenvolvimento de software que visa construir um sistema como um conjunto de serviços independentes, cada um é executado no seu próprio processo e comunica-se com os outros serviços por meio de APIs bem definidas.

3.2.1 – ESCALABILIDADE HORIZONTAL: POTENCIALIDADES

A escalabilidade horizontal, Figura 13, refere-se à capacidade de aumentar o potencial de um sistema, adicionando-lhe mais instâncias. Isso implica distribuir a carga de trabalho entre várias máquinas ou servidores. Em vez de aumentar os recursos de uma única instância, a escalabilidade horizontal procura lidar com a grande quantidade de pedidos, distribuindo-os entre várias instâncias físicas ou virtuais.

Como exemplo, vamos considerar uma aplicação web de comércio eletrónico que recebe um aumento significativo de tráfego devido a uma promoção de vendas. Para lidar com o aumento de utilizadores simultâneos, a escalabilidade horizontal pode ser implementada adicionando mais servidores ao *cluster* de aplicações.

Digamos que inicialmente o sistema possui três servidores de aplicações, cada um é responsável por uma parte da carga do trabalho. Com a escalabilidade horizontal, podemos adicionar mais servidores ao *cluster*, com mais dois servidores de aplicações. Agora, a carga de trabalho será distribuída entre cinco servidores, permitindo que o sistema lide com um maior número de utilizadores em simultâneo.

Figura 13 – Escalabilidade horizontal[5]

3.2.2 - IMPLEMENTAÇÃO DE UMA ARQUITETURA DE MICRO SERVIÇOS

A arquitetura de micro serviços é um estilo de desenvolvimento de software que ganhou popularidade nos últimos anos devido à sua flexibilidade e escalabilidade. Neste modelo, uma aplicação é dividida em vários projetos. Cada projeto corresponde a um serviço, independente e autónomo, conhecido como micro serviço, que comunicam entre si por meio de chamadas APIs, normalmente por pedidos *REST (Serviços HTTP que fornecem recursos, geralmente JSON)* [11]. Esta abordagem permite que cada micro serviço seja desenvolvido, implantado e dimensionado de forma independente, facilitando a manutenção e evolução do sistema como um todo.

- **Planeamento e definição dos micro serviços:** O primeiro passo no desenvolvimento de uma aplicação baseada em micro serviço é realizar um

[5] Retirado de: https://desorientado.dev/2018/04/09/modelos-armazenamento-hibrido/figura-1-2-escalabilidade-vertical-x-escalabilidade-horizontal-adaptado-de-marquesone-2017

planeamento cuidadoso e definir os micro serviços que constituem o sistema. Essa definição deve ser baseada nas funcionalidades principais da aplicação e na identificação de componentes que possam ser separados e tratados como serviços independentes. Ao definir os limites dos micro serviços, é importante considerar critérios como coesão, dependência e responsabilidade única. Por exemplo, numa aplicação de comércio eletrónico, podemos ter micro serviços para gerir autenticação, carrinho de compras, processamento de pagamentos, catálogo de produtos, entre outros.

- **Separação de responsabilidades:** Cada micro serviço deve ser responsável por uma única função especifica dentro do sistema. Essa abordagem é conhecida com **P**rincípio da **R**esponsabilidade **Ú**nica e é fundamental para o sucesso da arquitetura de micro serviços. Ao separar as responsabilidades em serviços independentes, facilitamos o desenvolvimento, o teste, a implantação e a manutenção de cada micro serviço. Além disso, essa separação permite que cada serviço evolua de forma independente, sem conflituar com os restantes. Por exemplo, o micro serviço de autenticação pode ser responsável apenas pela autenticação de utilizadores, enquanto o micro serviço de catálogo de produtos será responsável por gerir as informações sobre os produtos.
- **Comunicação entre micro serviços:** A comunicação entre micro serviços é um aspeto critico na arquitetura. Geralmente, essa comunicação é feita por meio de APIs, utilizando protocolos como *REST* ou troca de mensagens assíncrona. Cada micro serviço expõe uma API para que outros micro serviços possam consumir os seus serviços. É importante estabelecer uma estratégia adequada para lidar com a comunicação e garantir a integridade e a consistência dos dados. Existem diferentes abordagens para o intercâmbio de dados entre os micro serviços, como chamadas síncronas de API, eventos assíncronos e integração via troca de mensagens. É fundamental escolher a abordagem mais adequada com base nas características e requisitos específicos de cada aplicação.
- **Implantação e escalabilidade:** A implantação e a escalabilidade são aspetos fundamentais no desenvolvimento de aplicações baseadas em micro serviços. Cada micro serviço pode ser implantado num ambiente separado, como contentores Docker ou máquinas virtuais, o que permite a fácil escalabilidade horizontal. A escalabilidade horizontal significa adicionar mais instâncias de um micro serviço para lidar com o aumento de carga, em vez de escalar verticalmente adicionando recursos a uma única instância. Essa abordagem oferece maior flexibilidade e eficiência em termos de uso de recursos. É importante também considerar a utilização de ferramentas de orquestração de contentores, como *Kubernetes*, para a gestão da implantação, a escalabilidade e a monitorização dos micro serviços.

Exemplo Prático:

Para ilustrar o desenvolvimento de uma aplicação baseada em micro serviços, considere-se um exemplo prático de um sistema de comercio eletrónico. Neste caso, podemos identificar os seguintes micro serviços:

- **Micro serviço de Autenticação:** Responsável pela autenticação e autorização dos utilizadores.
- **Micro serviço de gestão de pedidos:** Responsável pela gestão dos pedidos de compra.
- **Micro serviço de catálogo de produtos:** Responsável pela gestão das informações sobre os produtos disponíveis.
- **Micro serviço de pagamento:** Responsável pelo processamento dos pagamentos das compras.
- **Micro serviços de entrega:** Responsável pela gestão e rastreamento das entregas.

Cada um destes micro serviços é desenvolvido, testado e implantado de forma independente. A comunicação entre eles ocorre por meio de APIs, por exemplo, o micro serviço de gestão de pedidos poderia chamar a API do micro serviço de pagamentos para processar o pagamento de um pedido.

Assim, conclui-se que a arquitetura de micro serviços oferece uma abordagem flexível e escalável para o desenvolvimento de aplicações. Ao dividir uma aplicação em vários serviços independentes, é possível facilitar a manutenção, a evolução e a escalabilidade do sistema como um todo. Através da definição clara de responsabilidades e da comunicação adequada entre os micro serviços, é possível criar sistemas mais robustos, eficientes e que atendam aos requisitos de negócio de forma mais precisa. Com exemplos práticos, como o sistema de comércio eletrónico apresentado, é possível visualizar como os micro serviços podem ser aplicados em cenários reais.

Capítulo 4 - FRAMEWORK .NET

A Framework .NET é uma plataforma de desenvolvimento amplamente utilizada para criar uma variedade de aplicações, desde aplicações desktop a aplicações web e até mesmo aplicativos móveis.

É uma plataforma de desenvolvimento criada pela Microsoft que oferece um ambiente de execução comum para diferentes linguagens de programação, como o C#, Visual Basic .NET e F#. Fornece uma biblioteca de classes abrangente, juntamente com um tempo de execução (*Common Language Runtime -CLR*) que permite a compilação e execução de código em várias linguagens [12].

- Características:
  - **Multiplataforma:** Suporta o desenvolvimento de aplicações para Windows, MacOS e Linux. Oferece uma maior flexibilidade para os programadores ao escolher a plataforma de implantação.
  - **Linguagens de programação:** Suporta várias linguagens de programação, como C#, Visual Basic .NET, F# e outras linguagens de terceiros. Permite que os programadores escolham a linguagem com a qual estão mais familiarizados ou que melhor se adapte às necessidades do projeto.
  - **Biblioteca de classes:** Possui uma ampla biblioteca de classes (*Class Library*) que oferece uma série de funcionalidades prontas para utilização, como manipulação de ficheiros, acesso a base de dados, criptografia, comunicação de rede e muito mais. Isso acelera o desenvolvimento, uma vez que muitas funcionalidades comuns já estão disponíveis.
  - **Ambiente de execução (CLR):** O CLR é a parte central da Framework .NET e fornece recursos como gestão de memória, *Garbage Collector*, segurança, gestão de exceções e suporte a várias linguagens de programação.
- Vantagens:
  - **Produtividade:** Oferece uma ampla gama de ferramentas, recursos e bibliotecas que aumentam a produtividade dos programadores. A biblioteca de classes abrangente reduz a necessidade de escrever código a partir do zero, enquanto as ferramentas de desenvolvimento, como o Visual Studio, oferecem suporte para depuração, testes e implantação simplificada.
  - **Ecossistema robusto:** Possui uma comunidade ativa e um ecossistema rico, com uma ampla gama de recursos disponíveis, como bibliotecas de terceiros, pacotes *NuGet*, fóruns de discussão e documentação abrangente, facilitando a colaboração, a resolução de problemas e o acesso a soluções prontas.

    - **Segurança:** Possui recursos de segurança integrados, como o sistema de permissões do CLR, mecanismos de criptografia e recursos de controle de acesso. Isso ajuda a proteger a aplicação contra vulnerabilidades e ataques maliciosos.
    - **Desenvolvimento escalável:** Foi projetada para suportar o desenvolvimento de aplicações escaláveis. Oferece suporte a arquiteturas robustas, como a arquitetura em camadas, que permitem a separação de responsabilidades e a escalabilidade horizontal e vertical conforme necessário.
- Desvantagens:
    - **Curva de aprendizagem inicial:** Possui uma curva de aprendizagem inicial especialmente para programadores juniores ou que estão a migrar de outras plataformas. Aprender a utilizar efetivamente todas as ferramentas e recursos disponíveis pode exigir tempo e esforço.
    - **Restrições da plataforma:** Embora seja multiplataforma, nem todos os recursos estão disponíveis em todas as plataformas. Alguns recursos específicos do *Windows* podem não ser suportadas noutras plataformas, limitando a portabilidade da aplicação.
- Exemplos práticos de utilização:
    - Desenvolvimento de aplicações desktop com Windows Forms ou WPF.
    - Desenvolvimento de aplicações web com ASP.NET ou ASP.NET Core.
    - Criação de serviços e APIs com o ASP.NET Web API.
    - Desenvolvimento de aplicações móveis com Xamarin.
    - Criação de aplicações de jogos com Unity, que utiliza a Framework .NET para lógica do jogo.

A Framework .NET é uma plataforma de desenvolvimento versátil e escalável que oferece uma ampla gama de recursos e ferramentas para criar aplicações .NET em várias plataformas. As bibliotecas de classes abrangentes, ambiente de execução robusto e ecossistema ativo, a Framework .NET aumenta a produtividade dos programadores e permite o desenvolvimento de aplicações poderosas e seguras.

Embora exija uma curva de aprendizagem inicial e possa ter restrições da plataforma, as vantagens gerais da Framework .NET tornam-na uma escolha popular para o desenvolvimento de aplicações .NET.

A .Net 8 representa uma versão mais recente e avançada da plataforma .NET. Faz parte da estratégia de unificação da Microsoft, que tem como objetivo unificar as várias implementações do .NET em uma única plataforma, tornando o desenvolvimento multiplataforma mais fácil. A .NET 8 oferece melhorias significativas de desempenho e eficiência, tornando-a adequada para aplicações de alto desempenho, incluindo aquelas em tempo real e sistemas de IA. Além disso, oferece suporte avançado para o desenvolvimento Web e na Nuvem, incluindo suporte a *gRPC, Blazor*, *Azure functions* e

muito mais. A .NET 8 também continua a oferecer suporte ao .Net Standard, garantindo que as bibliotecas existentes sejam facilmente utilizáveis. É uma plataforma de código aberto, o que significa que a comunidade também pode contribuir para o seu desenvolvimento e melhorias.

4.1 - PORQUÊ UTILIZAR .NET 8 PARA MICRO SERVIÇOS

A nova Framework .NET, especialmente o .NET 6 (e as suas versões subsequentes, como a .NET 7, .NET 8, etc.), oferece diversas vantagens significativas para o desenvolvimento de aplicações de micro serviços. Algumas dessas vantagens são:

- **Desempenho Aprimorado:** Foram projetadas com um foco significativo em desempenho. Apresentam melhorias substanciais na velocidade de inicialização de aplicações e no desempenho de execução, tornando-as ideais para micro serviços que precisam ser altamente responsivos e eficientes.
- **Portabilidade**: São altamente portáveis e podem ser executadas em várias plataformas, incluindo Windows, Linux e macOS. Isso facilita a implantação de micro serviços em ambientes heterogéneos.
- **Menor tamanho de implantação:** Oferece suporte ao conceito de "*publish single file*" (publicar como um único ficheiro), o que permite criar implantações muito menores. Isso é essencial para os micro serviços, pois reduz os requisitos de armazenamento e acelera a implantação.
- **Suporte a contentores:** Tem um excelente suporte a contentores, facilitando a aglomeração dos micro serviços nos contentores do Docker. Esta abordagem ajuda na implantação, escalabilidade e gestão de micro serviços em orquestradores de contentores como o K*ubernetes*.
- **Integração em APIs Modernas:** Têm suporte nativo para tecnologias e padrões modernos, como gRPC e RESTful APIs, facilitando a criação de micro serviços que podem comunicar facilmente com outras partes do sistema.
- **Ferramentas de desenvolvimento Avançadas**: O ecossistema de ferramentas .NET, incluindo o Visual Studio e o Visual Studio Code, oferece suporte avançado para o desenvolvimento de micro serviços, incluindo depuração, criação de perfil e geração de documentação.
- **Modelo de implantação Simplificado:** Oferece uma abordagem simplificada para a implantação e gestão de micro serviços. Com suporte a contentores e orquestração de contentores, o processo de escalabilidade e recuperação de falhar torna-se mais eficiente.
- **Segurança avançada:** Tem recursos avançados de segurança, incluindo autenticação e autorização integradas. Isso é fundamental para proteger os micro serviços, especialmente quando eles fazem parte de sistemas distribuídos complexos.

- **Escalabilidade Dinâmica:** A .NET e as tecnologias relacionadas, como o ASP.NET Core, permitem dimensionar micro serviços dinamicamente com base nos pedidos. Isto é essencial para lidar com cargas variáveis e picos de tráfego.
- **Maturidade e Comunidade Ativa:** É uma plataforma madura com uma comunidade ativa de programadores e ampla adoção na indústria. Isto significa que podemos contar com suporte continuo e acesso a uma grande variedade de bibliotecas e recursos de terceiros.

No geral, a .NET é uma escolha sólida para o desenvolvimento de micro serviços, especialmente nas versões mais recentes. Oferece um conjunto de recursos poderosos que facilitam o desenvolvimento, implantação e escalabilidade de micro serviços em ambiente complexos e distribuídos.

Capítulo 5 - PADRÕES, PRINCÍPIOS E MODELOS APLICADOS EM MICRO SERVIÇOS

A utilização dos padrões S.O.L.I.D., *Repository*, *Unit Of Work*, dos princípios do *Domain-Driven Design* (DDD) e do *Clean Architeture*, juntamente com modelos específicos aplicados aos micro serviços, são um pilar fundamental para garantir a eficiência, escalabilidade e manutenção em sistemas distribuídos [13].

5.1 - S.O.L.I.D

Os princípios S.O.L.I.D, Figura 14, são um conjunto de diretrizes que ajudam a projetar e desenvolver software de forma mais modular, flexível e fácil de manter. Foram introduzidos por Robert C. Martin (também conhecido com *Uncle Bob*) [14] e são amplamente utilizados na indústria de desenvolvimento de software. Abaixo explicar-se cada um dos cinco princípios S.O.L.I.D. e fornecesse exemplos práticos para cada um deles.

Figura 14 - Princípios S.O.L.I.D.[6]

- **Princípio da Responsabilidade Única (*Single Responsability Principle* - SRP):** Afirma que uma classe deve ter apenas uma razão para mudar, ou seja, deve ter apenas uma responsabilidade. Isto significa que uma classe deve ter um único propósito e ser responsável por uma única parte do comportamento do sistema. Isso ajuda a manter as classes coesas, de fácil compreensão e evolução.

[6] Retirado de: https://blog.betrybe.com/linguagem-de-programacao/solid-cinco-principios-poo

Exemplo: Considere uma classe chamada "*MailSender*" que é responsável por enviar e-mails. Deve ter apenas a responsabilidade de enviar e-mails, sem se preocupar com a lógica de autenticação do utilizador. Se a classe também tiver responsabilidade de autenticar o utilizador, ela estaria a violar o **SRP**. Nesse caso, seria melhor separar a responsabilidade de autenticação noutra classe.

- **Princípio do aberto/fechado (*Open/Close Principle* - OCP):** Estabelece que as entidades do software (classes, módulos, etc.) devem estar abertas para extensão, mas fechadas para modificação. Isso significa que podemos adicionar novos comportamentos ou estender a funcionalidade sem alterar o código existente. O objetivo é evitar a quebra de código já testado, validado e funcional.
  Exemplo: Supomos que temos uma classe chamada "Calculadora" com um método "Calcular". Se precisamos de adicionar uma nova operação matemática, em vez de modificar a classe "Calculadora", podemos criar uma nova classe que herda da "Calculadora" e implementa a nova operação. Isso permite que possamos estender a funcionalidade da calculadora sem modificar o código existente.
- **Princípio da Substituição de Liskov (*Liskov Subtitution Principle* – LSP):** Estabelece que as classes derivadas devem ser substituíveis por classes base sem afetar o correto funcionamento do programa. Isso significa que uma classe derivada deve poder ser usada em qualquer lugar onde a classe base é esperada, sem causar efeitos colaterais indesejados ou violar as pré-condições e pós condições de contrato da classe base.
  Exemplo: Considere uma hierarquia de classes onde temos uma classe base chamada "Animal" e uma classe derivada chamada "Cão". Se o código espera um objeto do tipo "Animal", tem de ser capaz de lidar com um objeto do tipo "Cão" sem qualquer tipo de problema. Obedecer ao LSP garante que o comportamento esperado do objeto seja mantido em todas as subclasses.
- **Princípio da Segregação de interfaces (*Interfaces Segregation Pincíple* - ISP): a**firma que as *interfaces* devem ser específicas para os clientes que as utilizam, evitando *interfaces* monolíticas e que os clientes dependam dos métodos que não usam. O objetivo é evitar a dependência desnecessária e garantir que as classes implementem apenas o que é necessário para o uso específico.
  Exemplo: Suponhamos que temos uma *interface* chamada "IImpressora" com os métodos "imprimir", "digitalizar", e "enviar mail". No entanto, num cliente específico, só preciso do método "Imprimir". Neste caso, seria melhor criar uma *interface* separada, "Imprimir Simples", contendo apenas o método necessário. Isso evita a dependência de métodos não utilizados e mantêm as *interfaces* coesas e focadas.
- **Princípio da inversão de dependência (*Dependency Inversion Principle* - DIP):** Estabelece que as classes de alto nível não devem depender das

classes de baixo nível diretamente. Em vez disso, ambas devem depender de abstrações. Isso promove o desacoplamento e facilita a substituição de implementação, permitindo maior flexibilidade e reutilização de código.
Exemplo: Em vez de uma classe de alto nível depender diretamente de uma classe de baixo nível, ambas devem depender de uma *interface* comum. Por exemplo, num sistema de gestão de pedidos, em vez de uma classe "*Order Manager*" depender diretamente da classe "*DatabaseConnection*", ambas podem depender de uma *interface* "*IDatabaseConnection*". Isto permite que diferentes implementações de conexão com a base de dados sejam facilmente trocadas sem afetar a classe de alto nível.

Estes são os princípios S.O.L.I.D., que fornecem diretrizes valiosas para o design do software de qualidade, facilitando a manutenção, extensibilidade e testabilidade. Ao aplicar estes princípios, podemos criar aplicações mais robustas e flexíveis.

5.2 - DOMAIN-DRIVEN DESIGN

O *Domain-Driven Design* (DDD), Figura 15, traduzido como Desenvolvimento Orientado ao Domínio, é uma abordagem de *design* e desenvolvimento de software que se concentra na compreensão profunda do domínio de negócio e na modelagem desse domínio em código. O DDD foi introduzido por Eric Evans no livro *"Domain-Driven Design: Tackling Complexity in the heart of software"* [15]. Essa abordagem visa criar um software que reflita com precisão as complexidades e as regras do domínio de negócio, resultando em sistemas mais flexíveis, escaláveis e alinhados com as necessidades reais do utilizador.

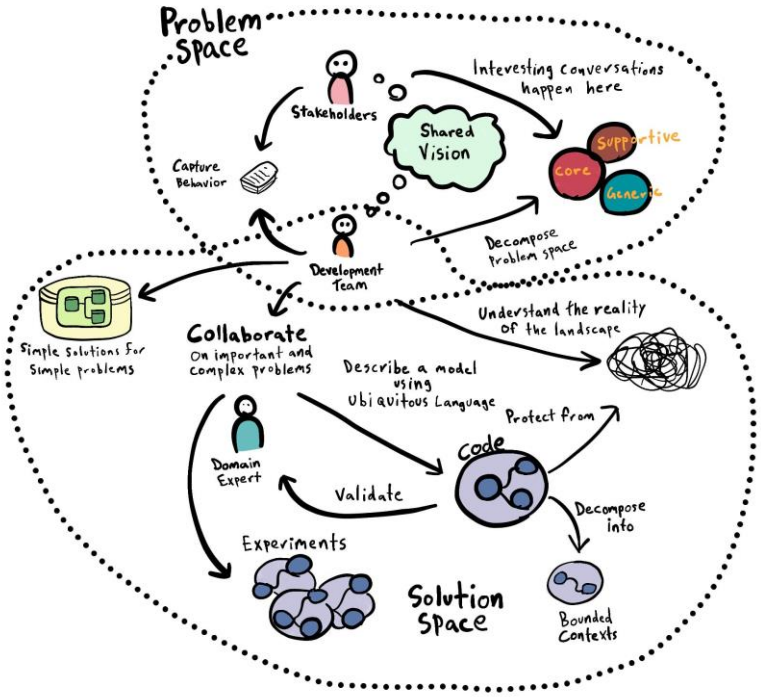


Figura 15 - Domain-Driven Design[7]

Conceitos fundamentais do DDD.

- **Linguagem Ubíqua:** É uma parte essencial do DDD. Consiste em estabelecer uma linguagem comum entre o programador e especialistas do domínio, eliminando ambiguidades e criando um vocabulário compartilhado. Essa linguagem é usada tanto na comunicação entre as equipas de desenvolvimento como na modelação do software.
- ***Bounded Contexts*:** É utilizado para delimitar a fronteira de um subdomínio específico dentro do sistema. Cada *Bounded Context* possui as suas próprias regras de negócio e é responsável por um conjunto limitado de funcionalidades relacionadas. Dentro de um *Bounded Context*, os modelos e as abstrações podem ser definidos de acordo com as necessidades do domínio específico, sem se preocupar com outros contextos.
- ***Aggregate Roots*:** São objetos de domínio que encapsulam um grupo de entidades relacionadas e são tratados como uma única unidade transacional. O *Aggregate Root* define as regras de consistência e as operações que podem ser realizadas nas suas entidades associadas. Essa abordagem garante a consistência e a integridade das entidades dentro do *Aggregate Root*.
- **Modelação do domínio:** É uma parte crucial do DDD, e envolve a criação de um modelo rico e expressivo que reflita as regras, os comportamentos e as interações do domínio de negócio. Essa modelação é realizada em conjunto com especialistas do domínio, através de *workshops* colaborativos e

[7] Retirado de: https://medium.com/raa-labs/part-1-domain-driven-design-like-a-pro-f9e78d081f10

interações continuas. Alguns elementos importantes da modelação do domínio são:

- **Entidades:** Representam objetos do mundo real com identidade própria. Possuem atributos e comportamentos, e são persistentes ao longo do tempo. As entidades são responsáveis por manter a integridade dos dados e implementar as regras de negócio relacionadas com esses dados.
- ***Value Objects*:** São objetos que não possuem uma identidade própria, mas são definidos pelos seus atributos. Representam conceitos imutáveis e são utilizados para encapsular valores e comportamentos específicos. Por exemplo, um *Value Object* poderá ser utilizado para representar uma data de nascimento ou uma morada.
- **Serviços de Domínio:** São responsáveis por executar operações que envolvem múltiplas entidades ou agregados. Encapsulam a lógica complexa e são utlizados quando uma operação não pertence exclusivamente a uma única entidade ou *Value Object*. Os serviços de domínio podem ser utilizados para validar regras de negócio, orquestrar interações entre objetos do domínio, entre outras tarefas.
- **Eventos de domínio:** São utilizados para comunicar mudanças significativas que ocorrem no domínio. Representam fatos ocorridos e podem ser consumidos por outros componentes do sistema. Os eventos do domínio permitem uma comunicação assíncrona entre os diferentes contextos delimitados.

- **Arquitetura Orientada ao Domínio:** A arquitetura de um sistema baseado em DDD deve ser orientada ao domínio, refletindo a estrutura do domínio do negócio. Alguns padrões e conceitos arquiteturais relacionados ao DDD são:
  - **Camadas Arquiteturais:** A arquitetura em camadas é habitualmente utilizada em sistemas baseados em DDD. Esta abordagem consiste em dividir o sistema em camadas distintas, como a camada de apresentação, a camada de aplicação e a camada de infraestrutura. Cada camada possui responsabilidades especificas e comunica com as outras camadas por meio de *interfaces* bem definidas.
  - **CQRS (*Command Query Responsability Segregation*):** É um padrão arquitetural que separa a responsabilidade de comandos (alteração de estado) e consultas (leitura de estado) em modelos de domínio diferentes. Essa separação permite otimizar a performance e a escalabilidade do sistema, além de permitir diferentes modelos de leitura específicos para cada necessidade.
  - ***Event Sourcing*:** É a técnica que consiste em armazenar todas as alterações de estado do sistema como uma sequência de eventos. Esses eventos são armazenados num log e podem ser utilizados para reconstruir o estado do sistema em qualquer ponto no tempo. O *Event*

Sourcing é especialmente útil quando se lida com sistemas complexos de negócio que envolvem ao longo do tempo.

- **Implementação e testes:** A implementação de um sistema baseado em DDD envolve a tradução do modelo de domínio em código concreto. É importante que o código reflita com precisão as abstrações e as regras definidas durante a modelagem do domínio. Além disso, é fundamental realizar testes adequados para garantir a fiabilidade do sistema. Alguns tipos de testes comuns em DDD são:
  - **Testes unitários:** Garantem que as entidades, os *Value Objects* e os serviços de domínio funcionem corretamente, de acordo com as regras de negócio estabelecidas. Estes testes devem abranger diferentes cenários e casos de utilização para garantir a integridade do modelo de domínio.
  - **Testes de aceitação:** São realizados em nível de funcionalidades completas do sistema. Validam o comportamento do sistema como um todo, garantindo que todas as partes estejam integradas corretamente e atendam aos requisitos estabelecidos pelos especialistas do domínio.

Para desenvolver software seguindo o modelo DDD, geralmente são utilizadas três camadas principais: a camada de apresentação (ou *interface* com o utilizador), a camada de domínio e a camada de infraestrutura. Vamos analisar cada uma delas, como se interligam e quais dependem umas das outras:

- **Camada de apresentação (*interface* com utilizador):** Responsável por lidar com a interação entre o utilizador e o sistema. Exibe informações para o utilizador e captura as ações efetuadas pelo mesmo. Esta camada depende da camada de domínio para aceder às regras de negócio e aos dados do domínio.
  Exemplo: Suponhamos que estamos a desenvolver uma aplicação para gestão de tarefas. Na camada de apresentação, podemos ter uma janela de utilizador que apresenta uma lista de tarefas e permite que o utilizador adicione, edite e apague tarefas. A camada de apresentação utiliza os serviços da camada de domínio para executar essas ações e exibir as informações relevantes para o utilizador.
- **Camada de domínio:** É o núcleo do modelo DDD, contém a lógica do negócio e representa os conceitos e as regras do domínio. Esta camada não deve depender diretamente de outras camadas e deve ser independente de qualquer tecnologia especifica.
  Exemplo: Continuando com o exemplo da aplicação de gestão de tarefas, na camada de domínio teremos a entidades como “Tarefa” e “Utilizador”. Estas entidades encapsulam os atributos e comportamentos relacionados a cada conceito. Além disso, podemos ter agregados, tais como o agregado “Lista de Tarefas”, que agrupa as tarefas relacionadas a um utilizador específico. A

camada domínio define as *interfaces* que são implementadas pela camada de infraestrutura.

- **Camada de infraestrutura:** Lida com detalhes técnicos, como persistência de dados, comunicação com serviços externos e outros aspetos relacionados à infraestrutura do sistema. Esta camada depende da camada de domínio para implementar as *interfaces* definidas na camada de infraestrutura.
  Exemplo: No contexto da aplicação de gestão de tarefas, a camada de infraestrutura pode incluir implementação concreta de *interfaces* definidas na camada de domínio, como um repositório para manter as tarefas numa base de dados. Além disso, pode conter componentes para autenticação do utilizador, integração com serviços externos, gestão de cache, entre outros.

A interligação entre as camadas ocorre através de *interfaces*. A camada de apresentação utiliza *interfaces* da camada de domínio para interagir com as entidades e os serviços do domínio. A camada de infraestrutura implementa essas *interfaces* e é responsável por lidar com os detalhes técnicos necessários.

No modelo DDD, é importante manter a separação de responsabilidades entre as camadas. A camada de domínio deve ser independente e não deve conhecer detalhes da camada de infraestrutura. Isso permite que a camada de domínio seja testada e evoluída independentemente da tecnologia da infraestrutura utilizada.

A camada de apresentação depende da camada de domínio para aceder às regras de negócio e aos dados do domínio. A camada de infraestrutura depende da camada de domínio para implementar as *interfaces* definidas. As *interfaces* fornecem as comunicações entre as camadas e garantem a separação de responsabilidades.

5.3 - CLEAN ARCHITETURE

A *Clean Architeture,* Figura 16, é um conceito proposto por Robert C. Martin, também conhecido como "*Uncle Bob*". Esta abordagem à arquitetura de software visa criar sistemas independentes de *Frameworks*, *interfaces* de utilizadores e detalhes externos, mantendo a lógica de negócio central limpa e desacoplada. Os princípios fundamentais da *Clean Architeture* incluem:

- **Independência de Framework (*Domain*):** A lógica de negócio deve ser independente de qualquer *Framework* externo, como bibliotecas gráficas, *Framework* web ou base de dados especificas.
- **Testabilidade:** A arquitetura deve permitir testar facilmente a lógica de negócio de forma automatizada, sem depender de *interfaces* de utilizador, base de dados ou outros componentes externos.
- **Independência de UI (*Presentation*):** A *interface* com utilizador (UI) é considerada uma camada externa e deve ser separada da lógica central. Isso

facilita a substituição ou atualização de *interfaces* do utilizador sem afetar a lógica subjacente.

- **Independência de dados (*Infrastructure Data*):** O acesso a dados, como base de dados e repositórios, é tratado como uma camada externa e não deve influenciar a lógica de negócio central.
- **Princípio de Inversão de Controlo (IoC):** A direção da dependência é invertida de modo que os módulos de alto nível não dependam dos módulos de baixo nível. Ambos devem depender de abstrações.
- **Separação de Responsabilidades:** A arquitetura promove a separação de responsabilidades, dividindo o sistema em camadas como Entidades de Negócio, Casos de Utilização, *Interfaces* do Utilizador e Infraestrutura.

A *Clean Architeture* destaca a importância de manter a lógica de negócio como o núcleo do sistema, protegendo-a de mudanças externas e facilitando a evolução e manutenção do software ao longo do tempo. Esta abordagem visa criar sistemas mais flexíveis, testáveis e sustentáveis.

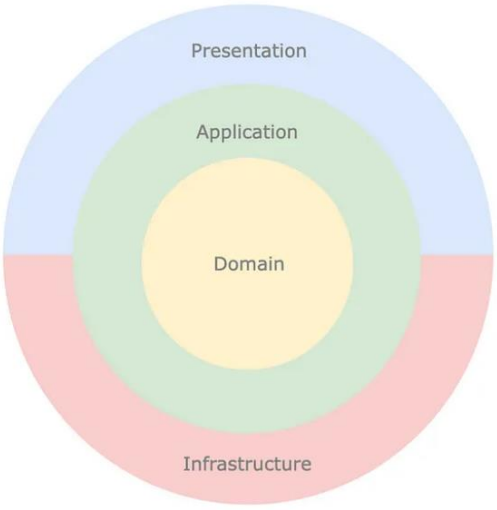


Figura 16 – Modelo do *Clean Architecture*[8]

5.4 - REPOSITORY PATTERN

Nesta secção será apresentado o *Repository Pattern,* Figura 17, e as suas principais vantagens e desvantagens.

[8] Retirado de: https://levelup.gitconnected.com/how-to-structure-net-project-with-clean-architecture-0192dd3c62c2

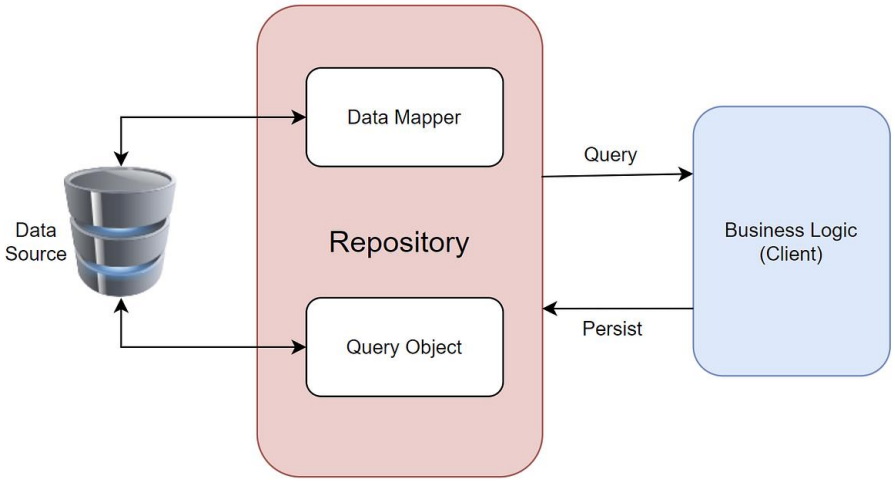


Figura 17 – Repository Pattern[9]

É uma abordagem de design de software que separa a lógica de negócio de acesso aos dados nas aplicações. Fornece uma camada de abstração entre os componentes de negócio da aplicação e o mecanismo de armazenamento de dados, seja base de dados, um serviço web ou qualquer outro meio [16].

Encapsula a lógica para recuperar e armazenar objetos de domínio de e para armazenamento de dados. Define uma *interface* comum para acesso a esses objetos, permitindo que os componentes de negócio da aplicação operem com eles sem precisar conhecer os detalhes específicos de como os dados são persistidos.

Por exemplo, numa aplicação de comercio eletrónico, em vez dos componentes de negócio interagirem diretamente com a base de dados para selecionar e guardar informações sobre produtos, pedidos e clientes, interagem com as interfaces do repositório especificas para cada tipo de entidade (*ProdutoRepository*, *PedidoRepository*). As interfaces do repositório definem métodos para as operações de CRUD.

É comumente utilizado em aplicações de software de médio e grande porte, especialmente aqueles que seguem a estrutura de camadas, como a arquitetura MVC (*Model-View-Controller*) ou MVVM (*Model-View-ViewModel*). Ajuda a manter a separação das preocupações e promove a reutilização do código, ao centralizar a lógica de acesso aos dados numa única camada da aplicação.

Vantagens:

- **Abstração de Dados:** Permite que os componentes de negócio funcionem com objetos de domínio sem se preocuparem com os detalhes de como os dados são armazenados e recuperados.

[9] Retirado de: https://medium.com/@pantaanish/understanding-repository-pattern-with-implementation-a-step-by-step-guide-ca1bf36be3b4

- **Facilidade de teste:** Facilita a criação de testes unitários, já que as implementações concretas do repositório podem ser substituídas por versões idênticas.
- **Centralização da lógica de acesso aos dados:** Torna mais fácil manter e modificar a lógica de acesso a dados, já que está centralizada numa única camada da aplicação.
- **Reutilização de Código:** Promove a reutilização de código ao fornecer uma interface comum para acesso a dados que pode ser partilhada entre diferentes componentes de negócio.

Desvantagens:

- **Complexidade adicional:** Introduz uma camada adicional de abstração, o que pode aumentar a complexidade do código.
- ***Overhead* de desenvolvimento:** Criar e manter interfaces e implementações do repositório pode adicionar *overhead* ao processo de desenvolvimento.
- **Possível sobrecarga de desempenho:** Em alguns casos, a camada adicional de abstração introduzida pelo *Repository Pattern* pode resultar numa pequena sobrecarga de desempenho, embora isso geralmente não seja significativo na maioria das aplicações modernas.

5.5 - UNIT OF WORK PATTERN

No desenvolvimento de software, especialmente em aplicações que interagem com base de dados, o conceito de "*Unit of Work*", Figura 18, desempenha um papel fundamental na gestão de transações de dados de forma eficiente e coesa.

É um padrão de design que agrupa um conjunto de operações relacionadas a dados numa única transação lógica. Representa uma única unidade de trabalho que deve ser concluída de forma consistente, garantindo que todas as operações de leitura e escrita de dados associadas sejam tratadas como uma única entidade [17].

Quando uma unidade de trabalho é iniciada, várias operações de acesso aos dados podem ser executadas, como adicionar, atualizar ou apagar registos na base de dados. As operações são agrupadas e executadas em conjunto dentro da transação da unidade de trabalho. Se alguma das operações falhar, a transação pode ser revertida, garantindo que o estado dos dados permaneça consistente.

O padrão da unidade de trabalho é comumente aplicado em cenários onde é necessário garantir a integridade dos dados ao realizar operações complexas que envolvem múltiplas interações com a base de dados. É frequentemente utilizado em conjunto com o *Repository Pattern*, o repositório é responsável por lidar com as

operações individuais do acesso aos dados, enquanto o *Unit Of Work Pattern* coordena essas operações e garante a consistência dos dados.

Vantagens:

- **Consistência dos dados:** Garante que todas as operações relacionadas a dados sejam tratadas de forma coesa, mantendo a integridade e consistência dos dados.
- **Atomicidade das transações:** As operações dentro da unidade de trabalho são tratadas como uma única transação atómica, ou seja, todas as operações são bem-sucedidas ou todas são revertidas em caso de falhar, isto evita inconsistência nos estados.
- **Redução da complexidade:** Ao agrupar operações relacionadas em uma única Unidade de trabalho, o código torna-se mais organizado e fácil de manter, reduzindo a complexidade da gestão das transações.

Desvantagens:

- **Overhead de desenvolvimento:** A implementação da unidade de trabalho pode adicionar algum overhead ao desenvolvimento, especialmente em casos simples onde as transações são diretas e não complexas.
- **Potencial para deadlocks:** Se não for implementado corretamente, pode aumentar o risco de deadlocks em sistemas concorrentes, onde múltiplas transações ocorrem em simultâneo.

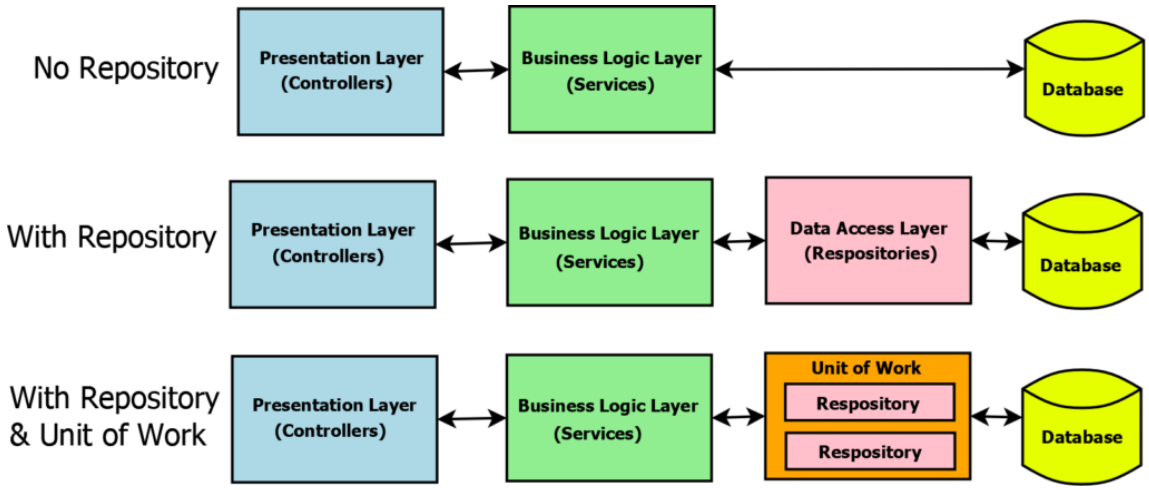


Figura 18 - Unit Of Work Pattern[10]

[10] Retirado de: https://medium.com/@edin.sahbaz/implementing-the-unit-of-work-pattern-in-clean-architecture-with-net-core-53efb7f9d4d

Capítulo 6 - TIPOS DE COMUNICAÇÃO ENTRE MICRO SERVIÇOS

6.1 - SISTEMA DE TROCA DE MENSAGENS

Num ambiente de micro serviços, onde várias partes independentes do sistema são implementadas como serviços independentes, a comunicação entre esses serviços é fundamental. É nesse contexto que os sistemas de troca de mensagens desempenham um papel crucial. Um sistema de troca de mensagens é uma infraestrutura que permite a troca assíncrona de mensagens entre os diferentes componentes de um sistema distribuído [18].

Num sistema de troca de mensagens, os serviços podem comunicar entre si enviando e recebendo mensagens assíncronas através de um intermediário, conhecido como o "*broker*". O *broker* é responsável por receber as mensagens dos produtores (serviços que enviam mensagens) e encaminhá-las para os consumidores (serviços que recebem e processam as mensagens). Existem dois principais padrões de troca de mensagens:

- **Padrão de Publicação/Subscritor (*Publish/Subscribe*):** Os serviços publicam mensagens num tópico específico e os serviços interessados em receber essas mensagens subscrevem esse tópico. O broker encaminha as mensagens para todos os consumidores inscritos no tópico. Este padrão permite uma comunicação flexível e escalável, onde os produtores não precisam conhecer os consumidores e vice-versa.
  Exemplo: Suponhamos um sistema de *e-commerce*, onde um serviço de *stocks* publica mensagens sobre atualizações de disponibilidade de produtos num tópico chamado "*Stock.atualizado*". Diversos serviços, como serviço de recomendação ou serviço de notificações, podem inscrever-se nesse tópico para receber as atualizações do *stock* e tomar ações apropriadas.
- **Padrão de Filas de Mensagens "*Message Queue*":** Os serviços enviam mensagens para uma fila específica e os consumidores recuperam as mensagens da fila de forma ordenada e processam uma de cada vez. Cada mensagem é processada por apenas um consumidor, garantindo um processamento sequencial e a divisão de carga entre os consumidores.
  Exemplo: Imaginemos um serviço de processamento de pedidos em um sistema de entrega. Os pedidos são enviados para uma fila chamada "fila_pedidos" e os trabalhadores da entrega recuperam os pedidos da fila, processam-nos um por um, garantindo que cada pedido seja tratado adequadamente.

Cada sistema de troca de mensagens possui as suas próprias características, desempenho, escalabilidade e recursos específicos, o que torna importante avaliar e

escolher aquele que melhor se adequa às necessidades do sistema pretendido de micro serviços.

Os sistemas de troca de mensagens desempenham um papel fundamental em arquiteturas de micro serviços, facilitando a comunicação assíncrona e desacoplada entre os serviços. Com o uso de padrões como *publish/subscribe* e filas, é possível criar sistemas escaláveis, flexíveis e resilientes. A escolha do sistema de troca de mensagens adequado depende das necessidades do sistema em termos de desempenho, confiabilidade e recursos específicos.

6.1.1 - EXEMPLOS POPULARES

- **Apache Kafka:** Projetado para fornecer uma solução de troca de mensagens altamente escalável, tolerante a falhas e de alto desempenho. É amplamente utilizado em cenários onde a latência e a capacidade de processamento em tempo real são essenciais, como *streaming* de dados, processamento de eventos, gestão de dados em tempo real, log de auditorias e monitorização de aplicações. Principais conceitos do Apache Kafka:
    - **Tópico (*Topic*):** É uma categoria ou canal de mensagens no Kafka. Os produtores publicam mensagens em tópicos específicos, e os consumidores leem essas mensagens a partir dos tópicos. Os tópicos são divididos em participações para permitir a distribuição e paralelização do processamento.
    - **Partição (*Partition*):** Cada tópico é dividido numa ou em mais partições. As partições são unidades de armazenamento e processamento no Kafka. Permitem que os dados sejam distribuídos em vários servidores Kafka e processados em paralelo.
    - **Produtor (*Producer*):** É a entidade responsável por enviar mensagens para os tópicos do Kafka. Os produtores publicam as mensagens em partições específicas ou permitem que o Kafka atribua automaticamente as partições.
    - **Consumidor (*Consumer*):** É a entidade que lê as mensagens dos tópicos do Kafka. Os consumidores podem ser configurados para ler das partições especificas ou podem beneficiar da distribuição automática do Kafka.
    - **Grupo de Consumidores (*Consumer Group*):** É um conjunto de consumidores que trabalham juntos para consumir mensagens de um tópico. Cada partição de um tópico é consumida por apenas um consumidor dentro de um tópico. Cada partição de um tópico é consumida por apenas um consumidor dentro de um grupo de consumidores.

- ***Broker*:** É um nó individual no *cluster* do Kafka. Cada *broker* é responsável por armazenar e servir as partições dos tópicos. Um *cluster* do Kafka é composto por vários *brokers* que trabalham em conjunto para garantir a tolerância a falhas e escalabilidade.

Características e benefícios do Apache Kafka:

- **Escalabilidade:** É altamente escalável e pode lidar com grandes volumes de dados e cargas de trabalho intensivas.
- **Alta taxa de transferência:** Oferece uma alta taxa de transferência de mensagens, permitindo o processamento eficiente de grandes fluxos de dados em tempo real.
- **Persistência:** As mensagens no Kafka são persistidas em disco, permitindo a recuperação de falhas e a nova tentativa de processamentos.
- **Tolerância a falhas:** É projetado para ser tolerante a falhas, garantindo a disponibilidade continua mesmo em caso de falha de um ou mais *brokers*.
- ***Streaming* em tempo real:** O Kafka é adequado para *streaming* de dados em tempo real, permitindo a gestão, processamento e análise contínuos de fluxos de dados em tempo real.
- **Ecossistema rico:** Possui um ecossistema robusto de ferramentas e integrações, permitindo utilização em conjunto com outras tecnologias com o *Hadoop*, *Spark* e base de dados *NoSQL*.

O Apache Kafka é amplamente adotado por empresas de diferentes setores para utilizações tais como processamento de eventos em tempo real, *log* de auditorias, análise de dados, integração de sistemas e muito mais.

- **RabbitMQ:** É um sistema de troca de mensagens *open source*, robusto e altamente flexível. Foi desenvolvido em Erlang e implementa o padrão de troca de mensagens com base em filas (*message queue*). O RabbitMQ é projetado para lidar com a troca de menagens assíncronas entre aplicações distribuídas e é amplamente utilizado em arquiteturas de micro serviços e sistemas distribuídos. Principais conceitos do RabbitMQ:
  - **Produtor (*Producer*):** É o componente responsável por enviar mensagens para o RabbitMQ. Os produtores publicam mensagens em uma fila especifica ou em um tópico de troca (*exchange*), que é responsável por encaminhar as mensagens para as filas corretas.
  - **Consumidor (*Consumer*):** É o componente que consome as mensagens do RabbitMQ. Os consumidores ligam-se a filas especificas para recuperar e processar as mensagens.

- **Fila (*Queue*):** É um mecanismo de armazenamento no RabbitMQ que contem as mensagens publicadas pelos produtores. Os consumidores recuperam as mensagens da fila de processamento.
- **Tópico de Troca (*Exchange*):** É um componente que recebe mensagens de produtores e as encaminha para as filas corretas com base em regras de encaminhamento. O RabbitMQ suporta vários tipos de troca, como trocas de encaminhamento direto, tópicos, *headers* e *fanout*.
- **Encaminhamento (*Routing*):** É o processo pelo qual o RabbitMQ direciona as mensagens para as filas corretas com base em regras de encaminhamento definidas pelo tópico de troca. As mensagens podem ser encaminhadas com base em padrões de encaminhamento, chaves de encaminhamento, cabeçalhos ou outras critérios.
- **Fila Durável (*Durable Queue*):** É uma fila no RabbitMQ que sobrevive a reinicialização ou falhas do servidor. As filas duradouras são armazenadas em disco para garantir que as mensagens não sejam perdidas em caso de falha.

Características e benefícios do RabbitMQ:

- **Alta flexibilidade:** Oferece uma variedade de padrões de troca e opções de encaminhamento, permitindo uma configuração flexível para atender a diferentes cenários de troca de mensagens.
- **Garantia de entrega:** Garante que as mensagens sejam entregues de forma confiável aos consumidores. Suporta confirmações de entrega (*acknowledgements*) para evitar perda de mensagens.
- **Filas prioritárias:** Permite a configuração de filas prioritárias, onde mensagens com prioridades mais altas são processadas antes das mensagens com prioridades mais baixas.
- **Escalabilidade e *clustering*:** Suporta *clustering*, permitindo a escalabilidade horizontal e a distribuição de carga entre vários nós do RabbitMQ.
- **Suporte a diversos protocolos:** Suporta protocolos de troca de mensagens como AMQP (*Advanced Message Queuing Protocol*), MQTT (*Message Queuing Telemetry Transport*) e o STOMP (*Simple Text Oriented Messageng Protocol*).
- **Integração com diferentes tecnologias:** Pode ser integrado facilmente com várias tecnologias, como *frameworks* de desenvolvimento, base de dados e sistemas de processamento de dados em tempo real.

O RabbitMQ é amplamente adotado em ambientes de micro serviços e sistemas distribuídos, onde a troca de mensagens assíncronas é necessária para

garantir a escalabilidade, resiliência e o baixo acoplamento entre os componentes do sistema.

- **ActiveMQ:** É um sistema de troca de mensagens *open source* que implementa o padrão JMS (*Java Message Service*). Fornece uma plataforma robusta e escalável para troca de mensagens assíncronas entre aplicações distribuídas. Principais conceitos do A*ctiveMQ*:
  - **Produtor (*Producer*):** É o componente responsável por enviar mensagens para o ActiveMQ. Os produtores podem enviar mensagens para filas (*queues*) ou tópicos (*topics*).
  - **Consumidor:** É o componente que consome as mensagens do ActiveMQ. Os consumidores conectam-se às filas ou tópicos para receber e processar as mensagens.
  - **Filas (*Queue*):** É uma estrutura de armazenamento no Active MQ que segue o modelo de troca de mensagens ponto a ponto. As mensagens são entregues a apenas um consumidor registado na fila.
  - **Topico (*Topics*):** É uma estrutura de armazenamento no ActiveMQ que segue o modelo de troca de mensagens de publicação/subscrição. As mensagens são entregues a todos os consumidores que subscreveram o tópico.
  - **Mensagem (*Message*):** É a unidade de informação trocada no ActiveMQ. Uma mensagem contém os dados a serem transmitidos, bem como meta dados com identificadores, tipos e propriedades.
  - **Persistência:** Suporta a persistência de mensagens em disco, garantindo que as mensagens não sejam perdidas em caso de falha do sistema.

  Características e benefícios do ActiveMQ:

  - **Escalabilidade e desempenho:** É altamente escalável e pode lidar com grandes volumes de mensagens. Suporta *clustering*, permitindo a distribuição da carga entre vários nós.
  - **Suporte a protocolos:** Suporta vários protocolos de comunicação, incluído o protocolo *OpenWire*, MQTT, AMQP e STOMP.
  - **Recursos avançados de troca de mensagens:** Oferece recursos como filtragem de mensagens, transações, mensagens persistentes, confirmações de entrega e colocação em fila por prioridades.
  - **Integração com o ecossistema Java:** É desenvolvido em Java e integra facilmente com outras tecnologias e *frameworks* Java.
  - **Suporte a diferentes tipos de clientes:** Pode ser utilizado por aplicações Java, bem como por aplicações em outras linguagens de programação, através de APIs e bibliotecas disponíveis.

- **Monitorização e gestão:** Fornece ferramentas e *interfaces* para monitorizar e gerir o desempenho, o estado e a disponibilidade do sistema.

O ActiveMQ é amplamente utilizado em arquiteturas de micro serviços, integração de sistemas, processamento de eventos, troca de dados assíncrona e muitos outros cenários em que a comunicação entre aplicações distribuídas é necessária.

6.2 - SISTEMA GATEWAY

Um sistema de *gateway* num software de micro serviços é um componente responsável por gerir o tráfego entre clientes externos e os diferentes micro serviços que compõem a aplicação. Atua como uma camada intermédia entre os clientes e os micro serviços, fornecendo recursos de encaminhamento, autenticação, autorização, monitorização, balanceamento de carga e outras funcionalidades [19]. Principais funções de um sistema de *gateway*:

- **Encaminhamento:** Direciona os pedidos dos clientes para os micro serviços corretos, com base em informações como o caminho do URL (*Uniform Resource Locator*), cabeçalho ou parâmetros.
- **Autenticação e autorização:** Pode autenticar os clientes, verificar as credenciais e autorizar o acesso aos micro serviços com base em políticas de segurança.
- **Balanceamento de carga:** Pode distribuir o tráfego de entrada entre várias instâncias dos micro serviços para garantir uma distribuição equilibrada da carga.
- **Transformação de protocolos:** Pode realizar a conversão de protocolos, permitindo que os clientes comuniquem com os micro serviços usando protocolos diferentes, se necessário.
- ***Cache*:** Pode armazenar em *cache* as respostas dos micro serviços, reduzindo a latência e o tráfego de rede em pedidos subsequente.
- **Monitorização e métricas:** Pode reunir informação sobre os pedidos e respostas, gerando métricas e registos para fins de monitorização e analise de desempenho.

6.2.1 - TIPOS DE SISTEMA

API Gateway: Virado para a exposição e gestão de APIs. Fornece recursos como controlo de versões de API, documentação automática, controle de acesso, monitorização do uso e políticas de segurança.

Exemplo prático: Um cliente faz um pedido HTTP para o *endpoint* específico do API Gateway. O *gateway* autêntica o cliente, verifica as permissões e encaminha o pedido para o micro serviço correspondente, retorna a resposta ao cliente.

6.2.2 - LIVRARIAS POPULARES

- **Ocelot:** É uma livraria .NET que oferece recursos de encaminhamento, autenticação, autorização e balanceamento de carga. É configurada usando ficheiros de configuração JSON ou C#.
- **Kong:** É uma plataforma de gestão de API de código aberto que pode ser usada como um g*ateway*, oferece recursos de encaminhamento, autenticação, autorização, *rate limiting*, transformação de dados e muito mais. O Kong é altamente escalável e pode ser implantado em ambientes distribuídos.
- **Tyk:** É outra plataforma de gestão de APIs e API g*ateway* de código aberto. Fornece recursos de encaminhamento, autenticação, autorização, *rate limiting*, transformação de dados e *logging*. O Tyk é conhecido pela sua simplicidade e facilidade de uso, permitindo que os programadores configurem rapidamente os *gateways* e as políticas de API.
- ***Service Mesh*:** É uma camada de infraestrutura que fornece recursos de rede e comunicação entre os micro serviços. Inclui um componente chamado "*sidecar*" que é implantado junto com cada micro serviço e lida com funções de *gateway*, como encaminhamento, autenticação, balanceamento de carga e criptografia. Exemplo prático: Cada micro serviço tem um sidecar associado que intercepta os pedidos recebidos, realiza autenticação e encaminhamento interno, e envia o pedido para o micro serviço correto. O sidecar também lida com recursos de impedimentos de rotas, tentativas de execução nas falhas e monitorização.
- ***Message Broker Gateway*:** Atua como intermediário entre sistemas de troca de mensagens e os micro serviços. Gere a comunicação entre os micro serviços por meio de filas ou tópicos, garantindo a entrega confiável de mensagens e permitindo a integração assíncrona.
  Exemplo prático: Os micro serviços publicam mensagens num tópico específico do *Message Broker Gateway*. O gateway recebe as mensagens e encaminha-as para os micro serviços inscritos nesse tópico. Garantindo a entrega confiável e o processamento assíncrono das mensagens.

É importante salientar que a escolha do tipo de sistema de *gateway* e da livraria a ser utilizada depende dos requisitos e das necessidades específicos do software de micro serviços. Cada tipo de *gateway* e livraria possui as próprias características, funcionalidades e casos de uso adequados.

6.3 - TROCA DE MENSAGENS *VERSUS* SISTEMA GATEWAY

A adoção de arquitetura de micro serviços tem crescido rapidamente devido aos benefícios que oferecem em termos de escalabilidade, flexibilidade e desenvolvimento ágil de aplicações. Dois componentes chave para o funcionamento eficiente de uma arquitetura de micro serviços são o sistema de troca de mensagens e o sistema de *gateway*. Ambos desempenham papeis importantes na gestão da comunicação entre os micro serviços e a integração com clientes externos.

- **Sistema de troca de mensagens:** O sistema de troca de mensagens, Figura 19, é responsável pela comunicação assíncrona entre os micro serviços numa arquitetura de micro serviços. Utiliza fila ou tópicos para o envio e receção de mensagens, garantindo a entrega confiável mesmo em cenários de alta carga e aumenta a resiliência do sistema. Alguns dos principais pontos a serem considerados sobre o sistema de troca de mensagens incluem:
    - **Escalabilidade e tolerância a falhas:** Oferece uma camada de comunicação independente entre os micro serviços, permitindo que operem independentemente. Facilita a escalabilidade horizontal, onde os micro serviços poder ser dimensionados individualmente com base nas necessidades especificas. Além disso, a tolerância a falhas é aprimorada, pois os micro serviços podem continuar a operar mesmo que um serviço esteja temporariamente indisponível.
    - **Assincronismo e flexibilidade:** A comunicação assíncrona proporcionada pelo sistema de troca de mensagens permite que os micro serviços trabalhem de forma independente, processando mensagens quando for mais conveniente. Resulta em uma maior flexibilidade na implementação de funcionalidades, pois os micro serviços podem responder a mensagens no seu próprio ritmo, reduzindo a dependência entre eles.
    - **Integração Heterogénea:** O sistema de troca de mensagens permite a integração entre micro serviços desenvolvidos em diferentes linguagens de programação e plataformas. Atua como uma camada intermedia que abstrai as diferenças tecnológicas entre serviços, possibilitando a comunicação eficiente e transparente.

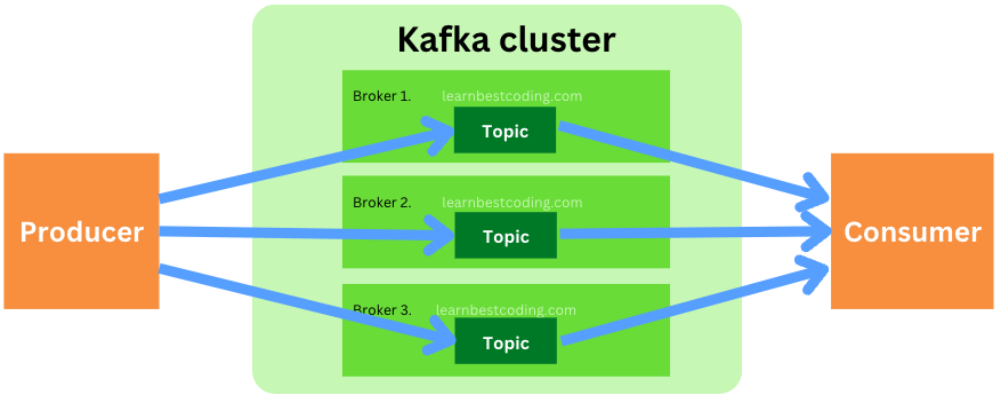


Figura 19 - Exemplo de sistema de troca de mensagens Kafka[11]

- **Sistema de *gateway*, Figura 20:** É responsável por fornecer uma *interface* unificada para os clientes externos acederem aos micro serviços em uma arquitetura de micro serviços. Atua como um ponto de entrada único, lidando com encaminhamento, autenticação, autorização e outras funcionalidades de gestão de APIs. Algumas das características do sistema de *gateway*:
  - **Exposição Controlada de Funcionalidades:** Permite que os micro serviços exponham as suas funcionalidades de forma controlada, fornecendo *endpoints* específicos para cada serviço. Simplifica a interação com os micro serviços, reduzindo a complexidade para os clientes externos e facilitando a evolução da arquitetura sem impactar diretamente os consumidores.
  - **Autenticação e Autorização Centralizada:** Pode centralizar a autenticação e autorização, permitindo que os micro serviços se concentrem nas regras de negócio específicas. O *gateway* pode autenticar os clientes, validar as permissões e controlar o acesso aos diferentes *endpoints* dos micro serviços.
  - **Encaminhamento e Balanceamento de Carga:** Lida com o encaminhamento dos pedidos dos clientes para os micro serviços correspondentes. Pode implementar regras de encaminhamento flexíveis, como basear-se no caminho do URL, cabeçalho ou parâmetros de solicitação. O *gateway* pode aplicar balanceamento de carga para distribuir os pedidos de forma equilibrada entre os micro serviços.

Exemplo prático: Considere um *gateway* que recebe pedidos de uma aplicação de comércio eletrónico. Autêntica o cliente, verifica as permissões e roteia os pedidos para

[11] Retirado de: https://www.linkedin.com/pulse/afinal-o-que-%C3%A9-apache-kafka-alex-jos%C3%A9-silva-msc--5gryf/?originalSubdomain=pt

o micro serviço apropriado. O *gateway* também pode aplicar políticas de limitações de taxa de pedidos para evitar sobrecarga nos micro serviços.

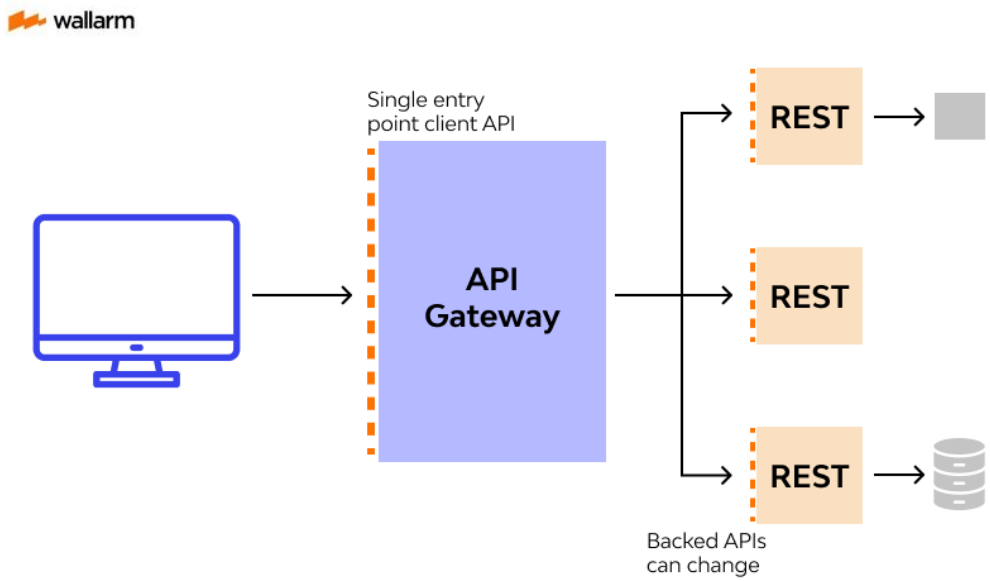


Figura 20 - Exemplo implementação de uma Gateway[12]

Comparando os sistemas de troca de mensagens e gateway com base em alguns aspetos chave:

- **Comunicação:** O sistema de troca de mensagens é virado para a comunicação assíncrona entre os micro serviços, enquanto o sistema de *gateway* lida com a comunicação síncrona entre clientes externos e os micro serviços.
- **Escopo:** O sistema de troca de mensagens abrange a comunicação interna entre os micro serviços, enquanto o sistema de *gateway* concentra-se na interface externa com os clientes.
- **Funcionalidades:** O sistema de troca de mensagens oferece recursos como filas, tópicos e garantia de fiabilidade. O Sistema de *gateway* fornece funcionalidades de encaminhamento, autenticação, autorização e controlo de tráfego.
- **Dependências:** Os micro serviços dependem do sistema de troca de mensagens para comunicação assíncrona, enquanto o sistema de *gateway* depende dos micro serviços para fornecer as funcionalidades expostas.
- **Utilização Combinada:** Em muitos casos, os sistemas de troca de mensagens e *gateway* são usados em conjunto, Figura 21. O sistema de troca de mensagens permite a comunicação entre os micro serviços, enquanto o

[12] Retirado de: https://www.wallarm.com/what/the-concept-of-an-api-gateway

sistema de *gateway* facilita a comunicação entre os micro serviços e os clientes externos.

Tanto o sistema de troca de mensagens quanto o sistema de *gateway* desempenham papéis cruciais na arquitetura de micro serviços. O sistema de troca de mensagens fornece uma forma eficiente de comunicação assíncrona entre os micro serviços, permitindo escalabilidade e flexibilidade. Por outro lado, o sistema *gateway* oferece uma interface unificada para clientes externos acederem aos micro serviços, garantindo controlo de acessos, segurança e encaminhamento adequado. Dependendo das necessidades do sistema, pode ser necessário implementar tanto o sistema de troca de mensagem quanto o sistema de *gateway* para alcançar uma arquitetura completa e funcional.

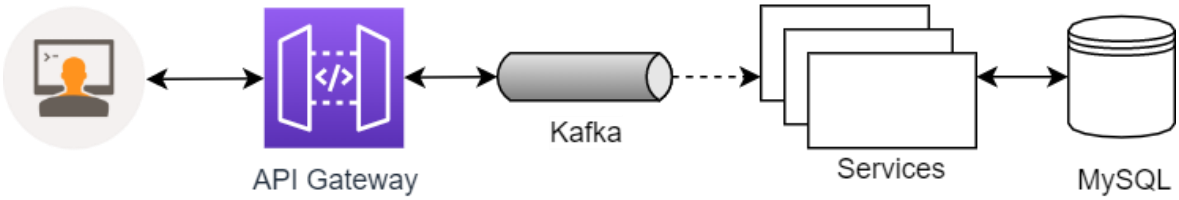


Figura 21 - Exemplo de combinação de gateway com troca de mensagens[13]

[13] Retirado de: https://arnoldgalovics.com/fault-tolerant-kafka-messaging

Capítulo 7 - TIPOS CACHE PARA MICRO SERVIÇOS

No desenvolvimento de aplicações Web, o uso de *cache* é uma prática comum para melhorar o desempenho e a eficiência. Dois tipos de *cache* amplamente utilizados são a memória *cache* e o Redis *cache*.

Embora ambos sejam usados para armazenar dados em memoria, a memoria *cache* e o Redis *cache* diferem em algumas características.

- **Estrutura de dados:** A memoria *cache* é geralmente implementada como um armazenamento simples de chave-valor, ou seja, uma chave guarda um valor e se essa chave se voltar a repetir num pedido será devolvido o mesmo valor, enquanto o Redis *cache* oferece uma ampla variedade de estruturas de dados, como listas, conjuntos, *hashes* e muito mais.
- **Recursos avançados:** O Redis *cache* fornece recursos avançados, como capacidade de executar operações atómicas e operações em lotes, além, de suporte a transações e pub/sub (publicação/subscrição) para comunicação entre aplicações.
- **Persistência:** O Redis *Cache* permite que os dados sejam armazenados em disco, garantindo a recuperação dos dados mesmo após uma reinicialização do servidor, enquanto a memória *cache* não oferece esse recurso.

Vantagens **memória cache**:

- Acesso extremamente rápido aos dados em memória, resultando melhor desempenho.
- Redução da carga nos recursos do servidor, como acesso a base de dados.
- Configuração flexível de políticas de expiração e invalidação dos dados em *cache*.

Desvantagens **memória cache**:

- Tamanho limitado pela capacidade da memoria RAM do servidor.
- A necessidade de mecanismos adicionais para lidar com a invalidação de *cache* e atualização de dados.

Exemplo prático: A *cache* de consultas de base de dados, onde os resultados de consultas frequentes são armazenados em *cache*, evitando a necessidade de consultar a base de dados repetidamente.

Vantagens **Redis *Cache***:

- Alta flexibilidade e suporte a diferentes tipos de dados e estruturas.
- Desempenho excecional em casos de uso complexos e operações avançadas.
- Recursos de persistência para garantir a recuperação dos dados em caso de falha do servidor.

Desvantagens **Redis *Cache***:

- Requere mais configuração e gestão em comparação com a memória da cache simples.
- Pode exigir mais recursos do servidor, especialmente em cenários de grande escala.

Exemplo prático: O armazenamento em cache de sessões do utilizador, onde os dados da sessão são armazenados no Redis Cache para permitir a recuperação rápida e o partilhar entre várias instâncias de aplicações.

Tanto a memória cache quanto o Redis Cache são ferramentas poderosas para melhorar o desempenho e a eficiência das aplicações Web. A escolha depende das necessidades específicas do projeto e dos recursos.

Capítulo 8 - LIVRARIAS E FERRAMENTAS UTILIZADAS EM MICRO SERVIÇOS

8.1 - DOCUMENTAÇÃO API SWAGGER

No desenvolvimento de aplicações web, a documentação e testes eficazes de APIs são fundamentais para facilitar a compreensão e a utilização corretas dessas *interfaces*. O Swagger é uma ferramenta amplamente adotada para documentar, testar e visualizar APIs em aplicações web [20].

- **O que é o Swagger?**

  O Swagger é um conjunto de ferramentas *open-source* que auxilia no desenvolvimento de APIs *RESTful*, disponibilizando recursos abrangentes de documentação e testes. Permite que os programadores e utilizadores finais visualizem e entendam melhor como interagir com as APIs, fornecendo uma documentação detalhada e uma *interface* interativa para testar os pedidos.

- **Funcionalidades do Swagger:**
  - **Documentação automatizada:** Permite que os programadores gerem automaticamente a documentação da API a partir do código fonte, fornecendo informações detalhadas sobre os *endpoints*, parâmetros, modelo de dados, métodos suportados e exemplos de utilização.
  - ***Interface* interativa:** O SwaggerUI, Figura 22, uma parte integrante do Swagger, fornece uma *interface* web amigável que permite aos utilizadores explorar e testar a API de forma interativa. Oferece um ambiente intuitivo para enviar pedidos, visualizar respostas e experimentar diferentes cenários.
  - **Validação de entrada:** Pode validar automaticamente os parâmetros de entrada dos pedidos, garantindo que sejam do tipo correto e atendam às restrições definidas.
  - **Geração de SDKs:** Com base na documentação fornecida pelo Swagger, é possível gerar automaticamente SDKs (*Software Development Kits*) em várias linguagens de programação, simplificando a integração e a utilização da API em diferentes plataformas.

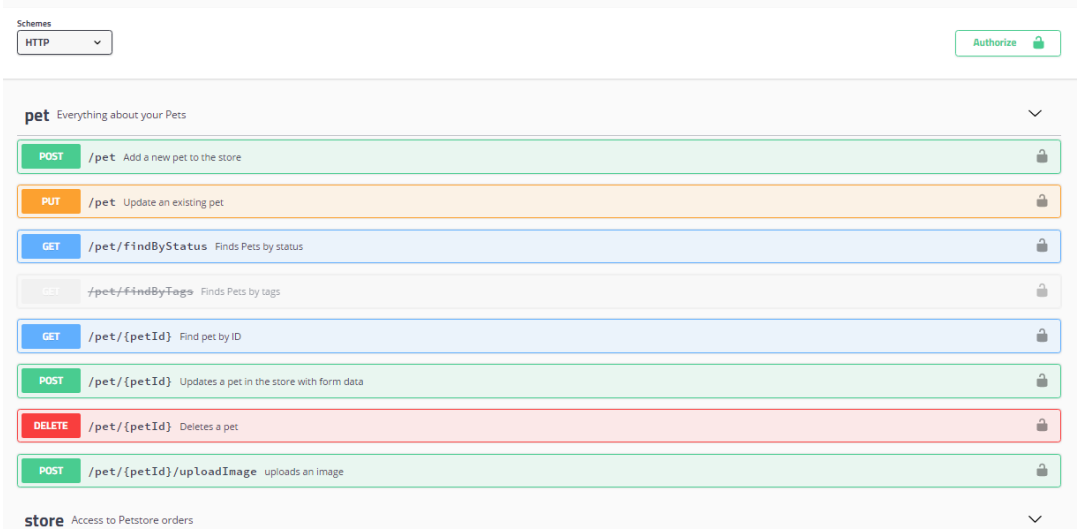


Figura 22 - Exemplo de janela do Swagger UI[14]

Vantagens:

- **Documentação precisa e atualizada:** Simplifica a tarefa de manter a documentação da API sincronizada com o código fonte, garantindo que as informações estejam sempre precisas e atualizadas.
- **Facilidade de utilização para programadores e utilizadores finais:** A *interface* interativa do Swagger UI permite que os programadores testem facilmente as APIs, entendam a estrutura dos pedidos e visualizem as respostas de forma clara e organizada.
- **Aceleração do desenvolvimento:** Com a geração automatizada da documentação e SDKs, o Swagger reduz o tempo necessário para criar e integrar APIs, agilizando o processo de desenvolvimento.

Desvantagens:

- **Curva de aprendizagem inicial:** Embora o Swagger seja amplamente adotado e bem documentado, pode haver uma curva de aprendizagem inicial para entender completamente todas as funcionalidades e opções disponíveis.
- **Dependência de integração continua:** Para aproveitar ao máximo o Swagger, é necessário integrá-lo a um processo de integração continua, o que pode exigir ajustes e configurações adicionais.

Exemplo prático de implementação do Swagger em uma API *Restful*: Vamos considerar um exemplo prático de implementação do Swagger em uma API *RESTful* para uma aplicação de gestão de tarefas. Usando o Swagger, podemos gerar automaticamente a documentação da API, descrevendo os *endpoints* disponíveis, os

[14] Retirado de: https://swagger.io/tools/swagger-ui/

parâmetros esperados, os códigos de status das respostas e exemplos de requisições e respostas.

Além disso, podemos utilizar o Swagger UI para fornecer uma *interface* interativa para testar a API. Os programadores e utilizadores finais podem enviar pedidos para criar, atualizar ou apagar tarefas, visualizar listas de tarefas ou pesquisar tarefas especificas. A *interface* do Swager UI facilita o envio dos pedidos, exibe as respostas correspondentes e permite a exploração completa da API.

O Swagger é uma ferramenta poderosa para a documentação e testes de APIs em aplicações web. Com capacidade para gerar documentação automatizada, oferecer uma *interface* interativa e simplificar a integração com diferentes plataformas, desempenha um papel crucial no desenvolvimento de aplicações baseadas em APIs. Embora tenha algumas desvantagens, as vantagens do Swagger superam amplamente os obstáculos, tornando-o uma escolha popular entre os programadores.

8.2 - ORM – OBJECT RELATIONAL MAPPER

É uma técnica de programação que permite mapear objetos num sistema orientado a objetos para tabelas numa base de dados relacional. Em outras palavras, um ORM facilita a interação entre o código da aplicação e a base de dados, abstraindo a complexidade das consultas SQL e fornecendo uma *interface* orientada a objetos para manipular os dados armazenados [21].

Um ORM, Figura 23, serve para simplificar o desenvolvimento de aplicações que utilizam base de dados relacionais. Permite que os programadores trabalhem com objetos e classes no código da aplicação, enquanto o ORM cuida da tradução e persistência desses objetos na base de dados. Isto elimina a necessidade de escrever consultas SQL manualmente e facilita a manutenção e o desenvolvimento ágil da aplicação.

Apesar das desvantagens potenciais, as vantagens do uso de um ORM, como produtividade, portabilidade e facilidade de manutenção, tornam-no uma escolha popular entre os programadores. Com a escolha certa de um ORM e um bom entendimento do funcionamento, é possível aproveitar ao máximo essa abordagem e acelerar o desenvolvimento de aplicações orientadas a dados.

Vantagens:

- **Abstração da base de dados:** Abstrai as complexidades da base de dados, fornecendo uma *interface* de programação consistente e orientada a objetos para interagir com os dados armazenados.
- **Produtividade e agilidade:** O programador pode concentrar-se na lógica de negócio em vez de escrever consultas SQL complexas. Isto permite obter um

desenvolvimento mais rápido e eficiente, aumentando a produtividade da equipa.

- **Portabilidade:** Pode fornecer uma camada de abstração entre a aplicação e a base de dados, permitindo que a aplicação seja suportada para diferentes sistemas de gestão de base de dados sem a necessidade de alterar significativamente o código.
- **Manutenção simplificada:** Facilita a manutenção de código da aplicação, pois as alterações no mapeamento objeto-relacional podem ser tratadas pelo ORM, evitando a necessidade de modificar consultas SQL em várias partes do código.

Desvantagens:

- **Curva de aprendizagem:** Alguns ORM podem ter uma curva de aprendizagem íngreme, especialmente para programadores inexperientes. Compreender os conceitos e as funcionalidades do ORM pode levar tempo.
- **Desempenho:** Nalguns casos, o uso de um ORM pode resultar em consultas SQL ineficientes ou complexas, afetando o desempenho da aplicação. Otimizações manuais podem ser necessárias para melhorar o desempenho em determinadas situações.
- **Limitações do mapeamento objeto-relacional:** O mapeamento entre objetos e tabelas da base de dados nem sempre é direto. Algumas situações complexas podem exigir soluções personalizadas ou consultas SQL Diretas.

Exemplo práticos de utilização de um ORM:

- **EntityFrameworkCore:** É um ORM amplamente utilizado para aplicações .NET, permitindo o mapeamento de classes .Net para tabelas de bases de dados relacionais.
- **Dapper (Micro ORM):** É uma biblioteca de mapeamento de objeto-relacional extremamente rápida e simples para .Net

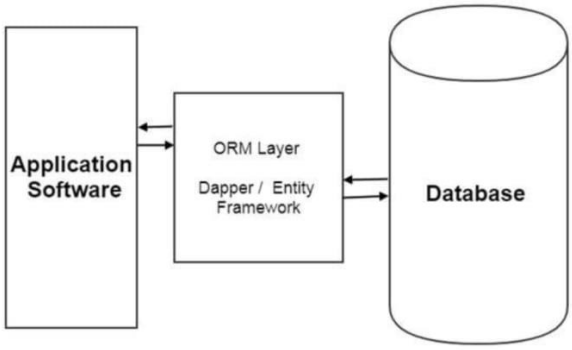


Figura 23 – Posicionamento do ORM[15]

[15] Retirado de: https://dev.to/shishsingh/the-case-study-dapper-vs-entity-framework-core-dj8

8.2.1 - MICROSOFT ENTITYFRAMEWORKCORE

No desenvolvimento de aplicações .NET, o EntityFrameworkCore [22] é uma ferramenta poderosas que simplifica a interação com base de dados relacionais por meio do mapeamento objeto-relacional, Figura 24.

A EntityFrameworkCore é uma versão moderna, flexível e multiplataforma uma atualização da anterior EntityFramework ORM (*Object-Relational Mapping*) é popular para os programadores de aplicações .Net. Permite que os programadores desenvolvam objetos e classes no código .NET, enquanto a EntityFrameworkCore preocupa-se com o mapeamento e persistência desses objetos na base de dados relacional. Funcionalidades:

- **Mapeamento objeto-relacional:** Permite mapear classes e propriedades em .NET para tabelas e colunas na base de dados relacional. Suporta diferentes estratégias de mapeamento, como mapeamento por convenção, atributos e *Fluent API*, esta última permite uma personalização avançada do mapeamento.
- **Linguagem de consulta LINQ:** Oferece suporte completo à linguagem de consulta LINQ (*Language Integrated Query*), permitindo que os programadores realizem consultas poderosas e expressivas na base de dados, utilizando sintaxe familiar e fortemente tipificada.
- **Migrações de base dados:** Facilita a migração do esquema de base de dados por meio das chamadas "*migrations*". As migrações permitem atualizar automaticamente a estrutura da base de dados à medida que o modelo de objetos evolui, garantindo a consistência entre a estrutura da base de dados e o código da aplicação.
- **Rastreamento das alterações:** Possui um recurso chamado "*Change Tracking*" que rastreia automaticamente as alterações feitas em objetos carregados na base de dados. Permite que as alterações sejam detetadas e persistidas na base de dados de forma eficiente.
- ***Cache* de consultas:** Possui um mecanismo de *cache* de consultas que armazena em memória os resultados das consultas frequentes, melhorando o desempenho da aplicação ao evitar idas desnecessárias à base de dados.

Vantagens:

- **Produtividade:** Permite que os programadores trabalhem com objetos e consultas num ambiente orientado a objetos, tornando o desenvolvimento mais produtivos e reduzindo a quantidade de código SQL manual necessário.
- **Portabilidade:** É multiplataforma, ou seja, pode ser executado em diferentes sistemas operativos e bases de dados e oferece maior portabilidade da aplicação.

- **Manutenção simplificada:** As alterações no modelo de objetos podem ser facilmente refletidas na base de dados por meio das migrações, simplificando a manutenção da estrutura da base de dados.
- **Segurança:** Ajuda a prevenir ataques de injeção de SQL, uma vez que as consultas são construídas internamente e os parâmetros são automaticamente tratados.

Desvantagens:

- **Curva de aprendizagem:** Pode exigir um tempo de aprendizagem inicial para entender os conceitos, configurações e recursos avançados, especialmente para programadores juniores.
- **Desempenho:** Em algumas situações, especialmente em consultas mais complexas ou operações em massa, pode apresentar um desempenho inferior em comparação com consultas SQL escritas manualmente. Nestes casos, será necessárias otimizações adicionais.

Exemplos práticos:

- Mapeamento de classes para tabelas de base de dados.
- Execução de consultas LINQ para recuperar dados específicos da base de dados.
- Utilização de migrações para atualizar automaticamente a estrutura da base de dados.
- Utilização de recurso de rastreamento das alterações para persistir alterações na base de dados.

A EntityFrameworkCore é uma ferramenta essencial no desenvolvimento de aplicações .NET, oferecendo uma abordagem orientada a objetos para interagir com base de dados relacionais. Tendo funcionalidades poderosas, como o mapeamento objeto-relacional, suporta as consultas LINQ, migrações de base de dados e *cache* para consultas. A EntityFrameworkCore melhora a produtividade do programador e simplifica a manutenção da aplicação. Embora possa ter uma curva de aprendizagem inicial e exigir otimizações em casos específicos de desempenho, os benefícios gerais superam amplamente as desvantagens.

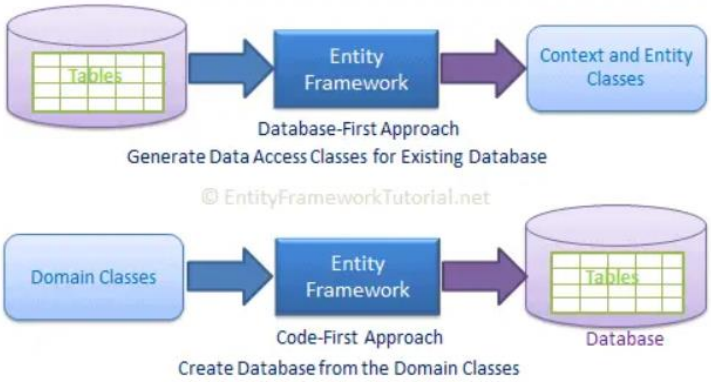


Figura 24 – EntityFrameworkCore DataBase-First vs Code-First[16]

8.2.2 - DAPPER

No desenvolvimento de aplicações .NET, o Dapper é uma biblioteca popular que oferece mapeamento objeto-relacional de forma simples e rápida.

É uma biblioteca de mapeamento objeto-relacional para .Net que permite executar consultas SQL e mapear os resultados diretamente para objetos .NET. Fornece uma camada fina de abstração sobre o ADO.NET (*ActiveX Data Objects*) padrão, oferece uma solução simples e de alto desempenho para interagir com base de dados relacionais. Funcionalidades:

- **Mapeamento simples:** Permite escrever consultas SQL e mapeá-las diretamente os resultados para objetos .NET. O mapeamento é realizado automaticamente, atribuindo os valores das colunas da base de dados às propriedades dos objetos.
- **Desempenho otimizado:** É conhecido pela sua performance superior, sendo extremamente rápido em comparação com outros ORMs. Possui um mecanismo de mapeamento eficiente que minimiza a sobrecarga e a complexidade das operações da base de dados.
- **Consultas personalizadas:** Oferece suporte a consultas personalizadas, permitindo a execução de consultas complexas e otimizadas, aproveitando ao máximo a flexibilidade do SQL.
- **Controle direto sobre consultas SQL:** Permite escrever as próprias consultas SQL quando necessário, oferecendo controle total sobre a lógica de acesso à base de dados.

Vantagens:

- **Desempenho superior:** Devido à abordagem leve e otimizada, oferece um desempenho superior em comparação com outros ORM's.

[16] Retirado de: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx

Especialmente vantajoso em cenários que exigem operações em massa ou consultas complexas.

- **Simplicidade:** É conhecido pela simplicidade e facilidade de uso. A API é mínima e intuitiva, exigindo menos configurações e tempo de aprendizagem em comparação com outros ORMs mais complexos.
- **Controlo granular:** Oferece um alto nível de controlo sobre as consultas, permitindo que os programadores otimizem consultas e lidem com casos especiais de maneira direta.
- **Suporte ao ADO.NET:** É construído em cima do *ADO.NET* padrão, o que significa que se pode aproveitar a funcionalidade existente do ADO.NET, como transações e parâmetros, quando necessário.

Desvantagens:

- **Responsabilidade manual do programador:** Como o Dapper permite escrita das próprias consultas SQL, existe uma maior responsabilidade por parte do programador para escrever consultas otimizadas e lidar com questões de segurança.
- **Mapeamento manual:** Embora possua um mecanismo de mapeamento automático, em alguns casos pode ser necessário um mapeamento manual para cenários complexos ou consultas personalizadas.

O Dapper é uma biblioteca de mapeamento objeto-relacional simples e de alto desempenho para aplicações .NET. Tendo uma abordagem leve, controlo granular e performance superior, o Dapper oferece uma solução eficiente para interagir com base de dados relacionais. Embora exija um pouco mais de responsabilidade do programador em relação à escrita de consultas SQL e mapeamento manual, as vantagens em termos de desempenho e simplicidade fazem do Dapper uma escolha popular entre os programadores .NET.

Capítulo 9 - ESTUDO E ANÁLISE DAS METODOLOGIAS DE GESTÃO DE PROJETO

As metodologias de desenvolvimento de software são abordagens estruturadas para planear, executar e controlar o processo de desenvolvimento de software. São cruciais para garantir a entrega eficiente e de qualidade do software, facilitando a colaboração entre as equipas de desenvolvimento e a comunicação com os clientes. Diversas metodologias foram desenvolvidas ao longo do tempo, cada uma com as suas próprias vantagens e desvantagens [23].

9.1 - METODOLOGIA CASCATA

A metodologia cascata, Figura 25, é uma abordagem tradicional e sequencial para o desenvolvimento do software, que divide o ciclo de vida do projeto em fases bem definidas, cada uma depende do resultado da fase anterior. Esta metodologia segue uma progressão linear, onde o progresso flui de cima para baixo, assim como a água numa cascata, daí o nome.

Vantagens:

- **Estrutura clara:** Oferece uma estrutura bem definida, com cada fase claramente delineada, desde a análise de requisitos até à implementação, testes e manutenção. Isto torna o processo fácil de entender e seguir.
- **Facilidade de gestão:** Como as fases são distintas e sequenciais, é mais fácil para os gestores de projeto acompanhar o progresso do mesmo e identificar potenciais problemas em fases iniciais.
- **Documentação abrangente:** Cada fase geralmente requer documentação detalhada, incluindo especificações de requisitos, *design* do sistema, planos de testes, entre outros. Resulta numa documentação mais abrangente, que pode ser útil para a referência futura e para garantir a consistência do produto final.
- **Controlo de mudanças:** Como as mudanças nos requisitos são geralmente mais difíceis de implementar em fases avançadas do projeto, a metodologia em cascata ajuda a controlar as mudanças, garantindo que os requisitos sejam bem definidos antes do início do desenvolvimento.

Desvantagens:

- **Adaptação limitada a mudanças:** Uma das maiores desvantagens desta metodologia é a rigidez em lidar com mudanças de requisitos durante o ciclo de vida do projeto. Uma vez que uma fase está concluída, é difícil retroceder e fazer alterações sem impactar severamente o cronograma e o orçamento.

- ***Feedback* tardio do cliente:** Como esta metodologia envolve a entrega do produto final apenas no final do ciclo de desenvolvimento, o cliente pode não fornecer *feedback* significativo até que seja tarde para fazer mudanças significativas sem grandes custos.
- **Risco de subestimação de requisitos:** Devido à natureza sequencial, é comum que os requisitos não sejam completamente compreendidos ou subestimados no início do projeto, o que pode levar a atrasos e custos adicionais à medida que são descobertos problemas mais tarde no ciclo de desenvolvimento.
- **Falta de flexibilidade:** Não é adequada para projetos onde os requisitos estão sujeitos a mudanças frequentes ou para aqueles que exigem uma abordagem iterativa e incremental. A falta de flexibilidade pode resultar em produtos que não atendem totalmente às necessidades do cliente.

Esta metodologia já foi amplamente utilizada no passado e ainda é aplicável a certos tipos de projetos, mas as limitações em lidar com mudanças de requisitos e falta de flexibilidade tornaram-na menos adequada para muitos projetos de desenvolvimento de software mais modernos.

L

Figura 25 – Metodologia Cascata[17]

9.2 - METODOLOGIA AGILE

É uma abordagem iterativa e incremental para o desenvolvimento de software, Figura 26. Centra-se na entrega contínua ao cliente através *sprints* curtos e interativos.

[17] Retirado de: https://metodologiasclassicas.blogspot.com/p/modelo-em-cascata.html

Realça a flexibilidade, a colaboração e a capacidade de resposta a mudanças durante o ciclo de vida do projeto.

Vantagens:

- **Flexibilidade:** Uma das principais vantagens é a capacidade de adaptação à mudança nos requisitos do cliente. Os projetos são divididos em *sprints* curtos, geralmente de uma ou duas semanas, o que permite ajustes rápidos e contínuos com base no *feedback* do cliente.
- **Entregas continuas:** A prioridade está na entrega contínua das funcionalidades do software. Isto significa que o cliente começa a ver e a usar as funcionalidades à medida que vão sendo desenvolvidas. O que permite uma validação mais rápida dos requisitos e uma maior satisfação do cliente.
- **Maior envolvimento do cliente:** Promove a colaboração contínua entre o consultor da equipa de desenvolvimento e o cliente durante todo o ciclo de vida do projeto. O cliente é incentivado a fornecer *feedback* constantemente e a participar ativamente no processo de desenvolvimento, garantindo que o produto final vá de encontro às suas espectativas.
- **Maior qualidade do produto:** Os testes às funcionalidades são integrados, desde o início do desenvolvimento. Ajuda a identificar e corrigir problemas mais cedo, resultando num produto final de maior qualidade e menor custo de suporte.

Desvantagens:

- **Requer alto ambiente colaborativo:** Exige uma equipa altamente colaborativa e auto-organizada. É um desafio numa cultura organizacional ou onde as personalidades individuais de cada elemento da equipa não favorecem o trabalho em grupo e a autogestão.
- **Documentação menos abrangente:** Em comparação com metodologias tradicionais, pode resultar em menos documentação formal, especialmente em projetos mais pequenos, sem grande relevância.
- **Possível resistência à mudança:** Poderá haver resistência por parte de alguns membros da equipa ou partes interessadas acostumadas a abordagens tradicionais. A mudança pode exigir tempo e esforço para educar e convencer sobre os benefícios desta abordagem.
- **Gestão de espectativas do cliente:** Mesmo com entregas contínuas e incrementais das funcionalidades do software seja uma vantagem desta metodologia, também pode levar as espectativas elevadas por parte do cliente. Conseguir gerir as espectativas do cliente e garantir a compreensão clara dos objetivos do projeto é essencial para o sucesso do mesmo.

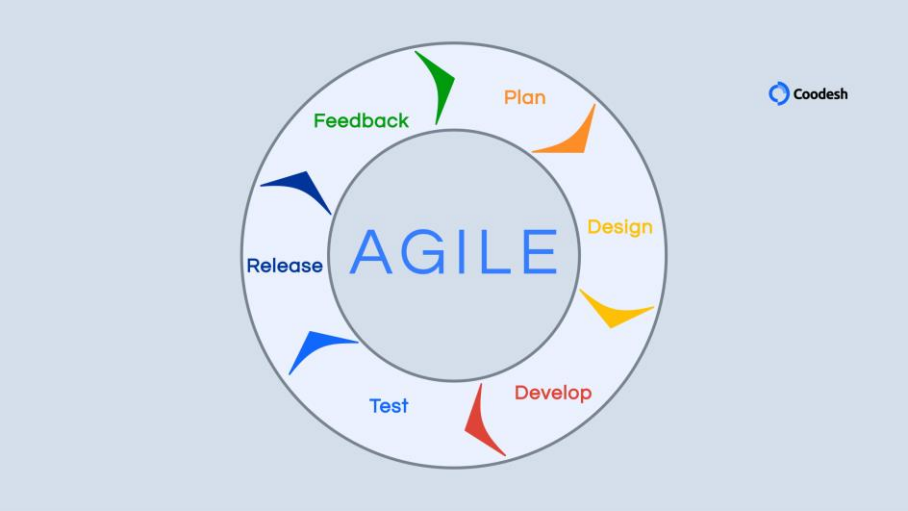


Figura 26 – Metodologia Agile[18]

9.3 - METODOLOGIA SCRUM

Uma das mais utilizadas no desenvolvimento de software, baseando-se no princípio da transparência e adaptação para a gestão de projetos de forma eficaz, Figura 27. Envolve ciclos de trabalho curtos, chamados de *sprints*, e promove a colaboração e a comunicação entre os membros da equipa.

Vantagens:

- **Estrutura clara e papéis definidos:** Oferece uma estrutura clara, com os papéis bem definidos para cada membro da equipa. Ajudando a clarificar as responsabilidades e a promover a colaboração eficaz.
- **Entregas frequentes e feedback rápido:** Com *sprints* curtos, geralmente de 1 a 2 semanas, promove entregas contínuas de funcionalidades do software. Permite um *feedback* rápido do cliente e a capacidade de ajustar os requisitos do projeto conforme o necessário.
- **Concentração no valor:** Concentra-se na entrega de valor ao cliente através das prioridades de cada funcionalidade com base no valor entendido. O gestor do lado do cliente é responsável por definir e priorizar o *backlog* do produto, garantindo que as funcionalidades mais importantes são implementadas primeiro.
- **Adaptação contínua:** Promove uma abordagem adaptativa ao desenvolvimento do software, permitindo que a equipa se adapte a mudanças nos requisitos. Através das reuniões diárias (*Daily Point*), normalmente de 15 minutos ao início da manhã e revisão do *sprint*,

[18] Retirado de: https://coodesh.com/blog/candidates/metodologias/metodologia-agile-guia-com-caracteristicas-beneficios-e-como-aplicar/

normalmente no último dia à tarde, a equipa pode identificar problemas e ajustar o planeamento do projeto conforme as necessidades.

Desvantagens:

- **Dificuldades em escalar:** Pode ser mais difícil de escalar para projetos maiores e mais complexos, quando envolvem várias equipas e requerem integração com várias áreas organizacionais. Pode ser um desafio, gerir dependências entre equipas e manter a coesão.
- **Compromisso total da equipa:** Depende do compromisso total da equipa com os princípios e práticas desta metodologia.

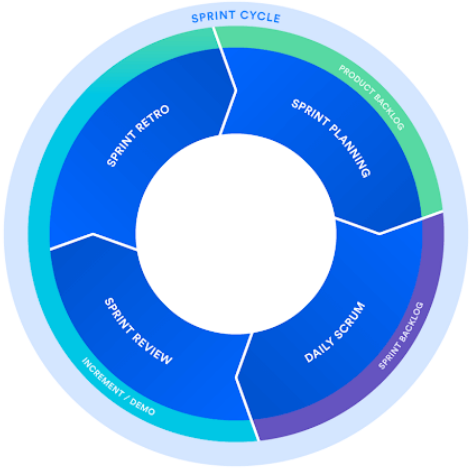


Figura 27 – Metodologia Scrum[19]

9.4 - METODOLOGIA KANBAN

Concentra-se na visualização do fluxo do trabalho e no progresso do trabalho, Figura 28. Tendo origem na indústria automóvel japonesa, esta abordagem tem sido cada vez mais adotada no mundo do desenvolvimento de software, devido à sua simplicidade e eficácia.

Vantagens:

- **Centrada na visualização do fluxo do trabalho:** Uma das principais vantagens é o grande foco na visualização do progresso do trabalho. Um quadro de Kanban, geralmente é composto por várias colunas, representando as diferentes etapas do progresso. Permite que a equipa

[19] Retirado de: https://www.atlassian.com/br/agile/scrum

visualize facilmente o estado atual das tarefas e identifique potenciais "estrangulamentos" ou áreas de melhorias.

- **Limitação do trabalho em progresso:** Promove a limitação de trabalho em progresso, o que significa que apenas um número definido de tarefas pode estar em andamento ao mesmo tempo. Evita sobrecargas da equipa e mantém o fluxo de trabalho mais consistente e previsível

Figura 28 – Metodologia Kanban[20]

9.5 - METODOLOGIAS UTILIZADAS

A metodologia de projeto desempenha um papel fundamental na implementação de soluções eficazes. Para a reengenharia desta aplicação, foram adotadas duas metodologias principais: *Scrum* e *Kanban*. Esta abordagens proporcionam um quadro estruturado e ágil para orientar o desenvolvimento das funcionalidades e a gestão do projeto como um todo.

A escolha do *Scrum* foi motivada pela capacidade de proporcionar um ritmo controlado e iterativo no processo de desenvolvimento. Com os *sprints* semanais, o Scrum permite um acompanhamento detalhado do progresso, facilitando o controlo das atividades e a identificação de eventuais desafios. Os pontos de situação ao início e ao final de cada sprint forneceram oportunidades para revisão e ajuste da estratégia, garantindo uma adaptação continua às necessidades do projeto.

Como complemento do *Scrum*, optou-se por integrar o *Kanban*, reconhecendo a sua eficácia na visualização do progresso dos trabalhos. Através das Dashboards intuitivas, o *Kanban* oferece uma representação visual clara do fluxo das tarefas e do estado atual do *sprint*. Esta visualização dinâmica permite uma compreensão rápida e precisa do

[20] Retirado de: https://www.accept.pt/kanban-gestao-tarefas/

estado do projeto, facilitando a identificação de possíveis problemas e a tomada de decisões para otimizar o desempenho.

Ao utilizar as duas metodologias em conjunto, é possível aproveitar o melhor de cada uma das abordagens, combinando a estrutura e a disciplina do *Scrum* com a transparência e a visualização do *Kanban*. Esta sinergia proporciona um ambiente de trabalho ágil e eficiente, que impulsiona o progresso do projeto e contribui para o sucesso dos objetivos.

A adoção das metodologias de *Scrum* e *Kanban* é essencial para o sucesso, fornece uma estrutura sólida e flexível para a gestão do projeto e garante um desenvolvimento eficaz das funcionalidades propostas.

Capítulo 10 - FERRAMENTA DE GESTÃO DE PROJETO - DEVOPS

No ecossistema do desenvolvimento de software, o Azure DevOps da Microsoft, Figura 29, desempenha um papel fundamental ao fornecer uma plataforma integrada que abrange todo o ciclo de vida do desenvolvimento de software, desde a gestão dos requisitos até à entrega contínua ao cliente.

Figura 29 – Azure Devops[21]

- ***Azure Boards*:** Permite a gestão de projetos ágeis com funcionalidades para criar e acompanhar *backlog* de produto, planear *sprints*, atribuir tarefas e gerir fluxos de trabalho personalizados. Também integra funcionalidades de controlo de versões, permitindo que as equipas trabalhem de forma colaborativa no código fonte.
- ***Azure Repos*:** Oferece repositórios GIT privados e ilimitados para armazenar o código fonte, com suporte para colaboração, revisão de código e rastreabilidade de alterações.
- ***Azure Pipelines:*** Permite a automatização das compilações, testes e implementação continua para aplicações em várias plataformas, incluindo Web, movel e desktop. Suporta integração com várias linguagens de programação e ambientes de desenvolvimento, proporcionando flexibilidade para criar pipelines de CI/CD personalizados.
- ***Azure Test Plans:*** Permite o planeamento, execução e acompanhamento de testes manuais e automáticos. Tem funcionalidades para criar casos de teste, gerir *suites* de teste, executar testes em vários ambientes e monitorizar os resultados.

[21] Retirado de: Azure DevOps – a evolução do VSTS (Visual Studio Team Services) – Digital Strategy and IT Innovation (leonardo-matsumota.com)

- **Azure Artifacts:** Oferece funcionalidades abrangentes para criar, hospedar e partilhar livrarias. Integra diretamente com o pipeline e os testes.

Capítulo 11 - VIABILIDADE DE REENGENHERIA

A migração de uma aplicação .NET 4.5 com 14 anos de existência, especialmente com muito código repetido e desorganizado, para uma arquitetura de micro serviços é um empreendimento complexo que requer uma análise cuidadosa de viabilidade [24]. Os principais aspetos dessa migração:

- **Escalabilidade melhorada:** Os micro serviços permitem escalabilidade granular, o que significa que se pode dimensionar apenas os componentes que estão a enfrentar altos pedidos, em vez de escalar toda a aplicação.
- **Manutenção Simplificada:** Cada micro serviço é uma entidade independente, facilitando a manutenção e a atualização de partes especificas da aplicação sem afetar as restantes.
- **Desenvolvimento Ágil:** As equipas de desenvolvimento podem trabalhar de forma independente em micro serviços distintos, acelerando o desenvolvimento e permitindo atualizações continuas.
- **Resiliência:** Se um micro serviço falhar, isso não necessariamente afetará toda a aplicação, tornando-a mais resiliente.
- **Desafios e considerações:**
  - **Custo:** A migração para micro serviços geralmente envolve custos significativos. É necessário reescrever partes substanciais da aplicação e investir em infraestrutura adicional.
  - **Tempo:** O tempo necessário para migrar dependerá do tamanho e da complexidade da aplicação. Projetos de migração podem levar meses ou até anos.
  - **Complexidade:** O código repetido e desorganizado da aplicação existente pode aumentar significativamente a complexidade da migração.
  - **Risco:** A migração de uma aplicação monolítica para micro serviços envolve riscos, como a possibilidade de introduzir novos erros e interrupções nos serviços existentes.
  - **Estratégia de migração:** É necessário definir uma estratégia sólida de migração, incluindo quais as partes da aplicação a migrar primeiro, como lidar com a integração entre micro serviços e como garantir a continuidade do serviço durante a transição.
  - **Testes e garantia de qualidade:** É crucial realizar testes extensivos para garantir que os micro serviços funcionem conforme o esperado. Isto pode consumir tempo significativo.
  - **Viabilidade:** A viabilidade da migração depende de vários fatores, incluindo o orçamento disponível, o tempo que se pode dedicar ao projeto, a importância da escalabilidade e da manutenção simplificada

para o negócio e a capacidade da equipa técnica. Antes de prosseguir, deve-se ter em atenção algumas etapas a seguir:

- **Avaliação Detalhada:** Realizar uma avaliação detalhada da aplicação existente para identificar áreas críticas, código repetido e desorganização.
- **Estimativa de Custo e Tempo:** Estimativas de custo e tempo de consultores de migração de micro serviços ou da equipa de desenvolvimento interna.
- **Priorização:** Identificar quais as partes da aplicação são candidatas ideais para migração inicial. Concentrar-se em áreas que trarão os maiores benefícios.
- **Plano de migração:** Desenvolver um plano de migração sólido, incluindo estratégias de teste e mitigação de riscos.
- **Piloto:** Realizar um projeto piloto de migração menor para avaliar a viabilidade em um ambiente controlado antes de migrar a aplicação inteira.
- **Revisão executiva:** Analisar a viabilidade e o plano de migração à liderança da empresa para obter aprovação e apoio financeiro.

Deve-se sempre ter em mente que a migração para micro serviços é um projeto de longo prazo e requer compromisso e recursos substanciais. A decisão deve ser baseada em uma avaliação completa dos benefícios e custos envolvidos, bem como na estratégia de negócios [25].

Capítulo 12 - PROPOSTA NOVA APLICAÇÃO WEB CLOUD

No modelo que se pode visualizar na Figura 30, é apresentada uma estrutura baseada em micro serviços, *Gateway* e *FrontEnd,* para a nova *Web Cloud*.

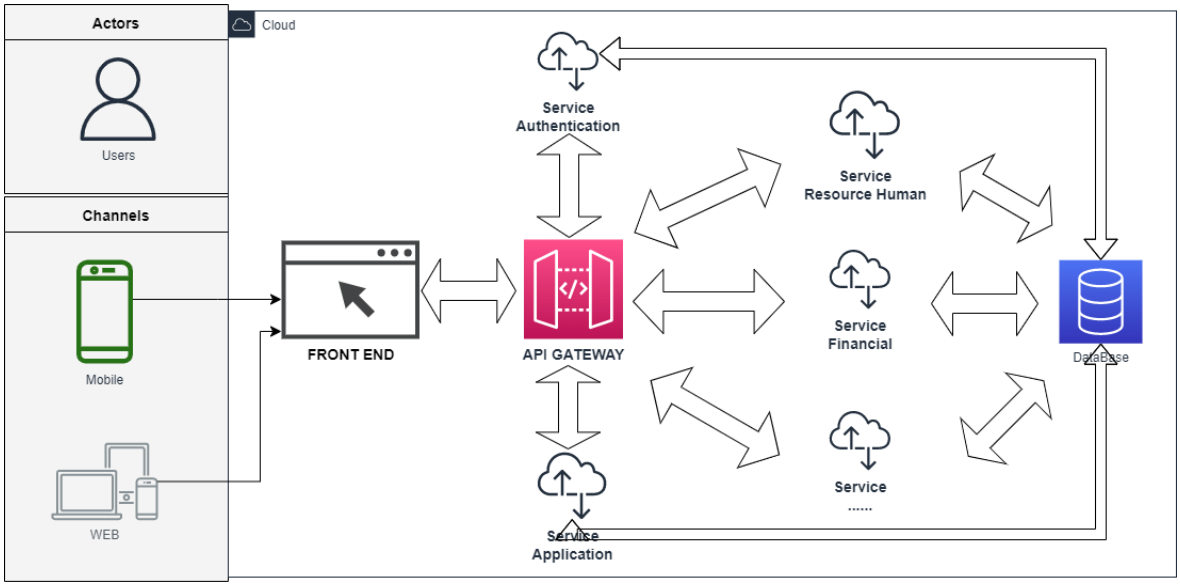


Figura 30 - Modelo do novo Web *Cloud*

O projeto de *FrontEnd* terá toda a parte visual de interação com o utilizador e será responsável por todo o design gráfico e pedidos.

De modo a centralizar todos os pedidos e a encapsular todos os endereços reais dos micro serviços, pretende-se criar um projeto do tipo API que, recorrendo ao *NuGet Ocelot*, fará essa ponte e será a primeira camada de *BackEnd*. Este *Gatway* permite que os endereços dos micro serviços sejam alterados sem que o utilizador final tenha essa perceção. Deste modo, a necessidade de expandir o sistema ou de o encolher torna-se bastante fácil, sendo este um sistema bastante elástico, que não necessita de alteração da capacitação do hardware e que só depende da ampliação da infraestrutura virtual de software.

Sabendo que a comunicação entre micro serviços é a mais complicada de gerir e que se deve respeitar ao máximo o que o próprio nome indica de “micro” (o mais pequeno possível) decidiu-se dividir os micro serviços em módulos, o que permite obter uma melhor manutenção e cada micro serviço ter apenas as suas responsabilidades, (para cada módulo um micro serviço). Isto permite que, tirando partido do padrão de repositório, cada micro serviço faça a gestão das próprias entidades, numa tentativa de cada um resolver internamente as suas responsabilidades, sem ter de comunicar com os outros, apenas com o *Gateway*.

Cada micro serviço comunica com a base de dados e tem a responsabilidade de toda a gestão das tabelas que lhe pertence, desde a criação, alteração e remoção das tabelas.

Para implementar a estrutura apresentada, iremos começar por definir as camadas da *Clean Architecture*.

Na figura 31 identificam-se claramente todos os projetos a criar dentro da solução e as suas dependências.

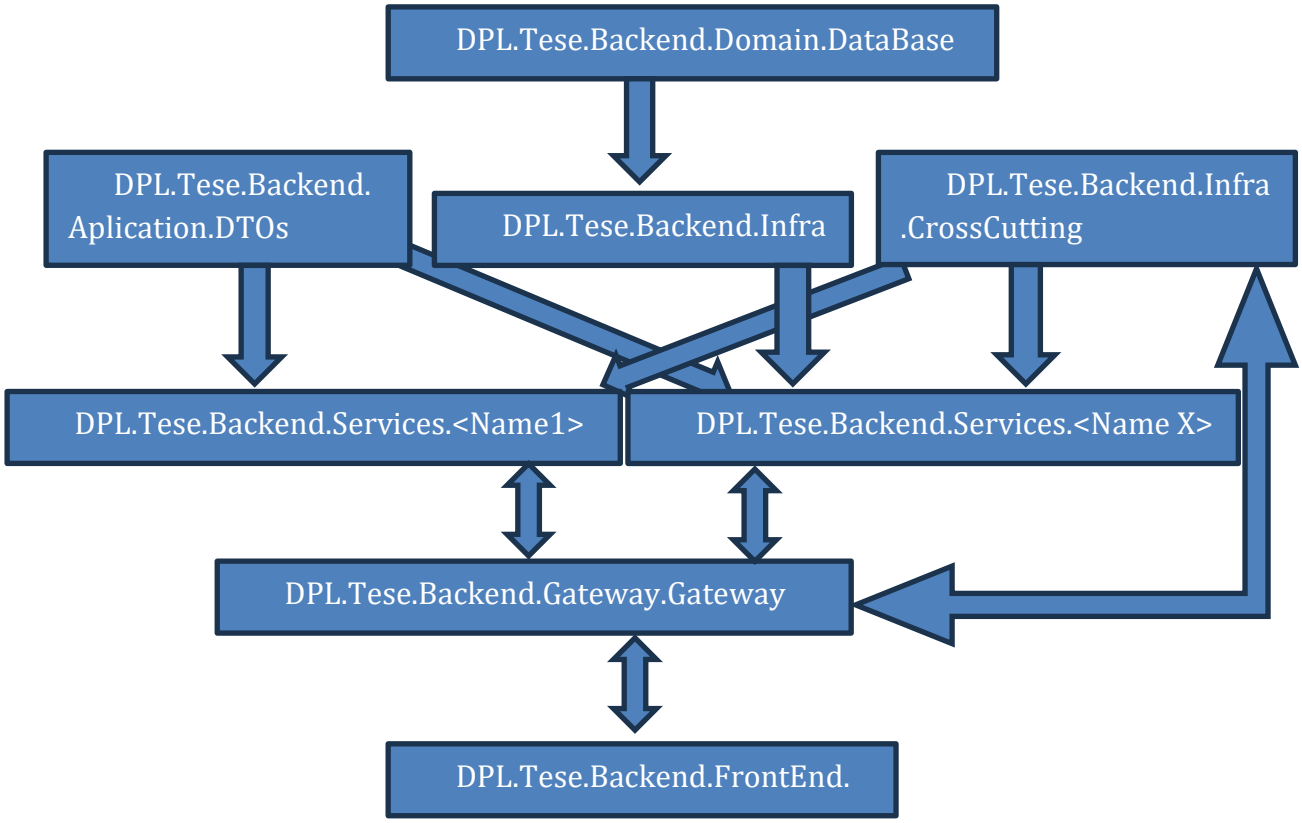


Figura 31 – Organização dos projetos do Web Cloud

- **Domain.DataBase:** Livraria que irá guardar as entidades com a lógica de negócio.
- **Application.DTOs:** Livrarias que irão guardar todos os DTOs da aplicação, responsável pela entrega e recolha do modelo de dados do lado do cliente.
- **Infra.Data:** Responsável pelo mapeamento com a base de dados e todas as interações com a mesma. Esta livraria depende da *Domain*, camada de negócio, uma vez que é responsável em transformar o código de C# em código SQL. Este método é chamado de *Code-First*.
- **Infra.CrossCutting:** Responsável por conter todas as classes e métodos que poderão ser transversais a vários projetos.
- **Service.<Name>:** Cada API responsável por ser um micro serviço que responde com as suas responsabilidades perante um pedido do *Gateway*.

- **Gateway.:** Responsável por efetuar a transcrição do endereço público para o endereço real do micro serviço. Deste modo efetua o encapsulamento. Podemos dizer que o *Gateway* funciona com o protocolo *NAT* dos routers.
- **FrontEnd.Web:** Responsável por toda a interface com o utilizador, entende-se como interface todas as janelas, campos e botões visíveis pelo utilizador.

Capítulo 13 - ESCOLHA DAS LINGUAGENS DE PROGRAMAÇÃO, MARCAÇÃO E ESTILOS

Neste capítulo serão descritas as linguagens utilizadas quer para o *BackEnd* quer para o *FrontEnd* da aplicação ou sistema desenvolvido.

13.1 - BACKEND

13.1.1 - C SHARP (C#)

Pronunciado como "C Sharp", é uma linguagem de programação orientada a objetos desenvolvida pela Microsoft. Foi introduzida no início dos anos 2000 como parte da plataforma .NET. O C# foi projetado com enfase na simplicidade, eficiência e segurança. Algumas das características notáveis do C# são:

- **Tipificagem Estática:** O que significa que os tipos de dados de todas as variáveis devem ser especificados em tempo de compilação. Isso ajuda a prevenir erros de tipos antes da execução do programa.
- **Orientação a Objetos:** É uma linguagem orientada a objetos, o que facilita a modelagem do mundo real por meio de objetos e classes. Suporta conceitos como encapsulamento, herança e polimorfismo.
- **Coleta o "lixo":** Utiliza um coletor de "lixo" (*Garbage Collector*) para gerir automaticamente a alocação e desalocação de memória, o que ajuda a evitar perdas de memoria.
- **Multiplataforma:** Com o advento do .NET Core (agora .NET 8), o C# tornou-se mais portátil e pode ser usado para desenvolver aplicações multiplataforma, incluindo Windows, Linux e macOS.
- **Biblioteca Padrão Rica:** O C# Possui uma biblioteca padrão rico (chamado de .NET Framework ou .Net Core ou .NET 8) que oferece uma ampla gama de classes e funcionalidades para desenvolvimento de aplicações.

O C# é frequentemente utilizado no desenvolvimento de BackEnd devido às suas características e ao ecossistema que o rodeia. Algumas razões para usar o C# no BackEnd:

- **Desenvolvimento rápido:** É uma linguagem de alto nível que permite o desenvolvimento rápido de aplicações. A sua sintaxe limpa e concisa ajuda a reduzir o tempo desenvolvimento.
- **Integração com o ecossistema .NET:** Faz parte do ecossistema .NET, que inclui bibliotecas e ferramentas poderosas para o desenvolvimento de servidores, serviços web, APIs e muito mais.

- **Segurança:** A tipificagem estática do C# ajuda a evitar erros de tipos comuns e torna o código mais seguro. Além disso, o C# possui recursos de segurança, como exceções controladas.
- **Confiabilidade:** É conhecido pela sua estabilidade e confiabilidade. Usado em muitos ambientes de produção críticos.
- **Suporte para serviços web:** Oferece suporte nativo para o desenvolvimento de serviços web RESTful e SOAP, tornando-o uma escolha sólida para a criação de APIs e sistemas distribuídos.
- **Integração com base de dados:** Possui ótimos recursos de integração com base de dados, como a EntityFrameworkCore, que simplifica a interação com o sistema de gestão de base de dados.

O C# é uma linguagem de programação versátil e poderosa que é amplamente utilizada no desenvolvimento de BackEnd devido à sua produtividade, segurança e integração com o ecossistema .NET. No entanto, a escolha da linguagem depende das necessidades especificas do projeto e das preferências da equipa de desenvolvimento.

13.2 - FRONTEND

13.2.1 - JAVASCRIPT

É uma linguagem de programação de alto nível, amplamente utilizada no desenvolvimento web. Foi criado originalmente pela Netscape e tornou-se uma linguagem fundamental para a criação de aplicações web interativas e dinâmicas. O JavaScript é uma linguagem de script interpretada, o que significa que não requer processo de compilação. É executada diretamente nos navegadores, permitindo a manipulação do conteúdo e do comportamento das páginas web [26].

O JavaScript é predominantemente utilizado no desenvolvimento FrontEnd, onde desempenha várias funções cruciais:

- **Interatividade com utilizador:** Uma das principais funções do JavaScript é tornar as páginas web interativas. Isso inclui a criação de elementos para *interfaces* dinâmicas para interação com o utilizador, como botões, formulários, carrosséis e menus que respondem às ações do utilizador em tempo real.
- **Manipulação da DOM:** O JavaScript permite a manipulação do *Document Object Model* (DOM) de uma página web. Isso significa que os programadores podem adicionar, remover ou modificar elementos e conteúdos da página sem necessidade de recarregá-la.
- **Validação de dados:** É usado para validar dados inseridos em formulários antes que sejam enviados ao servidor. Isso ajuda garantir que os dados sejam corretos e seguros.

- **Pedidos Assíncronos:** O JavaScript é essencial para a realização de pedidos assíncronos a servidores, possibilitando a procura e a exibição de dados em tempo real sem recarregar a página (tecnologia AJAX).
- **Controlo de Eventos:** Lida com os eventos do navegador, como cliques, movimentos do rato, teclas pressionadas e muito mais, tornando possível a criação de experiências interativas com o utilizador.
- **Integração com Mídias:** É usado para incorporar e controlar áudio, vídeo e gráficos em páginas web

Algumas vantagens do JavaScript são:

- **Compatibilidade:** É suportado por todos os principais navegadores, o que o torna uma escolha universal para o desenvolvimento Web.
- **Rapidez:** Por ser executado no navegador do utilizador, ações do utilizador podem ser processadas instantaneamente, melhorando a responsividade da página.
- **Comunidade Ativa:** Possui uma vasta comunidade de programadores, uma abundância de recursos, bibliotecas e *frameworks*.
- **Flexibilidade:** É uma linguagem versátil que permite aos programadores criar uma ampla variedade de aplicações, desde páginas web simples até aplicações móveis e até mesmo servidores (node.js).

As principais desvantagens do JavaScript são:

- **Segurança:** Como o código JavaScript é visível para o utilizador, é importante tomar precauções para evitar vulnerabilidades de segurança.
- **Incompatibilidade:** Navegadores mais antigos podem ter comportamentos inconsistentes em relação às especificações da linguagem, o que pode exigir trabalho extra para garantir a compatibilidade.
- **Desempenho limitado:** A execução no lado do cliente pode ser limitada pelo poder de processamento do dispositivo do utilizador, afetando o desempenho em aplicações intensivas em calculo.

Algumas *frameworks* de JavaScript:

- **React:** Uma biblioteca JavaScript mantida pela META para a criação de *interfaces* reativas e eficientes para o utilizador. É amplamente utilizado no desenvolvimento de aplicações web modernas.
- **Angular:** Uma *framework* JavaScript mantida pela Google que é para contruir aplicativos web de grande escala. Fornece ferramentas abrangentes para o desenvolvimento FrontEnd.
- **Vue.js:** Uma *framework* JavaScript progressiva que é conhecida pela sua simplicidade e flexibilidade. É especialmente popular para criar *interfaces* de utilizador interativos.

- **Ember.js:** Uma *framework* JavaScript focada em criar aplicações web ambiciosas. Possui convenções sólidas e um conjunto abrangente de ferramentas.
- **jQuery:** Uma biblioteca JavaScript que simplifica a manipulação do DOM, interações com Ajax e animações. Embora tenha perdido popularidade, ainda é amplamente usada em projetos grandes.

Em resumo, o JavaScript é uma linguagem de programação essencial para o desenvolvimento do FrontEnd, permitindo que os programadores criem páginas web interativas e dinâmicas.

13.2.2 - HTML

HTML, que significa *Hypertext Markup Language* (Linguagem de marcação de hipertexto), é uma linguagem de marcação fundamental para a criação de páginas web. Em vez de ser uma linguagem de programação, o HTML é uma linguagem de marcação que descreve a estrutura e o conteúdo de uma página web.

O HTML serve para estruturar o conteúdo de uma página web de maneira lógica e hierárquica. Permite que os programadores criem elementos visuais, tais como, títulos, parágrafos, listas, links, imagens, formulários e muito mais. Além disso, o HTML é a base para criar *hiperlinks* que conectam diferentes páginas web, formando a ***W**orld **W**ide **W**eb*.

A história do HTML remonta ao final dos anos 1980 e inícios dos anos 1990, quando Tim Verners-Lee, um cientista da computação britânico, desenvolveu a linguagem enquanto trabalhava no CERN (Organização Europeia para a pesquisa nuclear). A primeira versão, conhecida como HTML 1.0, foi criada com o objetivo de partilhar documentos de pesquisa de forma simples [27].

Conforme a web crescia, novas versões do HTML foram introduzidas, adicionando recursos como tabelas, formulários e formatação mais avançada. Em 1991, o HTML 4.01 tornou-se uma especificação amplamente adotada, trazendo suporte para folhas de estilo (CSS) para controlar o layout.

No entanto, a maior evolução ocorreu com o HTML 5, que começou a ser desenvolvido em 2008. Esta versão trouxe novos recursos, como suporte a vídeo e áudio, gráficos vetoriais, APIs JavaScript e muito mais. Em 2014, o HTML 5 foi formalizado como um padrão pela W3C (*World Wide Web Consortium*), marcando a nova era na web.

Hoje, o HTML 5 é amplamente utilizado para desenvolver páginas web modernas e interativas. Continua a evoluir junto com a Web, introduzindo novos recursos e possibilidades para programadores em todo o mundo.

13.2.3 - CSS

O CSS, ou *Cascading Style Sheets* (Folha de Estilos em cascata), é uma linguagem fundamental no desenvolvimento Web. É usada em conjunto com o HTML para controlar a aparência dos elementos de uma página Web. O CSS permite que os programadores apliquem estilos, como cores, fontes, margens, tamanhos e posicionamento, a elementos HTML, tornando possível projetar páginas web visualmente atraentes e bem organizados.

A história do CSS remonta ao início da Web. Antes do CSS, a formatação das páginas web era feita diretamente no HTML, resultando em código complexo e de difícil manutenção. A solução veio com a proposta do CSS em 1994 por Hakon Wium Lie e Bert Bos. O objetivo era separar a estrutura do conteúdo (HTML) da apresentação visual, facilitando a manutenção e melhorando a flexibilidade do design [28].

Em 1996, a primeira versão do CSS, conhecida como CSS1, foi lançada como uma especificação pela W3C (*Consortium World Wide Web*). Introduziu conceitos básicos de formatação, como cores, fontes e layout. Isso marcou o início da revolução no design Web.

Dois anos depois, em 1998, o CSS2 foi lançado. Esta versão trouxe recursos mais avançados, incluindo posicionamento, flutuação e folhas de estilo em cascata. O CSS2 foi marco importante no desenvolvimento de layouts complexos e designs mais sofisticados.

No início dos anos 2000, o desenvolvimento do CSS3 começou. Ao contrário das versões anteriores, o CSS3 foi dividido em módulos que introduziram novos recursos gradualmente. Isso incluía recursos como bordas redondas, sombras, animações e muito mais. Essa abordagem modular permitiu que os programadores implementassem apenas os recursos necessários para os projetos.

Hoje, o CSS3 evoluiu consideravelmente e tornou-se uma parte essencial do desenvolvimento web moderno. Os navegadores modernos oferecem suporte a uma ampla gama de recursos CSS3, permitindo que os programadores criem designs sofisticados e responsivos. O CSS desempenha um papel crucial na experiência do utilizador e na apresentação de conteúdo na web, tornando as páginas da web visualmente atraentes e funcionais.

Capítulo 14 - ESCOLHA DAS FRAMEWORKS PARA FRONTEND

14.1 - BOOTSTRAP

O Bootstrap é uma das *frameworks* de CSS mais populares e amplamente utilizadas no desenvolvimento web. Foi inicialmente criada pelo Twitter e é um conjunto de estilos e componentes prontos para utilização. A finalidade principal do Bootstrap é acelerar o processo de desenvolvimento web e melhorar a consistência visual das páginas web [29].

O Bootstrap foi concebido no Twitter em 2010 por Mark Otto e Jacob Thornton. Inicialmente, foi desenvolvido para uso interno, mas foi lançado como um projeto de código aberto. A abordagem inovadora e a capacidade de fornecer uma estrutura sólida para o desenvolvimento web rapidamente chamaram a atenção da comunidade de programadores.

O Bootstrap cresceu rapidamente em popularidade e tornou-se uma das *frameworks* de CSS mais utilizadas no mundo. Com a evolução, passou por várias versões, sendo a versão Bootstrap 5 a mais recente. A equipa de desenvolvimento do Bootstrap continuadamente trabalha para a melhorar e expandir o conjunto de recursos.

Algumas das vantagens do Bootstrap são:

- **Economia de tempo:** Oferece uma ampla gama de componentes prontos para utilização, economizando o tempo de desenvolvimento.
- **Consistência Visual:** Ajuda a manter uma aparência consistente em todo o site, garantindo um design atraente.
- **Layout Responsivo:** Facilita a criação de layouts que se adaptam automaticamente a diferentes dispositivos e tamanhos de ecrã.
- **Documentação Abundante:** Possui documentação detalhada e exemplos disponíveis, o que facilita a aprendizagem e a implementação.
- **Comunidade ativa:** Uma grande comunidade de programadores significa suporte e recursos adicionais disponíveis.

Algumas desvantagens do Boostrap são:

- **Aparência padrão:** Sites que usam Bootstrap podem parecer semelhantes, a menos que personalizados extensivamente.
- **Tamanho do ficheiro:** O BootStrap pode adicionar algum tamanho ao projeto, especialmente se usar muitos dos recursos.
- **Personalização complexa:** Personalizar o Bootstrap além das configurações padrão pode ser complexo e requer conhecimentos avançados de CSS.

Em comparação com outras *frameworks* de CSS, como Foundation e Bulma, o Bootstrap é conhecido pela vasta adoção e documentação abrangente. É amplamente utilizado em todo o mundo, o que significa que a comunidade é grande e há uma abundância de recursos disponíveis. No entanto, algumas outras *frameworks* podem oferecer uma abordagem mais flexível e personalizável, permitindo que os programadores criem design mais distintos e únicos. Portanto, a escolha entre o Bootstrap e outras *frameworks* depende dos requisitos específicos do projeto e das preferências do programador.

14.2 - SYNCFUSION EJ2 JAVASCRIPT (ES5)

O Syncfusion Essential JS2 (EJ2) é uma biblioteca de componentes JavaScript desenvolvida pela Syncfusion, uma empresa de software especializada em fornecer soluções de desenvolvimento para programadores de software. A biblioteca desempenha um papel fundamental no desenvolvimento de aplicações web interativas e rico em recursos [30].

Essencialmente, o EJ2 é projetado para simplificar o desenvolvimento de interfaces para o utilizador web, atraentes e funcionais. Oferece uma ampla variedade de componentes de interfaces para o utilizador prontos para usar, como tabelas de dados, gráficos, calendários, controlos de entrada e muito mais. Os programadores podem aproveitar esses componentes para criar rapidamente interfaces de utilizador complexas e interativas.

A história do EJ2 remonta à versão anterior, o Essential JS, que já era uma biblioteca popular no desenvolvimento web. No entanto, com a crescente procura por recursos e componentes mais avançados, a Syncfusion lançou o EJ2 como uma evolução para atender a essas necessidades. O EJ2 foi projetado para ser uma biblioteca robusta e flexível, capaz de atender a uma ampla gama de casos de uso.

Algumas das vantagens do Syncfusion Essentials JS2 são:

- **Ampla variedade de componentes:** oferece uma vasta gama de componentes prontos a utilizar, fornecendo aos programadores recursos suficientes para criar diversos tipos de aplicações.
- **Personalização:** Os componentes são altamente personalizáveis, permitindo que os programadores atendam aos requisitos específicos de design do projeto.
- **Documentação Detalhada:** A Syncfusion fornece documentação detalhada e exemplos, tornando o processo de aprendizagem e implementação mais fácil.
- **Suporte Técnico:** A empresa oferece suporte técnico para os seus produtos, o que pode ser valioso para empresas que enfrentam desafios técnicos.

Algumas das desvantagens do Syncfusion Essentials JS2 são:

- **Licenciamento:** Utiliza um modelo de licenciamento que pode não ser adequado para todos os projetos, especialmente projetos de código aberto ou pessoais.
- **Complexidade:** Devido à ampla gama de recursos e opções de personalização, a curva de aprendizado pode ser ingreme para iniciantes.
- **Tamanho do Pacote:** A utilização de muitos componentes de EJ2 pode resultar em um tamanho de pacote maior para as aplicações, o que pode afetar o tempo de carregamento.

Em comparação com outras ferramentas coma mesma finalidade, como bibliotecas concorrentes ou *frameworks* de *interface* de utilizador, a escolha do EJ2 depende dos requisitos específicos do projeto e das preferências dos programadores. A robustez, personalização e suporte técnico do EJ2 podem ser pontos a favor, mas é importante considerar fatores como o custo e complexidade antes de tomar uma decisão.

Capítulo 15 - ESCOLHA DA FERRAMENTA DE DESENVOLVIMENTO

No cenário em constante evolução do desenvolvimento de software, as ferramentas de desenvolvimento desempenham um papel crítico na produtividade, eficiência e qualidade dos projetos. Entre essas ferramentas, o Visual Studio 2022 destaca-se como uma das mais poderosas e versáteis. O Visual Studio 2022 apresenta-se como uma ferramenta de desenvolvimento essencial, destacando as características fundamentais e demonstrando por que é amplamente adotado por programadores de todo o mundo.

- **Fundamentação do Visual Studio:** Desenvolvido pela Microsoft, é uma suite de desenvolvimento integrado (IDE) que existe há décadas. Durante esse período, evoluiu constantemente para atender à crescente procura da indústria de desenvolvimento de software. O Visual Studio 2022 é a mais recente iteração dessa jornada de inovação.
- **Características-Chave:** O Visual Studio 2022 destaca-se pelas seguintes características-chave:
    - **Multiplataforma:** Oferece suporte a desenvolvimento multiplataforma, permitindo que os programadores criem aplicações para Windows, Linux, MacOs, dispositivos móveis e web.
    - **Linguagens de Programação:** Suporta várias linguagens de programação populares, incluindo C#, Visual Basic .Net, F#, Pyton, JavaScript e muito mais.
    - **Ferramentas de Depuração Avançadas:** Oferece um conjunto robusto de ferramentas de depuração que facilitam a identificação e correção de erros no código.
    - **Integração de Git:** Integra-se perfeitamente com o Git, facilitando o controle de versões e a colaboração em equipa.
    - **Desenvolvimento Web Moderno:** Fornece suporte avançado para o desenvolvimento web, incluindo ferramentas para ASP.NET Core, Blazor, React, Angular e muito mais.
    - **Inteligência Artificial:** Incorpora recursos de inteligência artificial e conhecimento da máquina para acelerar o desenvolvimento de aplicações inteligentes.
- **Importância para os programadores:** Porque o Visual Studio 2022 é essencial para os programadores:
    - **Produtividade:** Oferece uma ampla gama de recursos e ferramentas que aumentam a produtividade dos programadores, desde a sugestão de código até à geração automática de código.
    - **Comunidade Ativa:** A comunidade de programadores em torno do Visual Studio é vasta e ativa. Isso significa acesso a suporte, bibliotecas de terceiros e uma riqueza de recursos.

- **Integração de fluxo de trabalho:** O Visual Studio 2022 integra todo o ciclo de vida de desenvolvimento, desde a criação de código até a implantação, facilitando a gestão de projetos complexos.
- **Escalabilidade:** A ferramenta é escalável para atender às necessidades de projetos pequenos e grandes, tornando-a adequada para start-ups e grandes empresas.

O Visual Studio 2022 é muito mais do que uma simples ferramenta de desenvolvimento, é um ambiente completo para criar, depurar e implantar software de alta qualidade. A versatilidade, suporte à comunidade e conjunto abrangente de recursos tonam-no uma escolha essencial para programadores em todo o mundo.

Capítulo 16 - EXEMPLO ORGANIZAÇÃO DA NOVA PLATAFORMA WEB CLOUD

Para uma boa organização e entendimento do projeto é fundamental o líder da equipa de desenvolvimento reunir com a equipa e definirem em conjunto os protocolos da organização do projeto, nomenclaturas e standard. Deste modo, toda a comunicação é bem interpretada pelo emissor e pelo recetor. Devem ser definidos:

- Atribuição do nome base da solução, nome das pastas. Para o presente exemplo iremos assumir o nome base de DPL.Tese.
- Criação dos vários projetos e respetivas responsabilidades:
    - DPL.Tese.Backend.Domain.DataBase.
    - DPL.Tese.Backend.Infra.Data.
    - DPL.Tese.Backend.Gateway.Gateway.
    - DPL.Tese.Backend.Infra.CrossCutting.
    - DPL.Tese.Backend.Application.DTOs.
    - DPL.Tese.Backend.Services.<NameService>.
    - DPL.Tese.Frontend.Web.
- Definição da interligação e dependência dos vários projetos.
- Métodos de comunicação entre os serviços e a base de dados.
- Nome dos métodos a utilizar em cada serviço e a respetiva ligação a cada controlador da API.
- Definição da resposta do *BackEnd* para o *FrontEnd*.
- Definição dos *NuGets* base da *BackRnd*.
- Definição das *frameworks* do *FrontEnd*.
- Organização do *FrontEnd*.

16.1 - DEFINIÇÃO DE NOME BASE DA SOLUÇÃO, NOME DAS PASTAS E DOS PROJETOS

Começou-se por criar uma solução vazia e criar projeto a projeto, de modo a se poder controlar todo o crescimento da plataforma. Para o nome da solução decidiu-se usar a seguinte nomenclatura: <NomedaEmpresa>.<NomeDaSolucao>. Neste caso, iremos assumir que a empresa se chama DPL e a solução Tese, então o nome raiz da solução será DPL.Tese.

No Visual Studio 2022 escolhemos o projeto "*Blank Solution*", Figura 32.

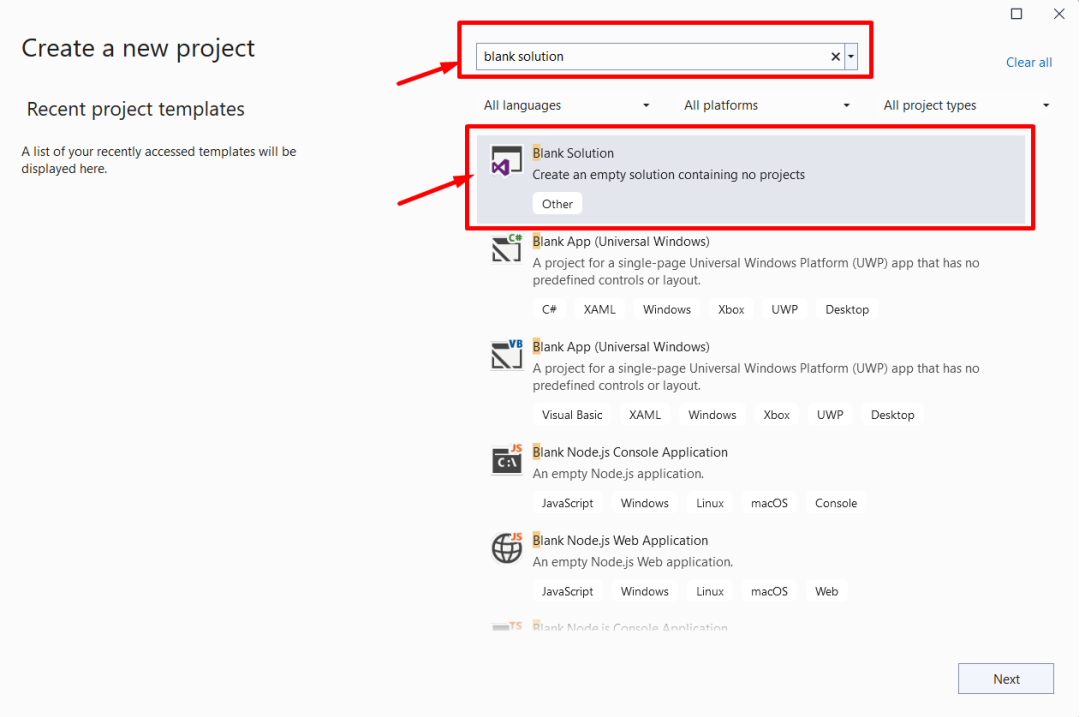


Figura 32 – Criação de uma solução vazia no *Visual Studio 2022*

Na janela seguinte, definimos o nome da solução e a localização onde o mesmo será guardado, Figura 33.

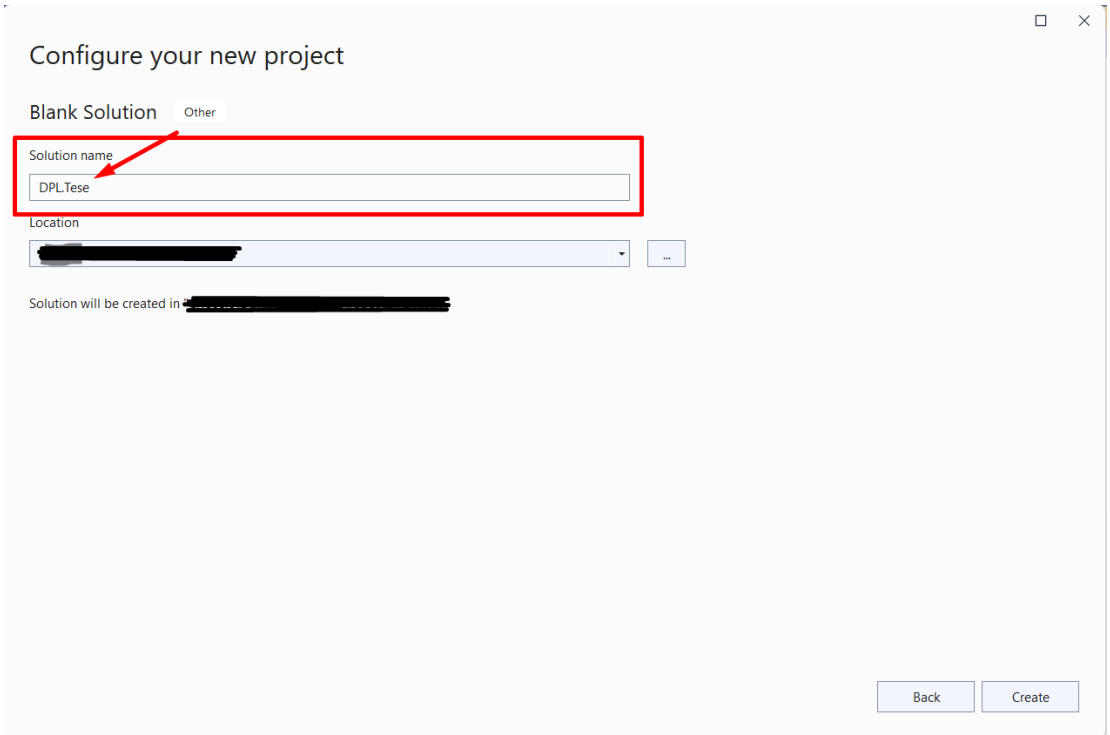


Figura 33 – Definição do nome da solução

No final temos no explorador de soluções a solução criada, Figura 34.

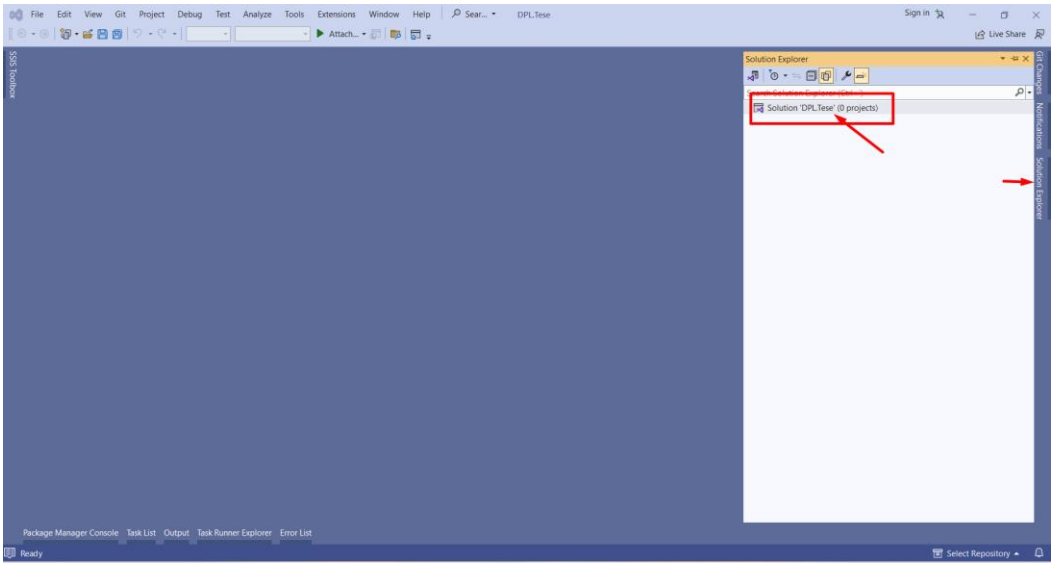


Figura 34 – Resultado da criação da solução vazia

Sabendo que o Visual Studio ordena o nome das pastas por ordem alfabética, decidimos criar três pastas na raiz com a seguinte nomenclatura <Posicao>.<TipoProjecto> e uma pasta para guardar toda a documentação e outros ficheiros relativos à solução. Assim, obtêm-se as seguintes pastas, Figura 35:

- .SolutionItems
- 1.BackEnd
- 2.FontEnd

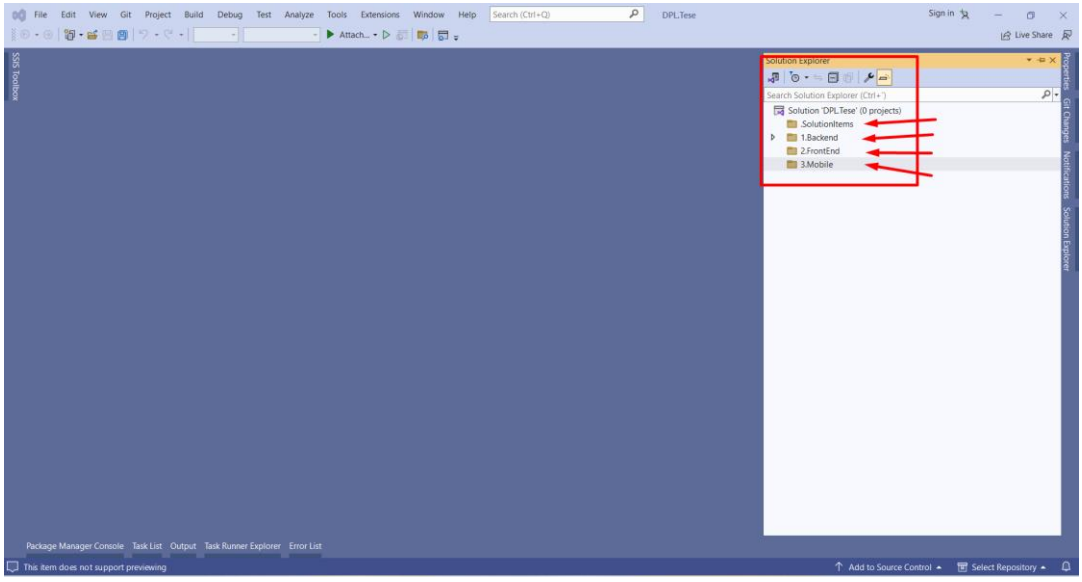


Figura 35 – Organização das pastas dentro da solução

Dentro da pasta de *BackEnd* decidiu-se criar as seguintes subpastas de modo a respeitar a definição de *Domain Driven Design*, Figura 36:

- 1.1.Domain
- 1.2.Infra
- 1.3.Application
- 1.4.Services
- 1.5.Gateway

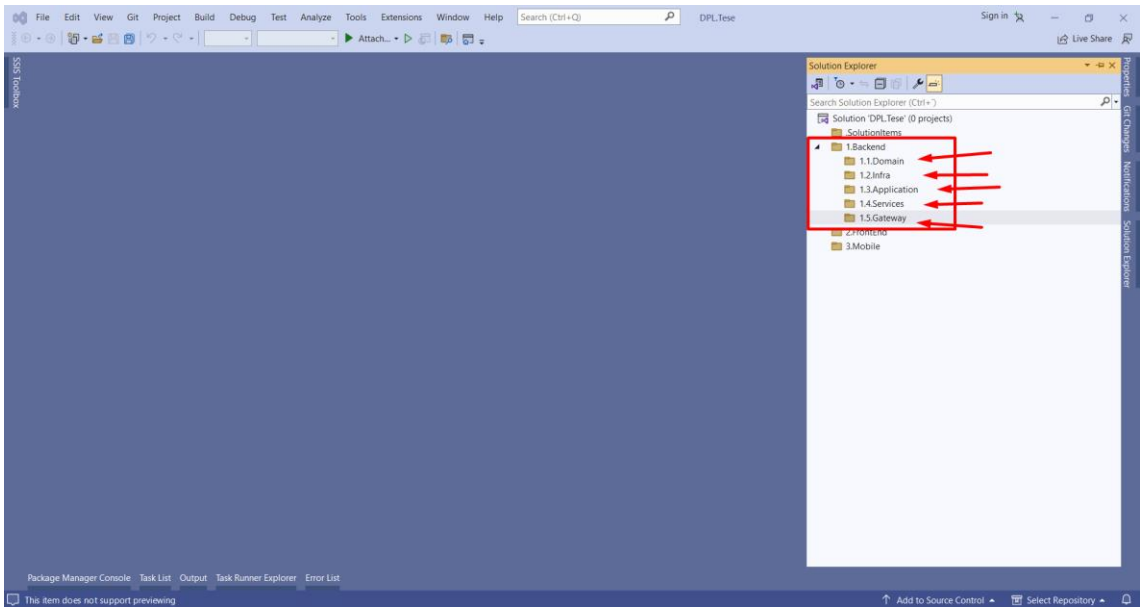


Figura 36 – Organização das subpastas do *Backend*

Neste momento já está feita a organização de pastas que irá alojar todos os projetos.

16.2 - CRIAÇÃO DOS VÁRIOS PROJETOS E RESPETIVAS RESPONSABILIDADES

16.2.1 - DPL.TESE.BACKEND.DOMAIN.DATABASE

Para tirar o maior partido das potencialidades da EntitiyFrameworkCore decidimos que a criação da base de dados será feita segundo a abordagem *Code-First*. O *Code-First* é uma abordagem de desenvolvimento de base de dados em que a estrutura da base de dados é criada com base no código da aplicação, neste caso em classes C#. Esta abordagem oferece algumas vantagens, tais como a capacidade de manter o foco na lógica da aplicação, uma vez que a estrutura da base de dados é gerada automaticamente. Além disso, torna mais fácil a gestão das alterações à base de dados,

uma vez que as alterações podem ser refletidas no código da aplicação e, em seguida, migradas para a base de dados. Para podermos utilizar a EntityFrameworkCore neste projeto é necessário instalar o *NuGet* Microsoft.EntityFrameworkCore, Figura 37.

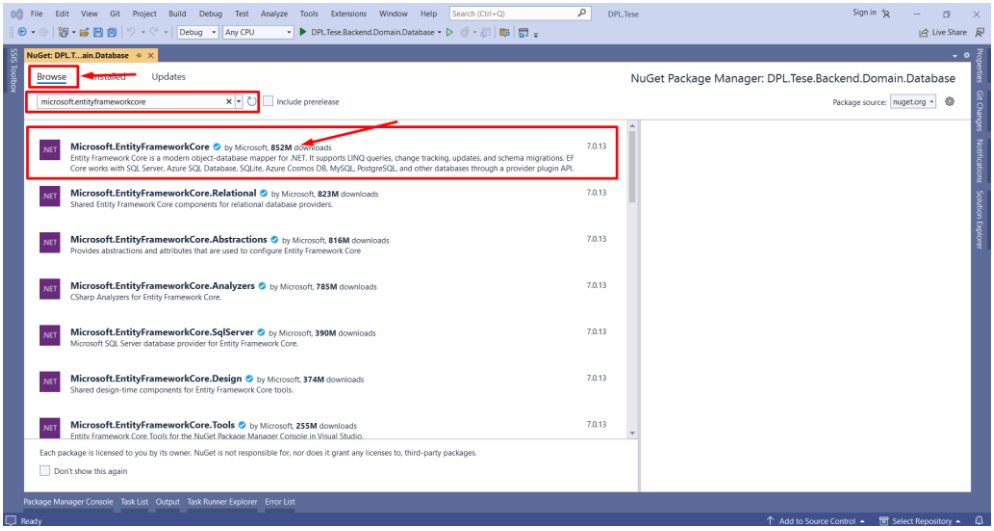


Figura 37 – Instalação do Nuget Microsoft EntityFrameworkCore

Para podermos utilizar noutros projetos da solução criamos um projeto do tipo *Class library* e definimos que dentro iriamos criar três pastas, Figura 38, 39 e 40:

- ***Interfaces*:** Responsável por guardar todas as *interfaces* do projeto.
- ***Tables*:** Responsável por guardar todas as classes que representam cada tabela na base de dados.
- ***Shared*:** Classe que irá conter todas as classes das quais será herdada as propriedades comuns.

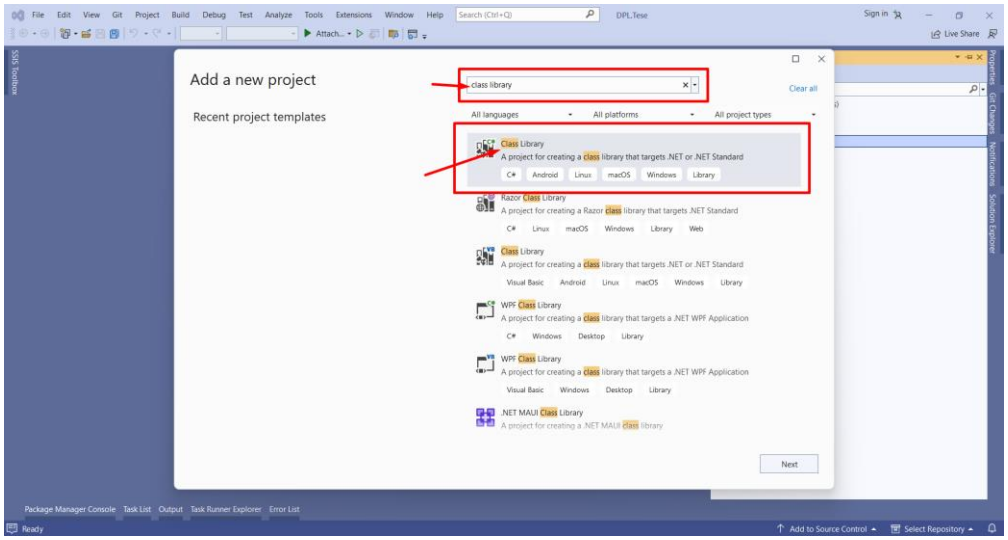


Figura 38 – Criação de um projeto tipo *Class Library*

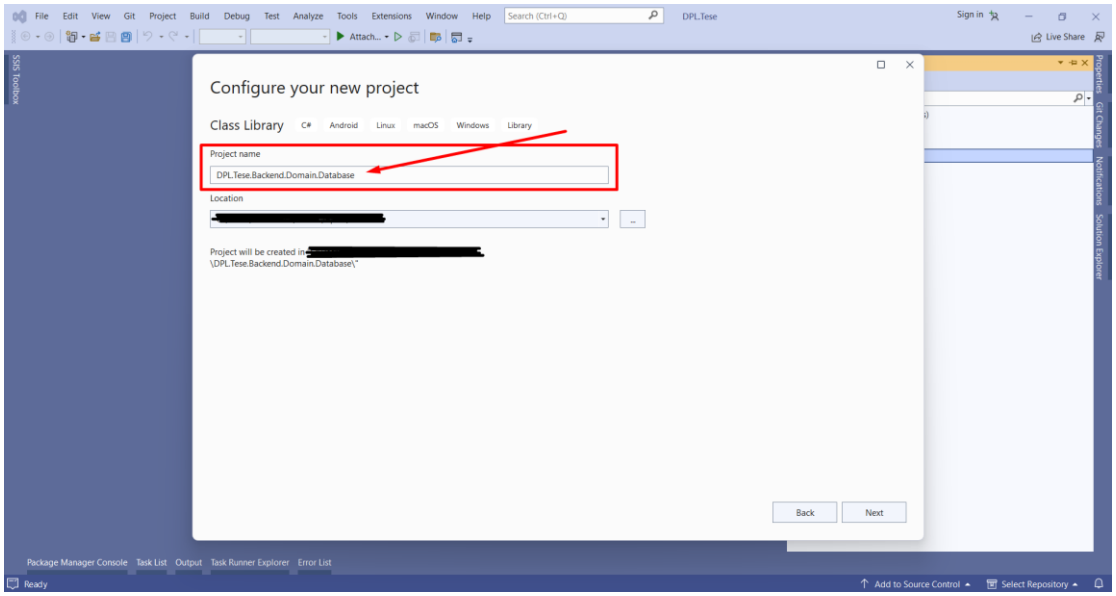


Figura 39 – Definição do nome do projeto *Domain*

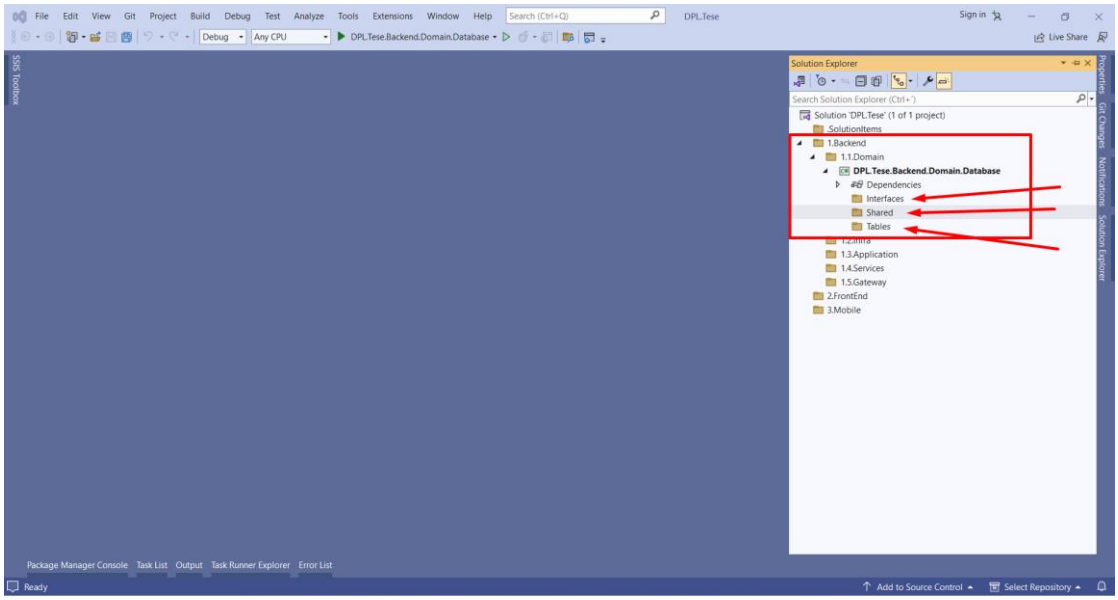


Figura 40 – Resultado da criação do projeto Domain

Uma vez que todas as tabelas irão ter atributos que são comuns, de modo a não repetir constantemente o código e otimizar tempo de desenvolvimento, decidimos criar duas interfaces, Figura 41:

- **IBaseEntity:** Responsável por conter todos os campos transversais a todas as tabelas, exceto o campo ID pois este poderá ser utilizado numa tabela que não necessite dos restantes campos transversais ou quando utilizado o

Reflection para percorrer todas as tabelas conseguirmos ter uma *interface* sem parâmetros.

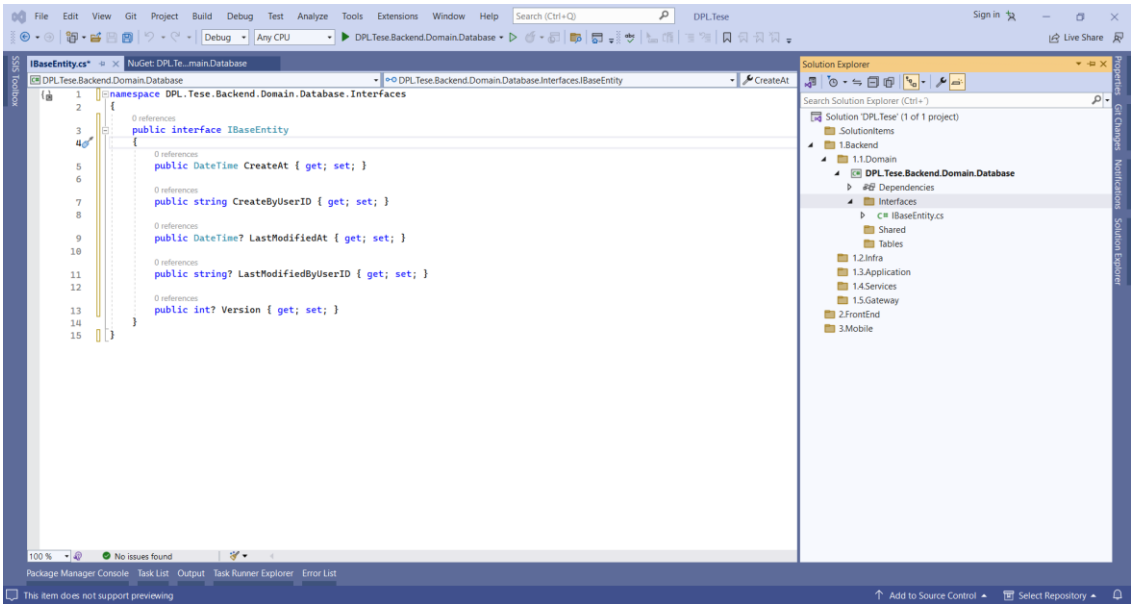


Figura 41 – Criação da interface IBaseEntity

- **IBaseID:** Responsável por definir que tipo de dados irá ser o ID da tabela, sendo o programador a definir aquando da criação de cada classe para cada tabela. Esta interface, Figura 42, irá ter apenas um atributo que será o ID e irá herdar da interface IBaseIdentity.

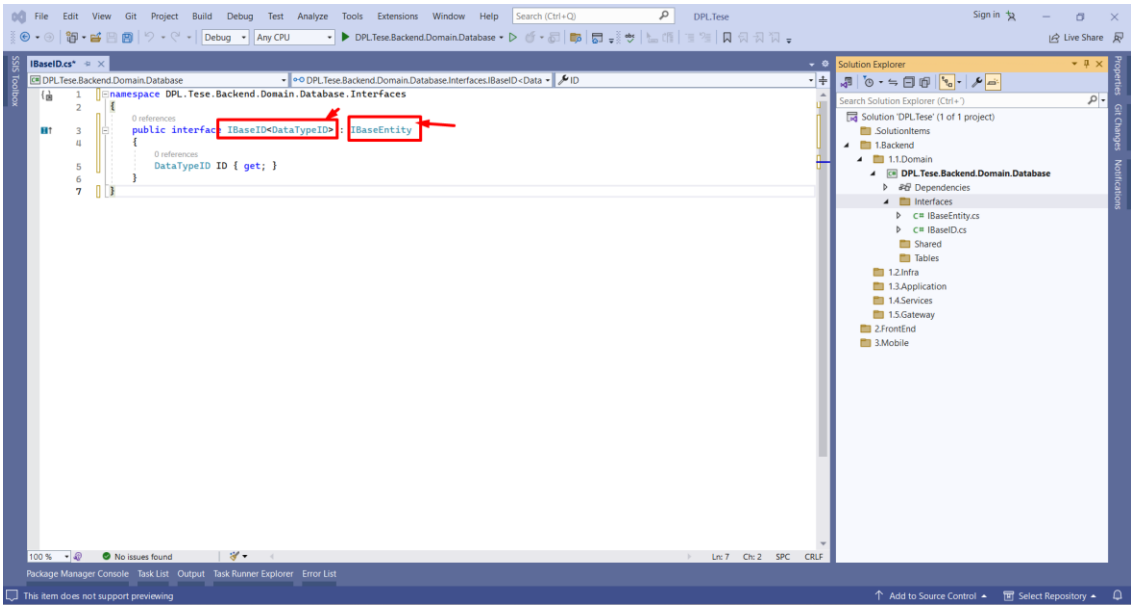


Figura 42 – Definição da *interface* IBaseID

Na pasta *Shared* decidimos criar a *class* BaseEntity, Figura 43, que herdará da interface IBaseID que definirá o tipo de dados do ID e outros atributos.

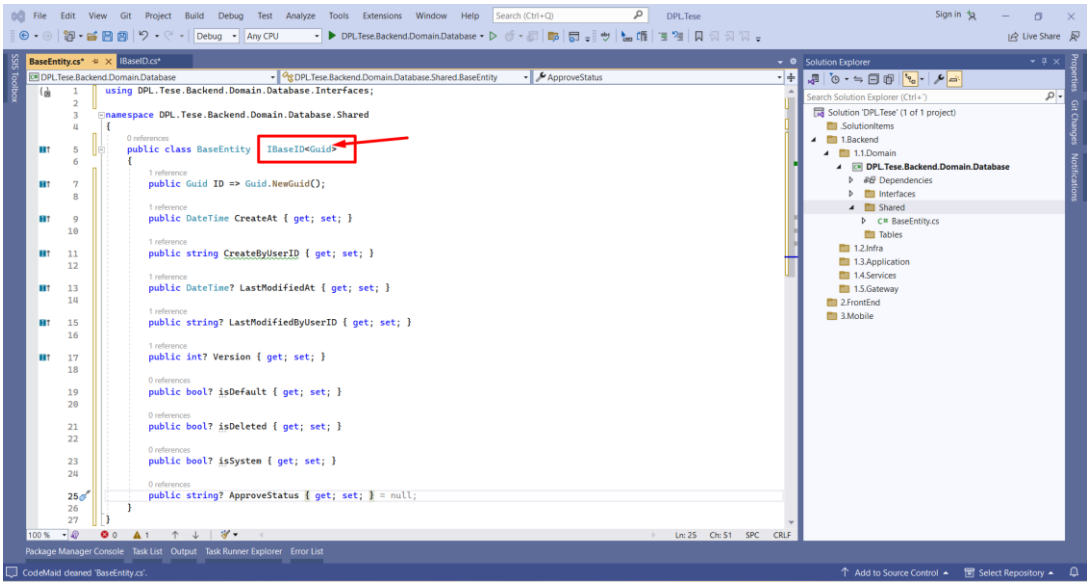


Figura 43 – Definição da class BaseIdentity

Para uma melhor organização do projeto, dentro da pasta *Tables* decidimos tentar dividir o máximo possível em subpastas cada módulo, Figura 44, a *título* de exemplo teremos as seguintes subpastas:

- ***Application*:** Terá as tabelas das configurações da aplicação.
- ***Authentication*:** Terá as tabelas referentes à autenticação.
- ***ResourceHuman*:** Terá as tabelas referentes aos recursos humanos.
- ***Financial*:** Terá as tabelas referentes ao modulo financeiro.

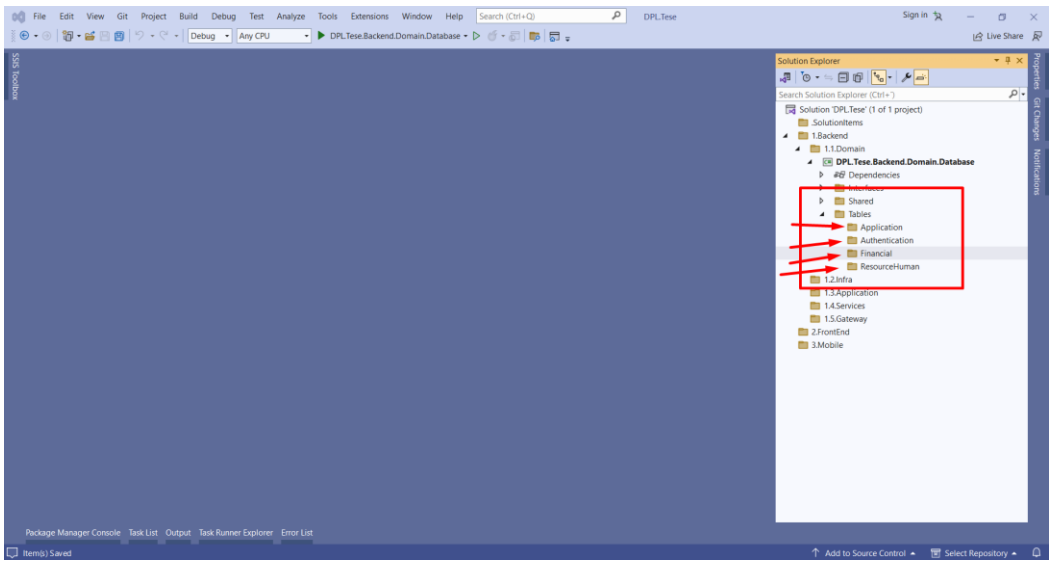


Figura 44 – Organização das pastas das entidades

Decidiu-se que o nome das classes representando as tabelas na base de dados iriam ser precedidas das duas letras que identifica o módulo. A título de exemplo teremos:

- **RHMaritialStatus:** Tabela que representa o estado civil da pessoa dentro do módulo de recursos humanos.
- **RHPerson:** Tabela que representa os dados de uma pessoa utilizadora do sistema, dentro do módulo de recursos humanos.
- **APPFile:** Tabela que guarda a base 64 do ficheiro da aplicação.
- **APPSettings:** Tabela que guardará as configurações das tabelas.

Todas estas classes irão herdar da classe BaseEntity, Figura 45 e 46.

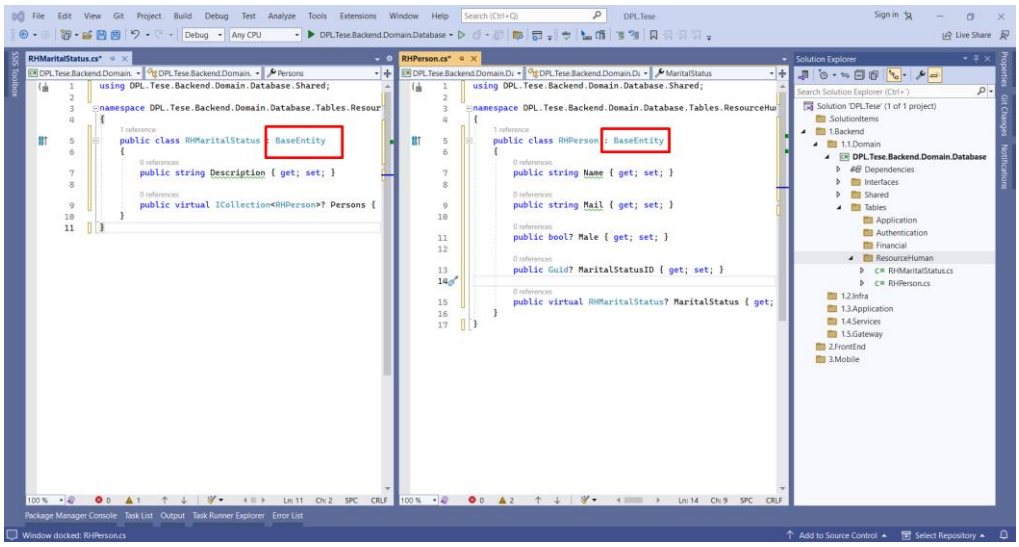


Figura 45 – Exemplo da criação de entidades que representam tabelas

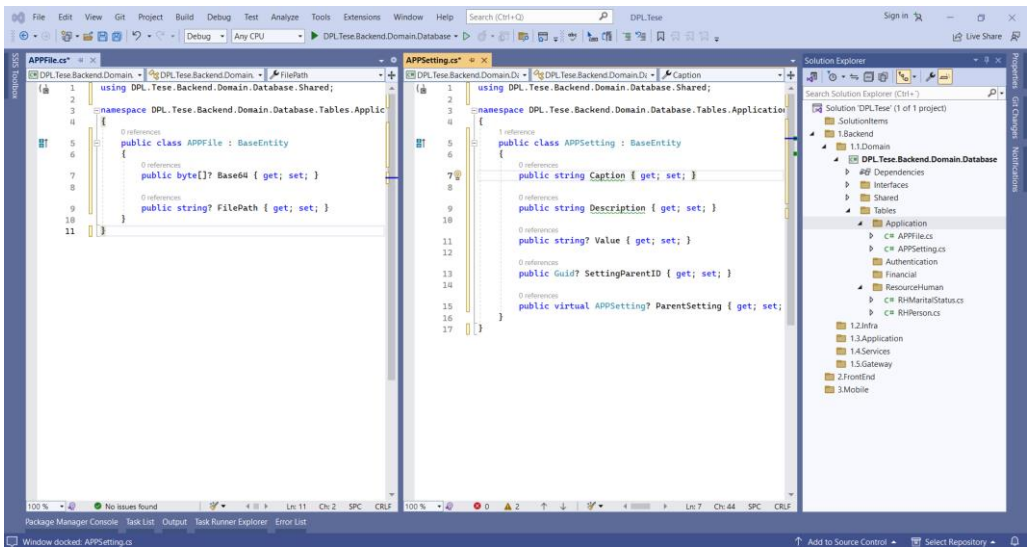


Figura 46 – Exemplo da criação de entidades que representam tabelas

Para criar as tabelas de autenticação, decidimos utilizar o *NuGet* da Microsoft *Identity,* Figura 47 e fazer a pequenas modificações para conter os atributos extras que necessitávamos [31].

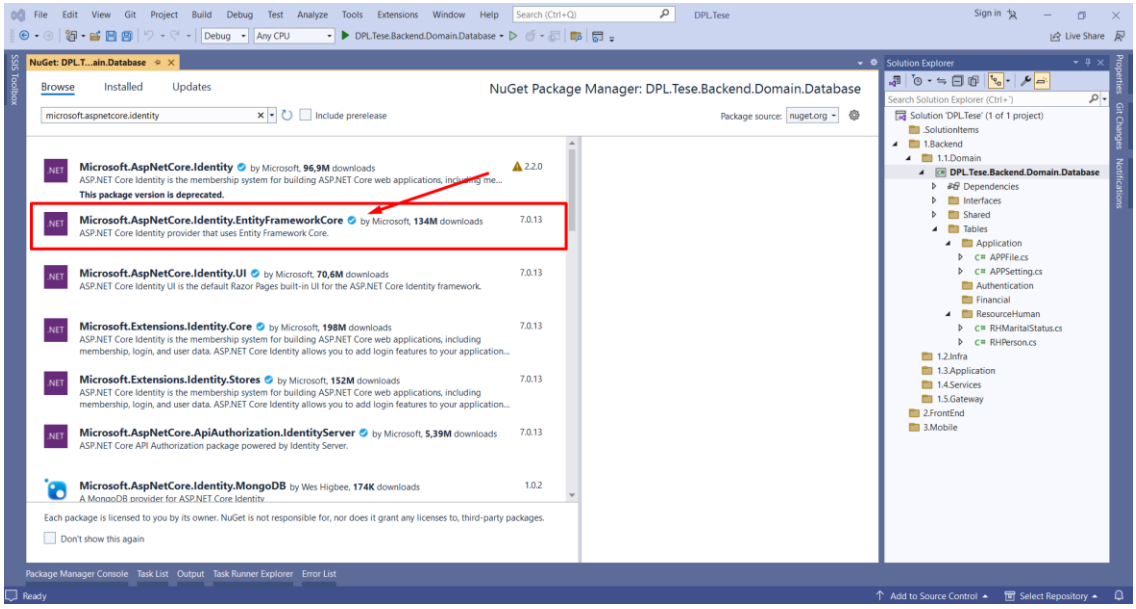


Figura 47 – Instalação do *Nuget* Microsoft *Identity*

A livraria do Microsoft *Identity* tem por base as seguintes tabelas para o sistema de autenticação:

- ***IdentityRole*:** Guarda os grupos de utilizador, ou as suas responsabilidades dentro da aplicação.
- ***IdentityRoleClaim***: Tabela de ligação entre as responsabilidades e as permissões.
- **IdentityUser**: Guarda os dados dos utilizadores.
- **IdentityUserRole:** Guarda a ligação entre os utilizadores e as várias responsabilidades que ocupam dentro do sistema.
- **IdentityUserClaim:** Guarda os dados da ligação entre um utilizador e as respectivas permissões.
- **IdentityUserLogin:** Guarda os logins externos que permitem fazer login na nossa aplicação.
- **IdentityUserToken:** Guarda os *token* das API externas, tais como Google, Facebook e etc.

De forma a seguir a nomenclatura por nós definida, decidimos criar classes com os nomes definidos e herdá-las das classes da Microsoft *Identity*, o que também nos permite adicionar novos atributos que necessitamos. Sendo assim, substituímos a

palavra *Identity* no nome das classes por Auth do módulo de autenticação, Figura 48, assim como nas tabelas de ligação, que correspondem às tabelas de relacionamento, colocamos um *underscore* (_) entre o nome das tabelas relacionadas.

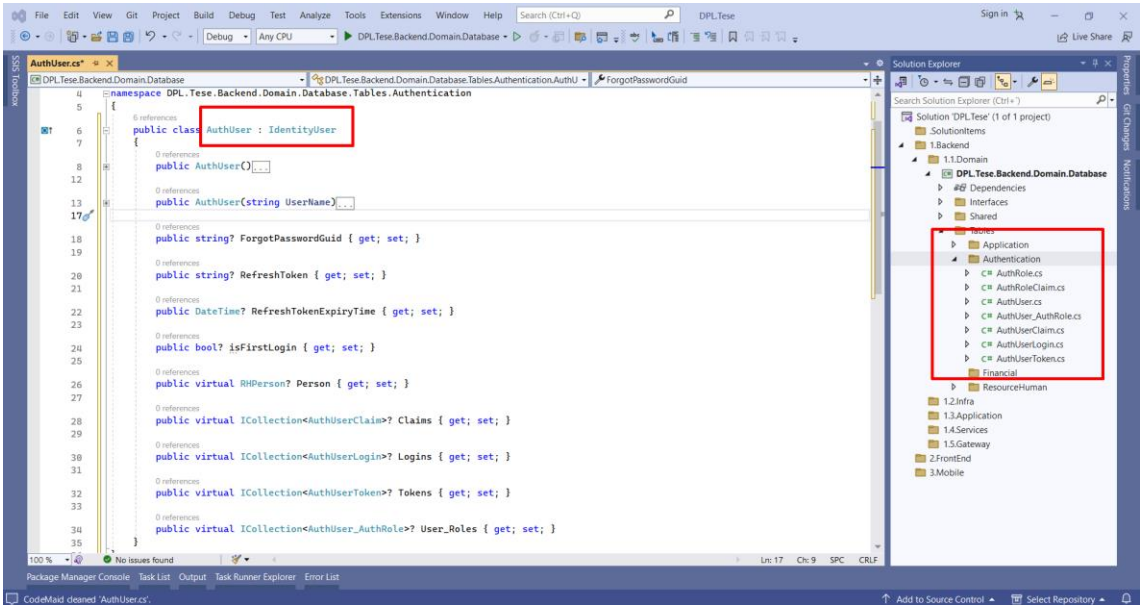


Figura 48 – Criação da entidade *AuthUser* para substituir a *IdentityUser*

Ficou decidido que esta *class library* define as tabelas da base de dados. Será utilizada para gerar as migrações, para criar e atualizar a base de dados de forma automática, ficando a responsabilidade do lado da *EntityFrameworkCore*. Este projeto não depende, nem pode depender de outros projetos na solução.

16.2.2 - DPL.TESE.BACKEND.INFRA.DATA

Estando a camada de *Domain* já inicializada e organizada, iremos criar a livraria(camada) responsável pela interação com o gestor de base de dados, que neste caso será o *SQL Server* da Microsoft. A presente camada irá depender da camada *Domain* pois será responsável por ler a informação de cada *class* que representa uma tabela, gerar o script automaticamente para depois ser executado na base de dados. Esta camada recorrerá ao *Nuget* da Microsoft *EntityFrameworkCore* e da Microsoft *Identity*. Além destes dois *Nugets* foi decidido que para efetuar as operações de CRUD (criar, ler, atualizar, apagar) na base de dados iremos utilizar outro micro ORM chamado Dapper, que é também disponibilizado via *Nuget*. A decisão de utilizar dois ORM em conjunto deveu-se a que a EntityFrameworkCore traz muitas vantagens a nível de criação, alteração e eliminação de tabelas de modo automático, mas é mais lenta e mais complexa a nível das operações de CRUD. O Dapper como executa consultas de SQL nativas, tem uma maior performance, facilidade e liberdade de

operações com dados. Este projeto irá ficar dentro da pasta infra, Figura 50, e será do tipo *Class Library,* Figura 49.

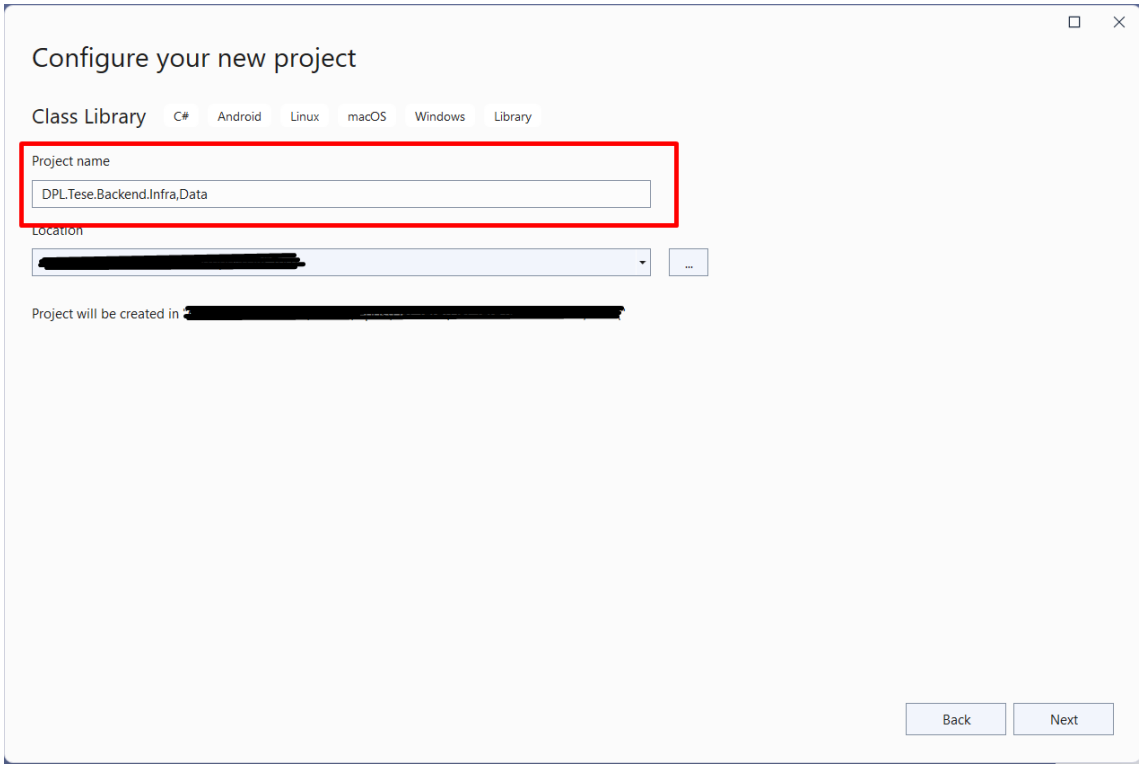


Figura 49 – Criação do projeto Data

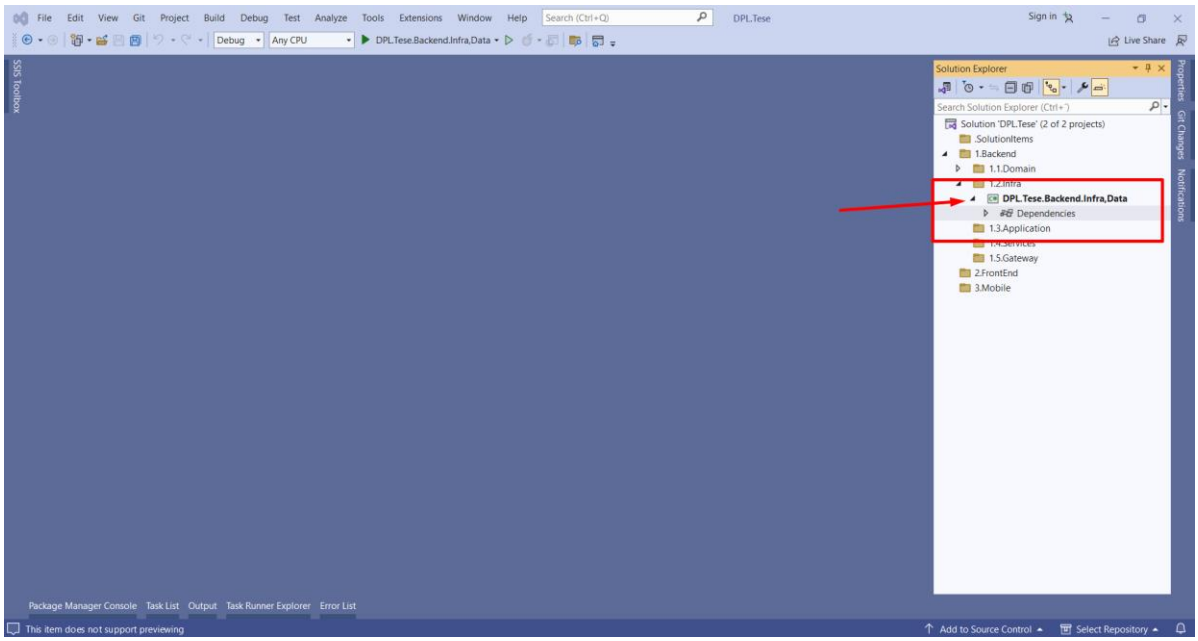


Figura 50 – Resultado da criação do projeto Data

Após a criação do projeto do tipo *Class Library*, procedemos à instalação do Nugets.

- Microsoft EntityFrameworkCore, Figura 51.
- Microsoft Identity, Figura 52.

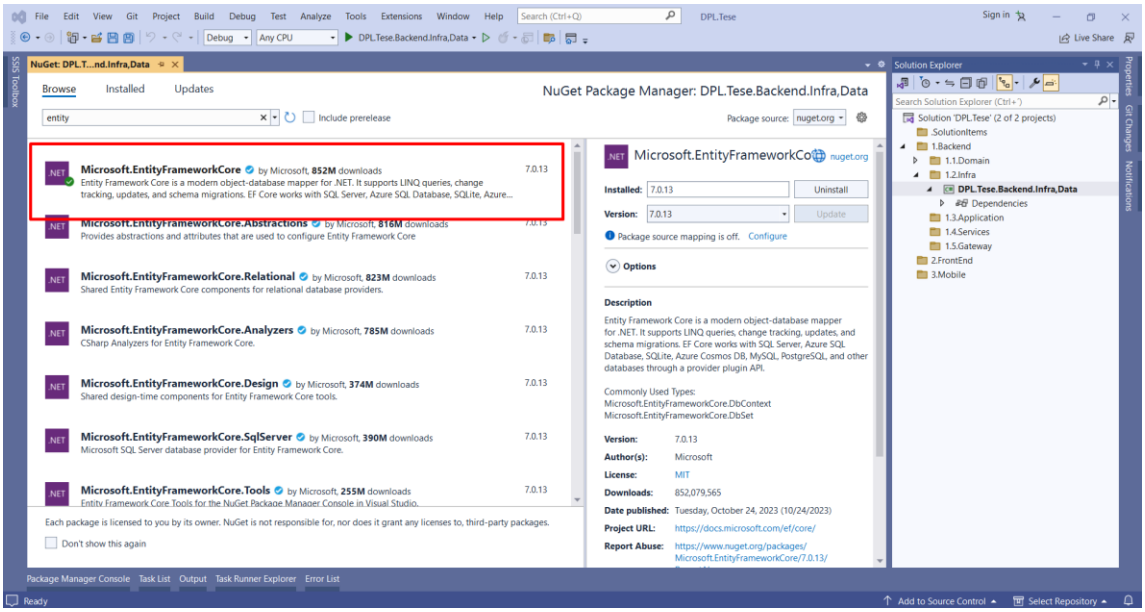


Figura 51 – Instalação do *Nuget* Microsoft *EntityFrameworkCore* no projeto Data

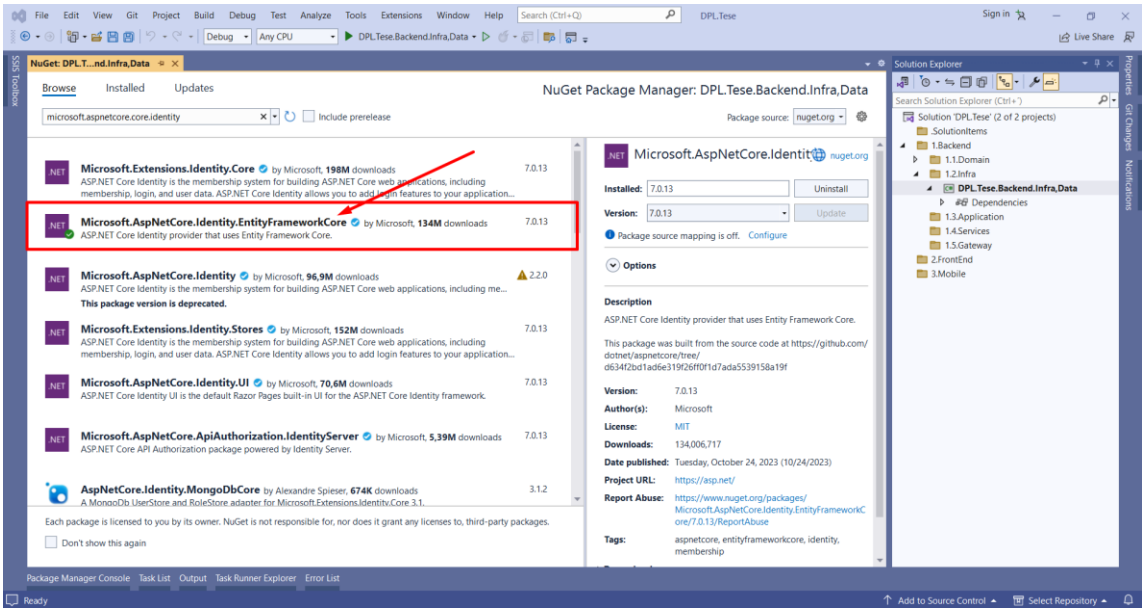


Figura 52 – Instalação do *Nuget* Microsoft *Identity* no projeto Data

Nuget Dapper, Figura 53.

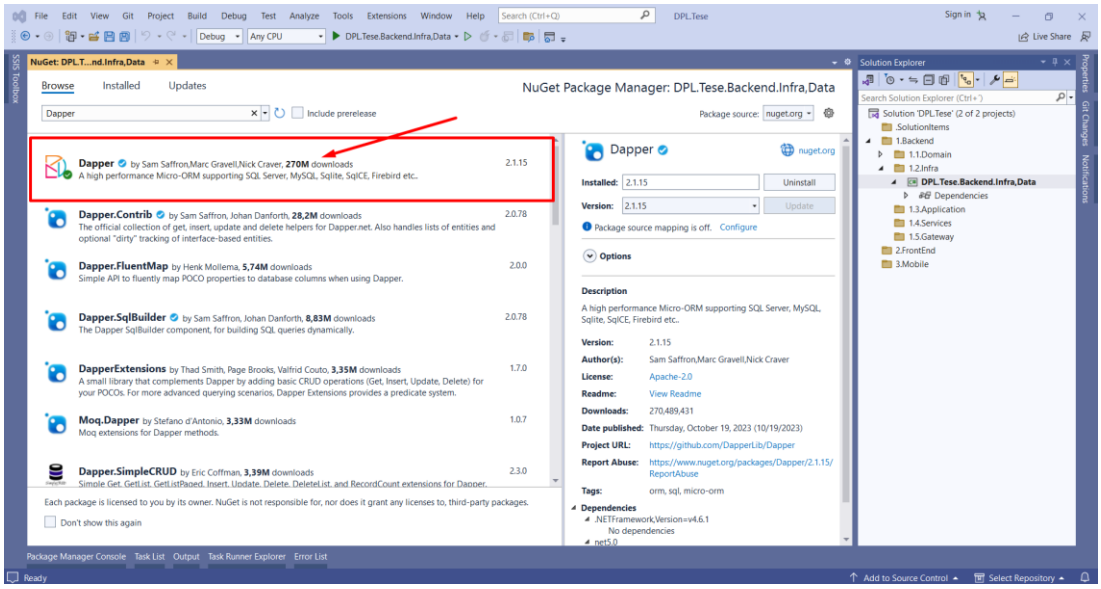


Figura 53 – Instalação do *Nuget* Dapper no projeto Data

Além destas dependências, necessitamos de inserir a dependência do projeto *Database* do *Domain,* Figura 54 e 55. para termos acesso às classes de representação das tabelas na base de dados, Figura 56.

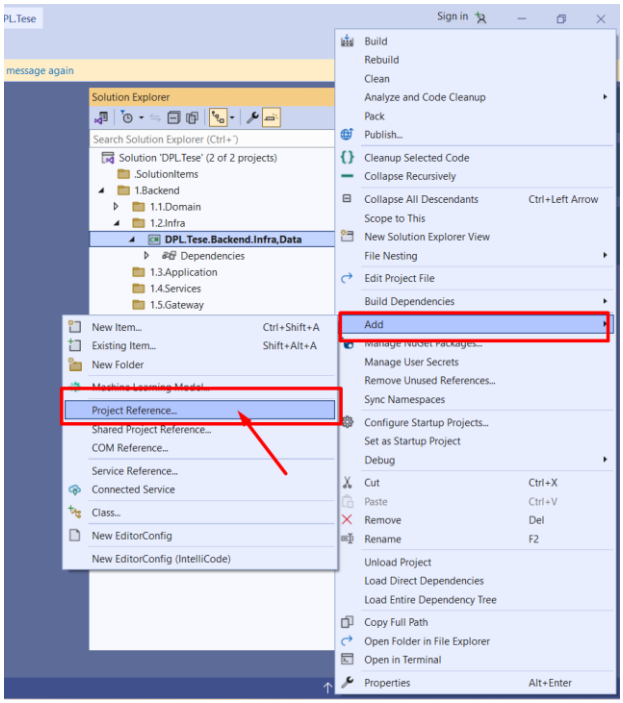


Figura 54 – Exemplo como adicionar a referência de um projeto

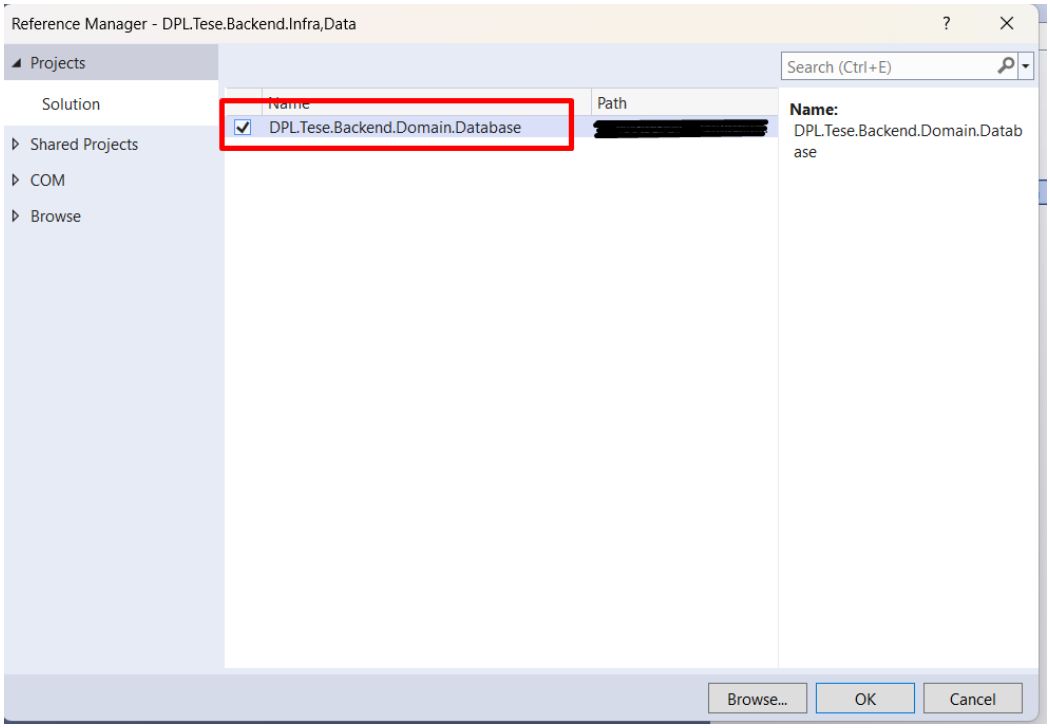


Figura 55 – Seleção do projeto a referenciar

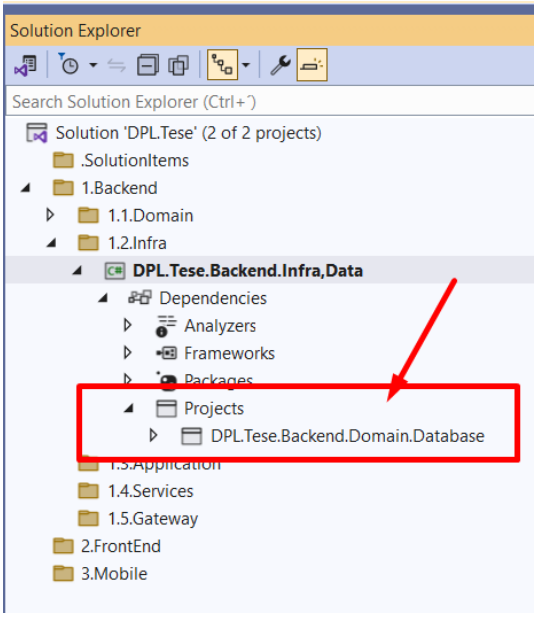


Figura 56 – Resultado da adição da referência ao projeto *Domain*

No projeto Data decidimos criar duas pastas distintas para cada ORM, Figura 57. Uma vez que iremos utilizar as duas, deveremos separar devidamente responsabilidades. Deste modo, quando existir alguma alteração ou correção, saberemos exatamente a localização, uma vez que na parte de criação, alteração e eliminação das tabelas é feito pela EntityFrameworkCore e as operações de criação,

consulta, alteração e eliminação dos registos será feita pelo Dapper. O nome das pastas da raíz são:

- Dapper.
- EntityFrameWorkCore.

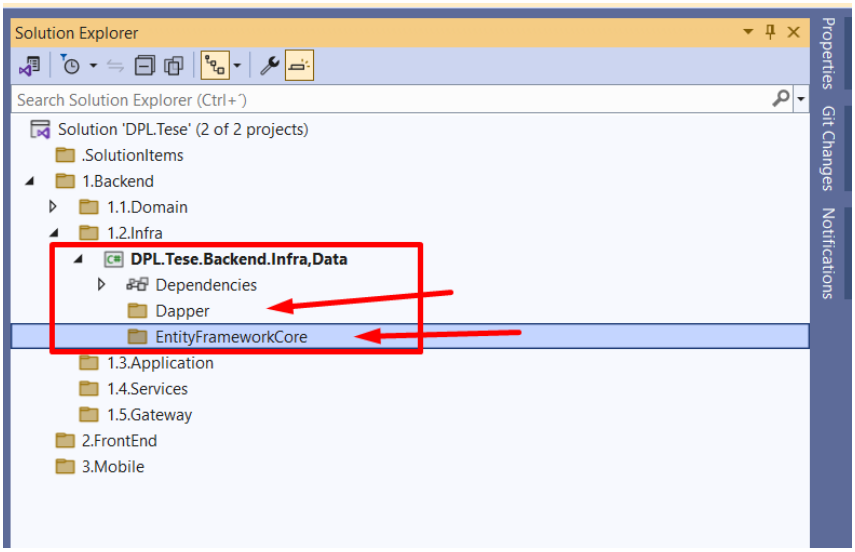


Figura 57 – Organização das pastas no projeto Data para os dois ORMs

Dentro da pasta Dapper ainda subdividimos em duas pastas, *UnitOfWork* e *Repository,* Figura 58. A *UnitOfWork* irá conter as classes e as *interfaces* bases de onde as restantes unidades de trabalho irão herdar. Por unidade de trabalho entende-se todas as interações feitas numa base de dados a cada pedido e é terminada quando é validada com sucesso a transação na base de dados. Por exemplo, um pedido pode conter duas instruções: apagar numa tabela e inserir noutra tabela. Estas duas ações são executadas na mesma unidade de trabalho e terminadas com um "*commit*" na base de dados. Neste caso a unidade de trabalho teve duas interações numa base de dados durante uma transação. A subpasta *Repository* irá conter as classes e as interfaces bases de onde os restantes repositórios irão herdar. Entende-se por repositório a representação das interações efetuadas numa dada tabela da base de dados, tais como criação, consulta, alteração e eliminação.

Dentro da pasta *EntityFrameworkCore* ainda conseguimos subdividir em duas pastas: *DBContext* e *Mapping,* Figura 58. A pasta DBContext irá conter as classes que irá representar a ligação para cada Sistema de Gestão de Base de Dados. No nosso caso apenas iremos ter uma para o *SQL Server,* mas caso no futuro seja necessário teremos que criar as restantes classes que herdam do *DBContext* da *EntityFrameworkCore* para cada provedor. A pasta *Mapping* será responsável por conter o mapeamento do tipo de dados entre as classes que representam cada

tabela na base de dados e o respetivo tipo de dados na base de dados. Na *EntityFramworkCore* existe a possibilidade de efetuar esta operação de duas formas: uma recorrendo à colocação de atributos em cima de cada propriedade da classe que representa cada tabela no projeto *Domain,* outra recorrendo à *Fluent* API da *EntityFrameworkCore* para fazer o mapeamento. Decidiu-se utilizar esta última por nos dar controlo total do mapeamento e por ser capaz de reconhecer cada classe a partir de comandos do *Refletion* recorrendo à identificação da *Interface* que utiliza, não sendo necessário adicionar uma a uma ao *DBContext*.

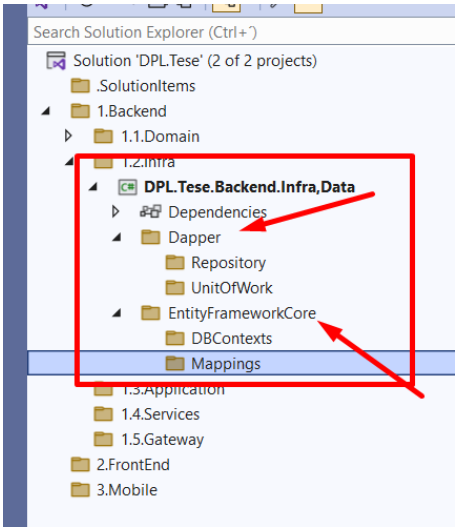


Figura 58 – Organização das subpastas dentro da pasta de cada ORM

Na pasta *Repository* dentro da pasta *Dapper* foi criado uma interface que conterá os cabeçalhos dos métodos a serem desenvolvidos na classe *BaseRepository*. A classe *BaseRepository* será responsável por ter os métodos inserir, selecionar, alterar e apagar, numa tabela de base de dados e como estes métodos são sempre idênticos em todos os repositórios, todos os repositórios irão herdar desta classe.

- **IBaseRepository:** Nesta interface, Figura 59, foram definidos:
  - Propriedades:
    - **QuerySelectByID:** Conterá a *string* da consulta para selecionar dados da tabela específica, localizada na classe filha que representa a tabela e que herda da *BaseRepository*. Como indica, a consulta irá devolver os dados correspondentes a um ID passado pelo programador.
    - **QuerySelectByWhereAndOrderBy**: Conterá a *string* da consulta para selecionar da tabela específica na classe filha que representa a tabela e que herda da BaseRepository. Como o nome indica, a consulta irá devolver os dados por uma clausula Where e OrderBy passada pelo programador.

- Métodos:
  - **InsertAsync:** Método assíncrono que irá inserir dados numa dada tabela.
  - **UpdateAsync:** Método assíncrono que irá atualizar os dados de uma tabela.
  - **DeleteByIDAsync:** Método assíncrono que irá apagar dados de uma tabela pelo respetivo ID.
  - **DeleteByWhereAsync:** Método assíncrono que irá apagar dados de uma tabela segundo uma clausula where.
  - **DeleteByWhereAndReturnIDsAsync:** Método assíncrono que irá apagar dados de uma tabela segundo uma cláusula where mas devolve os Ids dos registos apagados.
  - **SelectByIDAsync:** Método que irá efetuar uma consulta na tabela e devolver as linhas encontradas segundo o ID passado. Este método utiliza a propriedade QuerySelectByID anteriormente criado para saber que consulta executar.
  - **SelectByWhereAndOrderByAsync:** Método assíncrono que irá efetuar uma consulta na tabela e devolver as linhas encontradas segundo o parâmetro where e ordenado segundo o parâmetro OrderBy. Este método utiliza a propriedade QuerySelectByWhereAndOrderBy anteriormente definida para saber que consulta executar.

```
using DPL.Tese.Backend.Domain.Database.Shared;

namespace DPL.Tese.Backend.Infra_Data.Dapper.Repository
{
    1 reference
    public interface IBaseRepository<Entity> where Entity : BaseEntity
    {
        3 references
        public string QuerySelectAll {  get; set; }

        6 references
        public string QuerySelectByID { get; set; }

        7 references
        public string QuerySelectByWhereAndOrderBy { get; set; }

        1 reference
        Task<Entity> InsertAsync(IList<KeyValuePair<string, object>> Values);

        1 reference
        Task<int> UpdateAsync(IList<KeyValuePair<string, object>> Values, string? Where);

        1 reference
        Task<int> DeleteByIDAsync(Guid ID);

        1 reference
        Task<int> DeleteByWhereAsync(string Where);

        1 reference
        Task<IEnumerable<int>> DeleteByWhereAndReturnIDsAsync(string Where);

        1 reference
        Task<IEnumerable<DTOReturn>?> SelectAllAsync<DTOReturn>();

        1 reference
        Task<IEnumerable<DTOReturn>?> SelectByIDAsync<DTOReturn>(Guid ID, string? OrderBy);

        1 reference
        Task<IEnumerable<DTOReturn>?> SelectByWhereAndOrderByAsync<DTOReturn>(string Where, string? OrderBy);
    }
}
```

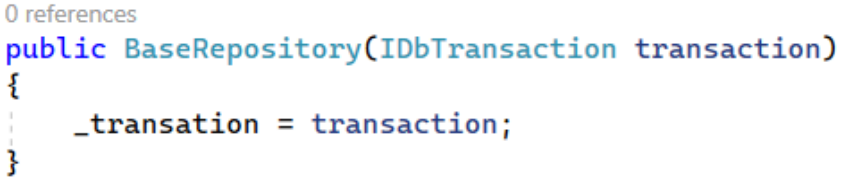
Figura 59 - Definição da interface do repositório base

A classe BaseRepository, Figura 60, herda da interface IBaseRepository e implementa os métodos da interface.

```
using Dapper;
using DPL.Tese.Backend.Domain.Database.Shared;
using System.Data;

namespace DPL.Tese.Backend.Infra_Data.Dapper.Repository
{
    1 reference
    public abstract class BaseRepository<Entity> : IBaseRepository<Entity> where Entity : BaseEntity
    {
        private readonly IDbTransaction _transation;

        3 references
        public string QuerySelectAll { get; set; } = string.Empty;
        6 references
        public string QuerySelectByID { get; set; }= string.Empty;
        7 references
        public string QuerySelectByWhereAndOrderBy { get; set; } = string.Empty;

        0 references
        public BaseRepository(IDbTransaction transaction)...

        1 reference
        public async Task<Entity> InsertAsync(IList<KeyValuePair<string, object>> Values)...

        1 reference
        public async Task<int> UpdateAsync(IList<KeyValuePair<string, object>> Values, string? Where)...

        #region DELETEs

        1 reference
        public async Task<int> DeleteByIDAsync(Guid ID)...

        1 reference
        public async Task<int> DeleteByWhereAsync(string Where)...

        1 reference
        public async Task<IEnumerable<int>> DeleteByWhereAndReturnIDsAsync(string Where)...

        #endregion

        #region SELECTs

        1 reference
        public async Task<IEnumerable<DTOReturn>?> SelectAllAsync<DTOReturn>()...

        1 reference
        public async Task<IEnumerable<DTOReturn>?> SelectByIDAsync<DTOReturn>(Guid ID, string? OrderBy)...

        1 reference
        public async Task<IEnumerable<DTOReturn>?> SelectByWhereAndOrderByAsync<DTOReturn>(string Where, string? OrderBy)...

        #endregion

    }
}
```

Figura 60 – Definição da *class* do repositório base

No construtor da classe passa-se a transação, Figura 61, para podermos identificar em que transação desejamos efetuar a operação.



```
0 references
public BaseRepository(IDbTransaction transaction)
{
    _transation = transaction;
}
```

Figura 61 – Injeção da transação no construtor do repositório base

No método InsertAsync recebemos por parâmetro uma lista de uma conjunto identificador/valor para identificar qual a coluna e respetivo valor na inserção do novo registo, Figura 62. Identificamos qual é a entidade que tem o nome idêntico na tabela e construímos o SQL para a inserção de valores.

```
7 references
public async Task<TEntity> InsertAsync<TEntity>(IList<KeyValuePair<string, object>> Values)
{
    TEntity? result = default(TEntity);
    string FieldsInsert = string.Empty;
    string ValuesInsert = string.Empty;

    string strSQL = "insert into " + typeof(Entity).Name + " ({0}) "
        + "OUTPUT INSERTED.*"
        + " values ({1})";


    Values.Add(new KeyValuePair<string, object>(Fields.CreateAt, DateTime.Now));

    foreach (KeyValuePair<string, object> KeyValue in Values)
    {
        FieldsInsert += KeyValue.Key + ",";
        ValuesInsert += "@" + KeyValue.Key + ",";
    }

    FieldsInsert = FieldsInsert.Substring(0, FieldsInsert.Length - 1);
    ValuesInsert = ValuesInsert.Substring(0, ValuesInsert.Length - 1);

    strSQL = strSQL.Replace("{0}", FieldsInsert).Replace("{1}", ValuesInsert);

    result = await _transation.Connection.QuerySingleAsync<TEntity>(strSQL, Values, _transation);

    return result;
}
```

Figura 62 – Método genérico para inserir na base de dados

No método UpdateAsync recebemos por parâmetro uma lista de um conjunto identificador/valor para identificar, qual a coluna e respetivo valor a atualizar segundo uma cláusula Where, também passada por parâmetro. Identificamos qual é a entidade que tem o nome idêntico na tabela, Figura 63, e construímos a clausula SQL para atualizar os valores. Retornamos o número de linhas afetas.

```
public async Task<int> UpdateAsync(IList<KeyValuePair<string, object>> Values, string? Where)
{
    int rowsAffected = 0;

    string strSQL = "UPDATE " + typeof(Entity).Name + " SET ";

    Values.Add(new KeyValuePair<string, object>(Fields.LastModifiedAt, DateTime.Now));

    foreach (KeyValuePair<string, object> KeyValue in Values)
        strSQL += KeyValue.Key + " = @" + KeyValue.Key + ", ";

    strSQL = strSQL.Substring(0, strSQL.Length - 2);

    if (!string.IsNullOrEmpty(Where))
    {
        strSQL += " Where " + Where;
    }

    rowsAffected = await _transation.Connection.ExecuteAsync(strSQL, Values , _transation);

    return rowsAffected;
}
```

Figura 63 – Método genérico para atualizar um registo na base de dados

No método DeleteByIDAsync recebemos um ID do tipo Guid, Figura 64, e construímos o SQL de modo a apagar o(s) registo(s) no final devolve o número de linhas afetadas.

```
0 references
public async Task<int> DeleteByIDAsync(Guid ID)
{

    int rowsAffected = 0;
    string strSQL = "Delete from " + typeof(Entity).Name + " Where ID=" + ID.ToString();

    rowsAffected = await _transation.Connection.ExecuteAsync(strSQL, null, _transation);

    return rowsAffected;
}
```

Figura 64 - Método para apagar um registo na base de dados a partir do ID

No método DeleteByWhereAsync recebemos um parâmetro do tipo string, Figura 64, que identifica a clausula para apagar. E devolve o número de linhas afetas.

```
1 reference
public async Task<int> DeleteByWhereAsync(string Where)
{
    int rowsAffected = 0;
    string strSQL = "Delete from " + typeof(Entity).Name + " where " + Where;

    rowsAffected = await _transation.Connection.ExecuteAsync(strSQL, null, _transation);

    return rowsAffected;
}
```

Figura 65 - Método para apagar um registo a partir de uma clausula *where*

No método DeleteByWhereAndReturnIDsAsync recebemos um parâmetro do tipo *string,* Figura 66, que identifica a cláusula para apagar. Devolve os Ids que foram apagados.

```
0 references
public async Task<IEnumerable<int>> DeleteByWhereAndReturnIDsAsync(string Where)
{
    IEnumerable<int>? IDs = null;

    string strSQL = "Delete from " + typeof(Entity).Name +
        " OUTPUT DELETED.ID" +
        " where " + Where;

    IDs = await _transation.Connection.QueryAsync<int>(strSQL, null, _transation);

    return IDs;
}
```

Figura 66 - Método para apagar registos a partir da clausula where e retorna os IDs

No método SelectAllAsync devolvemos os registos todos da base de dados, Figura 67.

```
1 reference
public async Task<IEnumerable<DTOReturn>?> SelectAllAsync<DTOReturn>()
{
    IEnumerable<DTOReturn>? result = null;

    if (!string.IsNullOrEmpty(QuerySelectAll))
        result = await _transation.Connection.QueryAsync<DTOReturn>(QuerySelectAll, null, _transation);

    return result;
}
```

Figura 67 - Método que seleciona todos os registos

No método SelectByIDAsync recebemos por parâmetro o ID, Figura 68, pelo qual iremos consultar os dados e caso seja desejável a forma de ordenação.

```
0 references
public async Task<IEnumerable<DTOReturn>?> SelectByIDAsync<DTOReturn>(Guid ID, string? OrderBy)
{
    IEnumerable<DTOReturn>? result = null;

    if (!string.IsNullOrEmpty(QuerySelectByID))
    {
        QuerySelectByID = QuerySelectByID + " Where ID = " + ID;

        if (!string.IsNullOrEmpty(OrderBy))
            QuerySelectByID += " Order By " + OrderBy;

        result = await _transation.Connection.QueryAsync<DTOReturn>(QuerySelectByID, null, _transation);
    }

    return result;
}
```

Figura 68 - Método que seleciona os registos a partir do ID

No Método SelectByWhereAndOrderByAsync recebemos por parâmetro a cláusula Where, Figura 69, e a cláusula de ordenação e devolvemos o resultado.

```
1 reference
public async Task<IEnumerable<DTOReturn>?> SelectByWhereAndOrderByAsync<DTOReturn>(string Where, string? OrderBy)
{
    IEnumerable<DTOReturn>? result = null;

    if (!string.IsNullOrEmpty(QuerySelectByWhereAndOrderBy))
    {
        if (!string.IsNullOrEmpty(Where))
        {
            QuerySelectByWhereAndOrderBy = QuerySelectByWhereAndOrderBy + " Where " + Where;

            if (!string.IsNullOrEmpty(OrderBy))
                QuerySelectByWhereAndOrderBy = QuerySelectByWhereAndOrderBy + " Order By " + OrderBy;

            result = await _transation.Connection.QueryAsync<DTOReturn>(QuerySelectByWhereAndOrderBy, null, _transation);
        }
    }

    return result;
}
```

Figura 69 - Método que seleciona os registos a partir de uma clausula *where* e *orderby*

Em alguns dos métodos verificou-se que é utilizada uma subclasse estática denominada *Fields,* Figura 70, que representa os vários campos da tabela correspondentes na base de dados. Caso seja alterada a estrutura da tabela na base de dados, basta efetuar também a alteração nesta sub classe Fields e toda a aplicação continuará a funcionar.

```
using Dapper;
using DPL.Tese.Backend.Domain.Database.Shared;
using System.Data;

namespace DPL.Tese.Backend.Infra_Data.Dapper.Repository
{
    1 reference
    public abstract class BaseRepository<Entity> : IBaseRepository<Entity> where Entity : BaseEntity
    {
        private readonly IDbTransaction _transation;

        3 references
        public string QuerySelectAll { get; set; } = string.Empty;
        6 references
        public string QuerySelectByID { get; set; }= string.Empty;
        7 references
        public string QuerySelectByWhereAndOrderBy { get; set; } = string.Empty;

        0 references
        public BaseRepository(IDbTransaction transaction)...

        1 reference
        public async Task<Entity> InsertAsync(IList<KeyValuePair<string, object>> Values)...

        1 reference
        public async Task<int> UpdateAsync(IList<KeyValuePair<string, object>> Values, string? Where)...

        DELETEs

        SELECTs

        2 references
        public static class Fields
        {
            0 references
            public static string ID => "ID";
            0 references
            public static string CreateByUserID => "CreateByUserID";
            1 reference
            public static string CreateAt => "CreateAt";
            0 references
            public static string LastModifiedByUserID => "LastModifiedByUserID";
            1 reference
            public static string LastModifiedAt => "LastModifiedAt";
            0 references
            public static string RowsAffected => "RowsAffected";
            0 references
            public static string ApproveStatus => "ApproveStatus";
        }

    }
}
```

Figura 70 - Resultado da classe completa do repositório base

Na pasta UnitOfWork foram criadas uma interface com os cabeçalhos da classe, Figura 71. Esta interface é responsável apenas por iniciar a transação e por terminá-la com sucesso ou cancelar toda a transação, fazendo a base de dados voltar ao seu estado inicial, ou seja, antes de iniciar a transação. A classe BaseUnitOfWork será herdada pelas UnitOfWork necessárias e terá já implementados os métodos referentes às transações.

- **IBaseUnitOfWork**: Interface responsável por ter os métodos que inicializam e termina a transação.

- Métodos:
  - **BeginTransaction**: Método que indica que se vai iniciar uma transação com a base de dados: uma transação pode ser considerada uma ou várias ações sobre uma ou várias tabelas.
  - **Commit:** Método que indica o término da transação e guarda as alterações na base de dados.
  - **Rollback:** Método que serve para cancelar todas as operações efetuadas na base de dados desde início da transação.

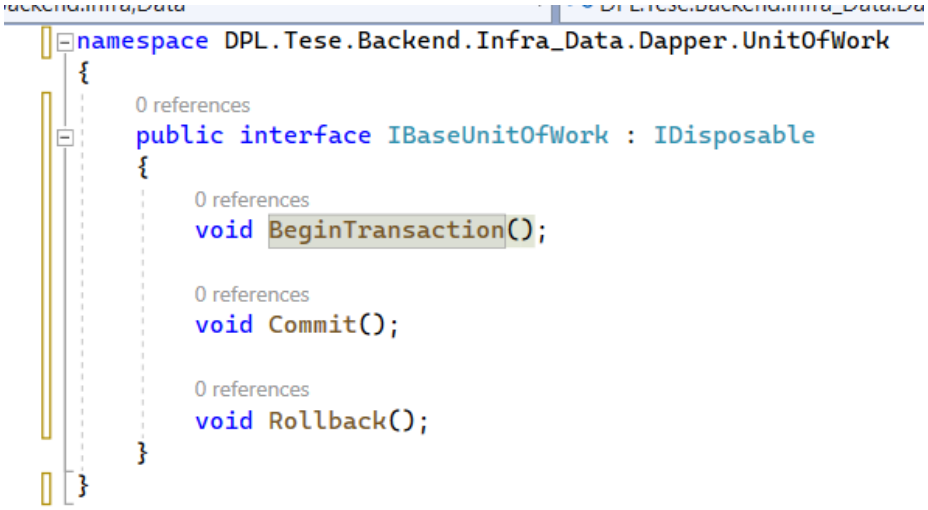


Figura 71 - Definição da Interface da unidade de trabalho

- **BaseUnitOfWork:** Classe que implementa os métodos da interface para a unidade de trabalho, Figura 72.
  - Propriedades:
    - **_connection:** Variável interna responsável pela conexão à base de dados;
    - **_Transation:** Variável responsável pelo estado da transação;
    - **_dispose:** Variável responsável pela destruição da alocação de memória da classe;
  - Métodos:
    - **BeginTransaction:** Responsável por iniciar a transação na base de dados.
    - **Commit:** Responsável por validar todas a operações dentro da transação.
    - **Rollback:** Responsável por anular todas as operações efetuadas dento transação.

```
using Microsoft.Data.SqlClient;
using System.Data;

namespace DPL.Tese.Backend.Infra_Data.Dapper.UnitOfWork
{
    2 references
    public class BaseUnitOfWork : IBaseUnitOfWork
    {
        protected IDbConnection _connection;
        protected IDbTransaction? _transation = null;
        private bool _dispose;

        0 references
        public BaseUnitOfWork(string connectionString)
        {
            _connection = new SqlConnection(connectionString);
            _connection.Open();
            BeginTransaction();
        }

        2 references
        public void BeginTransaction()
        {
            _transation = _connection.BeginTransaction();
        }

        1 reference
        public void Commit()
        {
            try
            {
                if (_transation != null)
                    _transation.Commit();
            }
            catch (Exception)
            {
                if (_transation != null)
                    _transation.Rollback();
                throw;
            }
            finally
            {
                _connection.Close();
            }

            Dispose();
        }

        1 reference
        public void Rollback()
        {
            try
            {
                if (_transation != null)
                    _transation.Rollback();
                Dispose();
            }
            catch (Exception)
            {

                throw;
            }
            finally
            {
                _connection.Close();
            }
        }

        2 references
        public void Dispose()
        {
            dispose(true);

            if (_connection != null)
                _connection.Close();

            GC.SuppressFinalize(this);
        }

        2 references
        private void dispose(bool disposing)
        {
            if (_dispose)
            {
                if (disposing)
                {
                    if (_transation != null)
                    {
                        _transation.Dispose();
                        _transation = null;
                    }
                    if (_connection != null)
                    {
                        _connection.Close();
                        _connection.Dispose();
                        _connection = null;
                    }
                }
                _dispose = true;
            }
        }

        0 references
        ~BaseUnitOfWork()
        {
            dispose(false);
        }
```

Figura 72 - Definição da *class* da unidade de trabalho

Estando a pasta *Dapper* configurada, iremos proceder à configuração das classes base que tirarão partido da *EntityFrameWorkCore* para gerir a criação a eliminação e alteração das tabelas e respetivos campos.

Dentro da pasta *DBContexts* iremos criar uma classe chamada de *SQLServerContext,* Figura 73, que caso não desenvolvêssemos também a parte de autenticação iria herdar da classe *DBContext* da *EntityFrameworkCore*. Mas uma vez que queremos implementar autenticação, teremos de instalar o *Nuget* da Microsoft *Identity* e herdar da Identity*DBContext* da Microsoft *Identity.* Uma vez que internamente já implementa todos os métodos de autenticação e a própria já tem os modelos base das tabelas necessárias na base de dados para gerir autenticações e permissões. Recorreu-se a esta livraria devido a ser a mais compatível com o tipo de projeto a desenvolver e está em constante atualização por parte da Microsoft de modo a seguir os melhores parâmetros de segurança no mundo Web.

Criamos a *class* SQLServerContext, Figura 74. que herda da class *IdentityDBContext* e depois criamos os modelos que herdam dos modelos da *Identity.* Para podermos personalizar com as nossas propriedades, teremos que na herança da classe identificar os novos modelos que serão utilizados.

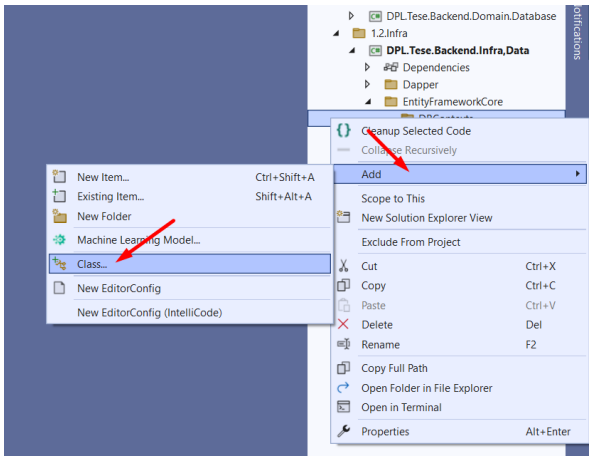


Figura 73 – Exemplo adição de uma classe ao projeto

```
using DPL.Tese.Backend.Domain.Database.Tables.Authentication;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

namespace DPL.Tese.Backend.Infra_Data.EntityFrameworkCore.DBContexts
{
    2 references
    public class SQLServerContext : IdentityDbContext<AuthUser, AuthRole, string,
        AuthUserClaim, AuthUser_AuthRole, AuthUserLogin, AuthRoleClaim, AuthUserToken>
    {
        0 references
        public SQLServerContext(DbContextOptions<SQLServerContext> options)
            :base(options)
        {
        }
    }
}
```

Figura 74 – Definição da *class* contexto para utilização da *EntityFrameworkCore*

Rescrevemos o método base *OnModelCreating,* Figura 75, de modo a não termos de recorrer ao método *DBSet* para adicionar, modelo a modelo, ao contexto e sim, poder-se fazer o mapeamento dos modelos para a base de dados de forma controlada. Iremos mostrar os mapeamentos mais abaixo. Para percorrer todos os modelos no *Domain* basta efetuar uma função com *Reflection* de modo a detetar quais os modelos a carregar.

```
0 references
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.ApplyConfigurationsFromAssembly(typeof(SQLServerContext).Assembly,
       t => t.GetInterfaces().Any(i => i.IsGenericType && i.GetGenericTypeDefinition() == typeof(IEntityTypeConfiguration<>) &&
           typeof(IBaseEntity).IsAssignableFrom(i.GenericTypeArguments[0])));

    base.OnModelCreating(builder);
}
```

Figura 75 - Carregamento das definições das entidades recorrendo ao Fluent API

Dentro da pasta *Mapping* iremos colocar as classes que irão fazer o mapeamento entre o tipo de dados do C# e o tipo de dados da base de dados. Este mapeamento é feito com recurso ao *Fluent* API da *EntityFrasmeWorkCore*. Dentro da pasta *Mapping* iremos fazer uma estrutura de pastas idênticas às do projeto *Domain,* Figura 76*,* de modo a ficarem emparelhadas.

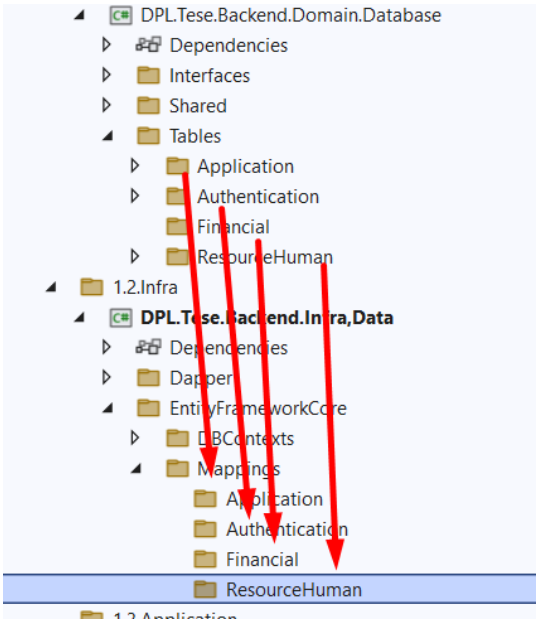
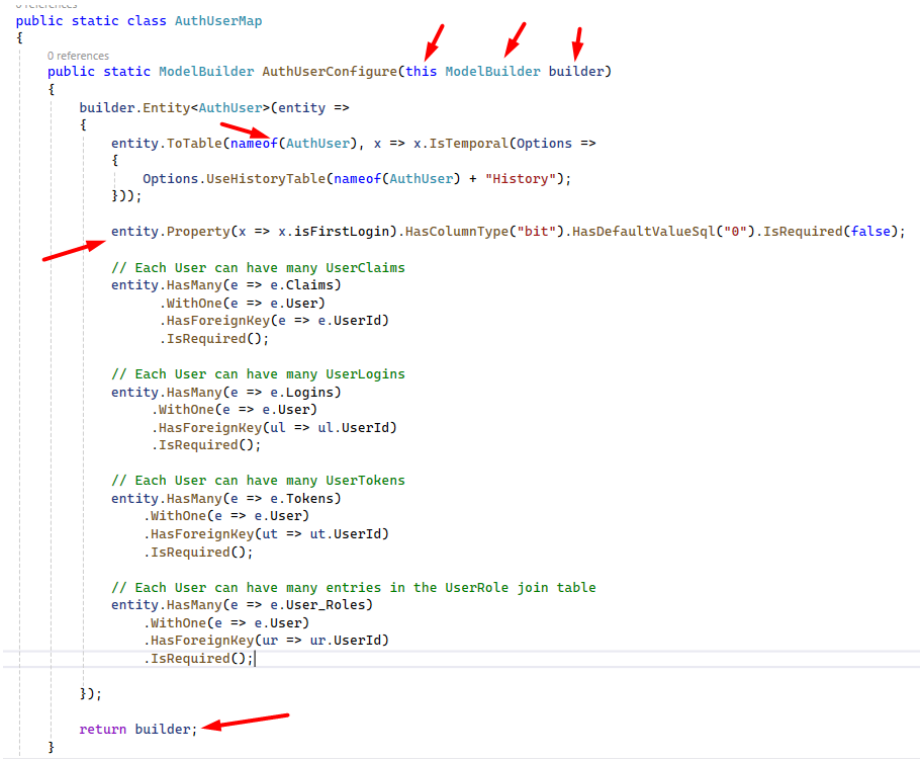


Figura 76 - Organização das subpastas de mapeamento da *Fluent* API da *EntityFrameworkCore*

Para generalizar a nomenclatura, foi decidido que as classes de mapeamento iriam ter o mesmo nome da entidade no *Domain* e que no final acrescentava-se Map ao nome: isto indica que estamos perante um mapeamento. Relativamente aos modelos já

existentes dentro do *Nuget* Microsoft *Identity*, não é possível efetuar um mapeamento idêntico às nossas entidades, recorrendo à interface *IEntityTypeConfiguration.* A única solução encontrada foi fazer uma classe extensão do *ModelBuilder*, estática e, após identificar a entidade, parametrizá-la à posteriori depois de carregado.

Exemplifica-se seguidamente a extensão para mapeamento da tabela AuthUser, Figura 77, que herda e substitui a *IdentityUser. N*este exemplo apenas acrescentamos um atributo para saber se é o primeiro login efetuado.



```
public static class AuthUserMap
{
    0 references
    public static ModelBuilder AuthUserConfigure(this ModelBuilder builder)
    {
        builder.Entity<AuthUser>(entity =>
        {
            entity.ToTable(nameof(AuthUser), x => x.IsTemporal(Options =>
            {
                Options.UseHistoryTable(nameof(AuthUser) + "History");
            }));

            entity.Property(x => x.isFirstLogin).HasColumnType("bit").HasDefaultValueSql("0").IsRequired(false);

            // Each User can have many UserClaims
            entity.HasMany(e => e.Claims)
                  .WithOne(e => e.User)
                  .HasForeignKey(e => e.UserId)
                  .IsRequired();

            // Each User can have many UserLogins
            entity.HasMany(e => e.Logins)
                 .WithOne(e => e.User)
                 .HasForeignKey(ul => ul.UserId)
                 .IsRequired();

            // Each User can have many UserTokens
            entity.HasMany(e => e.Tokens)
                .WithOne(e => e.User)
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            entity.HasMany(e => e.User_Roles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();

        });

        return builder;
    }
```

Figura 77 - Mapeamento da entidade AuthUser com Fluent API da EntityFrameworkCore

Na classe SQLServerContext, que diz respeito ao contexto da base de dados, temos de chamar o método que vai invocar o mapeamento da tabela AuthUser, Figura 78.

```
3 rererences
public class SQLServerContext : IdentityDbContext<AuthUser, AuthRole, string,
    AuthUserClaim, AuthUser_AuthRole, AuthUserLogin, AuthRoleClaim, AuthUserToken>
{
    0 references
    public SQLServerContext(DbContextOptions<SQLServerContext> options)
        :base(options)
    {
    }

    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.ApplyConfigurationsFromAssembly(typeof(SQLServerContext).Assembly,
           t => t.GetInterfaces().Any(i => i.IsGenericType && i.GetGenericTypeDefinition() =
               typeof(IBaseEntity).IsAssignableFrom(i.GenericTypeArguments[0])));

        base.OnModelCreating(builder);

        builder.AuthUserConfigure();
    }
}
}
```

Figura 78 – Invocação da extensão do mapeamento da entidade AuthUser

Ao iniciar os mapeamentos dos modelos criados e que se encontram no Domain, foi detetado que as entidades têm propriedades que se repetem neles todos, pois são herdados da entidade base. Desse modo, decidiu-se criar uma classe base de mapeamento, Figura 79, que irá mapear as propriedades comuns e nos mapeamentos de cada modelo apenas mapeamos os específicos de cada um. Deste modo, minimizamos a repetição de código e otimizamos a correção de erros.

Exemplo da classe base para mapeamento que mapeia as propriedades da classe base do domínio, Figura 80.

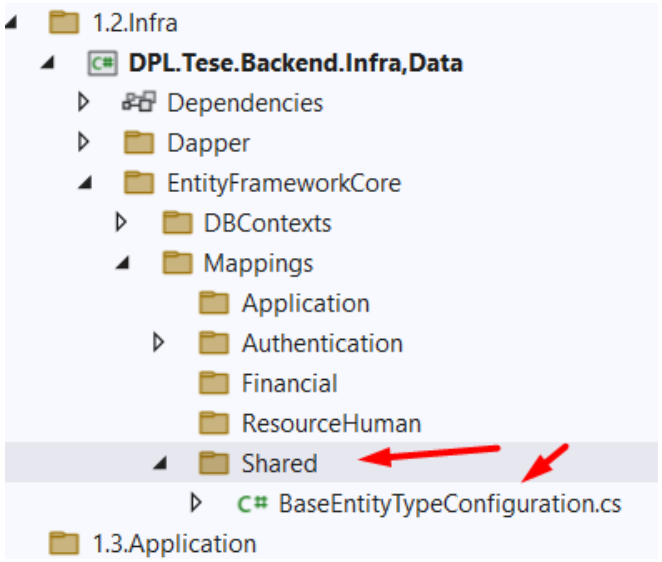


Figura 79 – Localização da class base de mapeamento da *Fluent* API *EntityFrameworkCore*

```
using DPL.Tese.Backend.Domain.Database.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace DPL.Tese.Backend.Infra_Data.EntityFrameworkCore.Mappings.Shared
{
    0 references
    public abstract class BaseEntityTypeConfiguration<TEntity> : IEntityTypeConfiguration<TEntity> where TEntity : BaseEntity
    {
        0 references
        public virtual void Configure(EntityTypeBuilder<TEntity> builder)
        {
            builder.HasKey(x => x.ID);
            builder.Property(x => x.isSystem).HasColumnType("bit").HasDefaultValueSql("0").IsRequired(false);
            builder.Property(x => x.ApproveStatus).HasColumnType("nvarchar(50)").HasDefaultValueSql("New").IsRequired(false);
            builder.Property(x => x.isDefault).HasColumnType("bit").HasDefaultValueSql("0").IsRequired(false);
            builder.Property(x => x.CreateAt).HasColumnType("datetime").HasDefaultValueSql("getdate()").IsRequired(true);
            builder.Property(x => x.CreateByUserID).HasColumnType("nvarchar(500)").IsRequired(true);
            builder.Property(x => x.isDeleted).HasColumnType("bit").HasDefaultValueSql("0").IsRequired(false);
            builder.Property(x => x.LastModifiedAt).HasColumnType("datetime").IsRequired(false);
            builder.Property(x => x.LastModifiedByUserID).HasColumnTy
            builder.Property(x => x.Version).HasColumnType("int").Has
        }
    }
}
```

class System.String
Represents text as a sequence of UTF-16 code units.

Figura 80 – Mapeamento da *class* base com *FLuent* API da *EntityFrameworkCore*

Exemplo da classe mapeamento do modelo RHPerson, Figura 81, que herda da class base anterior, Figura 82.

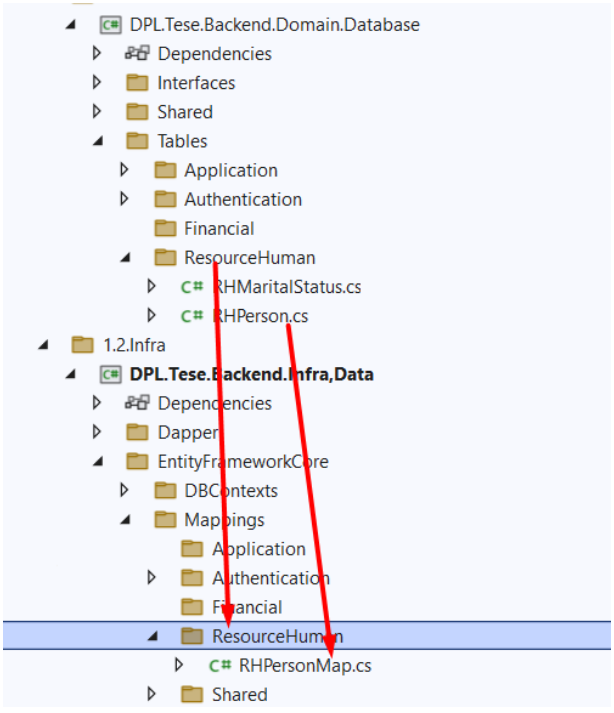


Figura 81 – Exemplo de organização das pastas para o mapeamento do Fluent API

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DPL.Tese.Backend.Infra_Data.EntityFrameworkCore.Mappings.ResourceHuman
{
    0 references
    public class RHPersonMap : BaseEntityTypeConfiguration<RHPerson>
    {
        2 references
        public override void Configure(EntityTypeBuilder<RHPerson> builder)
        {
            builder.ToTable(nameof(RHPerson), x => x.IsTemporal(true));

            builder.Property(x => x.Name).HasColumnType("nvarchar(500)").IsRequired(true);
            builder.Property(x => x.Mail).HasColumnType("nvarchar(250)").IsRequired(true);
            builder.Property(x => x.Male).HasColumnType("bit").IsRequired(false);
            builder.Property(x => x.MaritalStatusID).HasColumnType("bigint").IsRequired(false);

            builder.HasOne(x => x.MaritalStatus)
                   .WithMany(x => x.Persons)
                   .HasForeignKey(x => x.MaritalStatusID)
                   .IsRequired(true);

            base.Configure(builder);
        }
    }
}
```

Figura 82 - Exemplo do mapeamento da entidade RHPerson

Para todas as entidades criadas é necessário fazer o respetivo mapeamento para a base de dados. Deste modo consegue-se controlar todas as especificações da base de dados.

Se analisarmos a classe *SQLServerContext,* verificamos que existe uma linha de código que, através do *Reflection,* Figura 83, irá procurar todos os mapeamentos que herdam da classe *IEntityTypeConfiguration*.

```
0 references
protected override void OnModelCreating(ModelBuilder builder)
{
    builder.ApplyConfigurationsFromAssembly(typeof(SQLServerContext).Assembly,
        t => t.GetInterfaces().Any(i => i.IsGenericType && i.GetGenericTypeDefinition() == typeof(IEntityTypeConfiguration<>) &&
            typeof(IBaseEntity).IsAssignableFrom(i.GenericTypeArguments[0])));

    base.OnModelCreating(builder);

    builder.AuthUserConfigure();
}
```

Figura 83 - Exemplo de carregamento dinâmico utilizando *Reflection*

16.2.3 - DPL.TESE.BACKEND.GATEWAY.GATEWAY

Criou-se um projeto do tipo Web API *minimal* que irá ser responsável por ser o centro de ligação entre todos os serviços e interligar todas as livrarias e respetivos serviços, Figura 84. Esta API é *minimal* por ser apenas um *middleware* de encapsulamento para os restantes micro serviços.

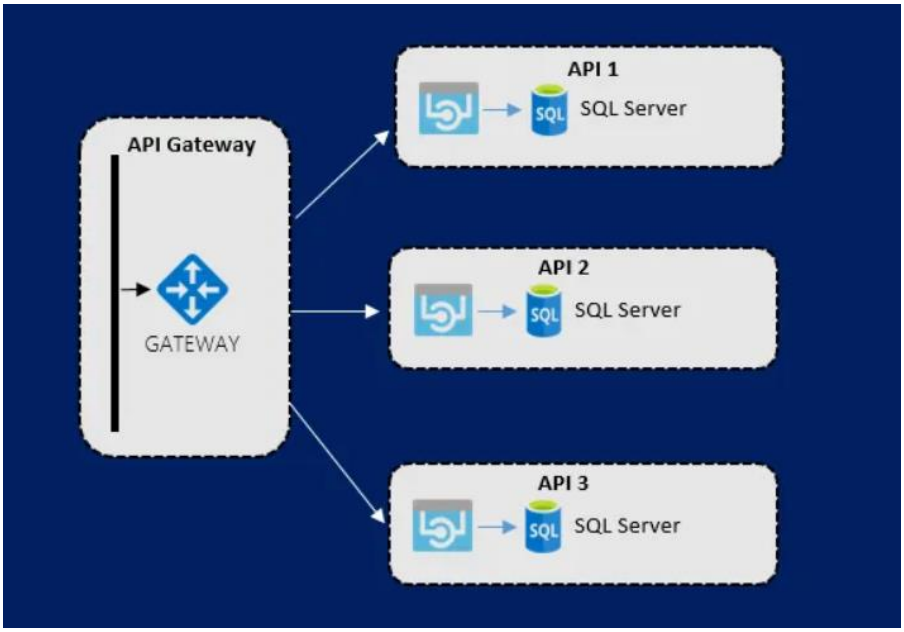


Figura 84 – Estrutura do sistema recorrendo apenas a *Gateway*[22]

Este projeto, Figura 85, será responsável por efetuar toda a comunicação entre os vários micro serviços, é neste projeto que será o ponto de ligação entre todos os projetos da plataforma.

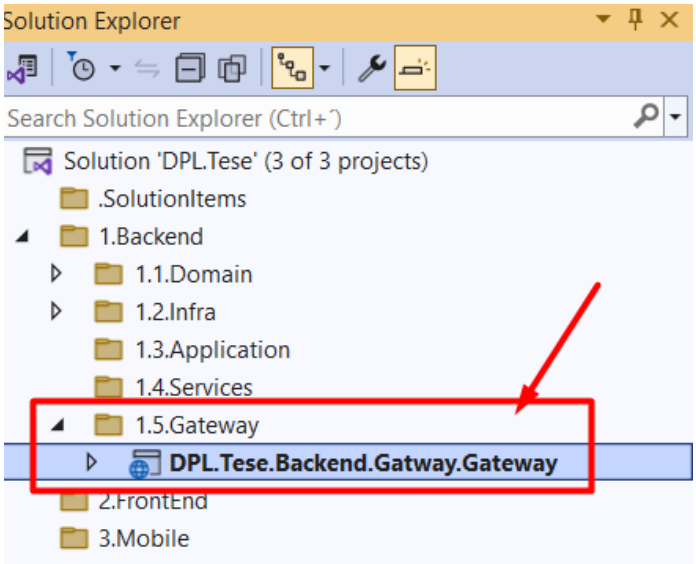


Figura 85 – Resultado da criação do projeto Gateway

[22] Retirado de: https://medium.com/xp-inc/criando-gateway-para-suas-api-s-com-aspnet-core-ocelot-e-docker-em-linux-21bd71ea6d94

Para construir o Gateway decidiu-se recorrer ao *NuGet* Ocelot, Figura 86, por ser de fácil aprendizagem, eficiente e de configuração minimalista.

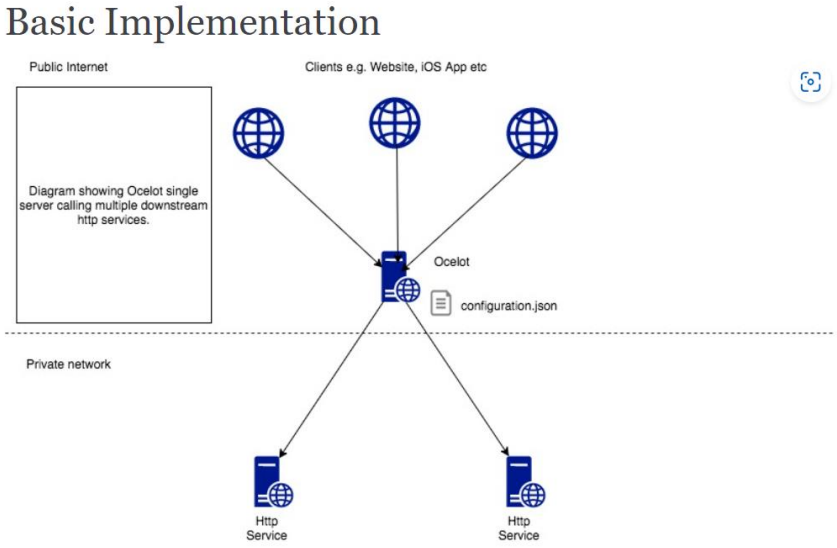


Figura 86 – Exemplo da implementação básica do *gateway* Ocelot[23]

Para configurar o Ocelot seguimos os seguintes passos:

- Instalamos o *NuGet* no projeto, Figura 87.

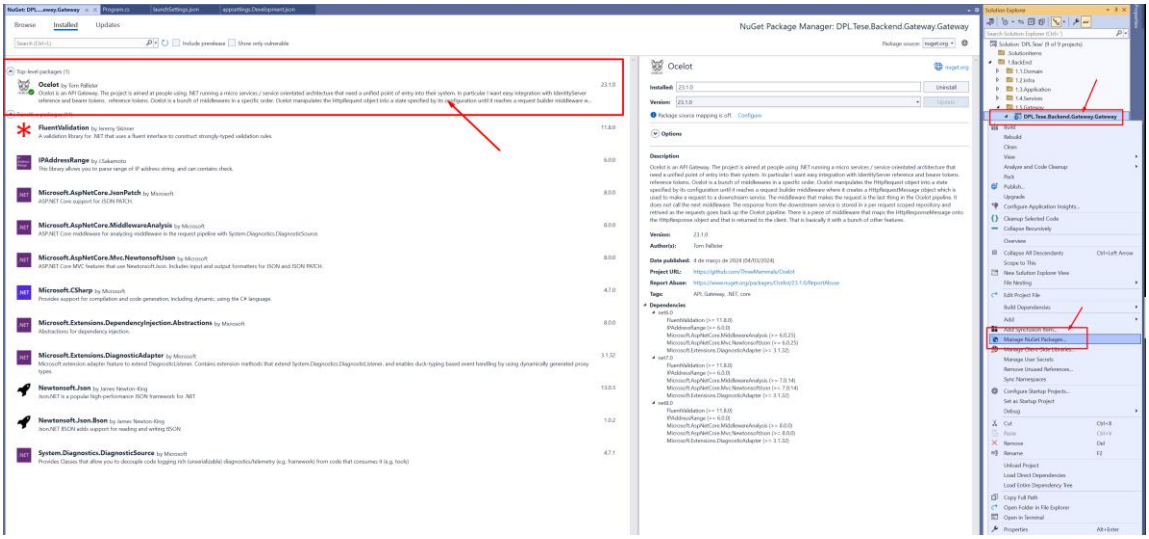


Figura 87 – Exemplo da instalação do Nuget Ocelot

[23] Retirado de: https://ocelot.readthedocs.io/en/latest/introduction/bigpicture.html

- Após a instalação do *NuGet*, foram criados os ficheiros JSON com as respetivas configurações do Ocelot. Para tal, adicionou-se na raiz do projeto, Figura 88. Três ficheiros, à semelhança do appsettings, que são um Ocelot.json, um Ocelot.Development.Json e um Ocelot.Production.Json, Figura 89. Cada um dos ficheiros irá conter as configurações correspondentes ao ambiente em que se irá trabalhar, Figura 90, ou seja, Ocelot.json será transversal a todos os ambientes, o Ocelot.Development.json será para o ambiente de desenvolvimento e o Ocelot.Production.json será utilizado para produção.

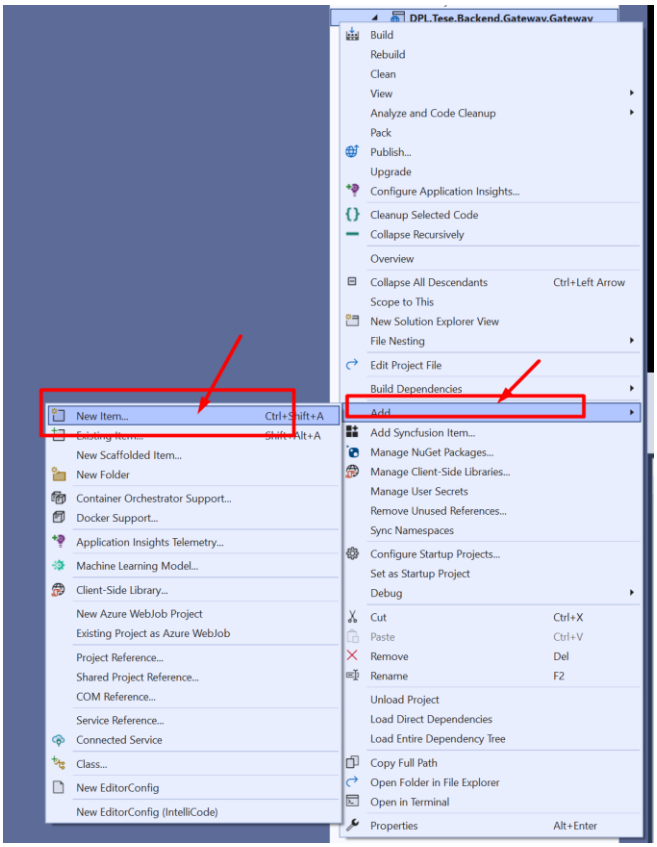


Figura 88 – Adicionar componente ao *gateway*

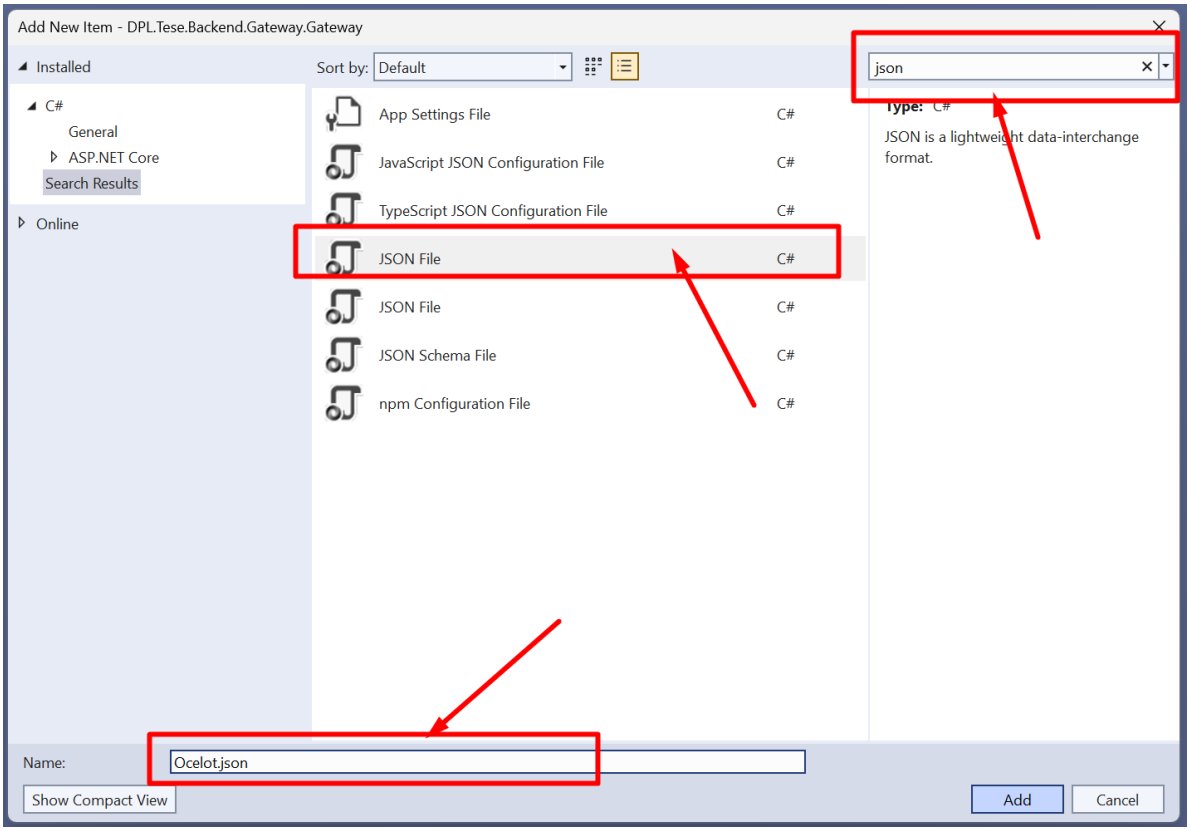


Figura 89 – Adicionar ficheiro json rotas ao *Gateway*

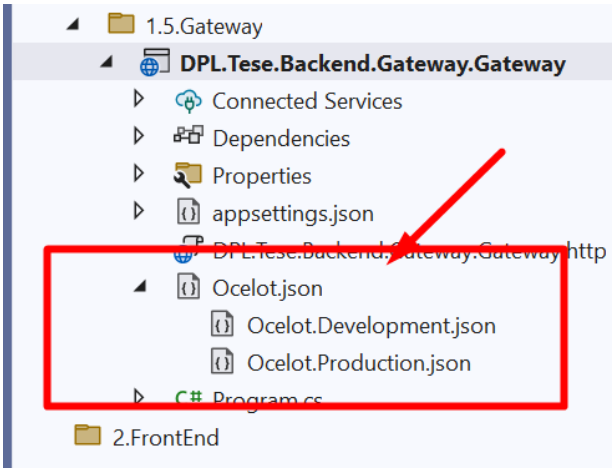


Figura 90 – Definição do ficheiro JSON para múltiplos ambientes

- Nos ficheiros Ocelot.<Ambiente>.json é necessário configurar os vários parâmetros, tais como:
  - **GlobalConfiguration, Figura 91:**
    - **BaseUrl:** Endereço de ponto único onde os clientes se irão ligar;

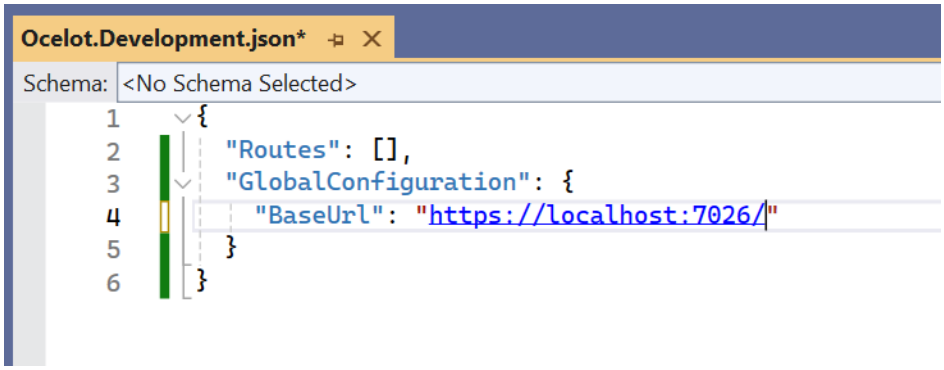


Figura 91 – Exemplo configuração global do JSON das rotas

- **Routes, Figura 92:** É uma lista de objetos com as configurações específicas de cada objeto, para onde deve ser redirecionado o pedido e que tem vários parâmetros:
  - **UpstramPathTemplate:** Restante caminho a ser adicionado ao caminho base que identifica o serviço a ser chamado, exemplo: "/auth/{Controller}/{Method}". O Auth é o serviço que desejamos chamar, a variável Controller é substituídoa pelo Controller que desejamos chamar e a variável Method é substituída pelo método que desejamos chamar.
  - **UpstreamHttpMethod:** Indica quais os métodos que poderemos chamar, tais como *Get*, *Post*, *Put* ou *delete*.
  - **DownstreamHostAndPort:** Identifica o endereço real da API que queremos consultar e a respetiva porta.
  - **DownstreamSchema:** Identifica o protocolo a utilizar.
  - **DownstreamPathTemplate:** Restante caminho a colocar no endereço real do serviço.

Figura 92 - Definição das rotas no JSON para os serviços

Após a configuração dos vários redireccionamentos nos ficheiros JSON, para cada serviço é necessário configurar no ficheiro Program.cs os serviços, Figura 93, de modo que, quando a API estiver a arrancar, esta mapeie todos os endereços. Para este passo, é necessário:

- Carregar os ficheiros JSON;
- Adicionar o serviço Ocelot ao contentor.
- Chamar a utilização do Ocelot para ficar à escuta.

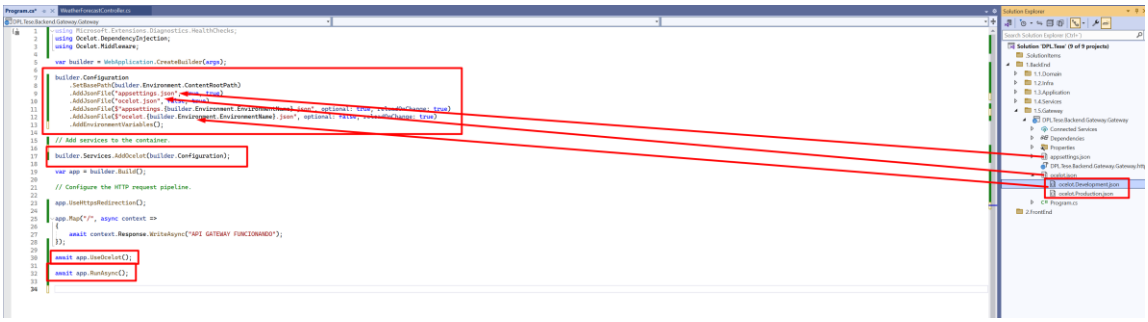


Figura 93 - configuração do arranque do *gateway* para Ocelot

Neste momento, temos o nosso Gateway pronto para começar a receber pedidos e os distribuir pelos respetivos serviços, sempre que seja necessário. Para criar um novo serviço basta ir aos ficheiros JSON do Gateway e adicionar. Vantagens de utilizar o gateway:

- Alteração dos endereços dos serviços, não tem implicações de alteração das aplicações nos clientes, ou atualização de aplicações móveis.
- Encapsulamento dos endereços verídicos dos serviços.
- Possibilidade de fazer múltiplas validações no gateway antes do pedido ser enviado para os serviços.

16.2.4 - DPL.TESE.BACKEND.INFRA.CROSSCUTTING

Criou-se um projeto do tipo "*Class Library*" que, tal como o nome indica, será transversal a todos os projetos que necessitem de utilizar classes e métodos que sejam necessários, Figura 94. Deste modo, dispensamos a necessidade de duplicar código. Como exemplo poderemos ter uma classe responsável por envio de mails. Pode conter os ficheiros JSON responsáveis pela organização dos menus, métodos que efetuem comunicação com API externas, tais como o Google Maps.

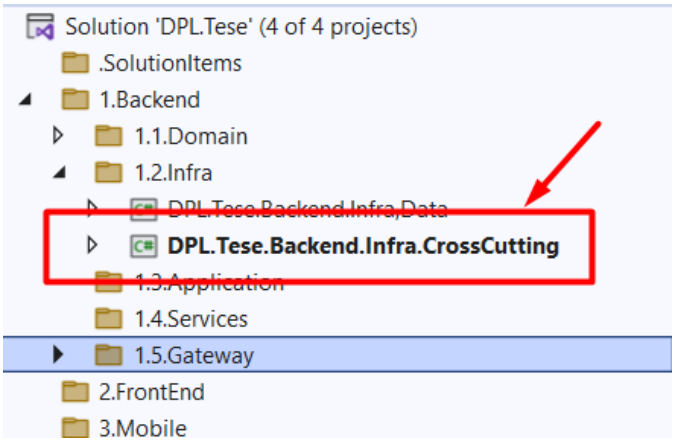


Figura 94 – Resultado da criação do projeto CrossCutting

De modo a automatizar o processo de registo dos serviços no contentor do Asp.net Core e sabendo que esta livraria é transversal a todos os projetos, inclusive aos dos serviços e uma vez que existem apenas três ciclos de vida para um serviço, que são Transient, Singleton e Scoped, decidimos criar nesta livraria três interfaces, uma para cada tipo. Deste modo poderemos de forma automática identificar cada serviço e registá-lo automaticamente, tirando partido do *Nuget* Scrutor, Figura 95.

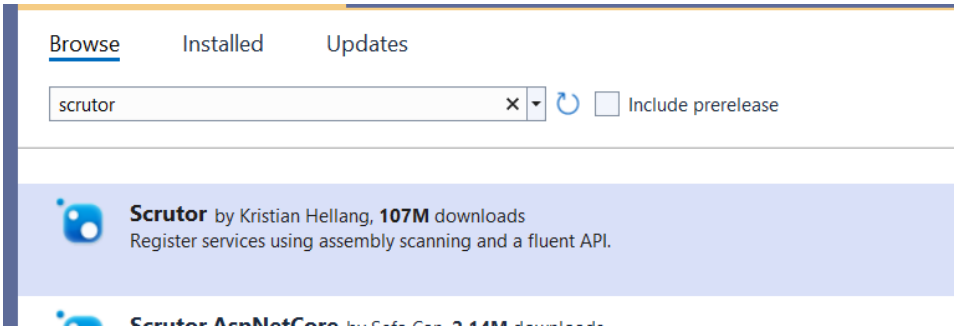


Figura 95 – Instalação do Nuget Scrutor

Para desenvolver este mecanismo procedemos da seguinte forma:

- Criamos uma pasta chamada Services na raiz da livraria. Dentro desta, criamos duas subpastas, uma chamada Interfaces que irá guardar as *interfaces* e outra chamada *Implementation* que irá guardar o desenvolvimento da classe, Figura 96.

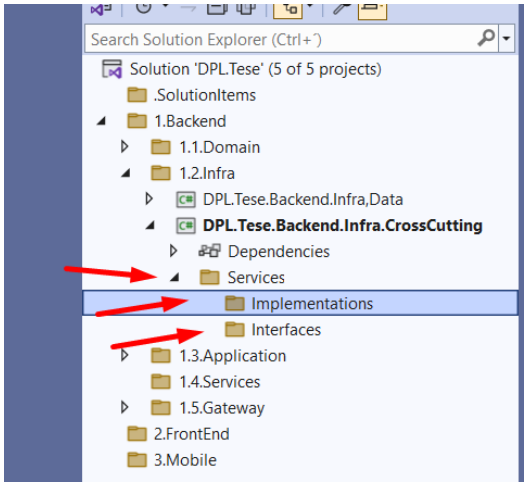


Figura 96 – Organização de pastas dentro do projeto *CrossCutting*

Dentro da pasta *interfaces* criamos uma subpasta chamada *ServicesRegistration* e dentro desta criamos três interfaces, uma chamada ITansientService, outra chamada ISingletonService e uma terceira chamada IScopedService, Figura 97. Um serviço *Transient*, é um serviço que é criado no instante do pedido e é destruído quando termina o pedido. Um serviço *Singleton* é criado quando é feito o primeiro pedido e já não é destruído. Um serviço *Scoped* é criado um por pedido e apenas é destruído quando todo o pedido for executado.

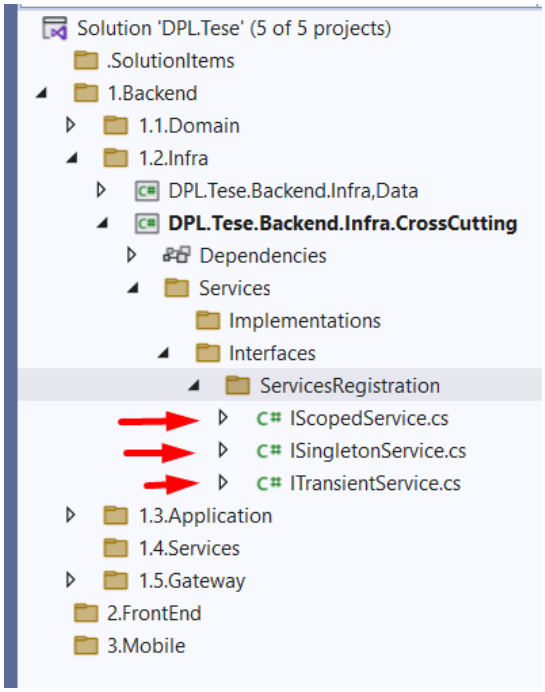


Figura 97 – Criação das interfaces para o ciclo de vida dos serviços

Deduzindo que no futuro, pudesse ser necessário implementar métodos obrigatórios, em todos os serviços, decidimos criar uma interface base, das quais os três tipos anteriores iriam herdar. Para isso, criamos uma pasta *shared* e dentro da pasta *shared* criamos a interface *BaseService,* Figura 98.

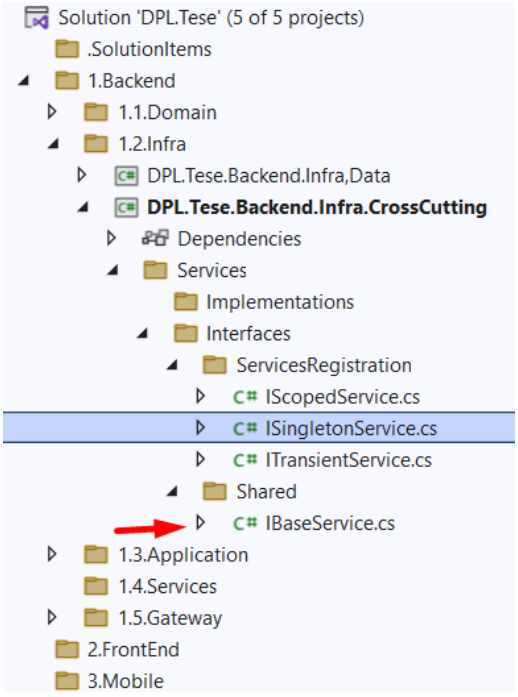


Figura 98 – Criação da interface genérica para todos os ciclos de vida

Alteramos todos os serviços do ServicesRegistration, Figura 99, para herdarem da IBaseService.

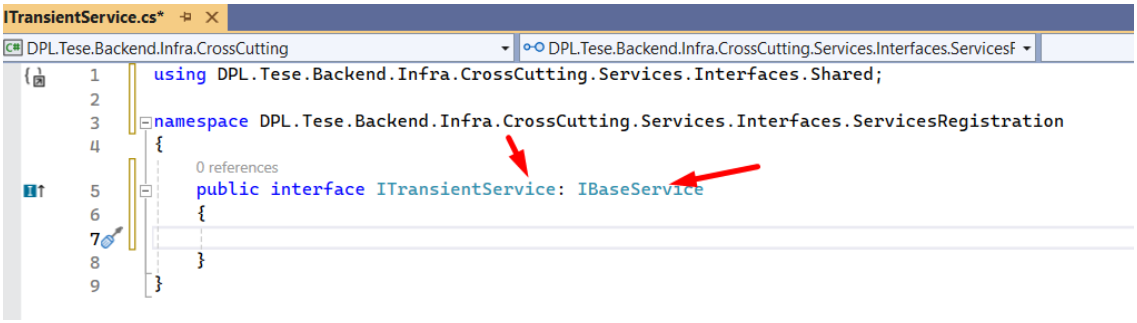


Figura 99 - Exemplo herança de uma interface ciclo de vida da interface genérica

Na pasta *Implementations* criamos uma pasta *shared* que ira ter as classes partilhadas, das quais as outras classes podem herdar e criamos uma classe para registar todos os serviços automaticamente, Figura 100, e que pode ser chamada em qualquer projeto.

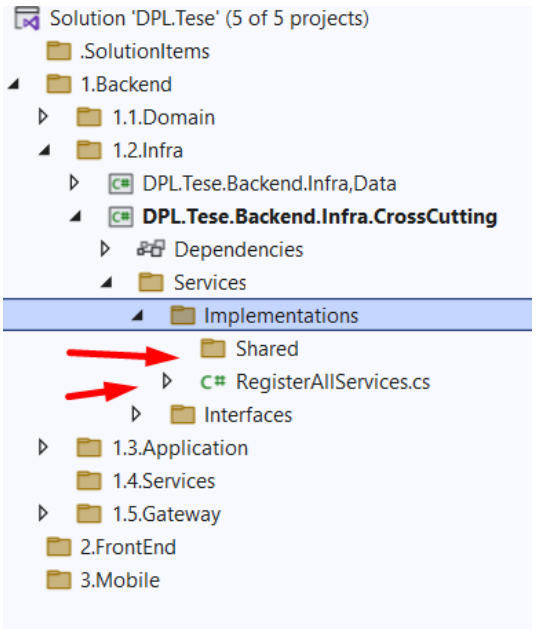


Figura 100 - Exemplo da localização da class que regista todos os ciclos de vida

Para conseguir injetar os serviços automaticamente também foi necessário instalar o *Nuget* Microsoft.Extensions.DependencyInjection e o Microsoft.Extensions.DependencyInjection.Abstrations, Figura 101.

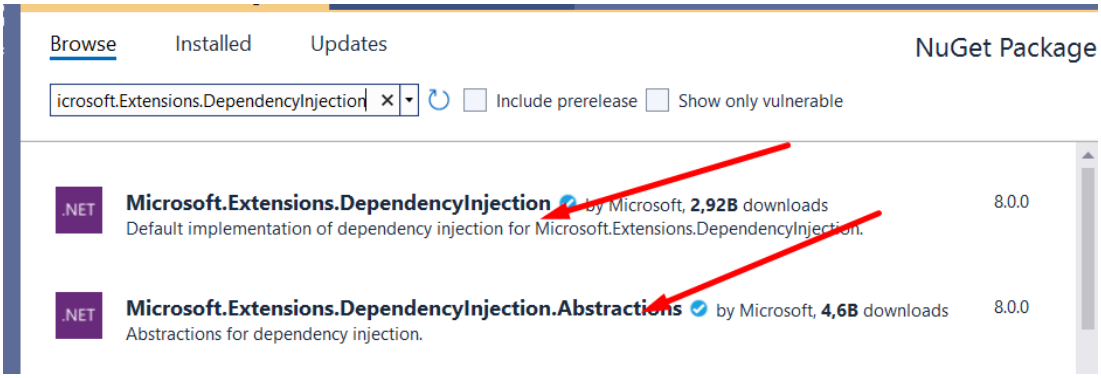


Figura 101 – Exemplo de instalação dos Nugets de injeção de dependência no projeto Corsscutting

Após a instalação dos *Nugets*, desenvolveu-se o método dentro da classe *RegisterAllServices* que irá ser um método estendido da interface *IServiceColletion,* Figura 101. Para isso foi necessário alterar a classe para ser estática, isto é, poder ser invocada sem ser instanciada. E todos os métodos dentro também serão estáticos. Com recurso ao *Scrutor* registamos os três tipos de "vida" dos serviços.

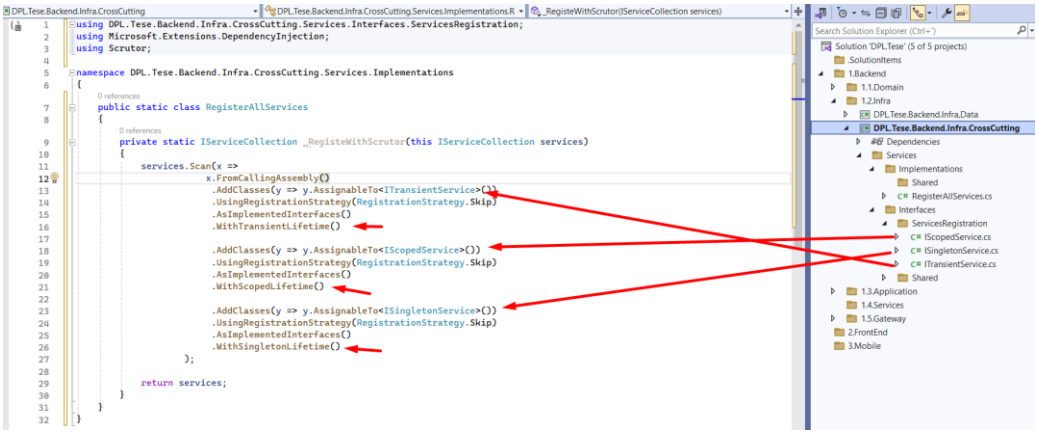


Figura 102 – Exemplo do código do Scrutor para registar os serviços globalmente

Desta forma basta as classes dos serviços herdarem das respetivas interfaces criadas, para que sejam injetadas no contentor de serviços.

16.2.5 - DPL.TESE.BACKEND.APPLICATION.DTOS

De modo a não expor a estrutura da base de dados para o exterior, foi necessário criar um projeto do tipo "Class Library" que terá as classes que representam os dados e que serão recebidos e devolvidos a partir de cada *view* do cliente, Figura 103.

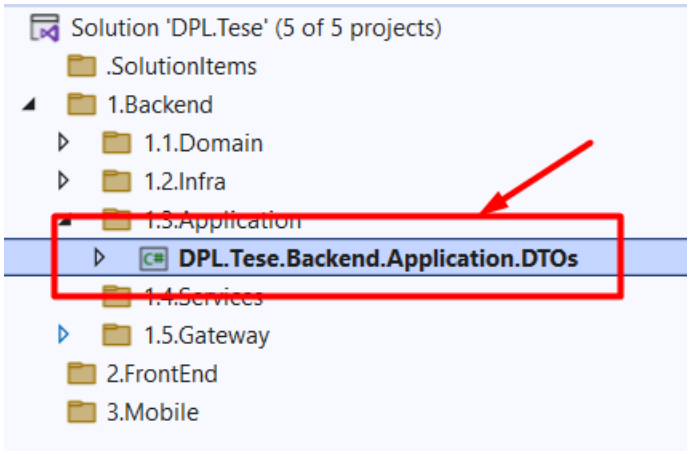


Figura 103 – Resultado da criação do projeto DTOs

Os DTO (Data Transfer Object) emergem como componentes cruciais para facilitar a comunicação eficiente entre sistemas. Os DTO representam uma prática comum em programação onde os objetos são criados para transportar dados ente subsistemas. Ao contrário das entidades de domínio, os DTOS são estruturas leves e geralmente contêm apenas dados relevantes para a transferência, evitando lógica de negócios complexa. Essa simplicidade torna-os ideais para otimizar a comunicação e reduzir a sobrecarga em operações de transferência de dados.

Decidiu-se que a organização de pastas para albergar os DTO serão idênticas à das subpastas das tabelas do projeto *Domain,* Figura104.

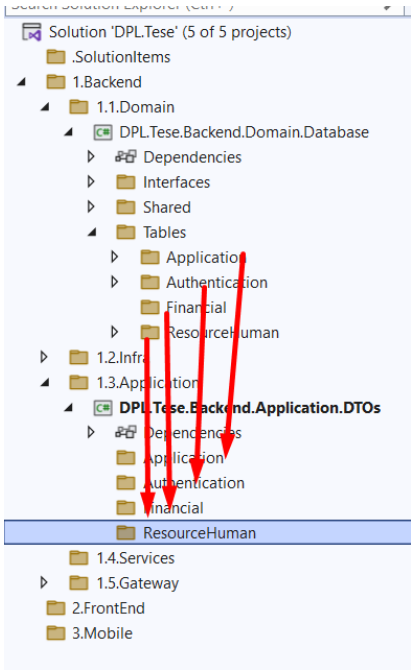


Figura 104 – Exemplo organização de pastas entre o Domain e o DTOs

Para uma melhor organização, ainda se decidiu criar uma subpasta para os vários DTO de cada tabela, Figura 105.

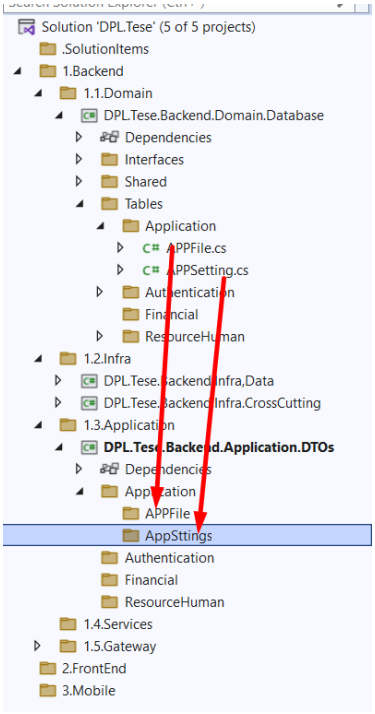


Figura 105 – Exemplo de organização das várias classes DTOs a partir do Domain

Como todos os DTO iriam ter alguns campos sempre em comum e de modo a não estar a repetir sempre o código, decidiu-se criar uma pasta chamada *Shared* e dentro desta criar um DTO base, Figura 106, dos quais os restantes herdariam. Foi ainda definido que no nome de cada DTO irá ter a terminação de DTO.

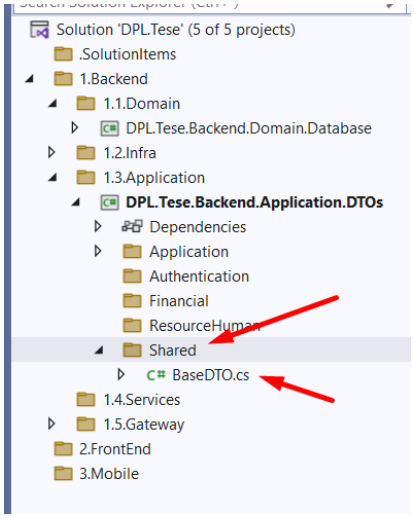


Figura 106 – Exemplo da localização da classe base do DTO, transversal aos restantes DTOs

A classe BaseDTO, Figura 107, será abstrata pois será herdada das outras e será definida sempre que tipo de dados é o ID.

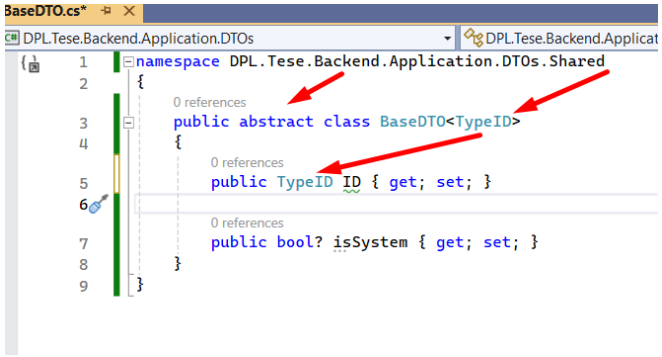


Figura 107 - Exemplo código do BaseDTO com tipo de dados do ID genérico

De modo a ter uma classe genérica e de possível expansão, criou-se uma interface, Figura 108 da qual a classe base abstrata herda, Figura 109.

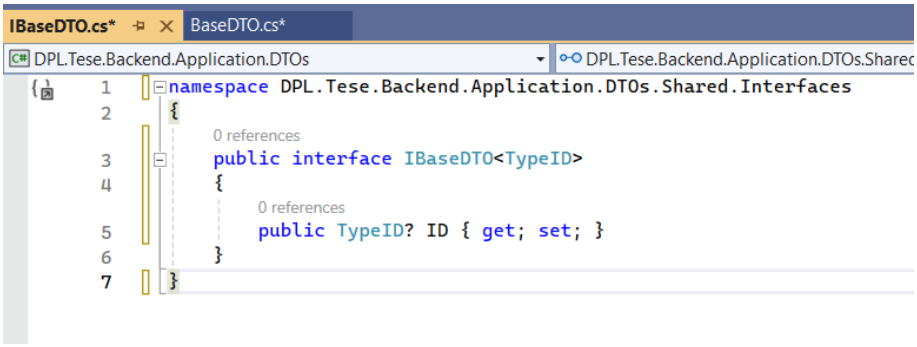


Figura 108 - Exemplo de interface genérica para a Base dos DTOs

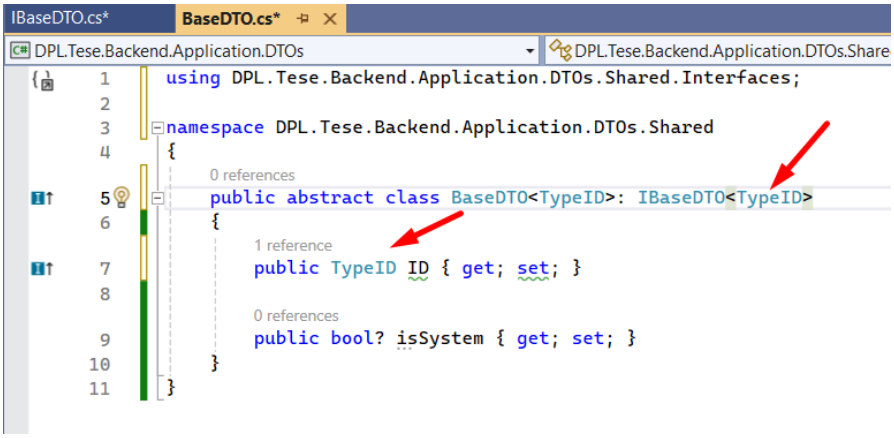


Figura 109 - Exemplo de herança da interface IBaseDTO para a BaseDTO

Todos os DTO da aplicação serão criados dentro desta livraria de modo a ficarem centralizados.

16.2.6 - DPL.TESE.BACKEND.SERVICES.<NAMESERVICE>

Na atualidade, a adoção de arquiteturas baseadas em micro serviços tem-se revelado uma peça-chave para alcançar a flexibilidade e agilidade necessárias no desenvolvimento de software. Ao enfrentar a complexidade inerente aos micro serviços, torna-se evidente que a comunicação eficiente entre as equipas de desenvolvimento é um fator determinante para o sucesso do projeto. Decidiu-se organizar os micro serviços de acordo com os módulos específicos de desenvolvimento, tais como, autenticação, financeiro, aplicacional e recursos humanos. Utilizaremos como exemplo demonstrativo o serviço de autenticação.

Criou-se um projeto do tipo Asp.net Core WebAPI e chamamos-lhe DPL.Tese.Backend.Services.AuthenticationService, Figura 110.

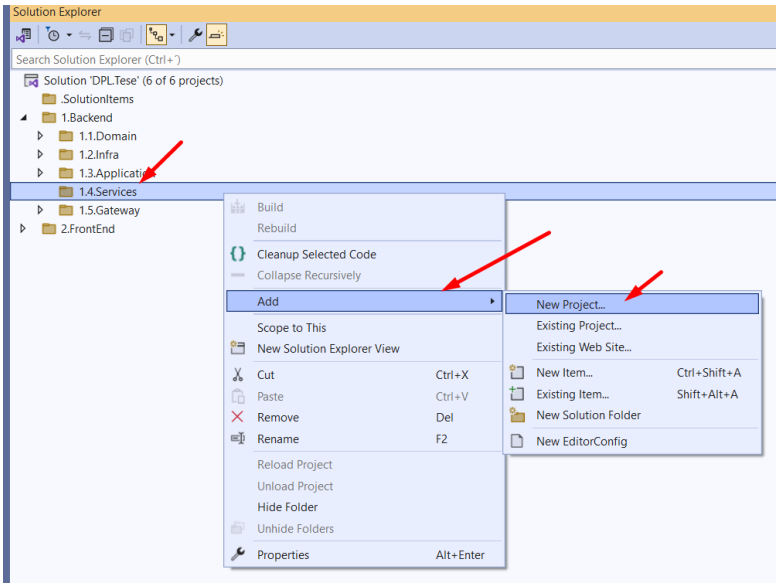


Figura 110 – Localização para a criação das APIs dos serviços

Escolhemos o *template* da Microsoft chamado ASP.NET Cores Web API, Figura 111.

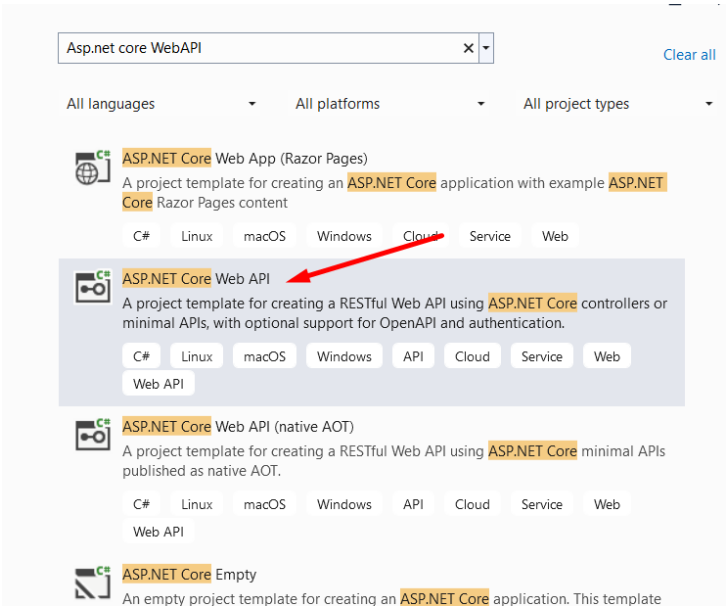


Figura 111 – Escolha do tipo de projeto para o serviço

Damos-lhe o nome de DPL.Tese.Backend.Services.AuthenticationService, Figura 112.

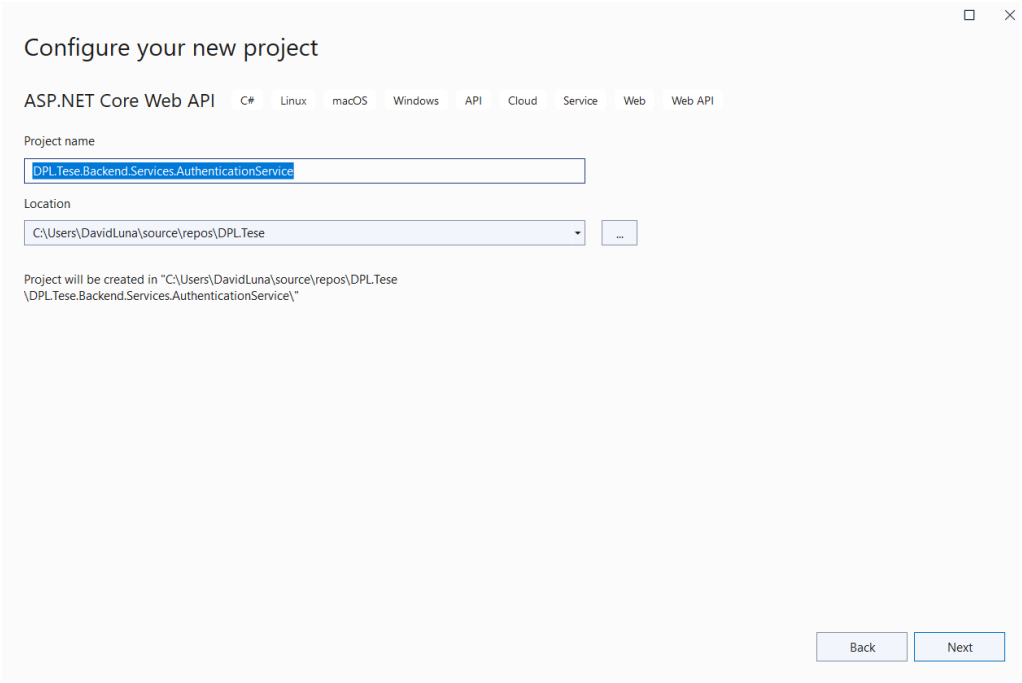


Figura 112 – Parametrização para criação do serviço AuthenticationService

Escolhemos a Framework .Net 8 e ativamos a funcionalidade da documentação e informamos que iremos utilizar *controllers,* Figura 113.

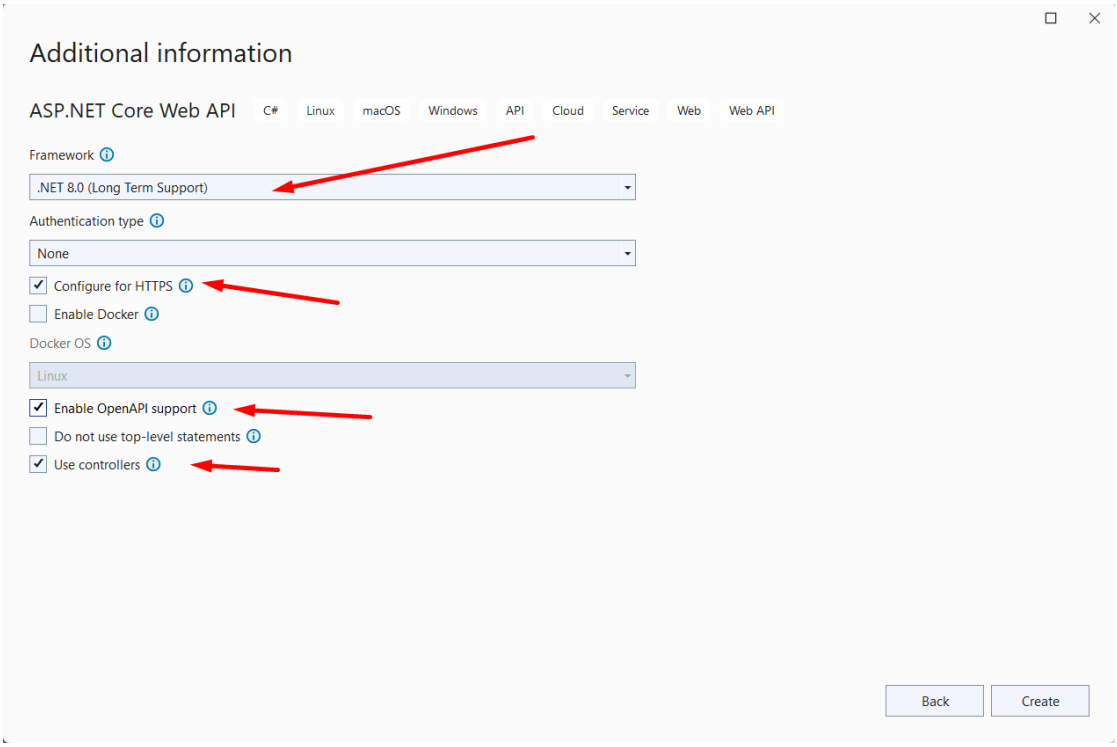


Figura 113 – Escolha dos componentes bases na criação do serviço AuthenticationService

Exemplo de quatro serviços criados, Figura 114.

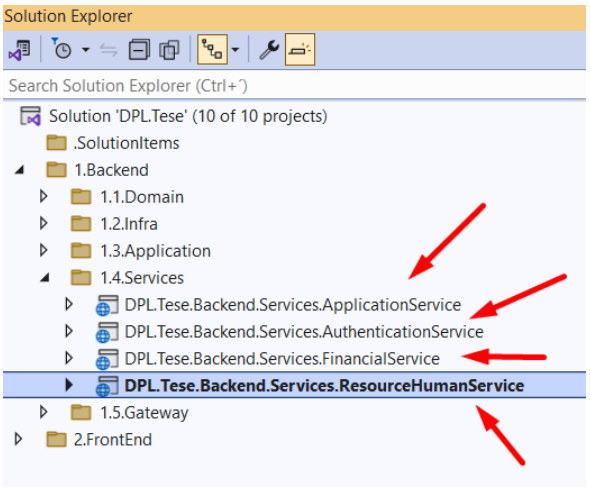


Figura 114 – Exemplo de quatro serviços, um para cada modulo

O serviço de autenticação irá gerir tudo o que tenha a ver com a autenticação, desde utilizadores, permissões entre outros. O serviço de autenticação é um projeto do tipo Web API e apenas terá as suas responsabilidades. Este serviço irá utilizar essencialmente os recursos do *NuGet* Microsoft *Identity* pois desta forma certificamo-nos que os padrões de segurança de autenticação serão garantidos pela Microsoft. Para configurar o serviço de autenticação, seguimos os seguintes passos:

- Adicionamos as referências, Figura 115 para o projeto Data e para o Crosscutting, uma vez que vamos utilizar serviços, Figura 116.

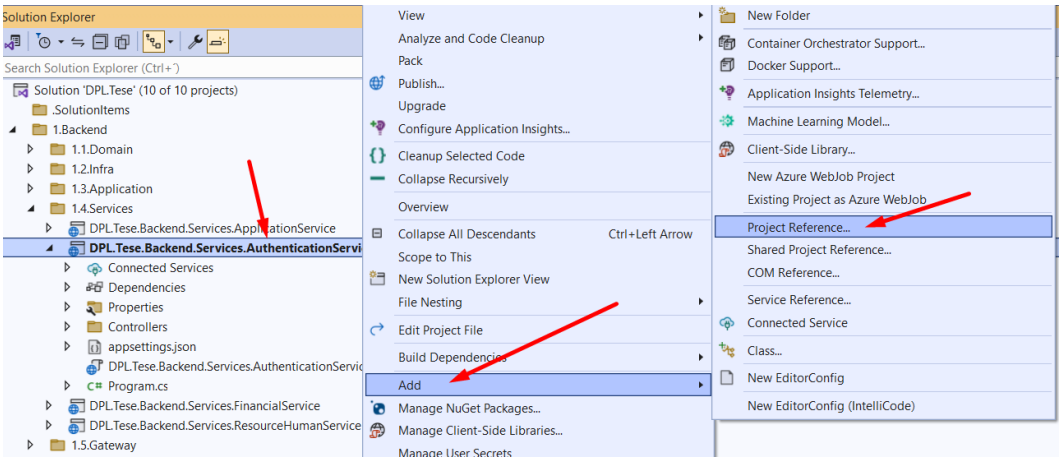


Figura 115 – Exemplo de adição de referências dos projetos aos serviços

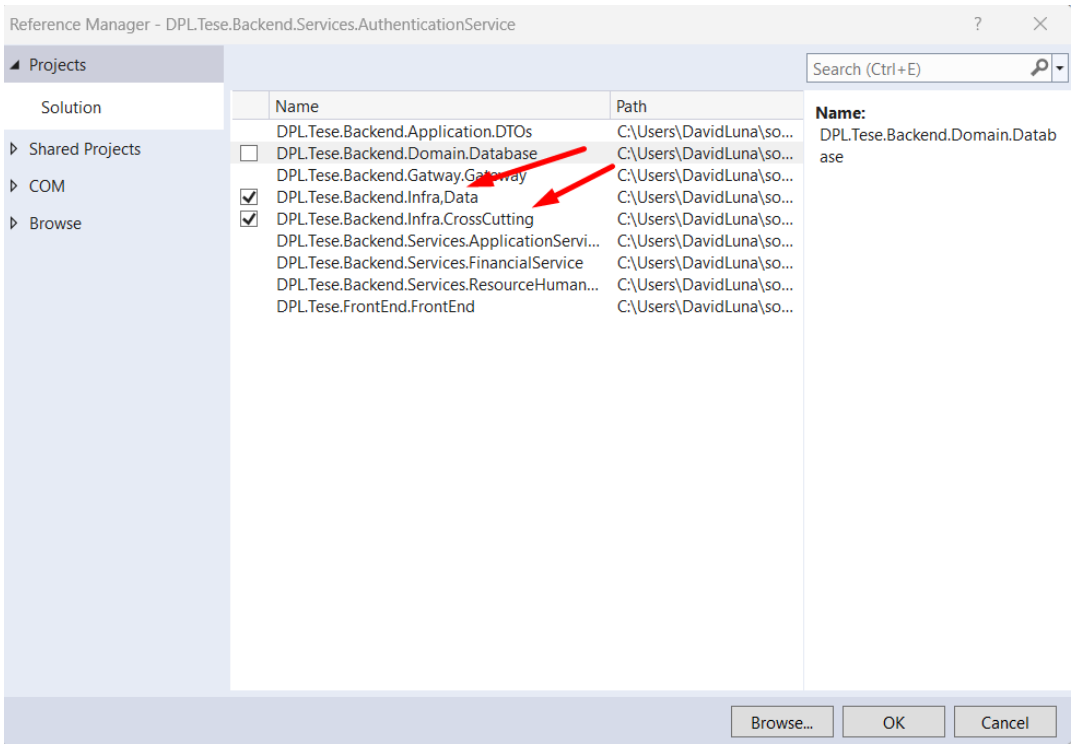


Figura 116 – Exemplo de escolha das referências dos projetos a adicionar ao serviço

- Instalamos os *Nugets* do Microsoft *Identity* e das EntityFrameWorkCore, Figura 117 e todos os que necessitamos para funcionar.

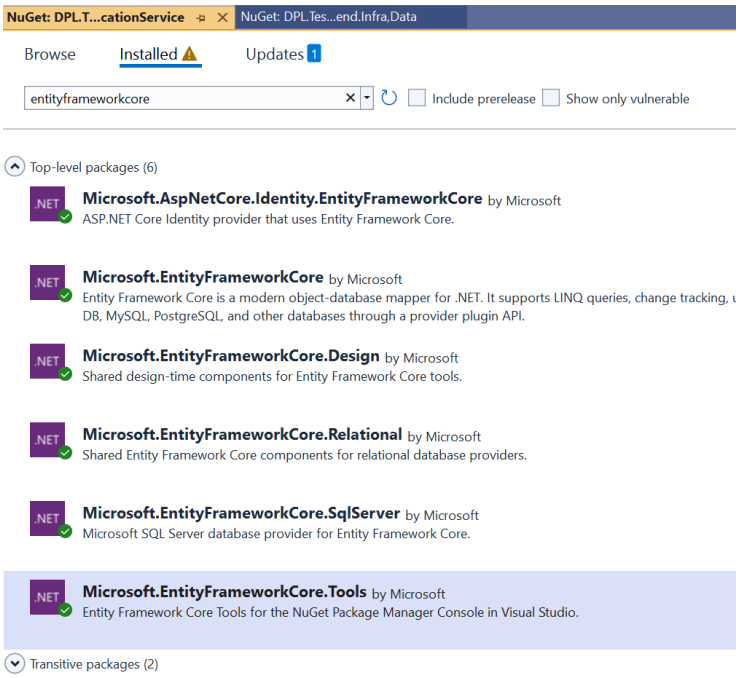


Figura 117 – Instalação dos Nugets da EntityFrameworkCore no serviço

- Criamos uma pasta chamada *Services*, na qual criamos duas subpastas uma para as interfaces e outra para a implementação, Figura 118. Na pasta de interfaces iremos criar os interfaces necessários para os vários serviços a serem registados no contentor. Cada tabela que for necessária utilizar, irá estar representada por uma interface. Deste modo assumimos que cada interface e respetiva classe apenas assumem as suas respetivas responsabilidades.

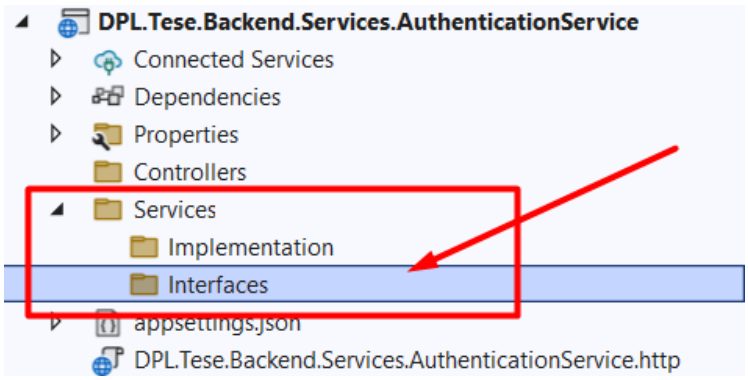


Figura 118 – Exemplo de organização das pastas que guardaram as classes e interfaces dos vários serviços

- Como exemplo, criamos uma *interface* chamada *IAuthUserService,* Figura 119 que será responsável por todos os métodos de autenticação, esta interface irá herdar da *interface ITransientService* da livraria *CrossCutting* de modo a definir o ciclo de vida do serviço. Na pasta de implementação criou-se a classe *AuthUserService,* Figura 119, que terá a implementação dos métodos para efetuar as operações de autenticação. Também será neste projeto que iremos criar todos os serviços referentes às Roles, *Permissions*, *Claims*, *Tokens*, entre outros.

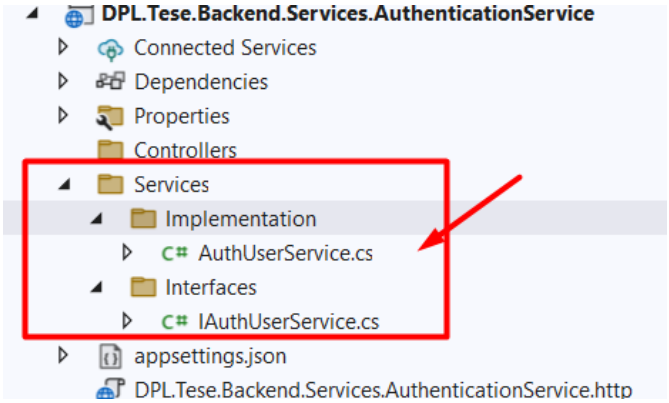


Figura 119 – Localização da interface e da class responsável por todos os métodos do utilizador

- Para poder utilizar tudo do Microsoft *Identity* é necessário registar a inicialização do mesmo no arranque da API. Para isso iremos ao ficheiro Program.cs acrescentar esta inicialização, Figura 120.

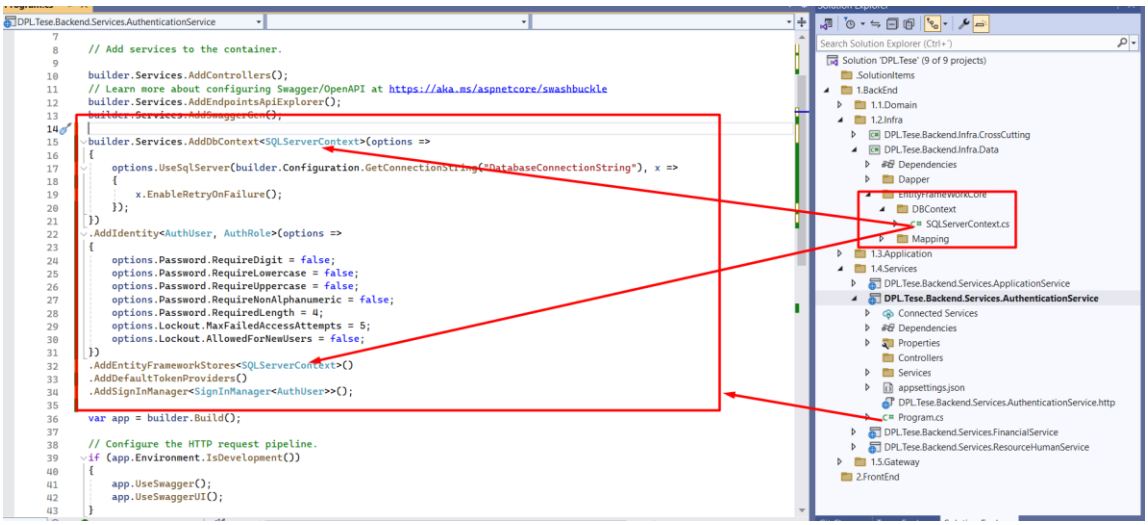


Figura 120 – Configuração da parametrização do SQLServerContext no arranque do serviço AuthenticationService

- Para comunicar com o exterior é necessário criar os *controllers* API, Figura 121. Neste exemplo iremos demonstrar como criar um *controller,* Figura 122, para o AuthUser, Figura 123.

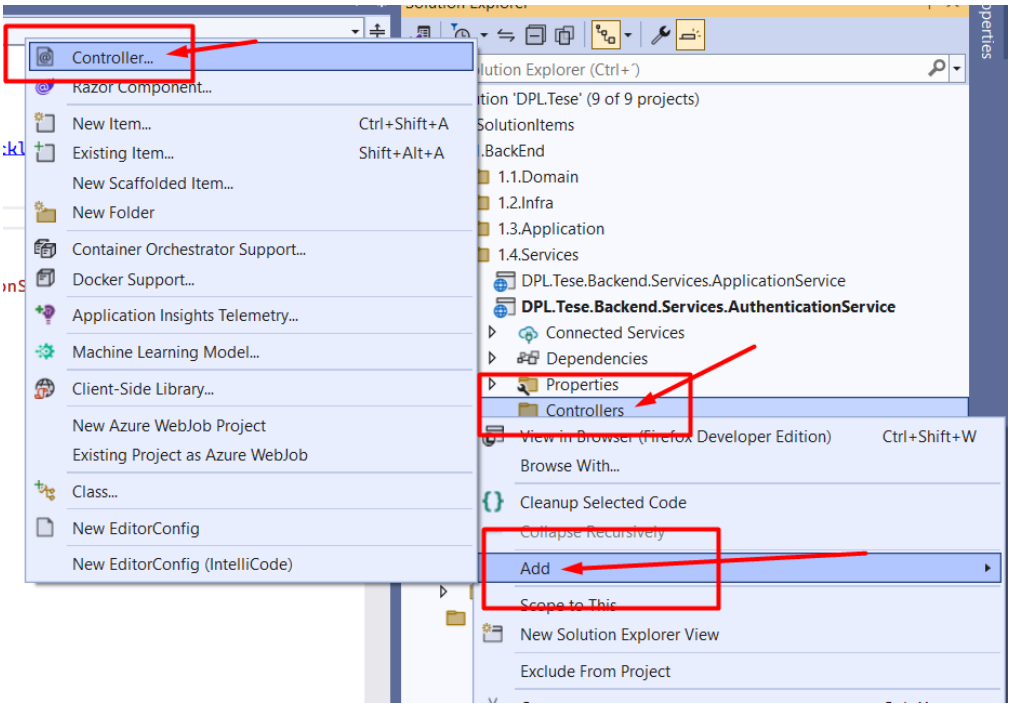


Figura 121 – Exemplo de adição de um controller ao micro serviço

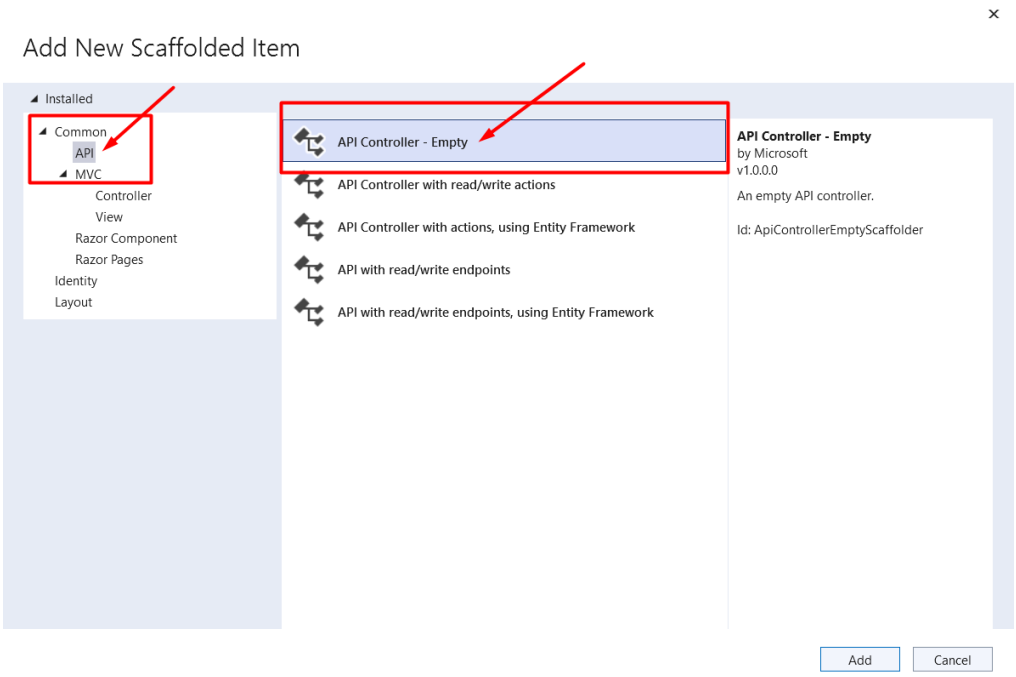


Figura 122 – Exemplo de escolha do tipo de controller

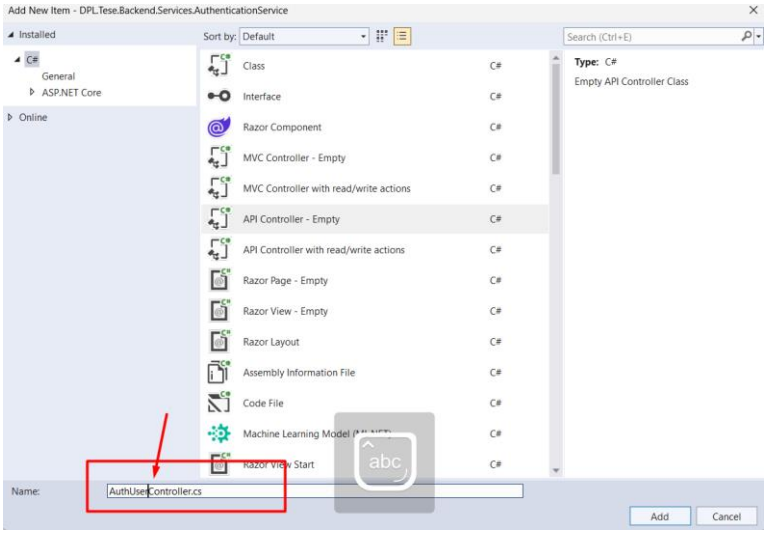


Figura 123 – Exemplo de nome a dar ao controller

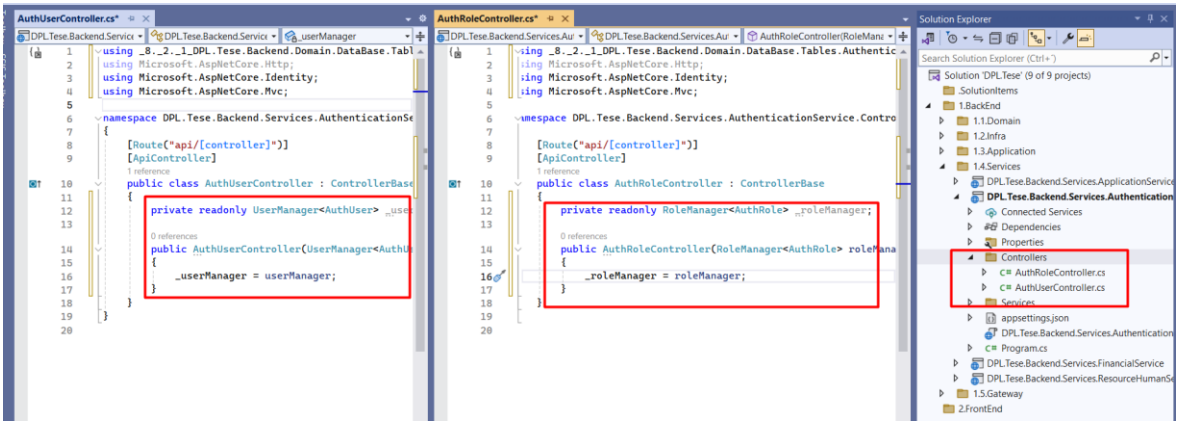


Figura 124 – Exemplo de injeção dos managers da Microsoft Identity no construtor do AuthUser e AuthRole controller

Para todos os módulos da aplicação iremos criar um micro serviço que será responsável pelas suas obrigações dentro do sistema.

16.2.7 - DPL.TESE.FRONTEND.WEB

Num mundo cada vez mais orientado para a tecnologia, o projeto de *FrontEnd* emerge como uma peça fundamental no desenvolvimento de software. Este é o ponto de contacto direto entre o utilizador e as aplicações, influenciando diretamente a perceção e interação dos utilizadores.

O projeto de *FrontEnd* concentra-se na criação e implementação da *interface* do utilizador (UI) de uma aplicação. Envolve o design e desenvolvimento da parte visual e

interativa, visando proporcionar uma experiência que seja não só funcional, mas também agradável para os utilizadores. Em termos simples, o *FrontEnd* representa a "cara" da aplicação, sendo o ponto de contacto direto com quem o utiliza.

Dentro do projeto de *FrontEnd*, encontramos componentes essenciais. O HTML (*Hypertext Markup Language*) é a base estrutural das páginas web, definindo a organização básica da informação. O CSS (*Cascading Style Sheets*) trata da esterilização e aparência visual da página e comunicação com o *BackEnd*. Além disso, *Frameworks* e bibliotecas, como React, Angular e Vue.js, simplificam e aceleram o desenvolvimento *FrontEnd*, oferecendo componentes reutilizáveis e facilitando a gestão de estados.

A importância do projeto de *FrontEnd* é evidente em diversos aspetos. A qualidade do *FrontEnd* influencia diretamente a experiência do utilizador. Uma interface bem desenhada facilita a navegação, reduz a curva de aprendizagem e aumenta a satisfação do utilizador. Além disso, o design estético e funcional contribui para a atividade visual da aplicação, criando uma primeira impressão positiva e refletindo na perceção global da marca. Garantir a compatibilidade com diferentes dispositivos e responsividade é essencial para proporcionar uma experiência consistente independentemente do dispositivo utilizado.

Podemos concluir, que o projeto de *FrontEnd* desempenha um papel fundamental na criação de *interfaces* que envolvem, cativam e satisfazem os utilizadores. Ao combinar elementos de design, desenvolvimento e interatividade, os profissionais de *FrontEnd* moldam a face visível das aplicações, contribuindo para o sucesso e aceitação num mercado competitivo de software.

Para criar o projeto de *FrontEnd* seguimos os seguintes passos:

- Na pasta 2.FrontEnd adicionamos um novo projeto com o nome DPL.Tese.FrontEnd.FrontEnd, Figura 125, do tipo Asp.net Core WebApp (Model-View-Control), Figura 126.

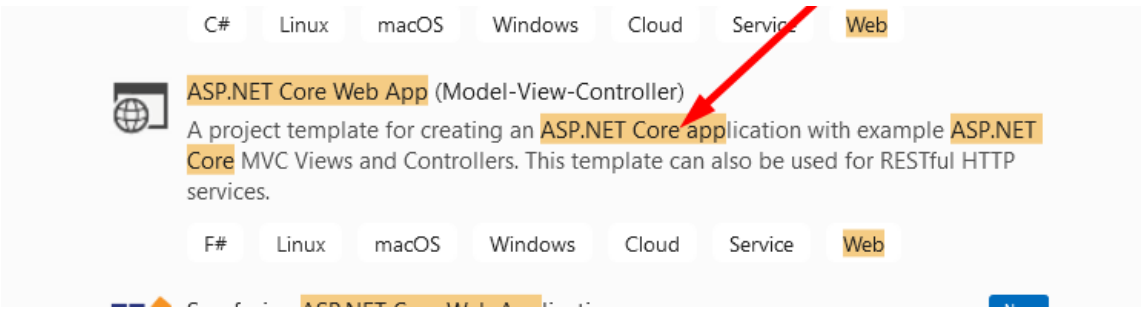


Figura 125 - Exemplo tipo de projeto para FrontEnd Web

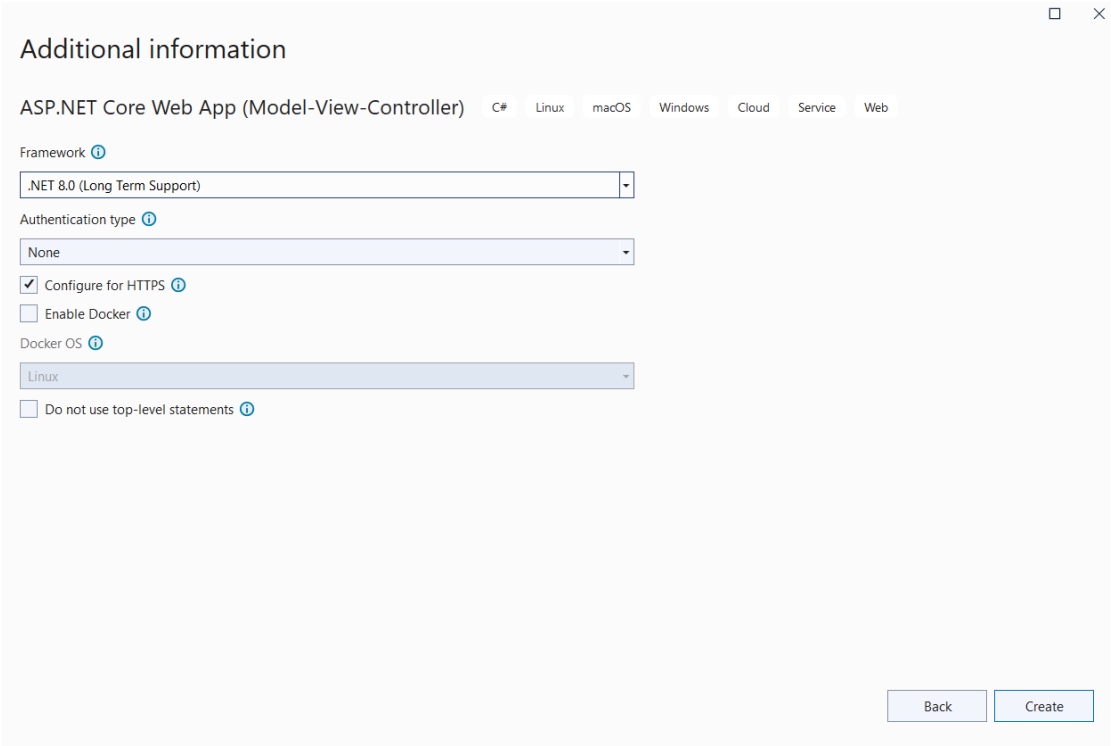


Figura 126 – Configuração para a criação do projeto FrontEnd

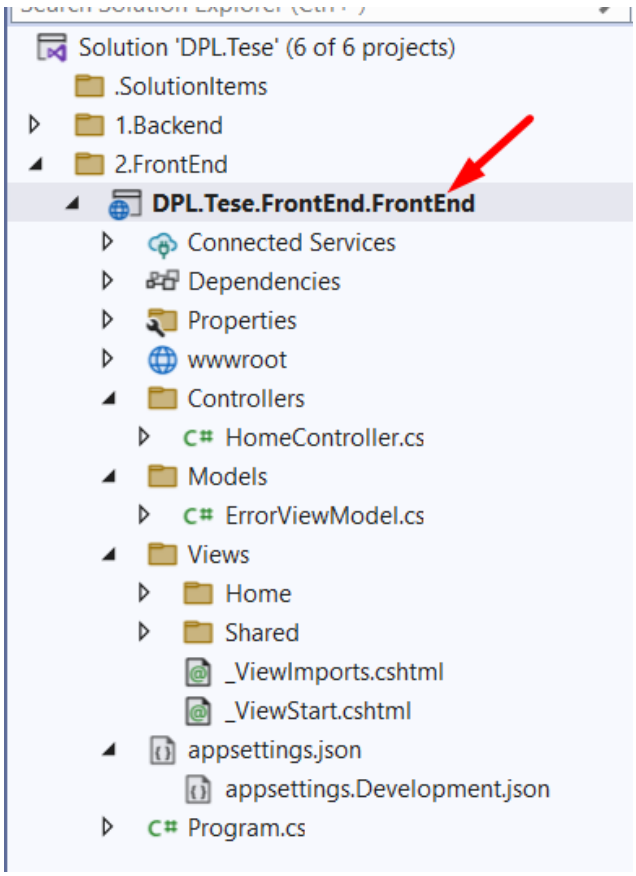


Figura 127 – Exemplo da estrutura do projeto FrontEnd

- Para podermos ter várias configurações, ao nível do ambiente de desenvolvimento, ou seja, é necessário ter parametrizações específicas para cada ambiente: uma para o ambiente de produção (final do cliente), outra para o ambiente de desenvolvimento e outra para o ambiente de qualidade (teste), Figura 128. O ficheiro appsettings.json é transversal a todos os ambientes, o ficheiro appsettings.Development.json contém as parametrizações específicas para o ambiente de desenvolvimento e as parametrizações finais do cliente vão para o ficheiro appSettings.Production.json.

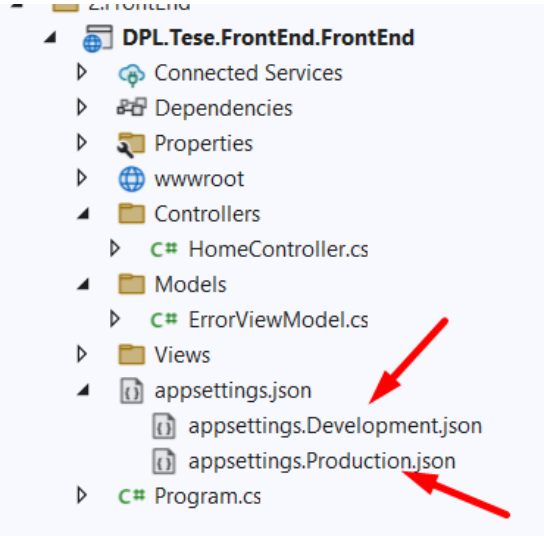


Figura 128 – Exemplo dos ficheiros de configuração para vários ambientes

- Ao criar o projeto detetou-se que os ficheiros de CSS e JS que se encontram na pasta *wwwroot* do projeto não se encontram minimizados, o que significa que os ficheiros podem ficar maiores para carregamento quando é feito o pedido. Os ficheiros minimizados retiram os espaços e colocam todo o código numa única linha, o que também o protege, uma vez que fica menos legível para quem tiver acesso a ele externamente. A opção foi criar uma pasta na raiz do projeto chamada CSS e outra chamada JavaScript, Figura 129. Tirando partido do Gulp, minimizou-se os ficheiros para a pasta wwwroot automaticamente.

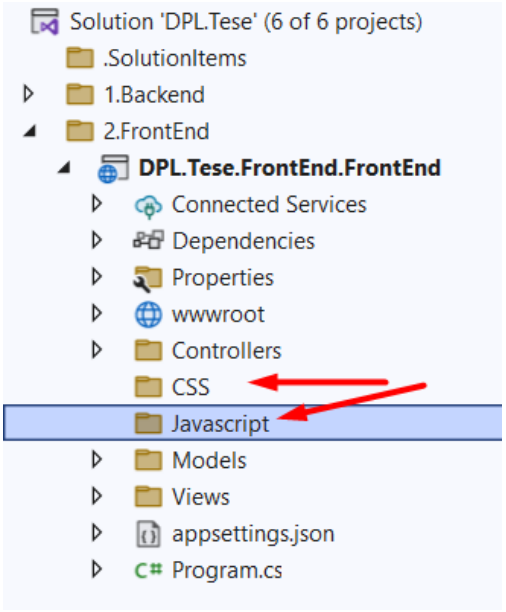


Figura 129 - Exemplos de pastas para guardar independente os Javascripts e os CSS

- Para utilizar o Gulp, é necessário instalá-lo no nosso projeto, mas para tal, primeiro iremos instalar o NodeIS, Figura 130, de modo a termos acesso a todos os módulos cliente, dando preferência a que o desenvolvimento do *FrontEnd* seja *client side*. Para instalar o nodeJS seguimos os seguintes passos:
    - Instalamos a última versão do nodeJS.

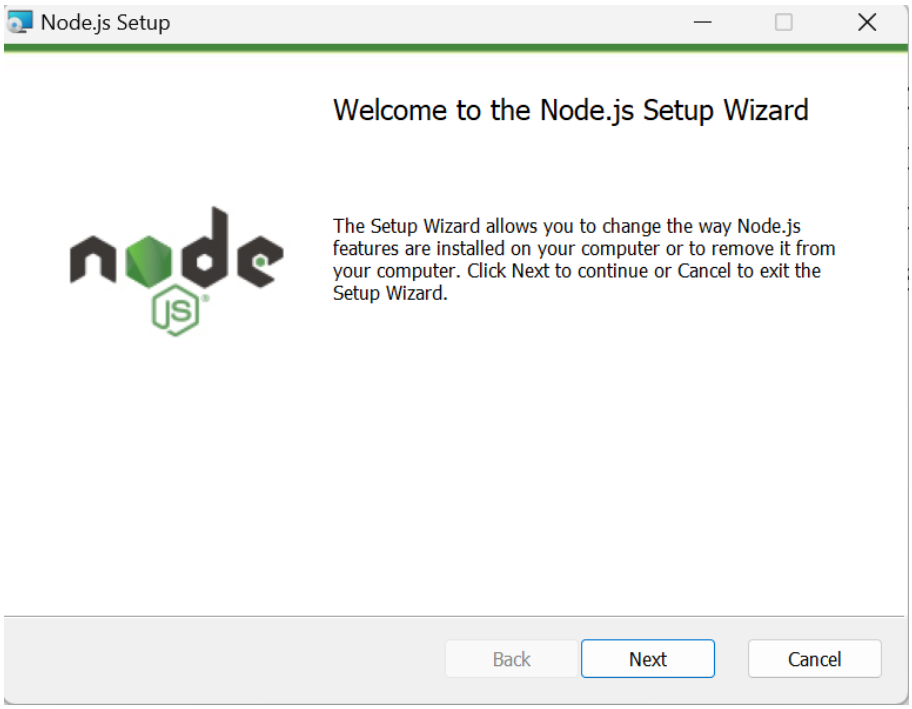


Figura 130 - Instalação do NodeJS no computador, após download

- Adicionamos um novo ficheiro npm, Figura 131, à raiz do nosso projeto, Figura 132.

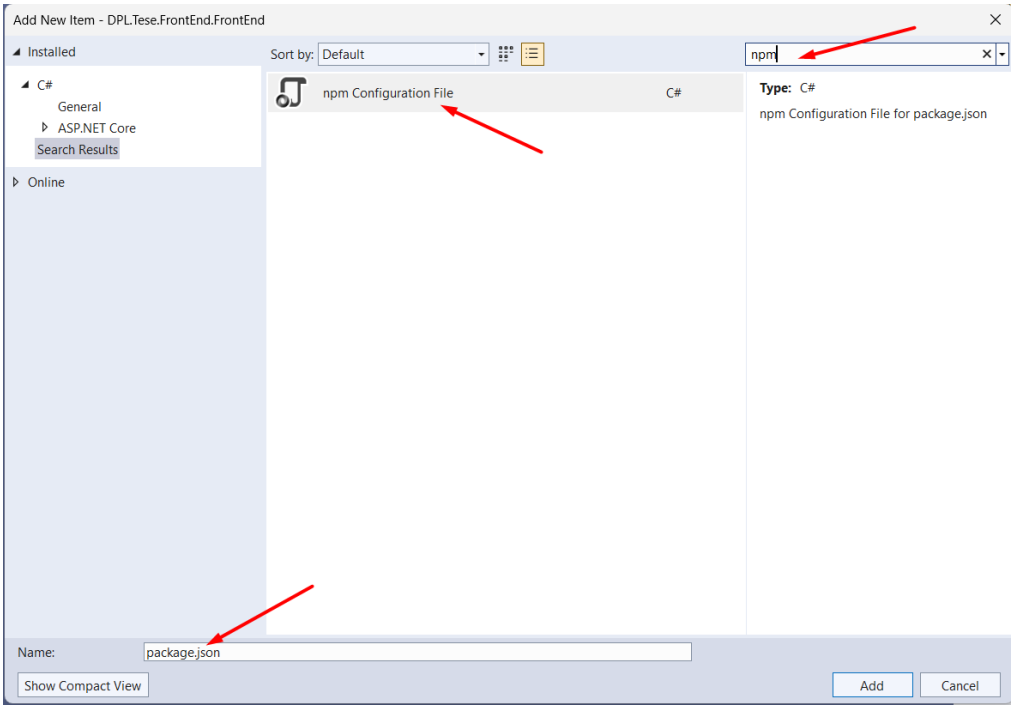


Figura 131 – Exemplo de adição do ficheiro de configuração npm para instalar os pacotes do NodeJS

Figura 132 – Exemplo do ficheiro npm, com os pacotes necessários a instalar

- Para ter a certeza que os pacotes são mesmo instalados no projeto, clicamos no botão direito do rato em cima do ficheiro package.json e escolhemos "restore packages", Figura 133.

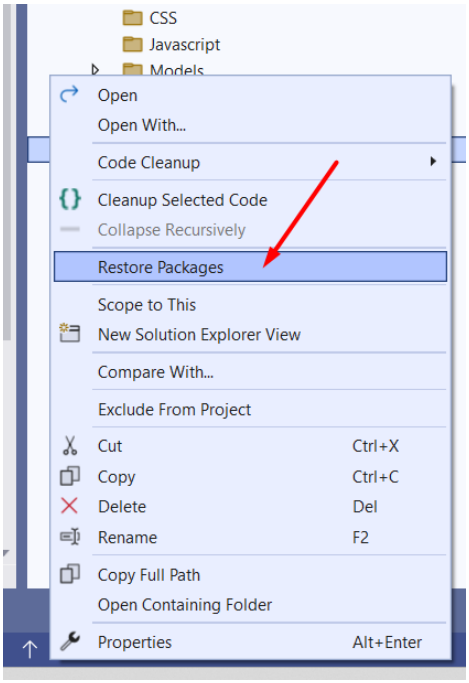


Figura 133 – Restaurar os pacotes do NodeJS

- Para tirar partido do Gulp, criamos um ficheiro chamado gulpfile.js na raiz do projeto, Figura 134: o nome tem de ser exatamente este e é sensível a maiúsculas e minúsculas (tem de ser sempre em minúsculas).

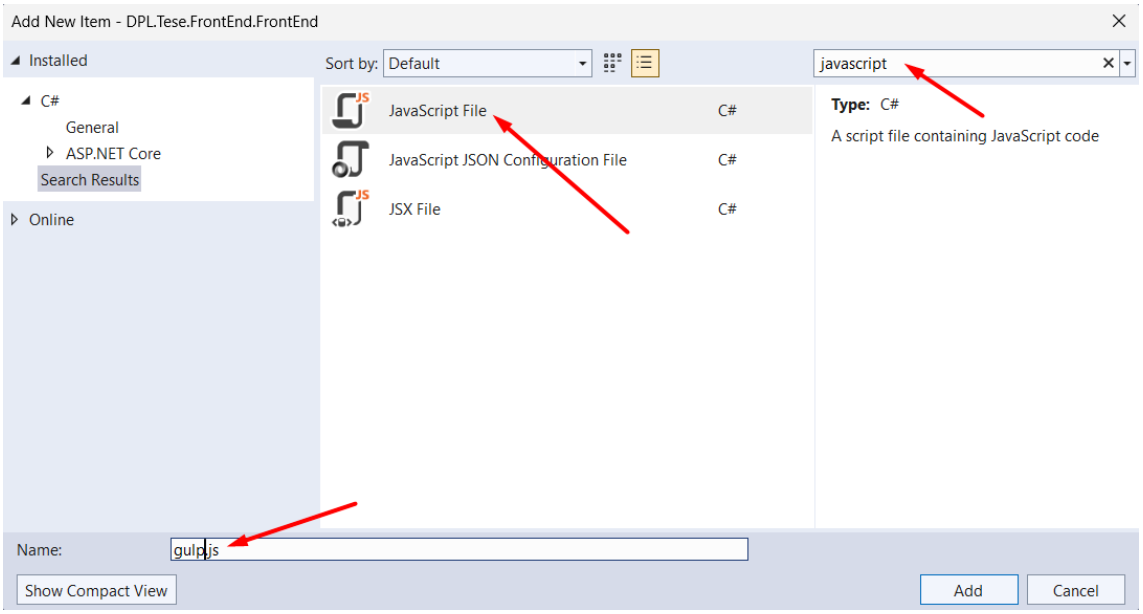


Figura 134 – Adição do ficheiro Javascript com o nome gulpfile.js

- No ficheiro Gulp desenvolvemos as funções para limpar e minimizar, Figura 135.

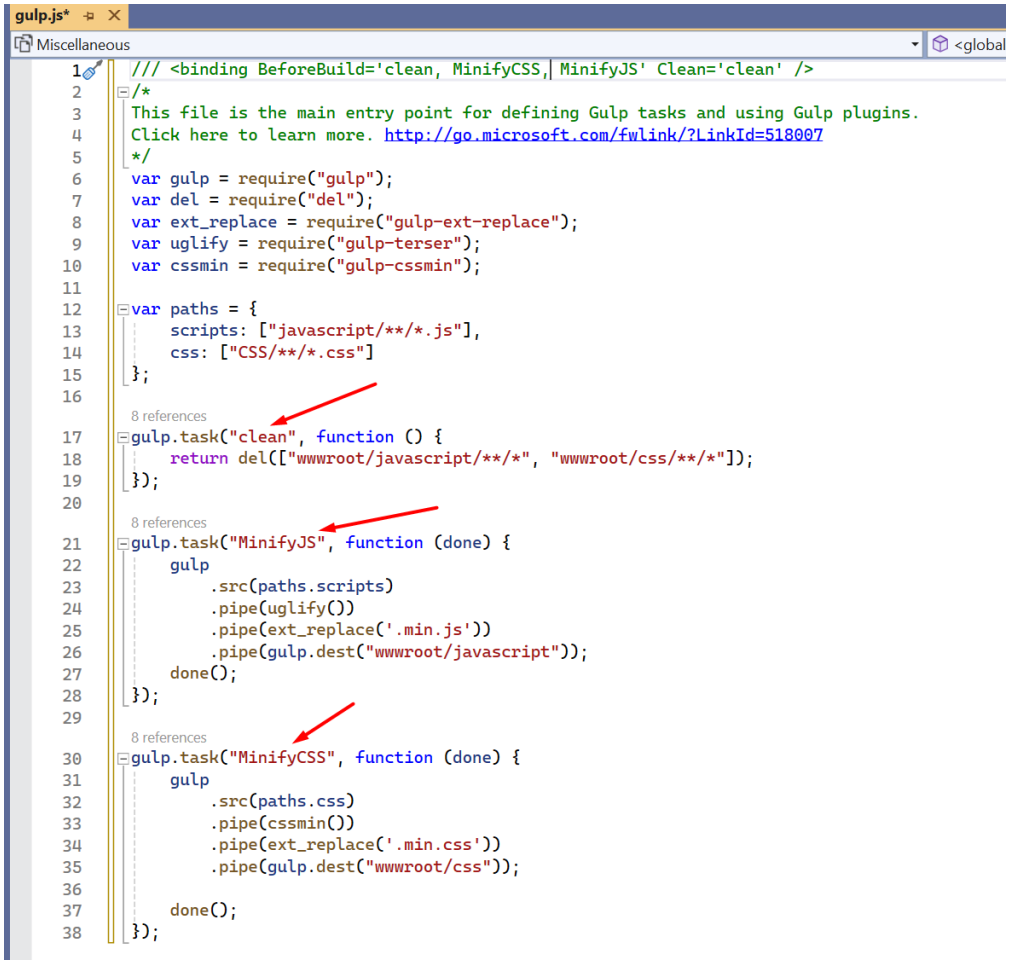


Figura 135 - Exemplo de código do GulpFile para minificar e copiar para a pasta wwwroot

- Para o *FrontEnd* ser maioritariamente *client side*, configuramos o ficheiro libman.json na raiz do projeto, Figura 137. Isto permite instalar as livrarias como o JQuery, Bootstrap e etc na pasta wwwroot/lib de forma automática. Para a instalação, seguimos os seguintes passos:
    - Clicamos com botão direito do rato em cima do projeto e escolhemos "*Manage Client-Side Libraries*", Figura 136, este irá criar o ficheiro libman.json na base do projeto.

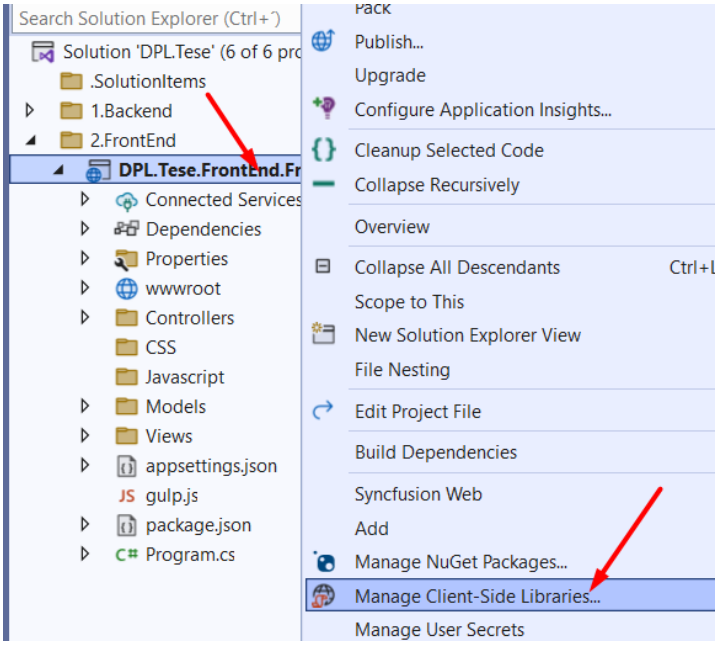


Figura 136 – Exemplo de como instalar livrarias do lado do cliente

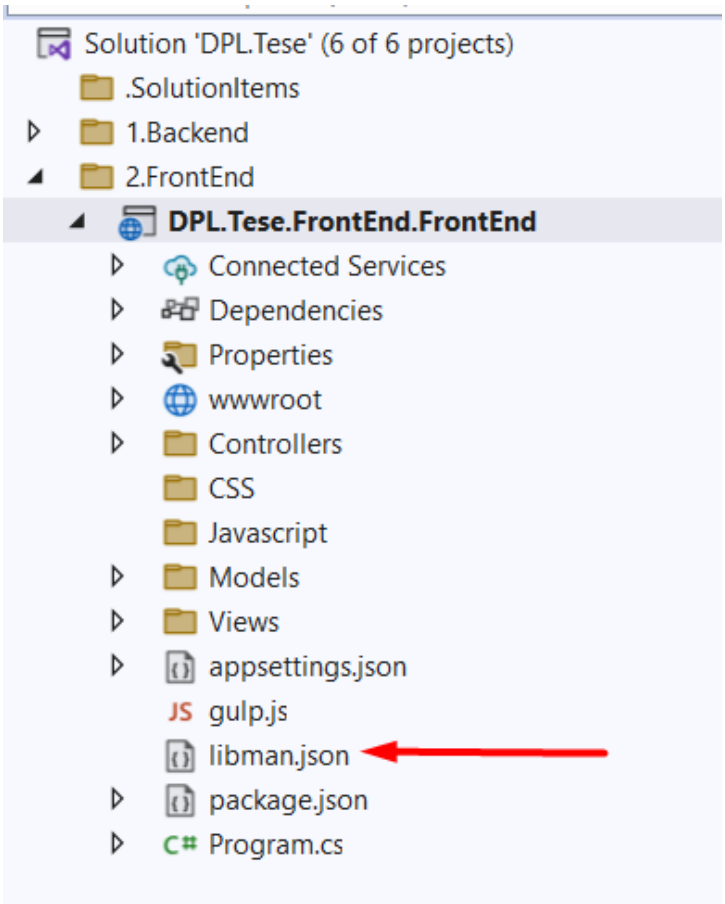


Figura 137 – Localização do ficheiro com as referências das livrarias cliente

- Colocamos as livrarias que desejamos instalar, Figura 138.

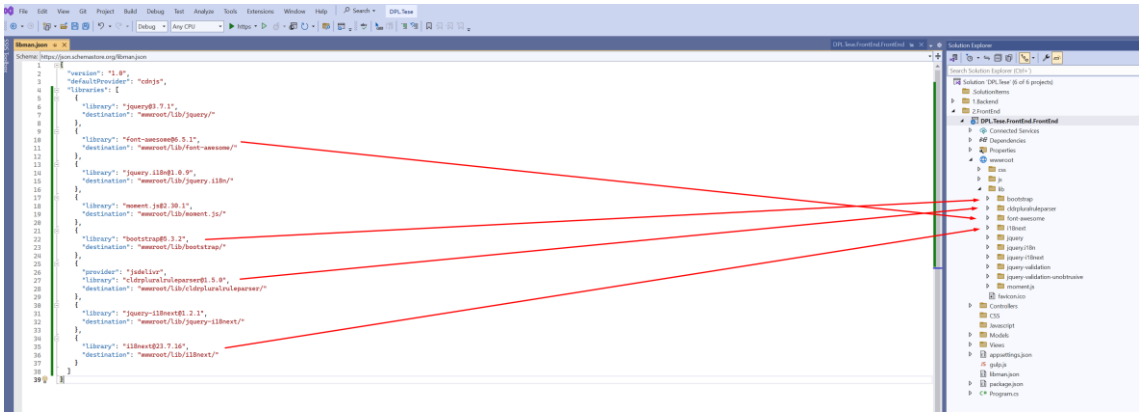


Figura 138 – Exemplo de livrarias clientes, instaladas

- Para minimizar nas várias etapas de compilação temos de abrir o "*Task Runner Explorer*", Figura 139 no menu "*View*" e submenu "*Other Windows*", Figura 140.

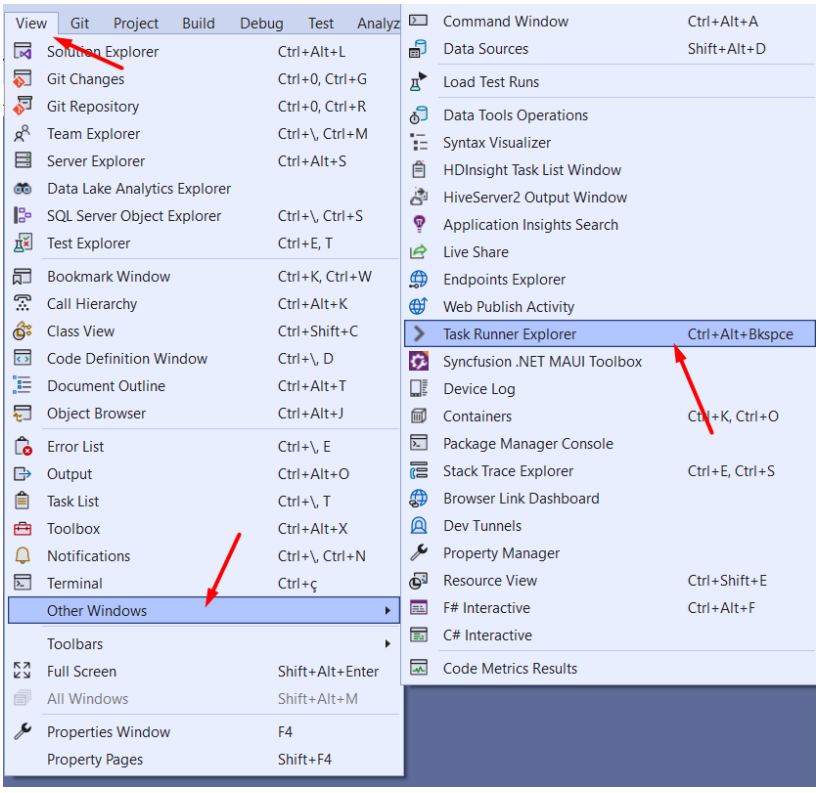


Figura 139 – Ativação da janela *Task Runner Explorer* para gerir as funções do GulpFile e visualizar o resultado dos mesmos

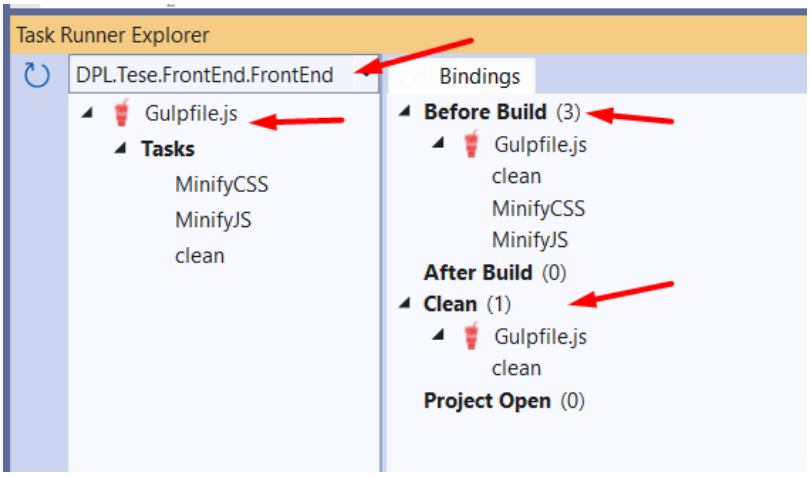


Figura 140 – Configuração do GulpFile no *Task Runner Explorer*

Para uma melhor organização e compreensão do código JavaScript decidiu-se centralizar todos os ficheiros na pasta JavaScript e subdividi-los principalmente em três pastas, Figura 141. Pasta *Helpers* que conterá todos os JavaScript referentes às funções globais para efetuar operações transversais a todo o *FrontEnd*. A pasta Controls, que conterá todos os JavaScript referentes a controlos herdados da Framework Syncfusion e adaptados para as necessidades do projeto. A pasta ViewModels que terá todos os ficheiros correspondentes a cada *view* que o utilizador visualizar.

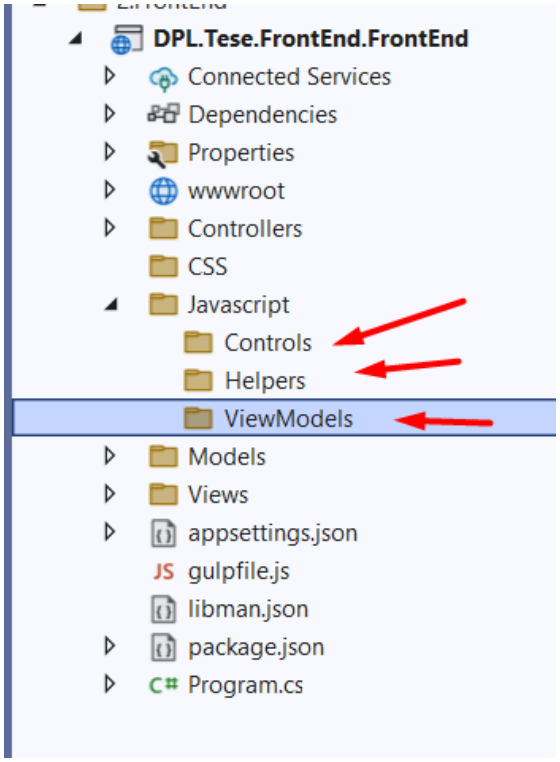


Figura 141 – Exemplo de subpastas dentro da pasta JavaScript para uma organização dos ficheiros

Para a criação dos ficheiros JavaScript definiram-se as seguintes regras:

- Utilizar o JavaScript orientado o máximo possível a objetos.
- Todos os ficheiros JavaScript têm de ter obrigatoriamente no início o '*use strict*' de modos a obrigar a utilizar todas as variáveis e detetar possíveis erros.
- A declaração de variáveis deverá ser o mais possível com a palavra reservada *let* em vez de var.
- As *functions* devem ao máximo serem *arrow functions* e serem inicializadas pela palavra reservada *const*.
- O nome dos ficheiros devem ser explicito.
- Os ficheiros da pasta ViewModels são obrigatórios terminarem em VM letra maiúsculas para serem identificados como sendo um *ViewModel*. Além que o nome antes da terminação VM tem de ser o mesmo do nome da *View* cshtml.
- Os *endpoints* serão todos centralizados num ficheiro JavaScript e terão o mesmo nome do *endpoint* do *backend*.

Um exemplo do ficheiro de variáveis de sessão pode ser visualizado na Figura 142.

```
'use strict'

0 references
export default class SessionData {
    ClearAll = () => sessionStorage.clear();

    TokenGet = () => {
        return sessionStorage.getItem('accessToken');
    };

    TokenSet = (Token) => sessionStorage.setItem('accessToken', Token);

    RefreshTokenGet = () => {
        return sessionStorage.getItem('refreshToken');
    };

    RefreshTokenSet = (RefreshToken) => sessionStorage.setItem('refreshToken', RefreshToken);

    UserIDSet = (ID) => sessionStorage.setItem('UserID', ID);

    UserIDGet = () => {
        return sessionStorage.getItem('UserID');
    };

    UserFirstLoginSet = (firstLogin) => sessionStorage.setItem('FirstLogin', firstLogin);

    UserFirstLoginGet = () => {
        return sessionStorage.getItem('FirstLogin');
    }

    RolesSet = (Roles) => {
        let stringRoles = JSON.stringify(Roles);
        sessionStorage.setItem('Roles', stringRoles);
    };

    RolesGet = () => {
        let stringRoles = sessionStorage.getItem('Roles');
        return JSON.parse(stringRoles);
    };

    ClaimsSet = (Claims) => {
        let stringClaims = JSON.stringify(Claims);
        sessionStorage.setItem('Claims', stringClaims);
    };

    ClaimsGet = () => {
        let stringClaims = sessionStorage.getItem('Claims');
        return JSON.parse(stringClaims);
    };

    LanguageSet = (language) => sessionStorage.setItem('Language', language);

    LanguageGet = () => {
        return sessionStorage.getItem('Language');
    }
}
```

Figura 142 – Exemplo de classe em JavaScript para gerir as variáveis de sessão

Um exemplo do ficheiro JavaScript para efetuar os pedidos REST pode ser visualizado na Figura 143.

```
1   'use strict'
2
3   import SessionData from '/Javascript/Helpers/SessionData.min.js'
4
    0 references
5   export default class RestRequest {
6       _Session = null;
7
        0 references
8       constructor() {
9           this._Session = new SessionData();
10      }
11
12      Method =...;
19
20      Config =...;
33
34      getBaseUrl = () =>...;
40
41      HeaderInicialiseAsync = async () =>...
54
55      //#region GETs
56
57      GetAsync = async (Url = '') => {
58          await this.HeaderInicialiseAsync();
59
60          this.Config.method = this.Method.Get;
61          this.Config.body = null;
62
63          const response = await fetch(Url, this.Config);
64
65          if (response.ok)
66              return await response.json();
67          else
68              throw await response.json();
69      };
70
71      GetHtmlAsync = async (Url = '') =>...;
84
85      GetWithDataAsync = async (Url = '', Data) =>...;
100
101     GetHtmlWithDataAsync = async (Url = '', Data) =>...;
116
117     //#endregion
118
119
120
121     PostAsync = async (Url = '', Data = null) =>...;
134
135     PutAsync = async (Url = '', Data = null) =>...;
148
149     DeleteAsync = async (Url = '') =>...;
162
163     DeleteWithDataAsync = async (Url = '', Data) =>...;
180
181
182 }
```

Figura 143 - Exemplo da classe, em JavaScript, responsável pelos pedidos REST

Para o ficheiro de *endpoints* definimos as seguintes cláusulas:

- Criar um objeto chamado Base, que conterá todos os campos que são comuns a todas as tabelas;
- Os objetos criados terão de ter o mesmo nome de cada tabela;
- Dentro de cada objeto estarão os seguintes subobjectos:
    - *Url*: Dentro do objeto *Url* ainda haverá um objeto que define cada método do *Rest*, isto é, terá um para os métodos *Post*, outro para os

métodos *Put*, outro para os métodos *Get* e outro para os métodos delete.

- *Fields*: Terá a representação de cada campo que virá no DTO com a informação de cada tabela.

Um exemplo do ficheiro *EndPoints* pode ser visualizado na Figura 145.

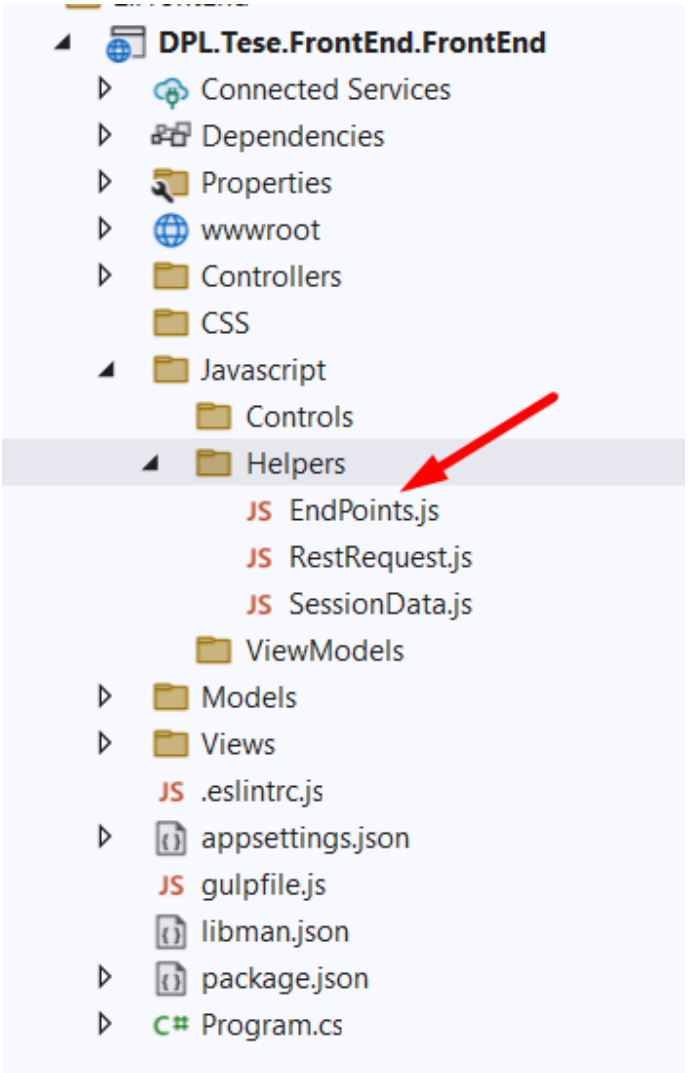


Figura 144 – Exemplo organização de alguns ficheiros Javascript

```
'use strict'

export const Base = {
    Url: {
        Get: {},
        Post: {},
        Put: {},
        Delete: {}
    },
    Fields: {
        ID: { field: 'ID', headerText: 'ID', visible: false, PrimaryKey: true },
        Id: { field: 'Id', visible: false, PrimaryKey: true },
        Data: { field: 'Data' },
        Succeeded: { field: 'Succeeded' },
        StatusCode: { field: 'StatusCode' },
        Message: { field: 'Message' },
        HasChildren: { field: 'hasChildren' },
        HasChild: { field: 'HasChild' },
        UserID: { field: 'UserID' },
        RowsAffected: { field: 'RowsAffected' }
    }
};

export const Auth = {
    Urls: {
        Post: {
            ForgotPassword: '/v1/Auth/ForgotPassword',
            ConfirmForgotPasswordCode: '/v1/Auth/ConfirmForgotPasswordCode',
            ChangePassword: '/v1/Auth/ChangePassword',
            Login: '/v1/Auth/AuthenticateAsync',
            LogoutAsync: '/v1/Auth/LogoutAsync'
        }
    },
    Fields: {
        UserName: { field: 'UserName', headerText: 'Utilizador' },
        Password: { field: 'Password', headerText: 'Password' },
        Mail: { field: 'Mail', headerText: 'Mail' },
        NewPassword: { field: 'NewPassword', headerText: 'Nova password' },
        Token: { field: 'Token' },
        UserID: { field: 'UserID' },
        TokenType: { field: 'TokenType' },
        RolesList: { field: 'Roles' },
        Expires: { field: 'Expires' },
        ClaimsList: { field: 'Claims' },
    }
};

```

Figura 145 - Exemplo da organização do ficheiro responsável pela centralização de todos os EndPoints do Backend

Relativamente às janelas de visualização e introdução de dados definiu-se que na pasta *views* se iria criar uma pasta para cada módulo, ou seja, uma pasta chamada *Authentication*, outra chamada *Application*, outra chamada Financial, outra chamada *ResourceHuman* e assim sucessivamente, Figura 146. Dentro de cada uma delas criou-se uma pasta para cada item do menu que iremos disponibilizar ao utilizador, desta forma caso exista algum erro que necessite de correção reportado pelo utilizador já poderemos cruzar o menu com a pasta onde se encontra o código.

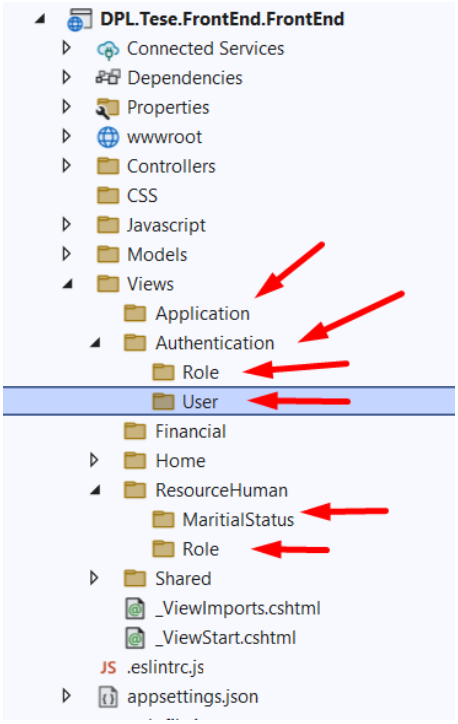


Figura 146 - Exemplo de organização da pasta das views, responsáveis por guardar as janelas de interação com utilizador

De forma a otimizar e diminuir o máximo possível a manutenção e ser de mais fácil compreensão, decidiu-se que se tentará ao máximo ter apenas duas *view* para cada pasta que representa em parte uma ou mais tabelas na base de dados. Ou seja, quando o utilizador clicar no menu na opção vai entrar numa *view* da pasta chamada Index.cshtml que será sempre a primeira página a ser apresentada e sempre que queira adicionar ou alterar valor será aberto um *popup* que será sempre uma *view*, chamada Edit.cshtml, Figura 147, e que será controlada por um estado para se saber se está no estado adicionar ou no estado de alteração de dados.

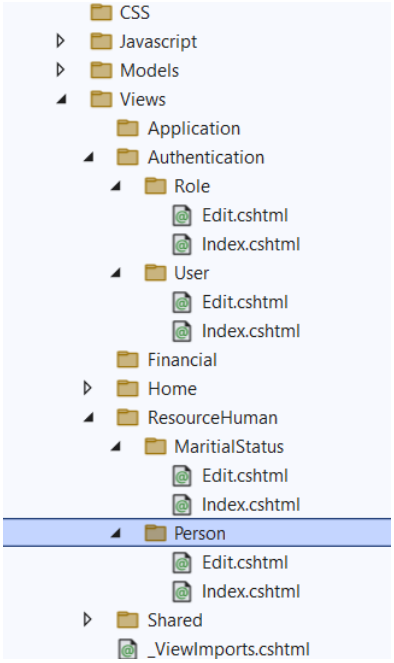


Figura 147 – Exemplo de views para o utilizador

Para cada *view* Index e Edit em cada pasta existirá um ficheiro de JavaScript correspondente dentro da subpasta ViewModels, Figura 148. Dentro da pasta ViewModels será criada a mesma estrutura de pastas que existe nas Views de modo a ser de fácil localização, aquando do crescimento do projeto.

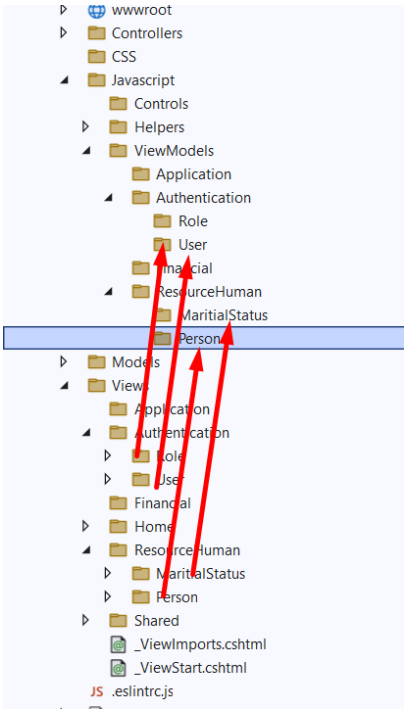


Figura 148 – Exemplo de emparelhamento de pastas entre o Javascript e as Views

Para cada *view* Index e Edit definiu-se que se iria criar um ficheiro JavaScript correspondente a cada *view*, cuja identificação é definida pelo nome da pasta concatenado com o nome da *view*, e tem como terminação o conjunto de letras VM em maiúsculo, Figura 149.

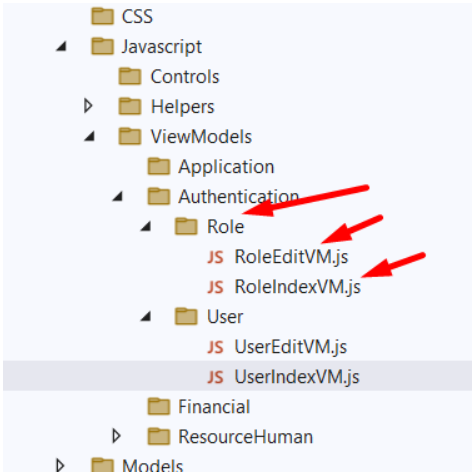


Figura 149 - Exemplo dos ficheiros JavaScript referentes a cada View

Tendo a *view* o respetivo JavaScript criado, referenciamos na *view* o respetivo JavaScript. Uma vez que teremos o ficheiro normal de JavaScript em modo desenvolvimento e minimizado em modo produção, teremos que efetuar duas referências na *view* independentes. Exemplo de referência na *view* Index da *role,* Figura 150.

Index.cshtml*

```
@{
}

<environment include="Development" exclude="Production">
    <script type="module">
        import RoleIndexVM from '/Javascript/ViewModels/Authentication/Role/IndexVM.js';
        new RoleIndexVM();
    </script>
</environment>

<environment include="Production" exclude="Development">
    <script type="module">
        import RoleIndexVM from '/Javascript/ViewModels/Authentication/Role/IndexVM.min.js';
        new RoleIndexVM();
    </script>
</environment>
```

Figura 150 - Exemplo de carregamento dos JavaScript para cada ambiente de desenvolvimento

De modo a otimizar o código e sabendo que existem objetos que serão sempre utilizados como é o caso dos pedidos *Rest*, decidiu-se criar uma pasta chamada *Shared*

dentro da pasta *viewmodels* do JavaScript e criar um ficheiro JavaScript chamado BaseVM que conterá todos os objetos e métodos transversais a todas as outras classes de JavaScript, Figura 151. As restantes classes das outras *views models* herdaram desta classe.

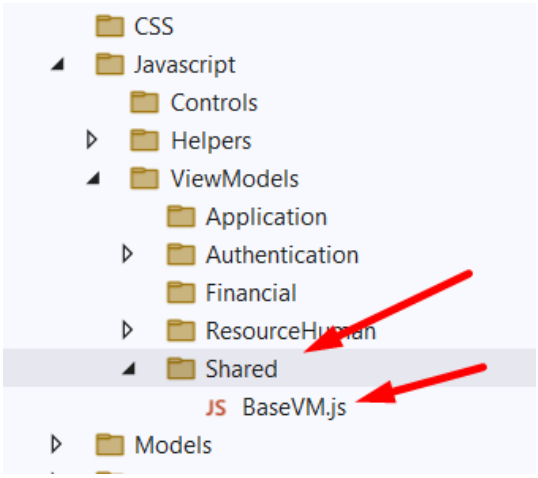


Figura 151 – Exemplo de localização da classe JavaScript, da qual, as outras classes herdarão

No ficheiro BaseVM foi criado uma variável que define o estado da ViewModel, e inicializa logo a classe de REST, Figura 152.

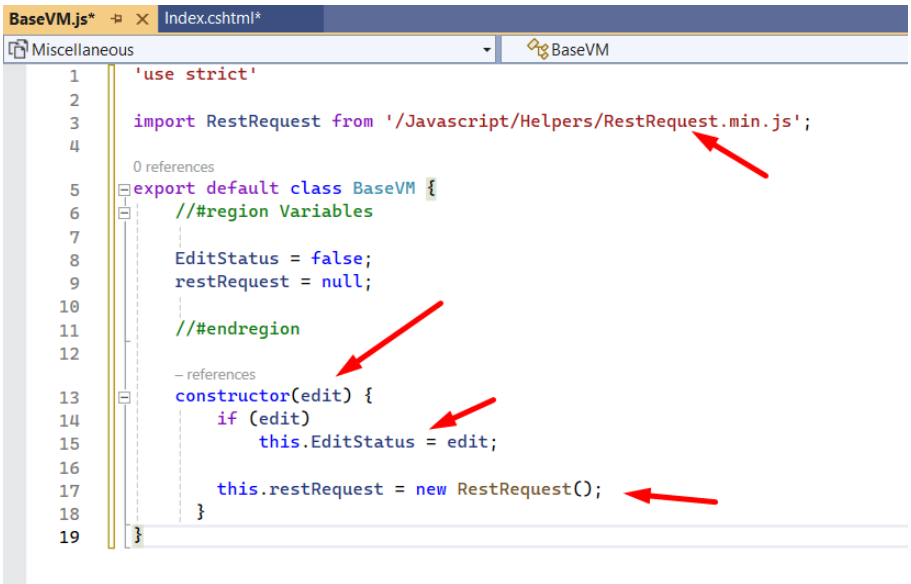


Figura 152 – Exemplo da definição da classe Base em JavaScript e respetiva inicialização

Todas as ViewModels herdaram da base. Exemplo para a Index do Role, Figura 153.

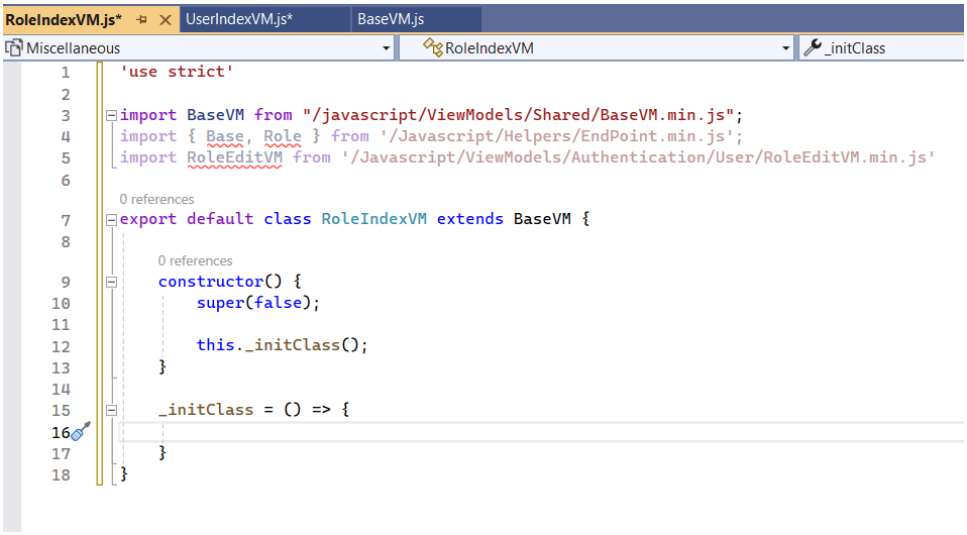


Figura 153 – Exemplo de uma class a herdar da base em JavaScript

Tendo o projeto já parametrizado e pronto, podemos iniciar o desenvolvimento específico de cada janela seguindo sempre as linhas definidas anteriormente de modo que o código fonte e a organização do código fique o mais legível possível e o menor possível de modo a ter uma boa base de compreensão.

Nos anexos podem ser visualizadas algumas capturas de ecrã da nova aplicação, principalmente de janelas de parametrizações, permissões e da construção de relatórios dinâmicos.

Capítulo 17 - CONCLUSÃO

A evolução das tecnologias de informação tem revolucionado a forma como desenvolvemos e implementamos software. Desde sistemas monolíticos tradicionais até modernas aplicações baseadas em SaaS (*Software as a Service*), a variedade de opções arquiteturais e práticas de desenvolvimento disponíveis têm crescido exponencialmente. Neste contexto, é crucial compreender o papel fundamental que a arquitetura de software e as práticas de desenvolvimento desempenham no sucesso de um projeto.

No desenvolvimento de software, uma das primeiras decisões que os programadores enfrentam é a escolha de uma arquitetura adequada. Em projetos de pequena escala, onde a simplicidade e a rapidez de desenvolvimento são prioridades, os sistemas monolíticos com ORM (Mapeamento objecto-Relacional) têm sido a escolha popular. *Frameworks* com o Microsoft *EntityFrameworkCore* oferecem uma forma rápida de mapear objetos de uma aplicação para tabelas de uma base de dados relacional, permitindo que os programadores se concentrem na lógica de negócio em vez de se preocuparem com os detalhes da persistência de dados.

No entanto, à medida que um projeto cresce em escopo e complexidade, torna-se evidente a necessidade de adotar abordagens mais avançadas de desenvolvimento e arquiteturas. É aqui que entram conceitos com o *Clean Architecture*, *Domain-Driven* Design (DDD), padrões como o *Repository Pattern* e *Gateway*. A *Clean architecture* propões uma separação clara entre as diferentes camadas de um sistema, garantindo que a lógica de negócio permaneça isolada das preocupações da infraestrutura, facilitando a manutenção e a evolução do software ao longo do tempo. Por sua vez, o DDD enfatiza a importância de modelar o software de acordo com o domínio do problema, criando uma linguagem ubíqua que liga as diferentes partes do sistema de forma coesa e compreensível.

A escolha entre sistemas monolíticos e arquiteturas mais distribuídas, como micro serviços, também se torna relevante à medida que um projeto cresce. Enquanto sistemas monolíticos oferecem simplicidade e coesão, os micro serviços permitem uma maior escalabilidade e flexibilidade, facilitando a manutenção dos sistemas complexos. No entanto, a transição de um para o outro requer uma cuidadosa consideração das necessidades do projeto e das capacidades da equipa de desenvolvimento.

Além da arquitetura, as práticas de desenvolvimento também desempenham um papel crucial no sucesso de um projeto de Software. O uso de testes automatizados, integrações e entregas contínuas (CI/CD), por exemplo, pode aumentar significativamente a qualidade e a confiabilidade de um software, reduzindo o tempo e os custos associados à manutenção do mesmo. Da mesma forma, a adoção de práticas ágeis, como o Scrum ou Kanban, pode melhorar a colaboração entre os membros da equipa e garantir que o software atenda continuamente às necessidades do cliente.

Por fim, é importante reconhecer que não existe uma abordagem única que sirva para todos os casos. Cada projeto é único, com as suas próprias necessidades, restrições e desafios. Cabe à equipa de gestão de projeto avaliar cuidadosamente esses fatores e tomar decisões informadas sobre a arquitetura e as práticas de desenvolvimento a serem adotadas. À medida que o campo das tecnologias de informação continua a evoluir, é fundamental permanecer aberto a novas ideias e abordagens, procurando constantemente maneiras de melhorar e aprimorar as práticas de desenvolvimento de software.

Em conclusão, a arquitetura de software e as práticas de desenvolvimento desempenham um papel fundamental no sucesso de um projeto de software. Desde a escolha da arquitetura inicial até à implementação de práticas de desenvolvimento ágeis e eficazes, cada decisão tem o potencial de impactar significativamente a qualidade, o desempenho e a manutenção do software. Ao reconhecer a importância desses aspetos e adotar uma abordagem cuidadosa e informada, as equipas de desenvolvimento podem aumentar as suas probabilidades de sucesso e criar software que atenda às necessidades dos utilizadores de forma eficaz e eficiente.

A migração de um sistema monolítico para uma arquitetura de micro serviços traz às empresas melhorias significativas em termos de desempenho, porque possibilita uma maior escalabilidade e agilidade nas operações. Esta transição resulta numa redução de custos operacionais, por haver menos necessidade de manutenções e suporte, o que permite uma alocação mais eficiente dos recursos humanos, infraestruturas e, por consequência, financeiros, promovendo ainda um funcionamento mais robusto e flexível, adaptado às exigências dinâmicas do mercado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Udemy, ".NET Core 3.1/NET 5/6 - C# API Com DDD na Prática," 2022.

[2] R. C. Martin, "Código Limpo. Habilidades Práticas Do Agile Software".

[3] Udemy, ".NET Core Microservices - The complete guide (.NET 8 MVC)," 2022.

[4] G. Blokdyk, Monolithic application The Ultimate Step-By-Step Guide, The art of service, 2021.

[5] Udemy, "C# - Aplicando Principios S.O.L.I.D. na prática," 2022.

[6] S. Newman, Building Microservices, 2019.

[7] A. Watt, Building Modern SaaS Applications with C# and .NET, packT, 2023.

[8] Microsoft, "Entity Framework Tutorial," [Online]. Available: https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx.

[9] Dapper Plus, "Learn Dapper," [Online]. Available: https://www.learndapper.com/.

[10] A. Alam, "Medium," 08 2017. [Online]. Available: https://medium.com/oceanize-geeks/code-smells-and-refactoring-c2c0e0642582.

[11] Udemy, "REST API's RESTFul do 0 à Azure com ASP.NET 8 e 5 e Docker," 2023.

[12] Wikipedia, "Wikipédia," 2023. [Online]. Available: https://pt.wikipedia.org/wiki/.NET_Framework.

[13] T. Williams, Microservices Design Patterns in .NET: Making sense of microservices design and architecture using .NET Core, Packt Publishing, 2023.

[14] S. Oloruntoba, "Digital Ocean," 03 11 2021. [Online]. Available: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

[15] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley , 2003.

[16] P.-E. Bergman, “Medium,” 20 Abril 2017. [Online]. Available: https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30.

[17] S. Bandara, “Medium,” 23 Julho 2023. [Online]. Available: https://medium.com/@susithapb/exploring-the-unit-of-work-pattern-b5163a9165db.

[18] T. Brito, “Medium,” 30 Dezembro 2019. [Online]. Available: https://medium.com/@devbrito91/mensageria-1330c6032049.

[19] Learn With Whiteboard, “Medium,” 30 Setembro 2023. [Online]. Available: https://medium.com/@learnwithwhiteboard_digest/understanding-what-is-api-gateway-and-how-it-works-with-examples-4762c26faca3.

[20] S. Musib, “Medium,” 17 Novembro 2019. [Online]. Available: https://medium.com/swlh/restful-api-documentation-made-easy-with-swagger-and-openapi-6df7f26dcad.

[21] I. V. Abba, “FreeCodeCamp,” 21 Outubro 2022. [Online]. Available: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/.

[22] J. Smith, Entity Framework Core in Action Rústica – 2, Manning, 2018.

[23] B. A. Peixoto, “invoicexpress,” 06 Abril 2022. [Online]. Available: https://invoicexpress.com/blog/produtividade-metodologias-gestao-projetos.

[24] R. S. P. e. B. R. Maxim, Engenharia de Software - Uma abordagem profissional (9ª Edição), McGraw Hill, 2021.

[25] u. A. e. J. Varajão, PLANEAMENTO DE SISTEMAS DE INFORMAÇÃO, FCA, 2007.

[26] V. Subramaniam, Rediscovering Javascript: Master ES6, ES7 and ES8, Pragmatic Bookshelf, 2018.

[27] InfoEscola, “InfoEscola,” 2024. [Online]. Available: https://www.infoescola.com/informatica/html/. [Acedido em 2023].

[28] librosweb, “librosweb.es,” librosweb, [Online]. Available: https://dis.um.es/~lopezquesada/documentos/IES_1213/LMSGI/curso/UT5/libroswebcss/www.librosweb.es/css/capitulo1/breve_historia_de_css.html. [Acedido em 2023].

[29] R. Queirós, Criação Rápida de Sites Responsivos com Bootstrap, FCA, 2017.

[30] Syncfusion, "Syncfusion," [Online]. Available: https://ej2.syncfusion.com/documentation/introduction.

[31] U. -. J. Casal, "Building Microservices with .NET - Security and Identity".

ANEXOS – PRINTS NOVO LAYOUT

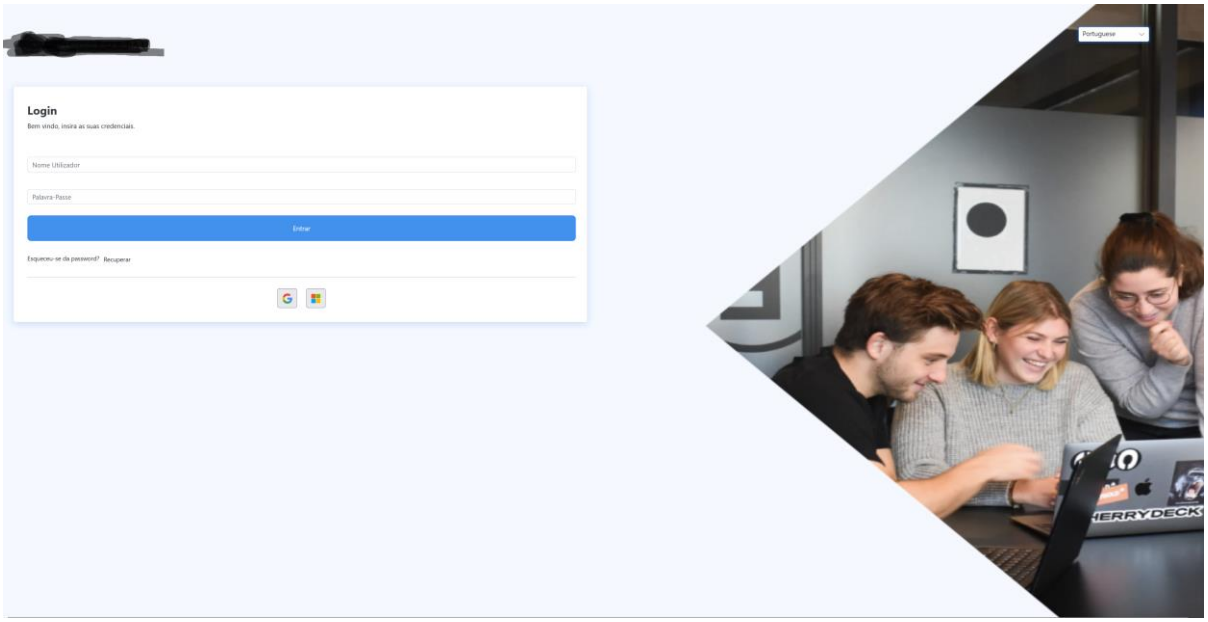


Figura 154 – Login nova aplicação

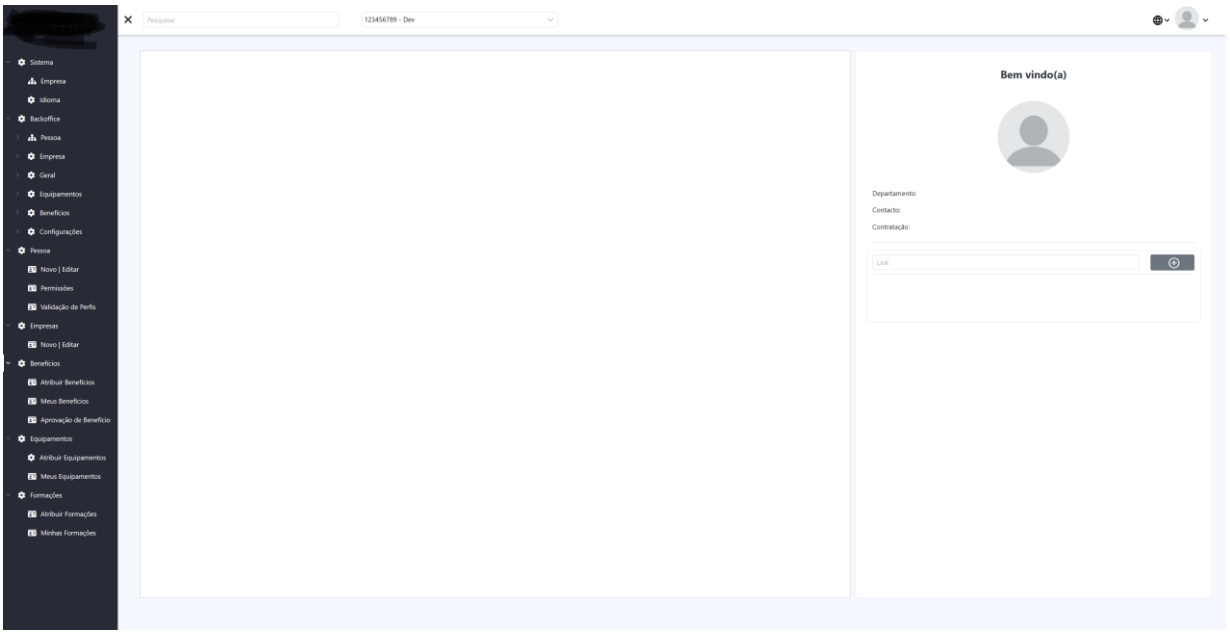


Figura 155 – Página Inicial nova aplicação

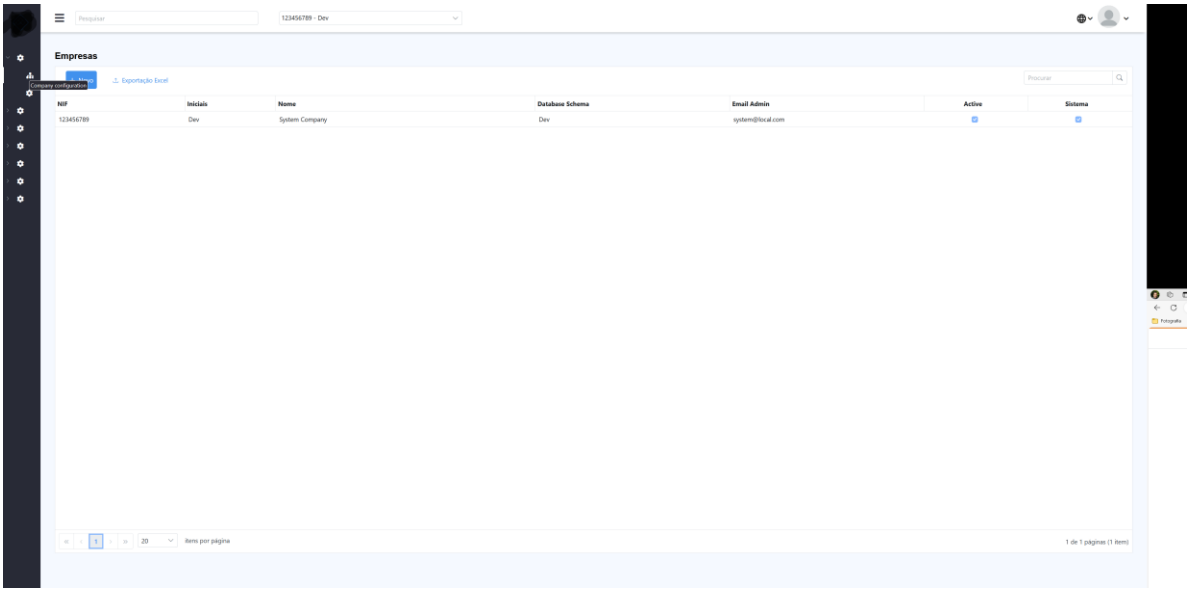


Figura 156 – Janela empresas, nova aplicação

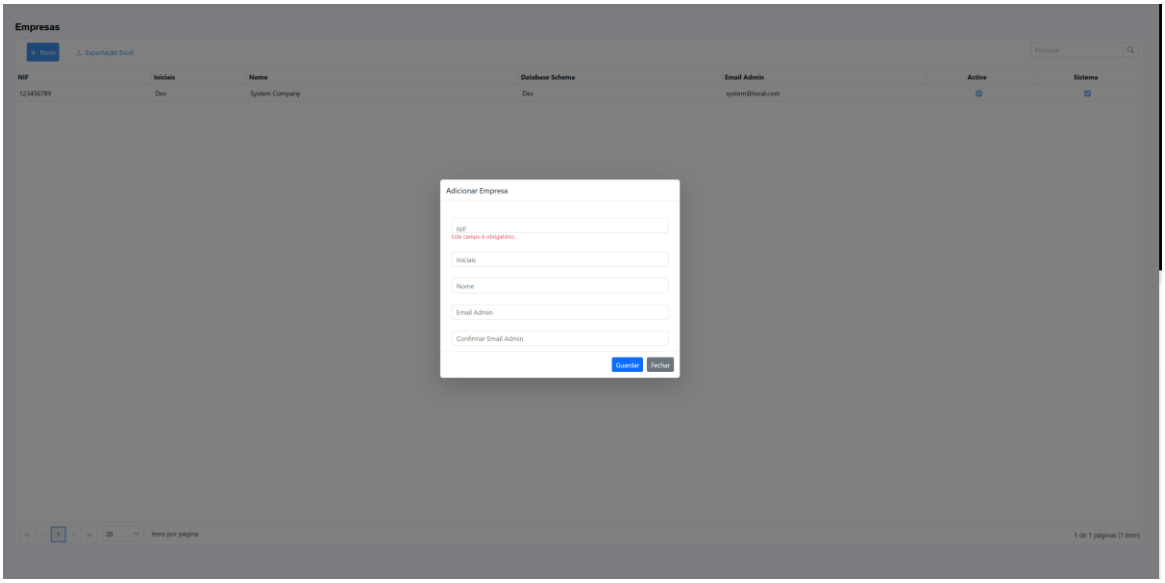


Figura 157 – Janela adicionar nova empresa, nova aplicação

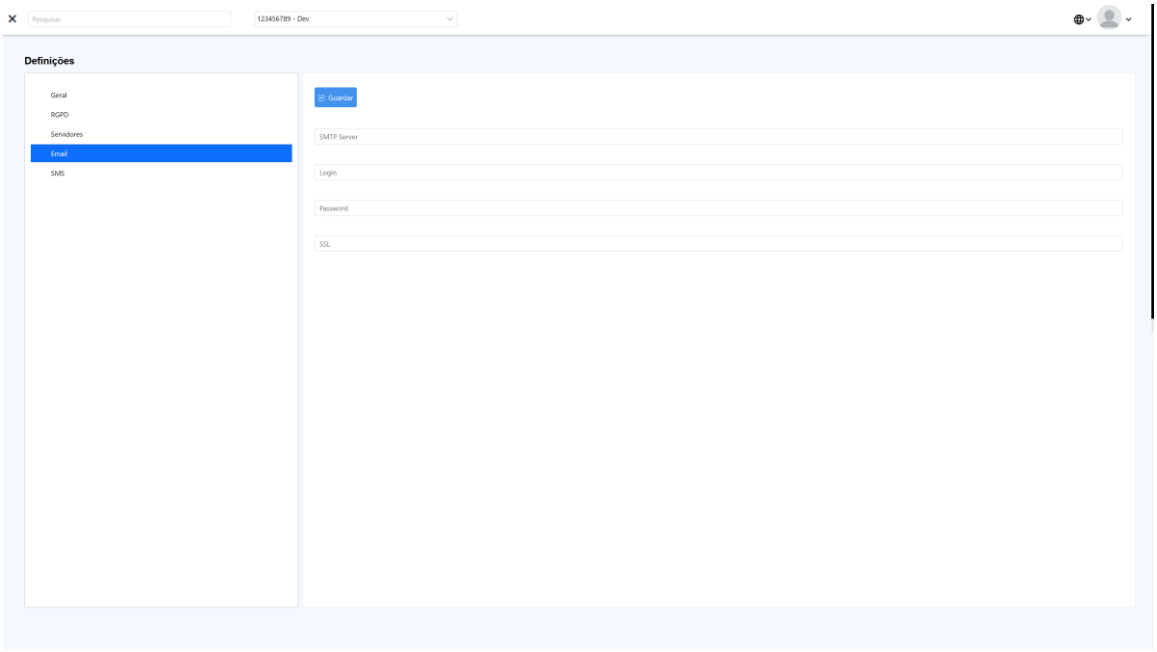


Figura 158 – Janela definições, nova aplicação

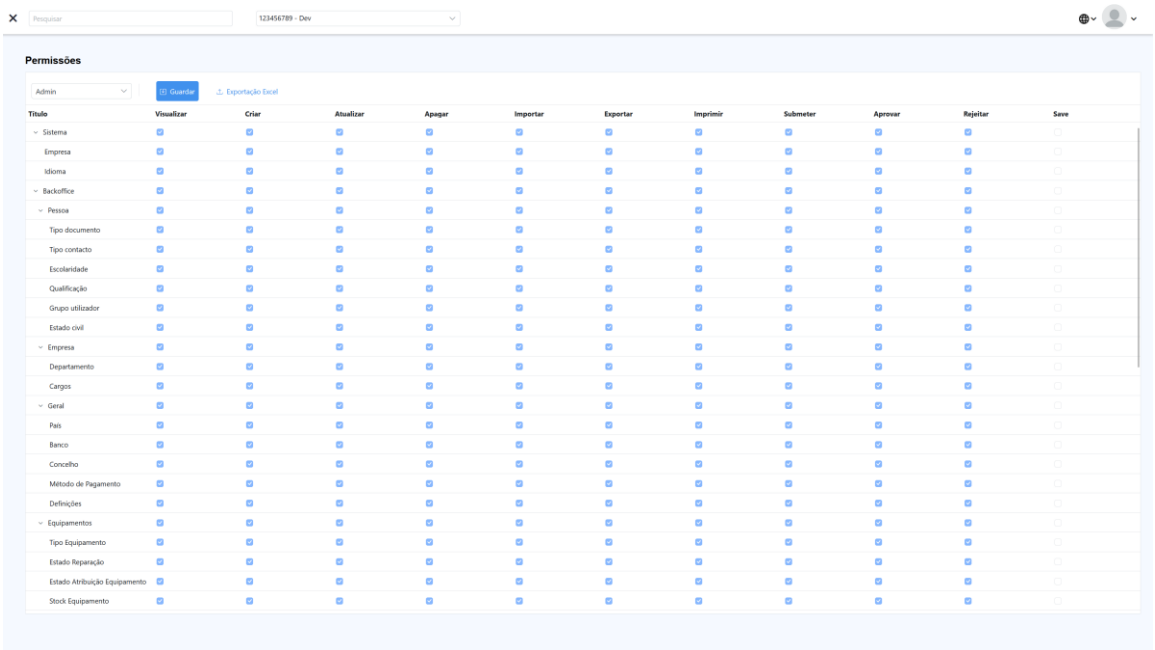


Figura 159 – Janela permissões menu, nova aplicação

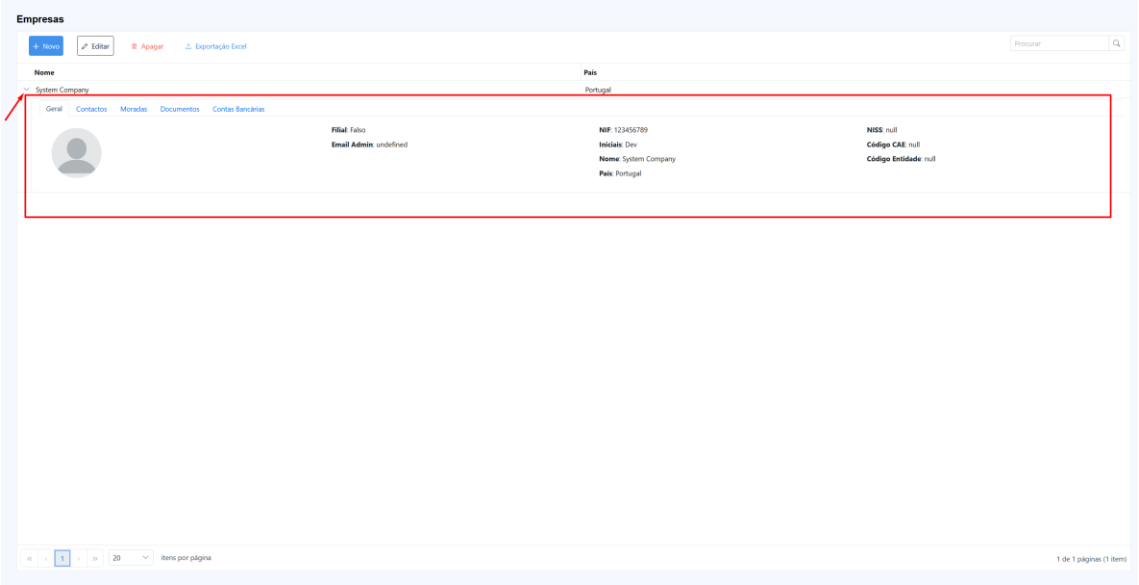


Figura 160 – Tabela hierárquica com detalhes, nova aplicação

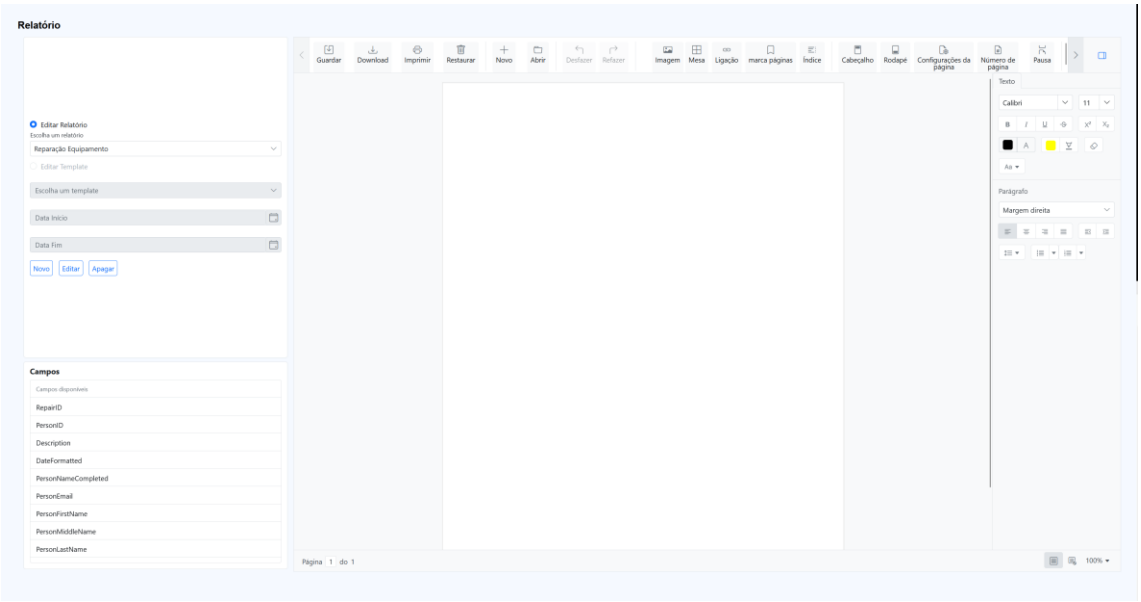


Figura 161 – Janela de construção de relatórios dinâmicos, nova aplicação