Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTAMENTO DE INFORMÁTICA E SISTEMAS

Mapeamento de Entidade-Relacionamento para Objeto-Relacional (ER2ORM)

Dissertação para a obtenção do grau de Mestre em Informática

Especialização em Desenvolvimento de Software

Autor

André Gonçalves Rodrigues

Orientadores

Prof. Jorge Bernardino, Instituto Politécnico de Coimbra

Prof. Vasco Pereira, Universidade de Coimbra

Coimbra, outubro de 2024

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Resumo

A conversão de modelos de base de dados em linguagens de programação é cada vez mais importante para diminuir a barreira conceptual entre estes dois modelos, permitindo assim aos programadores ter um melhor desempenho na integração dos mesmos. Neste trabalho foi desenvolvida a possibilidade de, através de um modelo conceptual de uma base de dados, poderem ser geradas as classes programáticas correspondentes a esse modelo. Esta solução foi desenvolvida através da integração de um microsserviço, com uma interface pública REST, que permite a conversão de scripts SQL em classes C#. Para desenvolver esta solução foi usada a plataforma Web ONDA (Online Database Architect) que permite a criação gráfica de modelos conceptuais, e a sua conversão automática no modelo físico, e posteriormente no script SQL. No final do desenvolvimento do microsserviço e da sua integração com a plataforma ONDA foi possível testar com sucesso o trabalho produzido nos motores de base de dados suportados pela plataforma. Este trabalho tornou possível que os utilizadores desta plataforma consigam pré-visualizar as classes a serem usadas numa solução de software para sua posterior integração com uma base de dados, facilitando assim o trabalho de arquitetura e desenho de uma solução de software numa fase inicial.

Palavras-chave: Mapeamento objeto-relacional, Entity Framework Core, Online Database Architect, Microsserviço.

Abstract

Converting database models into programming languages is becoming increasingly important in order to reduce the conceptual barrier between these two models, thus allowing programmers to perform better when integrating them. This work has developed the possibility of generating the programmatic classes corresponding to a database's conceptual model. This solution was developed by integrating a microservice with a public REST interface, which allows SQL scripts to be converted into C# classes. The ONDA (Online Database Architect) Web platform was used to develop this solution, which allows conceptual models to be created graphically and automatically converted into the physical model and then into the SQL script. At the end of the development of the microservice and its integration with the ONDA platform, it was possible to successfully test the work produced in the database engines supported by the platform. This work has made it possible for users of this platform to preview the classes to be used in a software solution for later integration with a database, thus facilitating the architectural and design work of a software solution at an early stage.

Keywords: Object-relational mapping, Entity Framework Core, Online Database Architect, Microservice.

Epígrafe

"Don't let the noise of others' opinions drown out your own inner voice."
Steve Jobs

Dedicatória

Uma mensagem especial para a minha mulher, os meus pais, os meus sogros, as minhas irmãs, e os meus avós que desde o primeiro momento me motivaram para a conclusão do mestrado.

Agradecimentos

Agradeço ao professor Jorge Bernardino, ao professor Vasco Pereira, e ao colega Gonçalo Carvalho por todo o apoio, força, e ajuda que me deram ao longo deste caminho por vezes sinuoso e duro. Sempre estiveram disponíveis desde o início, com reuniões muito frequentes, e sem os quais não seria possível a conclusão deste trabalho.

Ao Instituto Superior de Engenharia de Coimbra (ISEC), e respetivos docentes, por ao longo do meu percurso académico terem fornecido todo o conhecimento necessário para a elaboração deste trabalho.

Em particular, agradeço à minha mulher, família e amigos chegados, que apesar de todas as horas que não pude estar tão presente, sempre me apoiaram e motivaram para continuar este trabalho até ao fim.

A todos, os meus maiores agradecimentos.

Índice

Índice de tabelas

Índice de figuras

Lista de abreviaturas

API	*Application Programming Interface*
CLI	*Command-line interface*
CMS	*Content Management System*
CSS	*Cascading Style Sheets*
EF	*Entity Framework*
ER	Entidade Relacionamento
HTTP	*Hypertext Transfer Protocol*
HTML	*Hypertext Markup Language*
IDE	*Integrated Development Environment*
J2EE	*Java 2 Platform, Enterprise Edition*
KB	*Kilobytes*
Mbps	*Megabits* por segundo
NDA	*Non Disclosure Agreement*
ORM	*Object-relational mapping*
POO	Programação orientada a objetos
SQL	*Structured Query Language*
TCP	*Transmission Control Protocol*
UI	*User Interface*
URL	*Uniform Resource Locator*

Lista de siglas e acrónimos

ADO	*ActiveX Data Objects*
CISUC	Centro de Informática e Sistemas da Universidade de Coimbra
CRUD	*Create, Read, Update and Delete*
ISEC	Instituto Superior de Engenharia de Coimbra
JSON	*JavaScript Object Notation*
LINQ	*Language-Integrated Query*
ONDA	*Online Database Architect*
REST	*Representational State Transfer*
XAMPP	*Multiplatform, Apache, MySQL, PHP e Pearl*

1 Introdução

As bases de dados relacionais e os programas escritos em linguagens de programação orientadas a objetos (POO) são cada vez mais usados como parte integrante de soluções de software. Portanto, são necessárias novas soluções que facilitem a integração destes dois modelos distintos, cada qual com características muito peculiares.

Existe uma incompatibilidade de integração entre estes dois modelos devido às suas diferentes estruturas e modelos de dados. No caso das bases de dados relacionais a informação é representada em tabelas através de colunas e linhas, enquanto que nos modelos de POO é representada com propriedades e métodos.

No intuito de resolver estes problemas de correspondência têm sido desenvolvidos mapeadores objeto-relacional (ORM), os quais aproximam os modelos de bases de dados e os programáticos, implementando regras de conversão bem definidas. Este trabalho tem como objetivo permitir a conversão de diagramas entidade-relacionamento (ER) em classes programáticas escritas na linguagem C#.

Desde 1995, quando foi criado o TopLink para Java, o primeiro ORM, estes modelos de mapeamento têm-se tornado muito populares e usados por quase todas as linguagens de POO [1]. Ambler em [2] define o modelo de mapeamento objeto-relacional mais usado pela indústria, no qual são resolvidos os problemas de correspondência com as POO como hierarquias e relações. Este modelo define como os objetos no código programático podem ser mapeados em tabelas de base de dados, de forma a preservar a estrutura e as relações entre os objetos da aplicação, e vice-versa.

Para o desenvolvimento de um ORM deve ser tido em consideração o desempenho dos modelos de conversão existentes na literatura, para que consoante a necessidade de aproximação dos modelos de base de dados e programático se obtenha um produto otimizado ao cenário em questão. Ou seja, consoante as regras escolhidas no desenvolvimento de um ORM poderemos aproximar mais ou menos os dois modelos distintos, mas isso poderá trazer à solução impactos negativos de desempenho. Por isso, o ideal será sempre perceber antecipadamente qual o nível de aproximação dos modelos que pretendemos atingir.

A Microsoft criou o seu primeiro ORM, designado Entity Framewrok, através de um modelo chamado ADO.Net. Este modelo consiste numa biblioteca de classes que permite aos programadores usarem classes em C# para acederem às bases de dados. O modelo é baseado nas normas usadas por Chen em [3], onde é definido o modelo entidade-relacionamento. Muitas das decisões seguidas no desenvolvimento deste

ORM também seguem as linhas de implementação usadas por Ambler em [2].

Neste trabalho optou-se por usar um ORM já existente, por forma a alcançar os objetivos pretendidos de conversão de modelos ER em classes de programação numa plataforma Web. O ORM usado no desenvolvimento deste trabalho foi a Entity Framework Core (EF Core) definida como requisito inicial. A EF Core é uma biblioteca de código livre criada pela Microsoft, a sucessora da antiga biblioteca Entity Framework. A documentação desta biblioteca pode ser consultada em [4]. A decisão por este ORM deveu-se muito ao facto de ser uma biblioteca de código livre e especialmente por ser uma biblioteca multiplataforma, sendo assim possível a sua instalação em diferentes ambientes.

A realização deste trabalho tem essencialmente dois objetivos:

1. Criação de uma nova funcionalidade usando a linguagem C#, de criação de modelos programáticos a partir de scripts SQL, que será integrada numa plataforma Web já existente (ONDA);
2. O desenvolvimento de um microsserviço em .NET Core acessível através de uma Web API pública.

A plataforma Web ONDA desenvolvida no Centro de Informática e Sistemas da Universidade de Coimbra (CISUC) foi o ponto de partida para o desenvolvimento destes novos requisitos. O ONDA é uma plataforma Web que permite elaborar a arquitetura de uma base de dados desde a sua génese, através da criação do modelo ER, a geração automática de um modelo físico e a geração do script SQL correspondente. No âmbito deste projeto será integrada uma nova funcionalidade para visualização das classes programáticas geradas, para que numa fase de desenho e construção do modelo de dados o programador seja capaz de antever necessidades de arquiteturas e de implementação necessárias ao desenvolvimento de um software.

Os principais contributos deste trabalho são os seguintes:

- Conversão de modelos ER em classes programáticas na plataforma ONDA.
- Desenvolvimento de um microsserviço que apresenta uma Web API REST publica, responsável por todo o processo de conversão, compatível com tecnologia de container Docker.
- Adaptação da interface Web da plataforma ONDA para integrar esta nova funcionalidade de apresentação de classes C# ao utilizador.

Este documento está organizado em seis capítulos:

Capítulo 1: Onde foi feita uma introdução ao tema abordado fornecendo o contexto, a motivação, os objetivos, e a estrutura do documento.

Capítulo 2: Apresenta os trabalhos mais importantes realizados sobre ferramentas de

ORM, de conversão entre os modelos programáticos e os modelos de armazenamento de dados.

Capítulo 3: São introduzidos os conceitos necessários para entender a plataforma Web ONDA usada e modificada neste projeto, e introduzido o processo de *scaffolding* da EF Core (*Entity Framework Core*) da Microsoft.

Capítulo 4: Descreve a metodologia usada, a especificação dos requisitos para a solução desenvolvida, e as decisões tomadas para o desenvolvimento da solução.

Capítulo 5: É explicado a implementação da solução desenvolvida, descritos os testes funcionais executados, e por fim analisada e discutida a aplicação da solução criada.

Capítulo 6: É descrito o trabalho realizado e qual a sua importância. Quais as suas vantagens e limitações. E, por fim, são indicadas as possíveis aplicações noutros contextos, e apresentadas algumas ideias para trabalho futuro.

2 Trabalhos relacionados

Neste capítulo é apresentada uma revisão bibliográfica dos trabalhos mais importantes realizados sobre ferramentas de ORM, de conversão de modelos de armazenamento de dados em modelos programáticos.

A definição de um modelo de mapeamento de objetos em tabelas pode seguir diferentes metodologias. Em [2] são definidas várias regras para resolver problemas de mapeamento objeto-relacional como as hierarquias e as relações. Existem quatro maneiras diferentes de mapear hierarquias: mapear as classes numa única tabela, com todas as propriedades como colunas da única tabela criada; mapear cada classe concreta na sua tabela, repetindo as propriedades em todas as tabelas como colunas; cada classe ser mapeada na sua própria tabela, tendo cada tabela apenas as colunas específicas de cada classe concreta; ou mapear as classes numa estrutura genérica de tabelas. Para mapear as relações existem quatro abordagens consoante o tipo da relação: um para um, uma para muitos, muitos para muitos e mapeamento de coleções ordenadas por um determinado identificador com relações recursivas. Após a definição das regras deste modelo é explicado que os mapeamentos feitos com base neste modelo podem ter vários ajustes de desempenho devido às várias abordagens de conversão apresentadas. Em [5] é reforçado o uso deste modelo sendo explicadas as diversas formas como pode ser implementado em cenários reais e quais as implicações no modelo programático gerado.

Lorenz et al. em [6] reforça várias das regras do modelo mencionado em [2]. É feita uma análise comparativa de desempenho entre distintas abordagens destes modelos. O trabalho permite aos programadores retirarem conclusões sobre qual a abordagem que devem seguir. Assim, consoante o cenário abordado e a complexidade arquitetural, podemos optar pelo modelo mais adequado. É referido também que os ORM adicionam complexidade no acesso às bases de dados e que estas questões de desempenho devem ser tidas em consideração, porque uma pequena alteração no modelo poderá ter grande impacto. Para fazer esta análise de desempenho foram usadas as seguintes métricas: *Lookup*, *consumo de memória*, *RangeSelectPolymorph*, *TableScanPolymorph* e *AddAttribute*. A métrica *Lookup* mede o tempo de operações de CRUD (*Create*, *Read*, *Update*, *Delete*), a métrica *consumo de memória* mede a memória utilizada para guardar os objetos da hierarquia na base de dados, a métrica *RangeSelectPolymorph* mede o tempo de execução de consultas recorrendo a filtros por atributos indexados, a métrica *TableScanPolymorph* mede o tempo de execução de consultas recorrendo a filtros por atributos não indexados, e a métrica *AddAttribute* mede o tempo de execução de operações de

criação, alteração, e eliminação de tabelas.

Keller em [7] expõem vários exemplos práticos do modelo apresentado em [2, 5, 6], onde cria soluções para problemas concretos e analisa as devidas consequências. Cada exemplo é composto pelo problema, quais as forças analisadas, solução, estrutura, consequência, implementação e padrões relacionados. É abordada uma outra necessidade das ferramentas de mapeamento que é a questão dos tipos de dados entre entidades relacionais e objetos. Assim como em [8] os autores chegam à conclusão que existindo grande desajuste entre o modelo de dados e o modelo programático isso pode provocar um impacto negativo na eficiência dos ORM's. A escolha do ORM que melhor se adapta ao caso de uso e que ofereça boas práticas de lidar com o desajuste mencionado, e a implementação de otimizações personalizadas para ações mais complexas, ajudam na redução deste impacto negativo.

Para resolver problemas como a questão das relações recursivas no livro [8] Colley, Derek and Stanier, Clare and Asaduzzaman, também mencionadas em [7], definem como lidar eficientemente com relações recursivas em ORM's. As relações recursivas definem-se como aquelas em que uma entidade mantém uma relação consigo mesma. Estes casos apresentam desafios específicos de modelagem, consulta e desempenho. Para lidar com estas situações, devem ser identificadas as relações recursivas e configurá-las corretamente na EF Core, o que inclui mapear as entidades e definir as respetivas chaves estrangeiras. Nesta abordagem, é essencial evitar o problema de execução de consultas em excesso (problema N + 1), que ocorre quando uma consulta inicial desencadeia múltiplas consultas adicionais desnecessárias. Para tal, é recomendável otimizar as consultas de forma a garantir a eficiência de execução das mesmas. O estudo descrito neste livro permite um melhor desempenho no uso de relações recursivas na EF Core, ajudando assim a uma maior escalabilidade do sistema, e a uma melhor manutenção usando padrões de desenvolvimento. É referido que o uso do ORM nestes casos poderá ter um custo no desempenho da solução em operações mais complexas, dificuldade para encontrar falhas, e a dependência de funcionalidades específicas da EF Core.

O uso de ORM's na implementação de aplicações traz desafios e alterações ao processo de desenvolvimento assim como explicam os autores do artigo [9]. Este artigo estuda os impactos das alterações necessárias, ao longo de um processo de desenvolvimento de uma aplicação, no código produzido. Estas alterações poderão depender muito das alterações feitas à base de dados, e à alteração das relações entre as várias entidades na base de dados. Neste artigo foram usados três casos de uso de um parceiro comercial que usa o ORM JPA para Java no qual se verificou que existe uma preocupação acrescida na manutenção do código usando o ORM, e um sistema industrial. Os três casos de uso mencionados no estudo são: *Broadleaf Commerce*, que é um sistema de e-commerce utilizado em contextos de código aberto comerciais que permite funcio-

nalidades robustas para gerir transações comerciais online; *Devproof Portal*, que é uma aplicação de gestão de blogs personalizados; *JeeSite*, que é uma framework para gerir informações empresariais através de um sistema de gestão de conteúdos (CMS), e uma plataforma administrativa. Para além destes três casos de uso o estudo também incluiu um sistema industrial com mais de 300 mil linhas de código de uso global, mas sobre o qual os detalhes não são divulgados devido a um acordo de confidencialidade (NDA). Apesar de neste estudo mencionarem que o ORM pode ajudar os programadores a reduzir os custos de manutenção do código, referem que as alterações no código podem ser complexas devido ao ORM não conseguir encapsular completamente o acesso às bases de dados em objetos. Apesar de escreverem que o código usado pelo ORM deva ser independente da base de dados alertam para a possibilidade de nem sempre ser possível o que aumenta os problemas de manutenção. É realçado a importância de ferramentas de testes e validação do código gerado, e também a importância de revisões de código focadas na qualidade dos mapeamentos ORM.

Na indústria, há vários ORM's muito populares que aplicam os modelos de conversão supra mencionados. Alguns exemplos são: ActiveJPA [10], CoreData [11], Dapper [12], DjangoORM [13], Doctrine [14], EclipseLink [15], Entity Framework [16], Entity Framework Core [4], Linq2Db [17], MyBatis [18], NHibernate [19], Sequelize [20], e SQLAlchemy [21]. Neste trabalho foi usado o ORM EF Core como requisito de integração. Normalmente os ORMs mencionados são desenvolvidos para uma linguagem específica, sendo predominantemente para a linguagem Java ou C#. No projeto [22] Forsberg faz uma comparação entre dois ORM's a EF Core e o Dapper usando métricas como o desempenho, a facilidade de uso, a flexibilidade e a adequação a diferentes cenários aplicacionais. Através destas métricas são extraídas as vantagens e desvantagens de cada um destes ORM permitindo uma conclusão final acerca de ambos explicada a seguir. Através da análise percebe-se que a EF Core é de fácil uso, dispõem de muitas ferramentas de produtividade como migrações automáticas, faz uma modelagem completa dos dados incluindo herança, relacionamento e restrições, integração completa com o IDE Visual Studio, e abstrai muitos detalhes de manipulação direta com a base de dados via SQL usando o LINQ. São também listados aspetos negativos, como o desempenho em consultas mais complexas e em massa, a introdução de sobrecarga adicional no processamento de consultas, a pouca flexibilidade em consultas altamente otimizadas e especificas de um determinado motor de base de dados, e curva de aprendizagem acentuada. O Dapper apresenta um elevado desempenho, flexibilidade nas consultas bastante alta, adiciona pouca sobrecarga sendo assim muito leve, e é muito fácil de integrar no .NET. Quanto às desvantagens é de mais difícil manutenção devido à necessidade de código SQL espalhado pelo código C#, não tem suporte nativo para migrações e modelação de dados sendo o processo manual, não oferece validações de esquema da base de dados, e permite uma maior probabilidade de erros ao escrever SQL diretamente no código C# por este não ser compilado e só apenas validado em

tempo de execução. Este estudo conclui que a escolha entre os dois ORM depende sempre das especificidades de cada projeto. Embora a EF Core seja mais adequada a aplicações que necessitam de mais abstração do modelo de dados, e que têm a necessidade constante de migrações, o Dapper é ideal para cenários onde o desempenho e a flexibilidade são um requisito.

Em [23] os autores abordam o ORM Hibernate e como focar numa implementação eficiente no contexto da biblioteca J2EE. O Hibernate é das mais populares ferramentas de ORM que simplifica o processo de integração com as bases de dados para a linguagem Java. No artigo mencionado anteriormente são abordadas diversas vantagens e desvantagens da implementação deste ORM sendo que o autor realça as vantagens em termos de automação e a abstração para a camada de dados. Devem ser tidos em consideração os aspetos de desempenho e complexidade na construção de algumas consultas mais complexas.

Considerando a escolha da linguagem de programação C# para este trabalho, e consequente utilização do ORM EF Core, o estudo focou-se nas metodologias implementadas pela EF Core ao longo da sua evolução. A Microsoft na génese do seu ORM criou uma ferramenta chamada ADO.NET que implementa conversões do modelo de dados para o modelo programático, consistindo numa biblioteca de classes pertencente à entity framework. Esta ferramenta permite aos programadores de .NET abstraírem-se do modelo de dados e usar C# para aceder à base de dados. O modelo EDM (*Entity Data Model*) do ADO.Net é o modelo responsável por guardar como os mapeamentos entre o modelo de dados e o modelo programático são feitos. Este modelo criado pela Microsoft é baseado no modelo entidade-relacionamento descrito em [3]. Este modelo de conversão usa um ficheiro xml intermédio que é responsável por definir os mapeamentos entre as tabelas da base de dados e as classes em código C#. Os pontos fundamentais descritos por este modelo são: a definição dos tipos de entidades em que cada uma deve ter uma chave de entidade única e que vão representar as classes; os tipos de associações que são definidos por propriedades de navegação permitindo representar as relações uma ligação, nenhuma ou uma, ou muitas; e por fim, as propriedades que definem a estrutura das entidades e a forma das classes. Estas propriedades podem usar os dados primitivos, tais como Binary, Boolean, DateTime, Int32, Float, entre outros definidos em [24].

O ORM da Microsoft mencionado em [4] usa uma tecnologia de engenharia inversa que também é explicada por Premerlani, W.J. e Blaha, M.R., no livro [25]. Este livro apresenta uma abordagem de engenharia inversa a partir de esquemas físicos de uma base de dados existente. Esta abordagem é útil para a manutenção, migração, e modernização de sistemas desenvolvidos no passado. Quanto à metodologia desta abordagem é feita em três passos: Análise Estática, esta identifica tabelas, colunas, chaves primárias e estrangeiras; Recuperação dos metadados, são extraídos os metadados da

base de dados incluindo restrições de integridade e índices, Modelagem conceptual, criação de um modelo conceptual que representa a lógica da base de dados permitindo a visualização das relações entre entidades. Este livro define como vantagens desta abordagem uma melhor compreensão da base de dados especialmente em sistemas antigos, facilitar a migração de dados através de uma visão clara do esquema da base de dados, permitir mais facilmente a deteção de incoerências e inconsistências na base de dados, ajudar na melhoria continua de sistemas antigos adotando novas tecnologias e boas práticas de desenvolvimento. O processo de engenharia inversa também trás algumas desvantagens, como descrito, complexidade e demora do processo especialmente em bases de dados grandes, nem sempre é possível uma extração semântica eficaz e completa a partir dos modelos físicos, a eficiência do processo está muito dependente da qualidade dos metadados disponíveis na base de dados, exige a utilização de ferramentas especializadas que não são fáceis de usar e a sua disponibilidade pode não ser de fácil acesso, e por fim a resistência à mudança no uso de um novo processo ao implementar as novas técnicas e ferramentas.

A biblioteca Entity Framewrok tem evoluído até aos dias de hoje para se tornar multiplataforma e de código aberto, dando assim origem à EF Core, que será explicada no próximo capítulo. A escolha deste ORM deve-se ao facto de ser uma biblioteca de código aberto, e especialmente por ser uma biblioteca multiplataforma que permite a sua instalação em diferentes ambientes. Sendo este ORM multiplataforma também pode ser instalado em containers Docker com sistema operativo Linux ou Windows, como foi feito no âmbito deste trabalho.

3 Plataforma Web ONDA, *Entity Framework Core* e *Scaffolding*

Neste capítulo são introduzidos os conceitos necessários para entender a plataforma Web ONDA usada neste projeto, e para entender também o processo de *scaffolding* - engenharia inversa [26] da EF Core (*Entity Framework Core*) da Microsoft.

Na Secção 3.1 é apresentada a plataforma Web ONDA, usada neste projeto como interface e como fonte do código SQL que será posteriormente convertido num modelo programático.

Na Secção 3.2 é feita a descrição da EF Core e do *Scaffolding*.

Na Secção 3.3 é dado um exemplo de como funciona o processo de *Scaffolding* da EF Core.

Na Secção 3.4 é feita uma análise das transformações feitas pela EF Core em dois motores de base de dados (SQL Server e PostgreSQL).

3.1 Introdução à plataforma Web ONDA

O ONDA é uma plataforma Web de modelação de bases de dados, desenvolvida no Centro de Informática e Sistemas da Universidade de Coimbra (CISUC) [27], que utiliza tecnologias como HTML, CSS e Javascript. Esta ferramenta foi desenvolvida para facilitar a criação de bases de dados, sendo acessível tanto para iniciantes como para especialistas. Permite criar diagramas de entidades, definir relações entre tabelas e configurar tipos de dados e restrições lógicas das colunas. A ferramenta gera automaticamente o modelo físico a partir do modelo conceptual (ER), e posteriormente, os scripts SQL para um conjunto de motores de bases de dados, tornando-se uma solução bastante útil para programadores, estudantes, e profissionais na gestão de dados. Com o trabalho desenvolvido no âmbito deste projeto, a ferramenta agora também possui a funcionalidade de gerar classes programáticas em C# via ORM, após a geração automática dos scripts SQL.

A plataforma ONDA na sua interface gráfica tem uma barra de ferramentas no topo que permite fazer várias funções relacionadas com a criação visual de modelos ER. Na Figura 3.1 podemos ver dois botões essenciais da barra de tarefas, o número 1 que permite adicionar novas tabelas ao modelo visual ER, e o botão número 2 que permite adicionar novas relações entre tabelas presentes no modelo. Também na barra

de tarefas no topo podemos ver uma caixa de seleção que permite selecionar o motor de base de dados usado para gerar os scripts SQL, na Figura 3.1 podemos ver esta caixa selecionada com o valor PostgreSQL 9.6.1. Na vista presente no lado direito (área de propriedades) é possível editar detalhes das entidades adicionadas ao ecrã, como campos e os seus tipos de dados, e ainda outras propriedades mais avançadas como restrições, como se pode ver na Figura 3.2. Nesta área também é possível alterar os detalhes das relações entre as entidades como podemos ver na Figura 3.3. É aqui que são definidas as opções de cardinalidade entre as entidades envolvidas, e se a mesma é fraca ou não.

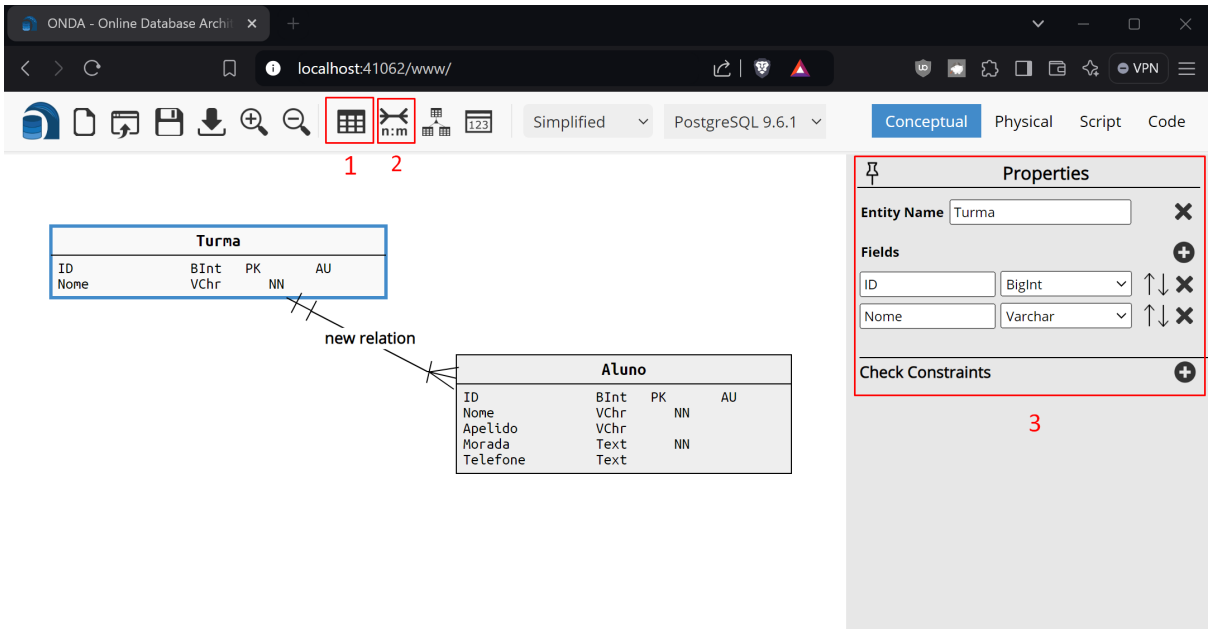

Figura 3.1: Interface gráfica do ONDA - criação de modelo ER.

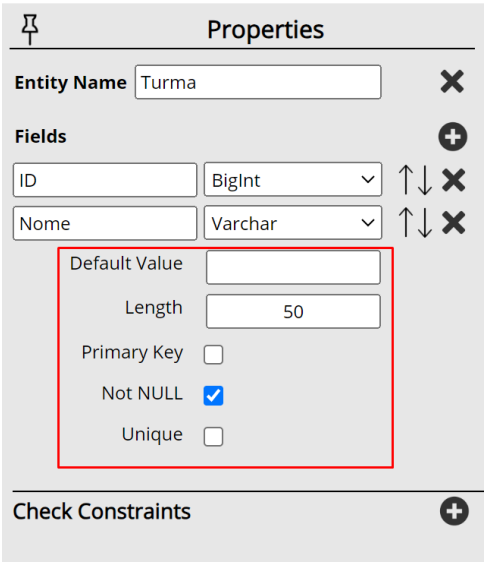

Figura 3.2: Interface gráfica do ONDA - propriedades de uma entidade e dos seus campos.

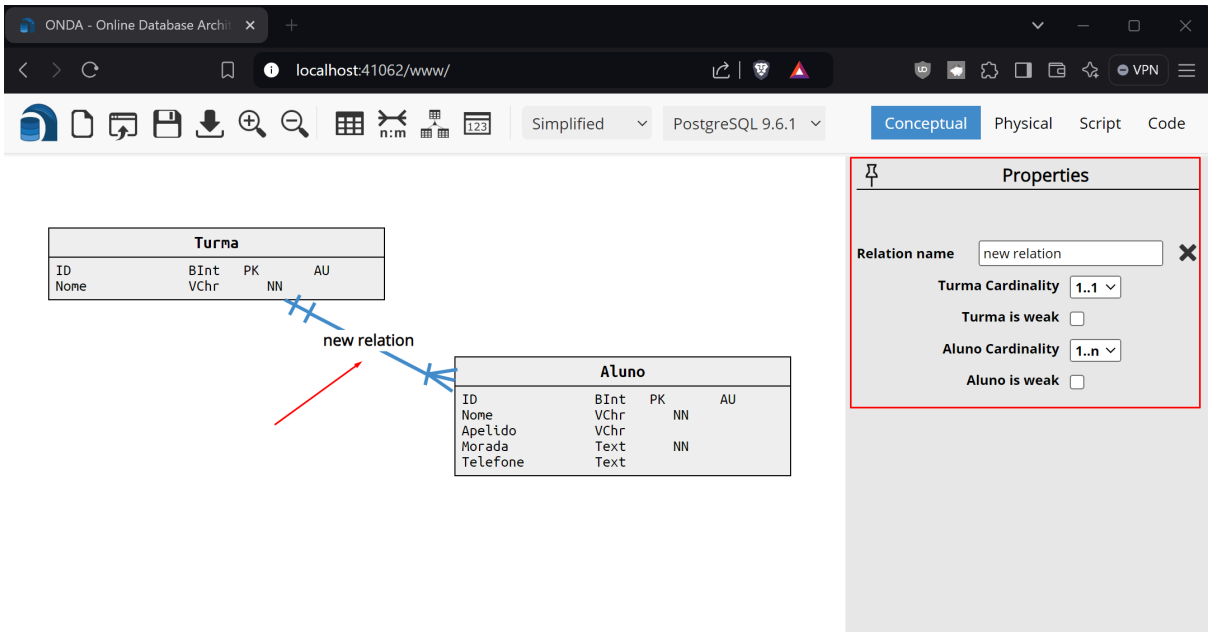

Figura 3.3: Interface gráfica do ONDA - propriedades de uma entidade e dos seus campos.

Após a criação manual do modelo ER, na interface gráfica do ONDA, é possível na barra de tarefas da plataforma clicar na aba "modelo físico", e ver o modelo gerado automaticamente pelo ONDA como podemos ver na Figura 3.4. Após a visualização do modelo físico pode-se navegar para a aba "script", e assim visualizar o script SQL produzido pelo ONDA que representa a base de dados desenhada no modelo ER no inicio do processo (ver Figura 3.5).

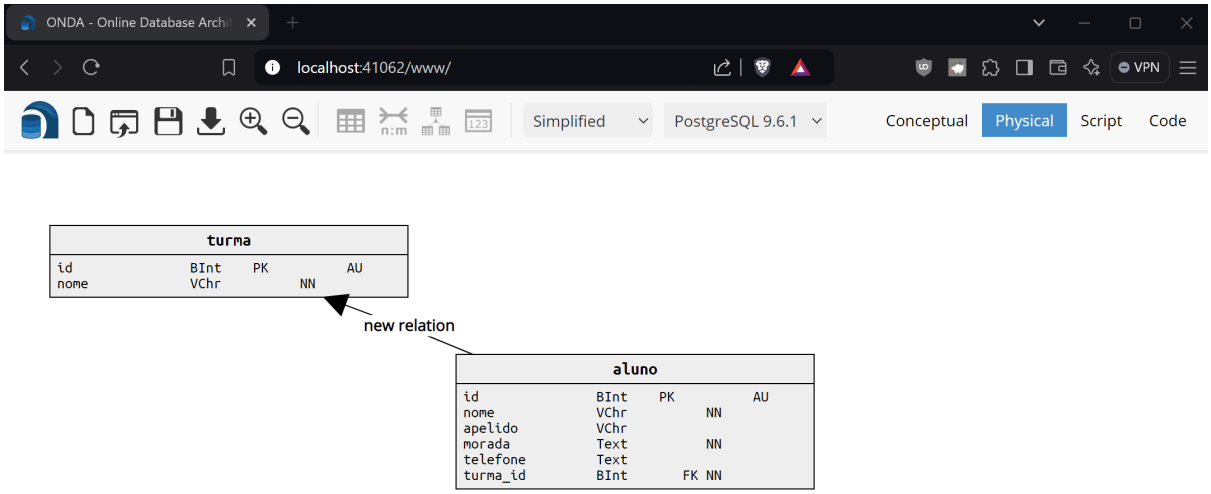

Figura 3.4: Interface gráfica do ONDA - modelo físico gerado automaticamente.

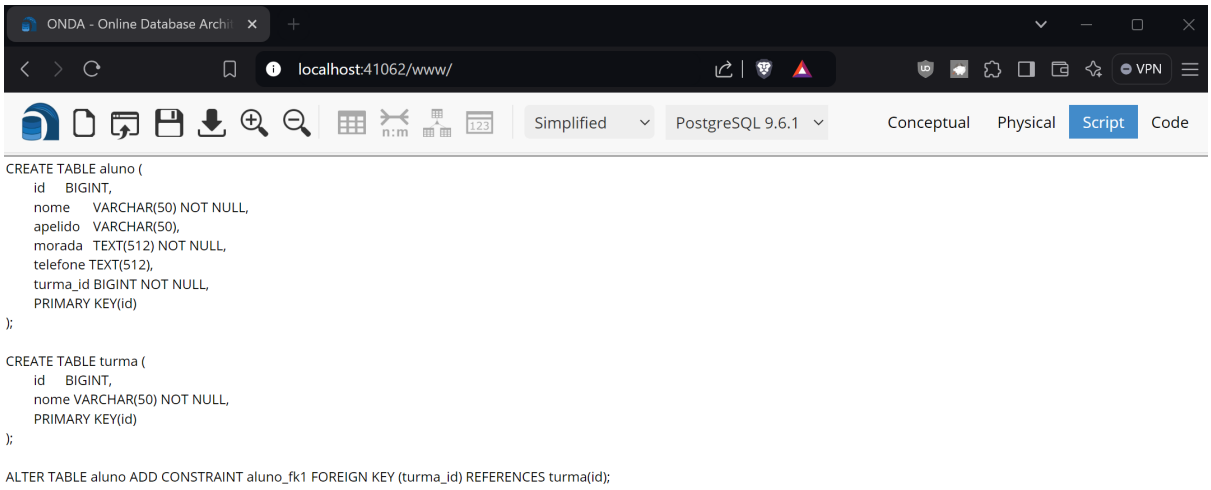

Figura 3.5: Interface gráfica do ONDA - script SQL gerados automaticamente.

A plataforma Web ONDA, introduzida nesta secção, será utilizada posteriormente na secção 5.2 para integrar o microsserviço desenvolvido neste projeto, que converte scripts SQL em classes C#.

3.2 Introdução ao EF Core

A EF Core é uma biblioteca de código aberto e multiplataforma que funciona como ORM que surgiu com o objetivo de alargar a anterior tecnologia da Microsoft *Entity Framework* conforme é explicado em [24].

Esta biblioteca desenvolvida em .NET permite aos programadores o mapeamento entre as entidades de uma base de dados e o código desenvolvido em C#, e vice-versa. Através desta biblioteca, é facilitada a escrita de consultas a bases de dados através de uma linguagem unificada, o C#. Assim como, a fácil transição da tecnologia da base de dados sem ser necessário a adaptação do código desenvolvido. Neste caso, o programador não precisa de ter conhecimentos de SQL pois todos os acessos feitos à base de dados podem ser feitos através da biblioteca LINQ (*Language-Integrated Query*) que é uma sintaxe unificada em C#, sendo esta compatível com todas as base de dados suportadas pela EF Core [28].

Para fazer a parte de conversão entre o modelo programático e modelo da base de dados, é possível usar a EF Core. Para isso existem três metodologias possíveis de abordar por parte dos programadores, sendo elas:

- Gerar automaticamente através do *scaffolding* da EF Core as classes C# a partir de uma base de dados existente.
- Escrever as classes em C# que correspondem à base de dados manualmente.
- Escrever, inicialmente, as classes em C# e gerar a base de dados a partir do código.

Neste caso, é preciso usar um modelo de migrações para atualizar a base de dados à medida que o código C# representativo da modelo da base de dados vai sendo alterado.

Como já mencionado acima, e em [26], o *scaffolding* do ORM EF Core permite gerar automaticamente as classes na linguagem C# a partir de uma base de dados já instanciada num servidor, que execute um dos motores de base de dados suportados, tais como, SQL Server, PostgreSQL, MySQL, Oracle, SQLite, entre outros [29]. Para usar esta ferramenta são necessários alguns pré requisitos e o uso da *command-line interface* (CLI) - linha de comandos do .NET - presente no IDE (ambiente de desenvolvimento integrado) Visual Studio, e em alguns outros IDE's compatíveis com o .NET CLI. É necessário instalar a biblioteca `Microsoft.EntityFrameworkCore.Design` presente em *Nuget Gallery* [30], que consiste num repositório de pacotes de .NET. Depois temos que instalar o pacote `Microsoft.EntityFrameworkCore.Tools`, e por fim instalar o conetor para a/as base/es de dados que estivermos a usar e sobre a/as qual/quais pretendemos usar o *scaffolding*.

Num projeto .NET, para se poder executar o *scaffolding* primeiramente deve ter-se definida a ligação à base de dados que nos indica o nome da base de dados a ser usada, qual o servidor onde está instanciada (IP, porto, nome da instância da base de dados, quais as credenciais de acesso (utilizador e palavra-passe), e qual o nome da base de dados (Catalog)). Podemos ver um exemplo desta configuração abaixo.

```
1 {
2   "ConnectionStrings": {
3     "MyDatabaseConnection": "Server=ENDERECO_SERVIDOR;
        Database=NOME_BASE_DE_DADOS;User Id=UTILIZADOR_BD;
        Password=PALAVRA_PASSE_BD;"
4   }
5 }
```

Código 3.1: Exemplo de configuração JSON da ligação à base de dados

Depois da ligação à base de dados definida para gerar as classe em C# deve ser executado o seguinte comando no .NET CLI.

```
1 dotnet ef dbcontext scaffold "Name=ConnectionStrings:
    MyDatabaseConnection" Microsoft.EntityFrameworkCore.SqlServer
```

Código 3.2: Exemplo de comando *scaffolding* em base de dados Sql Server

Este comando contém o nome da ligação à base de dados, supra definida, sendo a mesma usada para estabelecer uma ligação à base de dados, e contém o nome do pacote usado como biblioteca de conversão da base de dados para código C#. Alguns exemplos de plug-in de conversão disponíveis são Mi-

crosoft.EntityFrameworkCore.SqlServer, Microsoft.EntityFrameworkCore.Sqlite, MySql.EntityFrameworkCore, MongoDB.EntityFrameworkCore, Oracle.EntityFrameworkCore. Existem mais exemplos que estão disponíveis em [29] e os quais podem ser usados para integrar as bases de dados com a EF Core.

3.3 Exemplo do processo de *Scaffolding*

Para criar este processo exemplificativo foi usada uma base de dados genérica de uma plataforma de vendas numa instância de SQL Server com o diagrama presente na Figura 3.6. Esta base de dados permite gerir o armazenamento de encomendas de produtos feitas por clientes e atendidas por funcionários. Sendo, os produtos organizados por categoria e marca, e que fazem parte do stock de uma loja física. Também foi criado um projeto .NET de uma *Web API* para executar o processo de *scaffolding* e obter os outputs pretendidos.

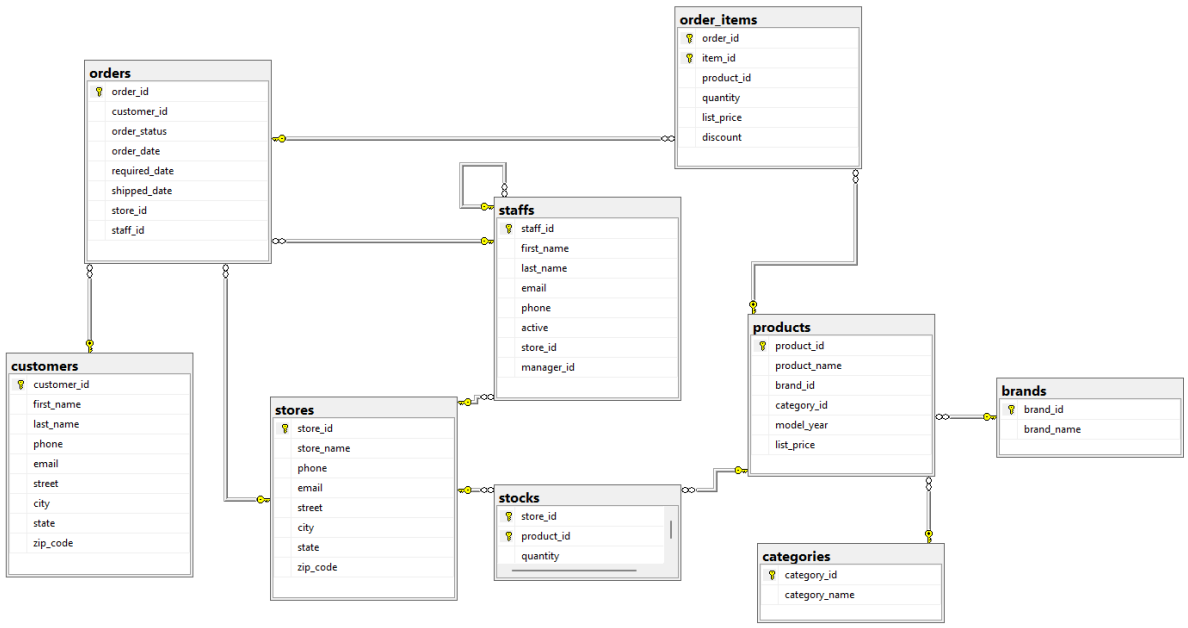

Figura 3.6: Diagrama de base de dados SQL Server de plataforma de vendas.

Em primeiro lugar, na aplicação foi adicionado no ficheiro `appsettings.json` a ligação à base de dados com o nome "*SalesConnection*", de acordo com o fragmento de código a seguir.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
```

```
7   },
8   "AllowedHosts": "*",
9   "ConnectionStrings": {
10    "SalesConnection": "Server=localhost;Database=Sales;User
        Id=sa;Password=*********;TrustServerCertificate=True;
        Trusted_Connection=False;"
11  }
12 }
```

Código 3.3: Configuração de testes para base de dados Sql Server

Seguidamente, foi preciso adicionar todas as bibliotecas necessárias, que falámos anteriormente:

- `Microsoft.EntityFrameworkCore.Design`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.EntityFrameworkCore.SqlServer`

Posteriormente, foi necessário executar o comando do excerto de Código 3.4 onde é possível definir a diretoria onde as classes são geradas através da opção `-output-dir NOME_DIRETORIA`, e também definir o projeto alvo com `-project PROJECT_NAME`. Este comando ao ser executado usa a configuração de ligação à base de dados com o nome "*SalesConnection*", e o plug-in de conversão `Microsoft.EntityFrameworkCore.SqlServer`. Os ficheiros gerados são guardados numa diretoria especificada chamada *Models*, denominando o projeto alvo por *TestSales*. Como podemos ver na Figura 3.7, foram gerados ficheiros C# correspondentes ao modelo de dados de testes usado, de acordo com a Figura 3.6.

```
1 dotnet ef dbcontext scaffold "Name=ConnectionStrings:
    SalesConnection" Microsfot.EntityFramewrokCore.SqlServer --
    output-dir Models --project TestSales
```

Código 3.4: Execução do comando de *scaffolding*.

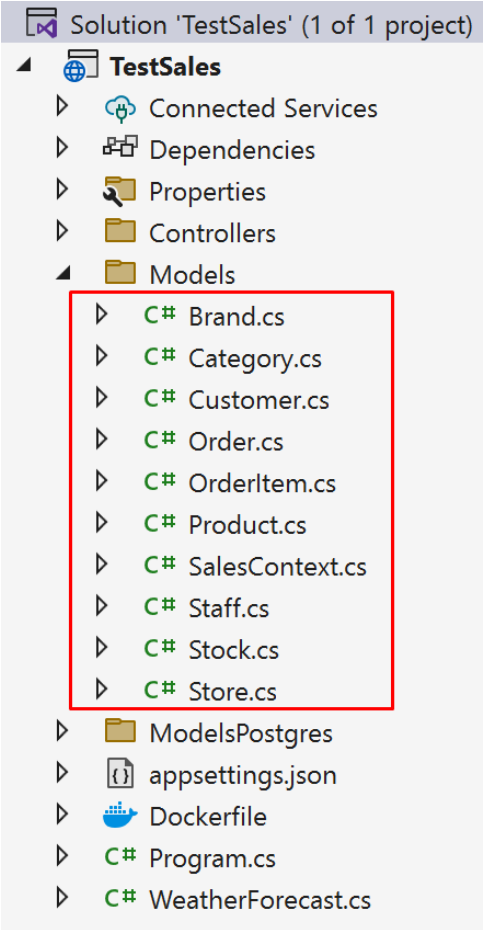

Figura 3.7: Diretoria com ficheiros gerados.

Seguidamente irá ser feita a comparação entre dois motores de base de dados diferentes, bastante usados no mercado, de forma a perceber o impacto da escolha do motor de base de dados no código gerado pela EF Core. Para ser possível a comparação com o código gerado através da base de dados SQL Server foi criada também uma base de dados em PostgreSQL exatamente com a mesma estrutura da já anteriormente criada (ver Figura 3.8). Foi necessária uma alteração a alguns tipos de dados para compatibilizar a base de dados com o PostgreSQL. No caso do SQL Server tinham o tipo de dados `tinyint` - inteiro sem sinal de 8 *bit* (1 *byte*) de tamanho, e na base de dados PostgreSQL ficaram com o tipo de dados `int2` - inteiro com sinal de 16 *bits* (2 *bytes*) de tamanho. O PostgreSQL não suporta nativamente tipos de dados sem sinal com tamanho de 1 *byte*, como se pode observar em [31], tendo assim sido usado na criação das tabelas em PostgreSQL o tipo de dados mais próximo `int2` - inteiro com sinal de 16 *bits* (2 *bytes*) de tamanho.

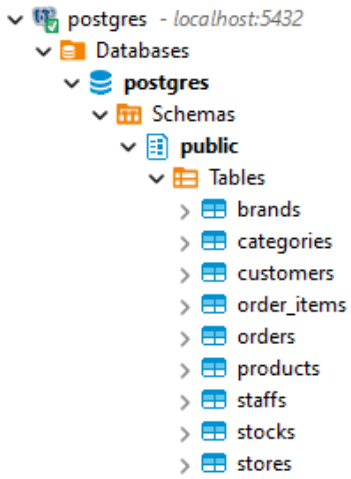

Figura 3.8: Base de dados PostgreSQL.

As classes geradas a partir do modelo de dados da base de dados PostgreSQL foram geradas com recurso ao comando infra citado.

```
dotnet ef dbcontext scaffold "Name=ConnectionStrings:
    SalesConnectionPostgres" Npgsql.EntityFrameworkCore.PostgreSQL
```

Código 3.5: Comando *scaffolding* para base de dados PostgreSQL.

3.4 Comparação do resultado das transformações nos dois motores de base de dados (SQL Server / PostgreSQL)

Nesta secção são mostrados alguns exemplos, de forma sucinta, de mapeamentos entre base de dados e classes em C#. É possível consultar na integra a documentação dos mesmos em [32, 33]. Também é feita uma comparação entre as diferenças observadas na geração das classes provenientes do motor de base de dados SQL Server e PosgreSQL. Isso permite perceber o impacto da escolha do motor de base de dados no código gerado pela EF Core, e se uma possível mudança futura do motor de base de dados trará algum impacto ao código produzido.

Analisando as Figuras 3.9 e 3.10 é possível verificar que a tabela produto é constituída por seis colunas sendo uma delas a chave primária da tabela (`product_id`) e duas delas chaves estrangeiras (`brand_id` e `category_id`). Também podemos ver na Figura 3.9 que a chave primária da tabela produto é usada como chave estrangeira em duas tabelas (`stock` e `order_items`). Quanto aos tipos de dados das colunas são usados tipos diferentes no PostgreSQL e no SQL Server. No PostgreSQL (Figura 3.10) são usados cinco tipos de dados: serial4 para a chave primária - inteiro que auto incrementa;

`varchar(255)` para a coluna do nome do produto que tem um tamanho variável de 255 caracteres; int4 - inteiro de 4 bytes com sinal para as colunas que representam chaves estrangeiras `brand_id` e `category_id`; `int2` - inteiro com sinal de 2 bytes para a coluna model_year; e por fim list_price `numeric(10,2)` - número decimal de dez dígitos com no máximo dois digitos à frente do ponto decimal.

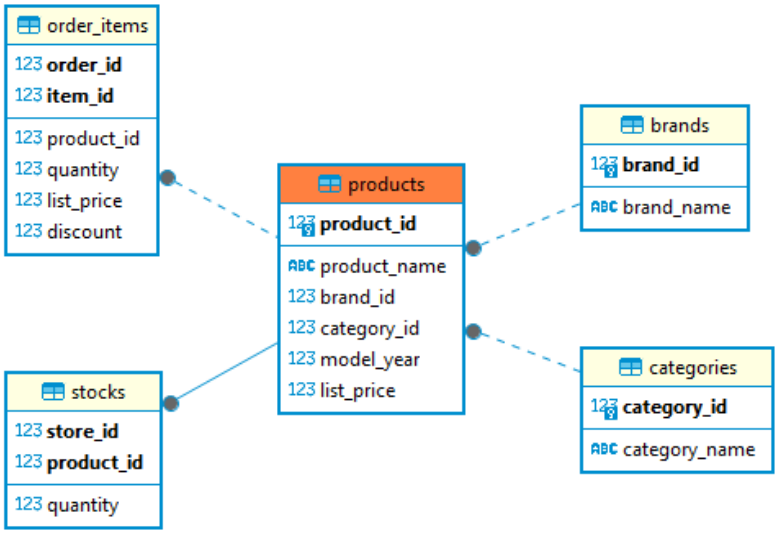

Figura 3.9: Representação ER da tabela produto.

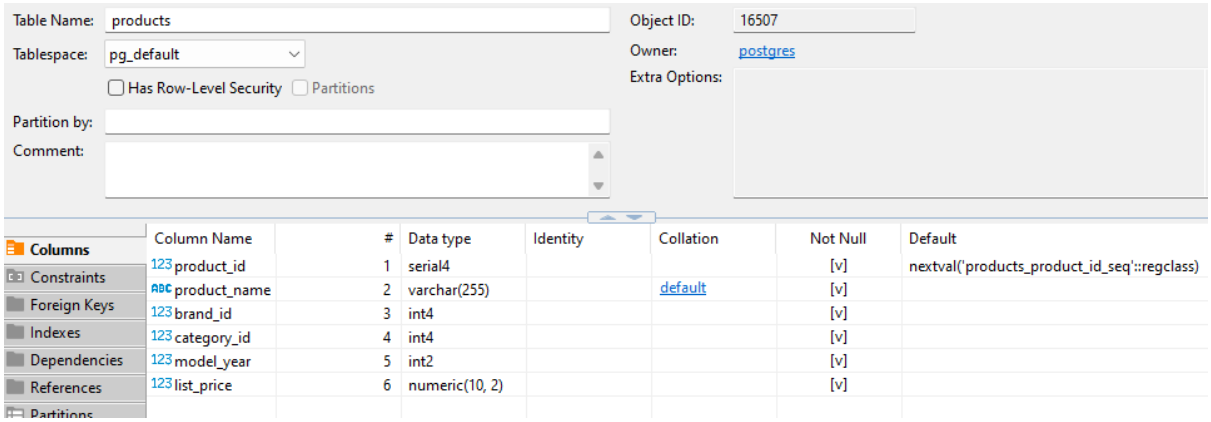

Figura 3.10: Estrutura tabela produto em PostgreSQL.

Na tabela em SQL Server podem ser vistas várias diferenças na Figura 3.11, a coluna da chave primária product_id é do tipo int - inteiro com sinal de 4 bytes; o tipo da coluna `product_name` mantém-se; as duas chaves primárias são do tipo `int`; a coluna `model_year` é `smallint` - um número inteiro com sinal de 2 bytes; e a coluna `list_price` decimal - um número decimal de dez dígitos com no máximo dois dígitos à frente do ponto decimal igual ao que estava definido no PostgreSQL.

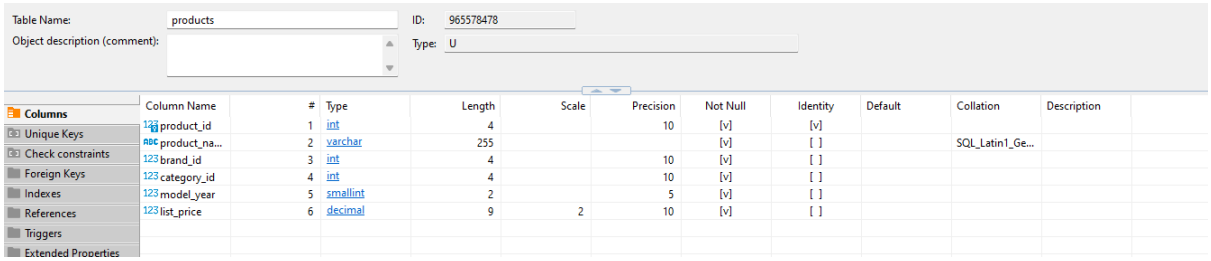

Figura 3.11: Estrutura tabela produto em SQL Server.

No Código 3.6 podemos ver o código gerado pelo *Scaffolding* da EF Core. No caso desta classe *Product* foi criada exatamente da mesma maneira a partir do PostgreSQL, e do SQL Server. Foi gerada uma propriedade do tipo `int` (inteiro) ProductId para a chave primária da tabela, uma `string` (frase) para o nome do produto ProductName, um inteiro para a chave estrangeira para a tabela *Category* CategoryId, um inteiro para a chave estrangeira para a tabela *Brand* BrandId, uma proriedade do tipo `short` (inteiro com sinal de 2 bytes) para o ModelYear, e um `decimal` para o ListPrice. Por fim, temos propriedades virtuais que indicam as ligações que podemos observar na Figura 3.9, a propriedade Brand é a marca associada ao produto, Category é a categoria ligada a este produto, os OrderItems são as compras que referenciam este produto através de uma chave estrangeira na tabela OrderItem, os Stocks são os stocks que referenciam este produto através de uma chave estrangeira na tabela Stock. Estas propriedades são virtuais para permitirem *LazyLoading*, isto é, as referências aos objetos são carregadas apenas no momento em que são acedidas poupando assim recursos ao sistema.

```
1  public partial class Product
2  {
3      public int ProductId { get; set; }
4      public string ProductName { get; set; } = null!;
5      public int BrandId { get; set; }
6      public int CategoryId { get; set; }
7      public short ModelYear { get; set; }
8      public decimal ListPrice { get; set; }
9
10     public virtual Brand Brand { get; set; } = null!;
11     public virtual Category Category { get; set; } = null!;
12     public virtual ICollection<OrderItem> OrderItems { get; set; }
           = new List<OrderItem>();
13     public virtual ICollection<Stock> Stocks { get; set; } = new
           List<Stock>();
14 }
```

Código 3.6: Classe C# gerada para a tabela produto.

Na Figura 3.12 é possível observar as únicas diferenças encontradas no modelo programático gerado entre as classes criadas a partir da base de dados em SQL Server e a base de dados PostgreSQL. A primeira diferença de byte para `short` deve-se ao facto já explicado de as bases de dados em PostgreSQL não permitirem nativamente inteiros sem sinal de 1 byte, gerando assim a partir de SQL Server uma variável do tipo `byte` - um inteiro sem sinal de 1 byte - e a partir de PostgreSQL um `short` - um inteiro com sinal de 2 bytes. Quanto à diferença entre DateTime e DateOnly em ambos os casos na base de dados o campo é do tipo data apenas sem tempo, mas devido a uma limitação do conversor de base de dados SQL Server para código C# é convertido para uma propriedade DateTime que permite data e tempo erradamente. Esta correção já está prevista em versões futuras do `Microsoft.EntityFrameworkCore.SqlServer` como é possível verificar em [34].

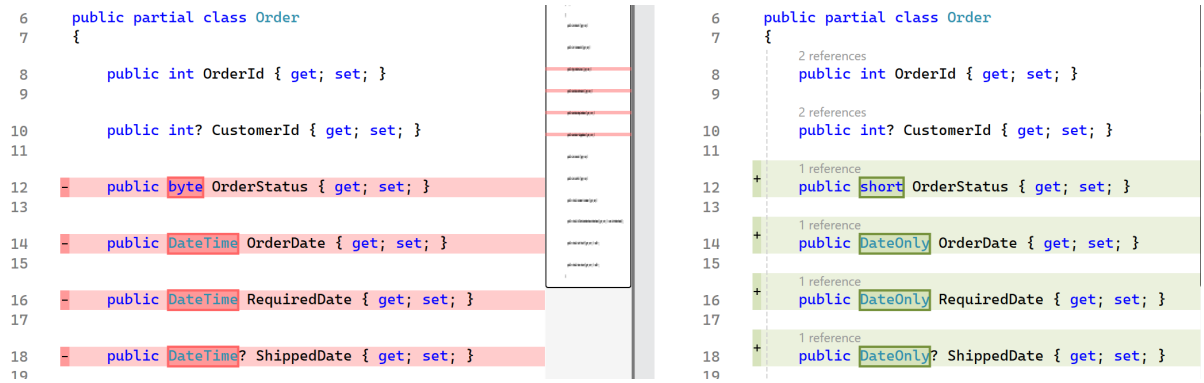

Figura 3.12: Diferenças na classe *Order*.

4 Requisitos, metodologia e arquitetura da solução

Neste capítulo serão especificados os requisitos, a metodologia e arquitetura usada, as decisões tomadas para o desenvolvimento da solução, os testes funcionais executados, e por fim será discutida a aplicação da solução criada.

4.1 Especificação

Nesta secção serão definidos os requisitos da aplicação (funcionais, interface externo, e não funcionais).

4.1.1 Requisitos funcionais / Casos de uso

Nesta sub-secção será feita uma listagem dos casos de uso e requisitos funcionais da solução proposta. O objetivo é fornecer uma visão das interações entre o utilizador e o sistema desenvolvido, bem como as funcionalidades essenciais a serem implementadas para atender as necessidades da aplicação Web ONDA.

Os casos de uso descrevem os cenários de interação do utilizador com o sistema, tendo estes por base os requisitos previamente identificados. Cada caso de uso identifica os atores envolvidos, define uma prioridade de desenvolvimento, identifica as pré-condições para o mesmo ser exequível, quais os resultados esperados e as etapas do processo.

Os requisitos funcionais indicam as funcionalidades que a aplicação deverá ter para garantir o correto funcionamento dos casos de uso descritos. Cada requisito é definido pela descrição, a prioridade de desenvolvimento, e qual o caso de uso relacionado.

A seguir, é apresentada a lista dos casos de uso e requisitos funcionais da aplicação.

Casos de uso:

O caso de uso descrito na Tabela 4.1 permite ao utilizador visualizar as classes C# geradas automaticamente pelo sistema, baseadas no motor de base de selecionado.

Tabela 4.1: [Caso de uso 1] - Visualizar classes C# geradas

[UC1]	**Visualizar classes C# geradas**
Descrição	O utilizador da plataforma ONDA deve conseguir aceder a uma nova aba ("Código") na plataforma, que ao ser clicada mostra a listagem de classes C# geradas automaticamente, e permite a navegação nas mesmas através de clique no nome das classes listadas. Sendo assim possível visualizar o conteúdo (texto) de cada uma das classes geradas.
Atores	Todos
Prioridade	Alta
Pré-condições	O utilizador deve previamente gerar um modelo ER em outra aba disponível, e gerar o script da mesma para ser possível a geração das classes C#.
Pós condições	Listagem das classes C# geradas visível, e a primeira classe visível no ecrã.
Fluxo de eventos	1. O utilizador clica na aba código; 2. O utilizador visualiza a listagem das classes C# geradas; 3. O utilizador visualiza o código da primeira classe gerada; 4. O utilizador tem disponível links nos nomes das classes geradas para alternar o texto da classe gerada visualizada no momento.

O caso de uso descrito na Tabela 4.2 permite ao utilizador realizar o download das classes que foram geradas pelo ONDA, permitindo o seu uso em outros projetos.

Tabela 4.2: [Caso de uso 2] - Download das classes geradas

[UC2]	**Download das classes geradas**
Descrição	O utilizador da plataforma ONDA deve conseguir fazer download de um arquivo ZIP com os ficheiros das classes geradas em C#.
Atores	Todos
Prioridade	Alta
Pré-condições	O utilizador deve previamente clicar na aba (“Código”) para que as classes sejam geradas.
Pós condições	No navegador Web utilizado será apresentado um ficheiro ZIP descarregado com os ficheiros C#.
Fluxo de eventos	1. O utilizador clica no botão de download; 2. O utilizador visualiza o arquivo ZIP descarregado no navegador Web.

O caso de uso descrito na Tabela 4.3 permite a troca do motor de base de dados utilizado pelo sistema para gerar as classes C#. Sendo este caso de uso útil quando se deseja adaptar o sistema a vários contextos com diferentes tecnologias de bases de dados.

Tabela 4.3: [Caso de uso 3] - Alterar o motor de base de dados usado para criar as classes

[UC3]	**Alterar o motor de base de dados usado para criar as classes**
Descrição	O utilizador da plataforma ONDA deve conseguir através de um menu suspenso alterar o motor de base de dados para gerar o script SQL, e posteriormente geração das classes C#.
Atores	Todos
Prioridade	Média
Pré-condições	O utilizador deve estar a visualizar a plataforma ONDA.
Pós condições	Deve ficar selecionado no menu suspenso o motor de base de dados pretendido.
Fluxo de eventos	1. O utilizador escolhe na dropdow o motor de base de dados pretendido; 2. O utilizador visualiza a opção escolhida no menu suspenso.

Requisitos funcionais:

O requisito funcional descrito na Tabela 4.4 indica que o utilizador deve ser capaz de aceder à aba "Código", onde poderá visualizar as classe C# geradas pelo sistema. Está diretamente relacionado ao caso de uso presente na Tabela 4.1.

Tabela 4.4: [Requisito funcional 1] - Clicar aba “Código”

[RF01]	**Clicar aba “Código”**
Descrição	O utilizador deve visualizar na plataforma ONDA uma nova aba “Código” clicável para aceder à nova secção.
Prioridade	Alta
Caso de uso relacionado	[UC01]

O requisito funcional descrito na Tabela 4.5 indica que o sistema deve listar todas as classes C# geradas pelo sistema. Este requisito é parte do fluxo do caso de uso presente na Tabela 4.1.

Tabela 4.5: [Requisito funcional 2] - Listar classes C# geradas

[RF02]	**Listar classes C# geradas**
Descrição	O utilizador deve visualizar na secção de "Código" do lado direito da janela uma coluna com todas as classes C# geradas, e qual a classe selecionada no momento.
Prioridade	Alta
Caso de uso relacionado	[UC01]

O requisito funcional descrito na Tabela 4.6 indica que o utilizador deve poder visualizar o código de uma classe C# numa área de texto dedicada no lado esquerdo da interface do ONDA. Este requisito está associado ao caso de uso presente na Tabela 4.1.

Tabela 4.6: [Requisito funcional 3] - Visualizar área de texto da classe C# selecionada

[RF03]	**Visualizar área de texto da classe C# selecionada**
Descrição	O utilizador deve ver uma grande área de texto do lado esquerdo da listagem de classes onde é possível ver o conteúdo da classe C# selecionada.
Prioridade	Alta
Caso de uso relacionado	[UC01]

O requisito funcional descrito na Tabela 4.7 indica que o utilizador deve ser capaz de clicar no nome de uma das classes geradas para visualizar o seu detalhe. Este requisito faz parte do fluxo descrito no caso de uso presente na Tabela 4.1.

Tabela 4.7: [Requisito funcional 4] - Clicar no nome de uma das classes geradas

[RF04]	**Clicar no nome de uma das classes geradas**
Descrição	O utilizador deve conseguir clicar no nome de uma das classes geradas e esta ficar visivelmente clicada, e aparecer à esquerda o código da mesma.
Prioridade	Alta
Caso de uso relacionado	[UC01]

O requisito funcional descrito na Tabela 4.8 indica que o utilizador deve ser capaz de fazer download das classes C# geradas, num arquivo ZIP, através do clique num botão. Este requisito está relacionado com o caso de uso presente na Tabela 4.2.

Tabela 4.8: [Requisito funcional 5] - Clicar no botão de download

[RF05]	**Clicar no botão de download**
Descrição	O utilizador pode clicar num botão de download, no topo da página na barra de tarefas, que lhe permite fazer download de um arquivo ZIP que contém todas as classes geradas.
Prioridade	Alta
Caso de uso relacionado	[UC02]

O requisito funcional descrito na Tabela 4.9 indica que o utilizador deve conseguir alterar o motor de base de dados utilizado para gerar as classes C#. Este requisito faz parte do fluxo do caso de uso presente na Tabela 4.3.

Tabela 4.9: [Requisito funcional 6] - Alterar o motor de base de dados usado

[RF06]	**Alterar o motor de base de dados usado**
Descrição	O utilizador pode selecionar num menu suspenso, localizado na barra de tarefas, o motor de base de dados que pretende usar para criação do script SQL e posterior geração de classes.
Prioridade	Média
Caso de uso relacionado	[UC03]

4.1.2 Requisitos de interface externo

Nesta sub-secção serão descritos os requisitos relacionados com as interfaces externas do sistema, tendo em conta aspetos de software e de comunicação. Este requisitos garantem que o sistema consegue integrar-se com outros sistemas garantindo um funcionamento como esperado e eficiente. A seguir, são abordadas as principais interfaces externas e as suas características.

Interfaces de software

- Containers Docker com os motores de base de dados suportados para conversão de tabelas em classes;
- Web API responsável pela conversão das classes num container com .NET Core;
- Aplicação Web ONDA num container com XAMPP Apache;
- JSZIP biblioteca de criação de ficheiros ZIP para Javascript;

- Entity Framework Core para conversão de tabelas de uma base de dados para classes C#;
- jQuery para facilitar a utilização do Javascript.

Interfaces de comunicação

- Comunicação HTTP entre aplicação Web ONDA e a Web API REST ONDAto-ORM;
- Comunicação via TCP com as bases de dados temporárias.

4.1.3 Requisitos não funcionais

Nesta sub-secção serão apresentados os requisitos não funcionais, estes que desempenham um papel fundamental no desenvolvimento de uma solução, pois definem características de qualidade e restrições que o sistema deve respeitar para além das funcionalidades. Os requisitos não funcionais definidos a seguir desempenham um papel fundamental no impacto direto na performance e usabilidade da aplicação desenvolvida.

Desempenho

- Em menos 10 segundos converter um script SQL, com duas entidades, em classes C#. Este tempo deve ser ajustado proporcionalmente para um maior número de entidades;
- Criar um ficheiro ZIP com 3 classes em poucos segundos (menos de 20 segundos excluindo o tempo de download que pode variar com a largura de banda disponível). Tal como na conversão, este tempo deve ser ajustado proporcionalmente ao número de ficheiros incluídos no ZIP.

O tempo de menos 10 segundos foi escolhido tendo em conta a complexidade da tarefa a executar. O tempo de resposta para um utilizador percebido como imediato situa-se entre 0,1 e 1 segundo, e para tarefas mais complexas esse tempo pode ser considerado aceitável até 10 segundos sem que o utilizador perca o foco. Tendo em conta a complexidade do processamento de conversão de scripts SQL em classes C# em modelos de bases de dados mais complexos, e tendo alguma tolerância para estes processos definindo uma margem de segurança, achou-se aceitável aumentar um pouco o tempo de espera. Esta tolerância reflete o reconhecimento da tarefa como intensiva, com a duração a variar conforme a complexidade do motor de base de dados usado, e a complexidade das duas entidades a converter, podendo alcançar até 10 segundos.

Escalabilidade

- A arquitetura da solução deve permitir escalabilidade, possibilitando atender um maior ou menos volume de pedidos conforme a procura.

Limitações nos pedidos HTTP de conversão:

- O tamanho do script SQL convertido para base64, presente no pedido de conversão, pode ser configurado no .NET Core Kestrel Web Server para um tamanho máximo que ronda os 8,59 GB.

Segurança

- Autorização na Web API REST com autenticação Bearer. Este tipo de autenticação foi escolhido porque permite a comunicação segura com serviços externos, pois muitos sistemas aceitam tokens Bearer, o que simplifica a autenticação entre microsserviços ou serviços distribuídos. Além disso, os token Bearer são autocontidos - isto porque contém toda a informação necessária para identificar o utilizador sem a necessidade de aceder a uma base de dados, curtos - pois tem um formato compacto e codificado, podem ter data e hora de expiração - o que reduz os riscos de segurança, e são independentes do estado, não sendo necessário o servidor guardar informação dos tokens - o que facilita a escalabilidade da API.

Qualidade

- Capacidade de conversão dos scripts SQL através dos motores de base de dados suportados no ONDA;
- Apresentação das classes C# na interface gráfica do ONDA;
- Capacidade de criação de um ficheiro ZIP com as classes geradas.

4.2 Metodologia, arquitetura e decisões de implementação de software

Nesta secção será explicada a metodologia e arquitetura usada no desenvolvimento da solução, e também explicadas as decisões mais importantes no desenvolvimento da mesma.

Como primeiro passo para o desenvolvimento da solução de conversão de modelos ER em classes programáticas, já existia uma plataforma Web chamada ONDA [27], desenvolvida no Centro de Informática e Sistemas da Universidade de Coimbra (CISUC). Esta plataforma permite a criação de modelos ER de forma gráfica e gera automaticamente um script SQL para criar as tabelas.

Através do estudo do trabalho relacionado existente foi possível perceber que já existem diversas soluções de conversão de bases de dados em linguagens programáticas. Apesar do objetivo inicial deste trabalho ser a criação de um ORM de raiz, optou-se pela integração de um ORM existente, evitando assim que se estivesse a desenvolver algo que já existe no mercado com vários anos de testes, e que segue as normas e regras estudadas na literatura. Sendo assim, decidiu-se a integração do ORM da Microsoft, Entity Framework Core, por ser uma solução multiplataforma e de código aberto.

Devido à aplicação Web ONDA estar desenvolvida em HTML e Javascript não era possível integrar a EF Core com esta solução Web de uma forma direta. Sendo a EF Core desenvolvida em .NET Core, a sua integração só é possível através da linguagem C# que não é compatível com Javascript. Para esta integração ser possível foi desenvolvida uma arquitetura "cloud ready"que está dividida em 3 módulos principais (Aplicação Web ONDA, microsserviço ONDAtoORM, e Bases de dados temporárias, ver Figura 4.1), que poderão ser containers Docker isolados numa arquitetura cloud, assim como em "on-premises"(servidores locais).

Este tipo de arquitetura baseado em containers Docker permite-nos ter mais flexibilidade e escalabilidade na solução para atender um número de pedidos que se altere frequentemente durante o tempo. Permite também uma melhor eficiência nos recursos consumidos porque os containers partilham o kernel do sistema operativo. Outra vantagem importante de salientar é a alta disponibilidade e tolerância a falhas através de sistemas de orquestração como Kubernetes. Este tipo de arquitetura apresenta diversos desafios, como a complexidade na implementação de sistemas de orquestração, a gestão de dados, uma vez que os containers são *stateless* e requerem armazenamento externo, a portabilidade que pode não ser assegurada entre diferentes fornecedores de ambientes cloud e, por fim, a dependência de ferramentas específicas como Docker e Kubernetes, que possuem uma curva de aprendizagem considerável.

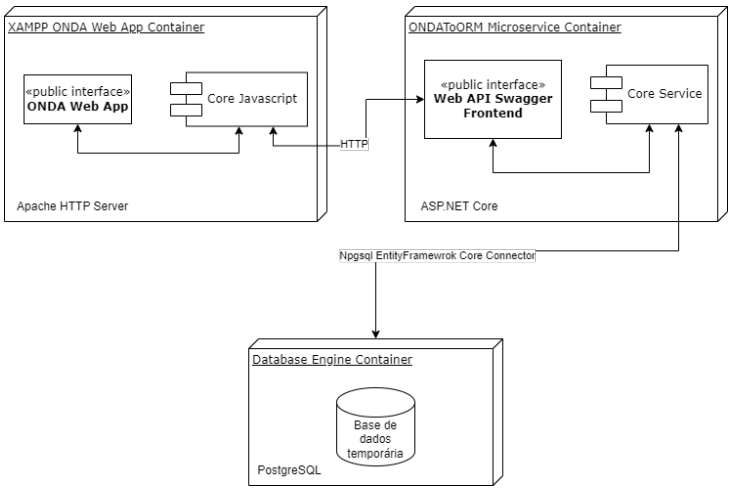

Figura 4.1: Arquitetura da solução.

Na Figura 4.1 podemos observar os principais módulos da aplicação. Em primeiro lugar temos um container Docker com um servidor Apache HTTP que permite alojar a aplicação Web, ONDA, e que expõem para o exterior a interface para o utilizador. Este primeiro container comunica via HTTP com um segundo container que corre aplicações em ASP.NET Core, neste caso uma interface HTTP mais propriamente uma Web API REST que permite fazer autenticação e receber pedidos de conversão de scripts SQL em ficheiros em código C#. Por fim, devido à necessidade de implementar os scripts SQL num motor de base de dados físico, para posteriormente ser convertido em classes C# pela EF Core, é necessário termos containers para cada um dos motores de base dados suportados para conversão. Estes containers apenas terão bases temporárias que são apagadas após a conversão feita pelo serviço principal do microsserviço ONDAtoORM. Na prática a aplicação ONDA via Javascript faz uma chamada HTTP a uma Web API REST, mais concretamente um pedido POST, para fazer autenticação, e seguidamente com o Bearer token devolvido faz uma nova chamada POST para fazer a conversão de um script SQL para um determinado motor de bases de dados. O microsserviço responsável pela conversão implementa o script num container especifico, e através da EF Core gera os ficheiros C# correspondentes e devolve-os à interface Web, que for fim os mostrará ao utilizador.

Na solução desenvolvida de raiz em .NET Core foi usado a arquitetura cebola (Onion), porque torna o código mais escalável, mais testável, e com melhor possibilidade de manutenção. Esta arquitetura caracteriza-se pelo desacopolamento das camadas internas para as camadas externas, estando assim os acoplamentos direcionados para o centro, o Domínio, como podemos ver na Figura 4.2. Tendo esta arquitetura as camadas bem definidas, e tendo uma estrutura genérica pode ser aplicada num contexto de microsserviços assim como foi aplicada neste projeto no serviço de conversão. Este tipo de arquitetura num contexto de microsserviços tem um papel fundamental na fácil aplicação de testes unitários no sistema, clara separação da lógica de negócio tornando as alterações mais fáceis, e o isolamento das camadas através de interfaces promovendo assim a produção de bom código e com uma estrutura organizada. Portanto, a arquitetura Onion está estreitamente ligada a arquiteturas de microsserviços que apesar de adicionar alguma complexidade à estrutura dos projetos traz muitas vantagens ao programador que esteja familiarizado com a arquitetura.

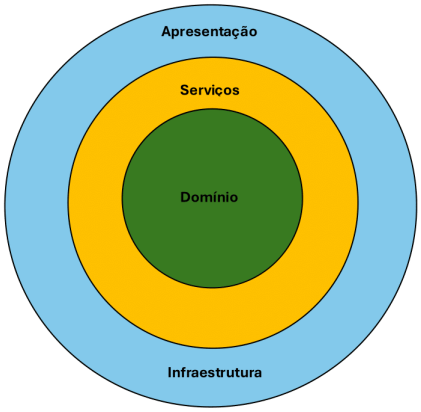

Figura 4.2: Arquitetura cebola.

Seguindo esta arquitetura iniciou-se o desenvolvimento da solução através do IDE Visual Studio, com a seguinte estrutura como mostra a Figura 4.3.

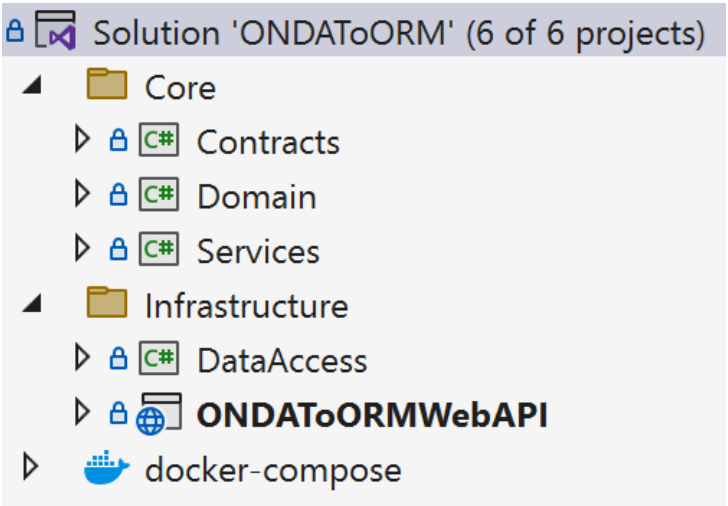

Figura 4.3: Estrutura da solução desenvolvida.

Temos assim a solução organizada com a arquitetura cebola: a pasta infraestrutura que contém os dois projetos da camada mais exterior, o de apresentação (WebAPI), e de infraestrutura (DataAccess); a camada de serviços que é composta pelo projeto Services, e Contracts, ambos os projetos permitem fazer o elo de ligação entre a camada exterior e a camada mais interna de domínio; por último a camada de domínio onde estão definidas as interfaces dos repositórios e entidades. Estas camadas estão interligadas através de interfaces e contratos sempre no sentido de fora para dentro e não o contrário, como se pode ver na Figura 4.4.

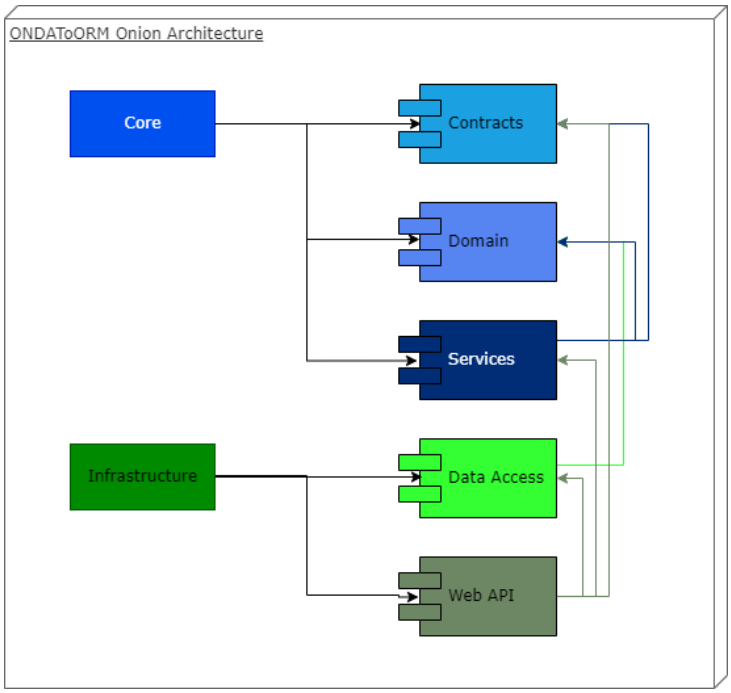

Figura 4.4: Dependências entre projetos da solução.

Com base no levantamento de requisitos realizado neste capítulo, e na escolha da metodologia e arquitetura da solução a implementar, prosseguiu-se para o desenvolvimento da solução, apresentado no próximo capitulo (Capítulo 5).

5 Implementação da solução proposta

Neste capítulo será explicada a Web API REST (ONDAtoORM) desenvolvida neste projeto, detalhada a integração deste microsserviço com a plataforma Web ONDA, descritos os testes funcionais realizados, e por fim feita uma análise e discussão dos resultados obtidos.

5.1 ONDAtoORM Web API REST

Este projeto .Net Core para além de outros padrões de arquitetura usou dois padrões muito importantes para a manutenção do código e futura implementação de novos motores de base de dados compatíveis no microsserviço de conversão. Esses padrões de desenvolvimento são o *Factory Method* e o *Strategy*. O padrão *Factory Method* foi usado para criar os diferentes tipos de estratégias usados para a conversão consoante o motor de base de dados passado pelo cliente no pedido HTTP. Podemos ver a implementação no excerto de código 5.1.

```
1  public ITemporaryDatabaseRepository
       CreateTemporaryDatabaseRepository(string databaseEngine)
2  {
3      if (string.IsNullOrWhiteSpace(databaseEngine))
4      {
5          throw new ArgumentException("Database engine cannot be
               null or empty.", nameof(databaseEngine));
6      }
7
8      var connectionString = configuration[$"ConnectionStrings:{
           databaseEngine}"];
9
10     if (string.IsNullOrWhiteSpace(connectionString))
11     {
12         throw new ArgumentException($"Connection string for '{
               databaseEngine}' not found.", nameof(connectionString))
               ;
13     }
14
15     return databaseEngine.ToLower() switch
16     {
```

```
17          "postgresql" => new PostgreSQLDatabaseRepository(
                connectionString),
18          "oracle" => new OracleDatabaseRepository(connectionString)
                ,
19          "mysql" => new MySQLDatabaseRepository(connectionString),
20          "mariadb" => new MySQLDatabaseRepository(connectionString)
                ,
21          "sqlite" => new SQLiteDatabaseRepository(connectionString)
                ,
22          _ => throw new ArgumentException($"Database engine '{
                databaseEngine}' is not supported.", nameof(
                databaseEngine))
23      };
24  }
```

Código 5.1: Padrão *Factory Method*.

O padrão *Strategy* foi usado para facilitar a implementação da conversão de SQL para código C# em novos motores de base de dados sendo apenas necessário criar uma nova classe que implemente a interface *ITemporaryDatabaseRepository* (estratégia) e implementar cada um dos seus métodos. Pode ser visto no excerto de código 5.2 a implementação da estratégia usada nos repositórios dos diferentes motores de base de dados.

```
1  public interface ITemporaryDatabaseRepository
2  {
3      Task<bool> CheckDatabaseExistance();
4      Task<bool> DropDatabase();
5      Task<bool> CreateDatabase();
6      Task<bool> RunSQLScript(string sqlScript);
7      Task<List<FileDto>> GetCSharpFilesFromDatabase();
8  }
```

Código 5.2: Padrão *Strategy*.

Para além do padrão *Strategy* cada um dos repositórios deriva de uma classe base que implementa o código genérico para conversão de uma base de dados, que foi previamente instanciada, em código C#. No excerto de código 5.3 podemos ver a implementação de um repositório que deriva da classe base *BaseTemporaryDatabaseRepository* e implementa a estratégia *ITemporaryDatabaseRepository*, e o código 5.4 que representa a classe base.

```
1  public class MySQLDatabaseRepository(string connectionString) :
       BaseTemporaryDatabaseRepository(connectionString),
       ITemporaryDatabaseRepository
2  {
```

```
    public async Task<bool> CheckDatabaseExistance()
    {
        ....
    }

    public async Task<bool> CreateDatabase()
    {
        ....
    }

    public async Task<bool> DropDatabase()
    {
        ....
    }

    public async Task<bool> RunSQLScript(string sqlScript)
    {
        ....
    }

    public async Task<List<FileDto>> GetCSharpFilesFromDatabase()
    {
        ....
    }
}
```

Código 5.3: Implementação de uma *Strategy*.

```
public abstract class BaseTemporaryDatabaseRepository(string
   connectionString)
{
    internal readonly string databaseName = "db_onda_to_orm_temp";
    internal string ConnectionString = connectionString;

    internal Task<List<FileDto>> GetCSharpFilesWithEFCore(
       IReverseEngineerScaffolder scaffoldService, string?
       connectionStringSufix = null, List<string>? schemas = null)
    {
        ....
    }
}
```

Código 5.4: Classe base da estratégia.

Através da utilização desta abordagem de desenvolvimento futuramente será muito mais fácil um programador adicionar a compatibilização de novos motores de base de dados neste serviço. Tendo para isso que apenas criar uma nova estratégia e implementar as funções da mesma com o código específico desse motor de base de dados, e adicionar este novo repositório à função de criação de objetos pelo nome do motor de base de dados.

Nesta aplicação foram desenvolvidos dois endpoints, que permitem a autenticação, e a chamada do serviço de conversão, implementando o verbo POST. Nas Web API REST é possível a utilização de cinco ações para executar os pedidos HTTP chamadas de verbos (GET, POST, PUT, DELETE, PATCH), que permitem por definição, respetivamente: obter dados, criar dados, alterar dados, apagar dados, e edição parcial de dados [35]. Estes endpoints chamam serviços da camada de serviços que por sua vez devolvem resultados em JSON para o cliente que os invoca. Sem qualquer tipo de autenticação o endpoint de conversão não funcionará e devolve o erro 401 Unhauthorized.

O sistema de autenticação implementado funciona através de um pedido HTTP POST que recebe um nome de utilizador e uma palavra-passe, e devolve um Bearer Token com uma duração de expiração definida. Através do uso desse token devolvido poderá ser chamado o endpoint de conversão que no corpo da mensagem POST deve conter um script SQL para um determinado motor de base de dados.

A seguir, são apresentados exemplos dos dois endpoints desenvolvidos na Web API do microsserviço ONDAtoORM (autenticação e conversão). Em cada exemplo, é definido o URL do pedido, a estrutura do corpo do pedido HTTP, e um exemplo do corpo da resposta HTTP.

Pedido de autenticação para obter Bearer Token

POST URL: /api/Identity/token

Corpo do pedido:

```
{
  "username": "{USERNAME}",
  "password": "{PASSWORD}"
}
```

Código 5.5: Corpo do pedido de token

Resposta do pedido:

```
{
  "data": {
    "user": {
      "id": {USER_ID},
```

```
      "username": "{USERNAME}",
      "password": "",
      "role": "{USER_ROLE}"
    },
    "token": "{BEARER_TOKEN}"
  },
  "errors": [],
  "isValid": true
}
```

Código 5.6: Corpo do pedido de token

Pedido de conversão de script SQL para classes C#

POST URL: /api/Converter/convert

Corpo do pedido:

```
{
    "DatabaseEngine": {Motor de base de dados suportado (p.e.
       : SQLite, Oracle, SQLServer)},
    "SqlContentInBase64": {Texto do script SQL codificado em
       base64}
}
```

Código 5.7: Corpo do pedido de conversão

Resposta do pedido:

```
{
    "data": [
        {
            "code": "{Codigo C\# codificado em base64}",
            "name": "{FILENAME_1}.cs"
        },
        {
            "code": "{Codigo C\# codificado em base64}",
            "name": "{FILENAME_2}.cs"
        }
    ],
    "errors": [],
    "isValid": true
}
```

Código 5.8: Corpo do pedido de token

Para ser executado este serviço de conversão no cabeçalho do pedido deve ser usado o token obtido no pedido anterior para que a autenticação Bearer seja feita com sucesso (exemplo de cabeçalho HTTP: *Authorization: Bearer {token}*), no corpo do pedido deve ser passado um json que contém o motor de base de dados usado na conversão "*DatabaseEngine*", e o script SQL em forma de texto codificado em base64 "*SqlContentInBase64*". O serviço de conversão usa um fluxo para verificar no container do motor de base de dados se a base de dados necessária está instanciada ou não. Se estiver, ela é eliminada. Seguidamente é criada a base de dados com um nome temporário, e executado o script SQL sobre a mesma para serem criadas as tabelas e devidas relações. Estas bases de dados temporárias são instanciadas em container Docker com os motores de base de dados compatíveis e que foram criados com recurso a comandos Docker como podemos ver no excerto de Código 5.9.

```
1 docker run -d -p 1521:1521 -e ORACLE_PASSWORD=PASSWORD -v oracle-
    volume:/opt/oracle/oradata gvenzl/oracle-free
2 docker run --name tese-mysql -e MYSQL_ROOT_PASSWORD=PASSWORD -d
    mysql:latest -p 3306:3306
```

Código 5.9: Comando Docker para criar motores de base de dados.

Após isto, através da integração via código do ORM EF Core, são criadas as classes C#. Finalmente é apagada a base de dados temporária, e devolvidos os ficheiros .cs (C#) codificados em base64 através de uma resposta HTTP com a estrutura JSON que é mostrada no Código 5.8.

O container Docker responsável pela conversão e que aloja a aplicação explicada anteriormente foi publicado num repositório público de imagens Docker.

Este container pode ser acedido em https://hub.docker.com/r/a21190325/ondatoormwebapi [36], permitindo futuras integrações com outros projetos. Ele pode ser executado em qualquer sistema operativo compatível com Docker, utilizando o comando mostrado no excerto de Código 5.10. Este comando que permite executar a aplicação de conversão, e também mapeia o porto 8080 do container com o exterior para poder ser acedido no *host*, e monta dois volumes, um que permite o acesso a uma base de dados externa SQLite, e o outro que permite a personalização do ficheiro de configurações da aplicação para definir os *hosts* e palavras-passe dos motores de base de dados usados pelo serviço de conversão.

```
1 docker run -p 8080:8080 -v C:\PATH_TO_SQLITE_DB/TempSQLiteDb.db -v
     C:\PATH_TO_ONDAtoORM\ONDAToORM\appsettings.json:/app/
    appsettings.json --name ONDAToORM -d a21190325/ondatoormwebapi
```

Código 5.10: Comando Docker para executar serviço ONDAtoORM.

5.2 Integração da aplicação ONDAtoORM na plataforma Web ONDA

Nesta secção será explicada a integração necessária realizada para acomodar as novas funcionalidades fornecidas pelo microsserviço ONDAtoORM na plataforma Web ONDA previamente descrita em 3.1.

A plataforma ONDA está dividida em vários separadores para cada uma das funcionalidades que suporta. Seguindo esta abordagem gráfica da plataforma, para integrar a aplicação ONDAtoORM no ONDA foi criado um novo separador nesta aplicação que permite mostrar e navegar na interface para o utilizador (UI) entre os ficheiros C# gerados, como se pode ver em Figura 5.1.

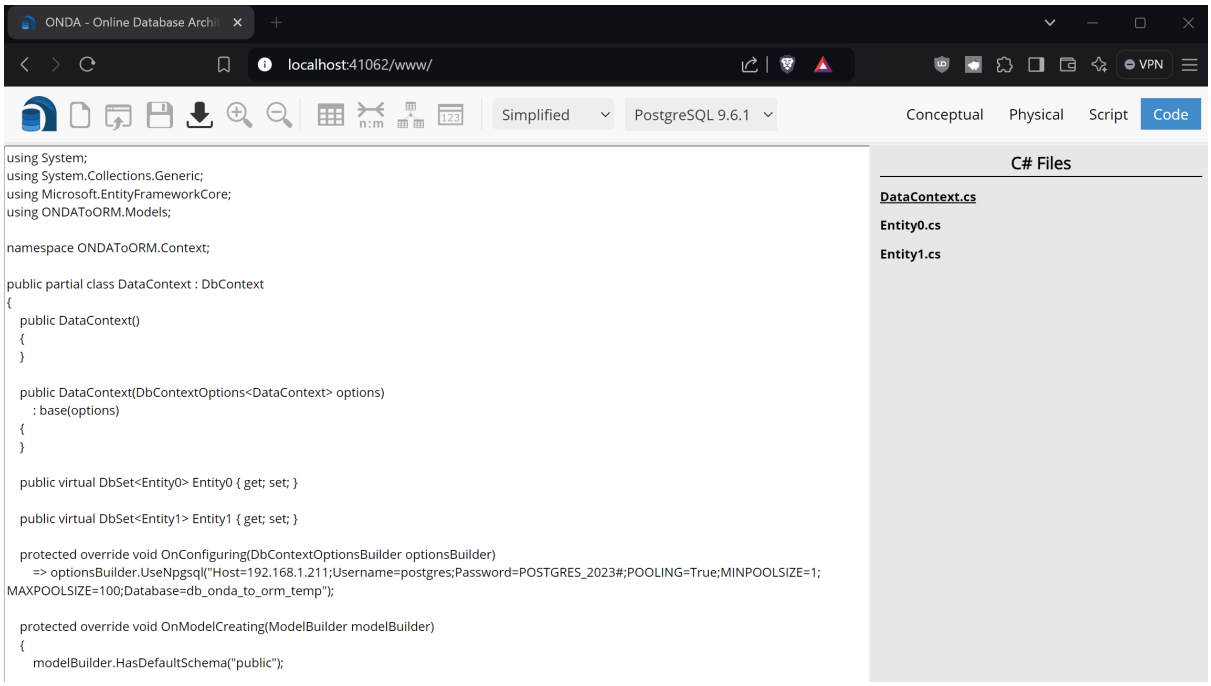

Figura 5.1: Interface gráfica do ONDA onde é mostrado o código gerado.

Para implementar esta integração para a parte da UI foi usado escrita em linguagem Hyper Text Markup Language (HTML), e produzido Cascading Style Sheets (CSS) para adicionar estilos à UI produzida.

A parte de lógica programática que invoca a aplicação ONDAtoORM foi desenvolvida em linguagem Javascript. Nesta etapa foram usadas duas bibliotecas de terceiros: a biblioteca jQuery v2.1.1 disponível em [37], que já se encontrava no projeto; e a biblioteca JSZIP v3.10.1 disponível em [38].

Através da biblioteca jQuery foi possível simplificar a forma como se interage com os elementos gráficos presentes no HTML, e também para executar as chamadas HTTP ao ONDAtoORM. As chamadas HTTP ao ONDAtoORM são executadas assincronamente

sem que a UI fique bloqueada para o utilizador que está a operar a plataforma.

A biblioteca JSZIP ajudou no desenvolvimento de uma funcionalidade que permite ao utilizador fazer download dos ficheiros gerados em C# num único ficheiro comprimido (Zip). A partir do conteúdo dos ficheiros C# este biblioteca gera um ficheiro Zip, e permite também a criação de um elemento blob, que é um objeto do tipo arquivo que permite ser transferido pelo navegador.

O código produzido em Javascript também é responsável pelo tratamento de respostas, de sucesso e erro, recebidas através da API, e consequente disponibilização na UI para que se perceba o que está a acontecer nestas chamadas assíncronas.

Devido a existir uma conexão externa da plataforma ONDA à Web API REST ONDAtoORM foi criada uma configuração em JSON, que facilita a sua alteração em diferentes ambientes de instalação. Pode-se ver um exemplo da estrutura dessa configuração no Código 5.11.

```
AppConfigsJSON = `{
    "ORMServiceEndPoint": {
        "BaseUrl": "https://{IP}:{PORTO}/api/",
        "Username": "{UTILIZADOR}",
        "Password": "{PALAVRA_PASSE}"
    }
}`;
```

Código 5.11: Configuração da Web APIU ONDAtoORM no ONDA

5.3 Testes funcionais

Nesta secção serão explicados os testes funcionais executados durante a implementação deste trabalho.

Para a realização dos testes funcionais foi usada a plataforma ONDA, através da sua interface Web foi criado um modelo conceptual simples de testes com duas tabelas e uma relação entre elas como podemos ver na Figura 5.2. Na Figura 5.3 podemos ver o setup de containers docker necessário para executar a solução, e assim poder realizar os testes funcionais: acima temos os containers aplicacionais myXampp (onde é executado o servidor aplicacional apache com a aplicação ONDA), e o container ONDAtoORM (onde é executado o serviço de conversão desenvolvido no âmbito deste projeto); abaixo temos os container com os motores de base de dados suportados por esta aplicação sendo eles MySQL, PostgreSQL, Oracle, MariaDB, e ainda está presente também no container ONDAtoORM um ficheiro local que é usado para a base de dados SQLite.

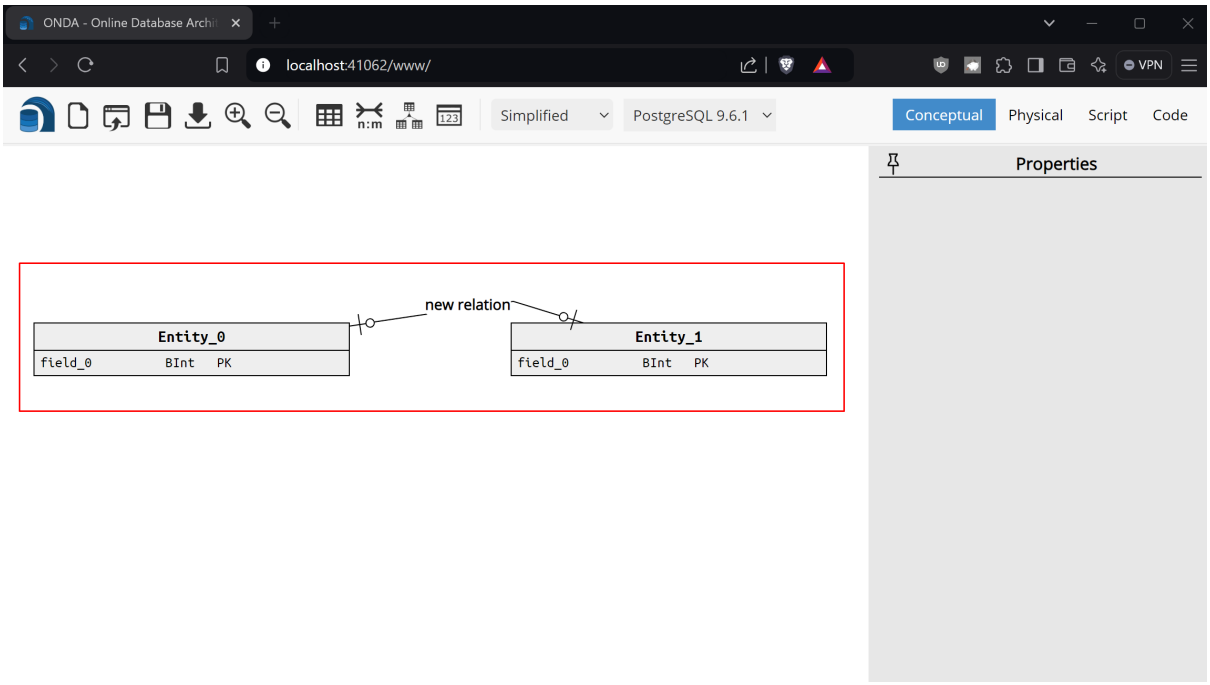

Figura 5.2: Modelo conceptual criado no ONDA manualmente.

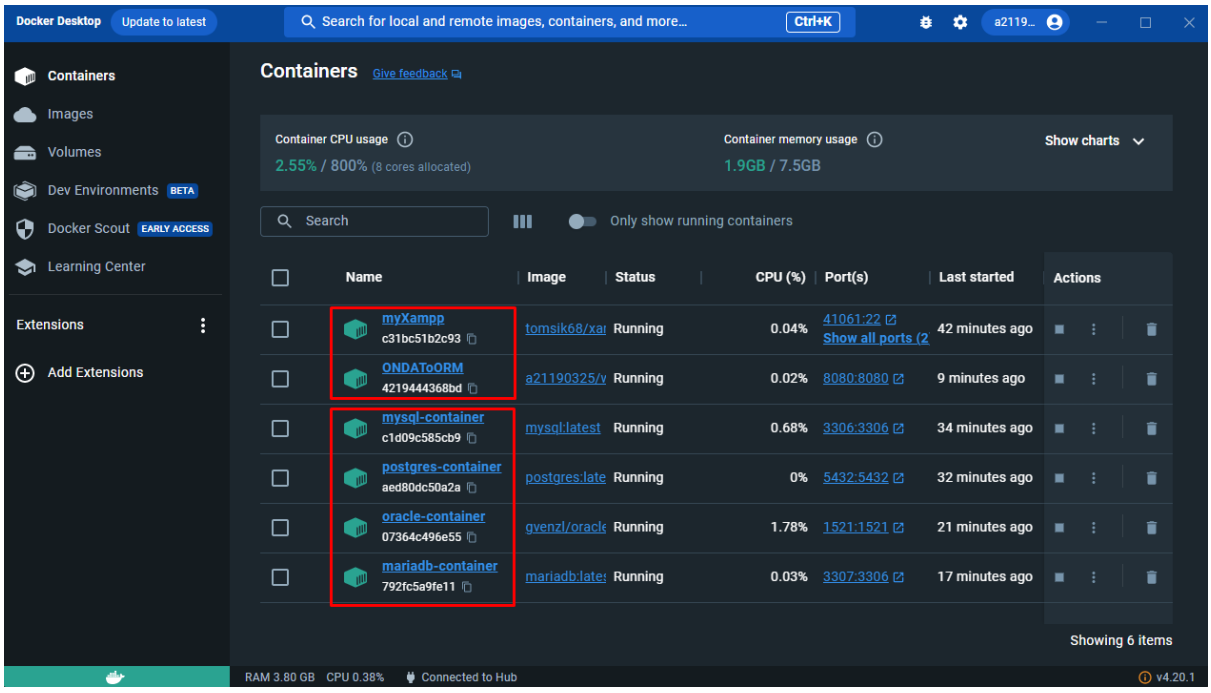

Figura 5.3: Containers da solução desenvolvida.

Usando este modelo conceptual presente na Figura 5.2 foram testados os motores de base de dados compatíveis no ONDA (PostgreSQL, Oracle, MySQL, MariaDB e SQ-Lite), nos quais a partir do script SQL gerado (ver Figura 5.4) foi analisada a geração do código C# na nova aba "*Code*"(ver Figura 5.5). Através de todos estes motores de base de dados foi possível gerar as classes C# correspondentes às tabelas presentes

no modelo conceptual, criado inicialmente, com sucesso. Foram usados modelos conceptuais de maior complexidade, mas percebeu-se que o único impacto é o tempo de conversão que aumenta.

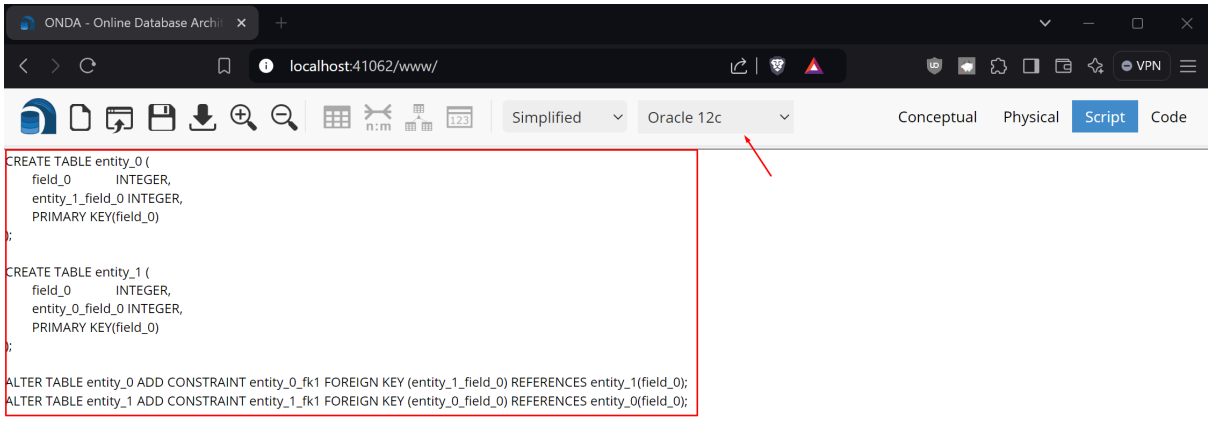

Figura 5.4: Script SQL criado automaticamente pelo ONDA.

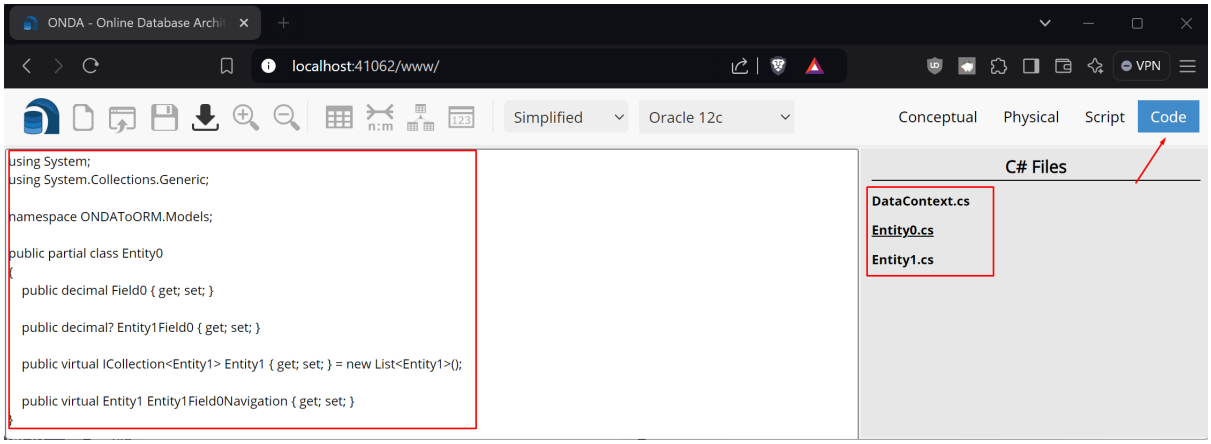

Figura 5.5: Código C# criado automaticamente pelo ONDA através do serviço ONDAtoORM.

Pelo facto da integração do microsserviço com a plataforma ONDA ter sido feita numa fase posterior deste projeto, foi usada a aplicação Postman [39] que permite executar pedidos HTTP tornando assim possível testar os endpoints criados e simular as chamadas da interface Web, e conseguir assim validar as respostas devolvidas. Na Figura 5.6 podemos ver um exemplo de um desses pedidos em que é criado um pedido HTTP POST de conversão de um script SQL codificado em base64 neste caso para o motor

de base de dados SQLite. Através desta aplicação foi possível criar pedidos para testar o outro endpoint de autenticação e também testar a conversão com todos os outros motores de base de dados compatíveis com a solução.

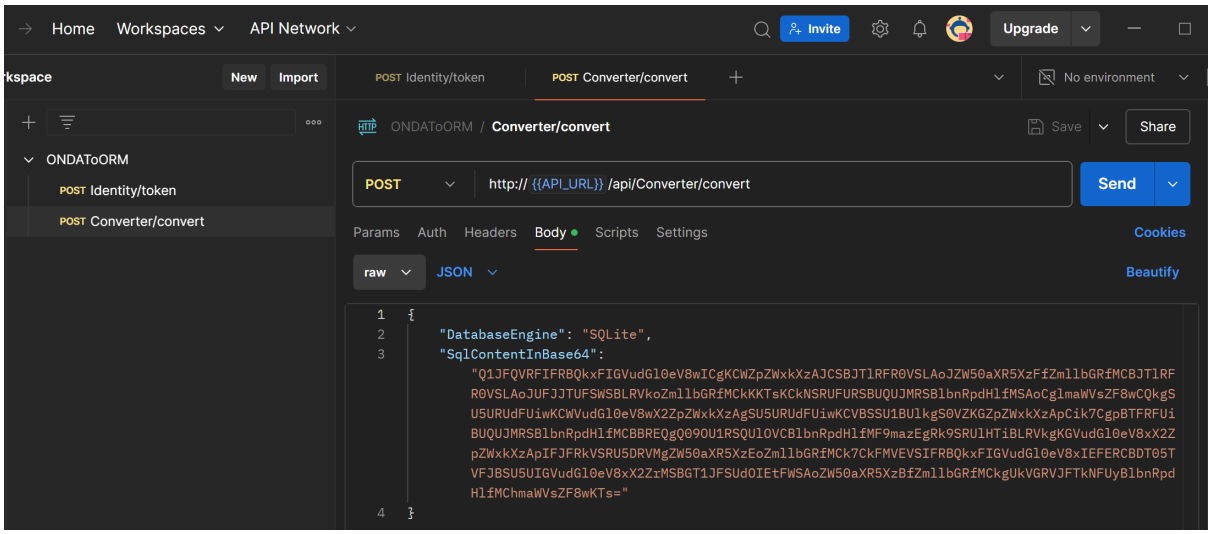

Figura 5.6: Exemplo de pedido HTTP POST através da aplicação Postman.

Após desenvolver a integração do microsserviço ONDAtoORM com a plataforma Web ONDA, foi possível executar os testes de integração. Quanto à avaliação dos tempos de resposta dos pedidos efetuados pela aplicação ONDA ao serviço de conversão ONDAtoORM, para autenticação, podemos analisar na Figura 5.7 que o tempo para o pedido de autenticação é quase instantâneo (cerca de 3 milissegundos). Na Figura 5.7 também podemos analisar que os pedidos de conversão variam consoante o motor de base de dados escolhido para gerar as classes programáticas em C#. Os tempos de resposta para as bases de dados SQLite e Oracle foram 92 milissegundos e 3 segundos, respetivamente. Podemos concluir que a complexidade da estrutura do motor de base de dados influência diretamente os tempos de resposta obtidos, e que a complexidade do script, ou seja, o número de objetos a serem convertidos em classes C#, afeta proporcionalmente o tempo de conversão.

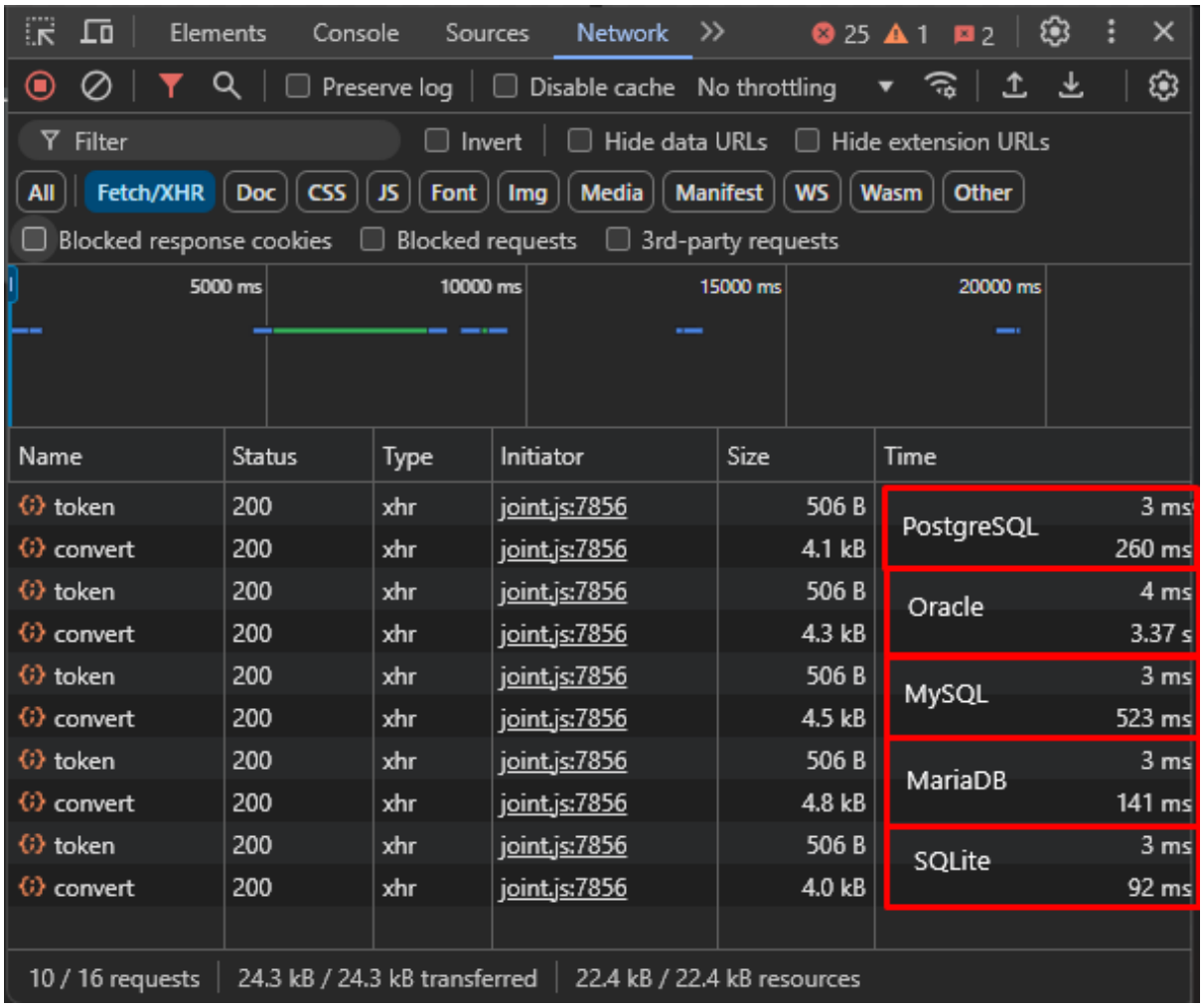

Figura 5.7: Tempos de resposta do serviço ONDAtoORM.

Sendo um dos requisitos o download das classes C# num ficheiro zip também foi avaliada essa funcionalidade. Assim como mostram as Figuras (5.8, 5.9, 5.10) podemos observar que foi pedido o download das classes num ficheiro com o nome "Code.zip"(Figura 5.8), este que é processado pelo browser e transferido para o sistema de ficheiros do computador (Figura 5.9), e seguidamente podemos ver no sistema de ficheiros do sistema operativo a pasta comprimida com as classes C# geradas no seu interior (Figura 5.10). O tempo de execução deste processo pode variar, dependendo da largura de banda disponível e do tamanho dos ficheiros a serem transferidos. Por exemplo, com uma conexão de 100 Mbps é possível fazer o download de um ficheiro de 100 KB (equivalente por exemplo, ao tamanho de um ficheiro ZIP com arquivos C#) em menos de 1 segundo. Se a largura de banda for maior ou menor, o tempo poderá variar proporcionalmente. Neste caso, como se tratam apenas de ficheiros de texto, o impacto será menor pois o tamanho do arquivo comprimido raramente será excessivo.

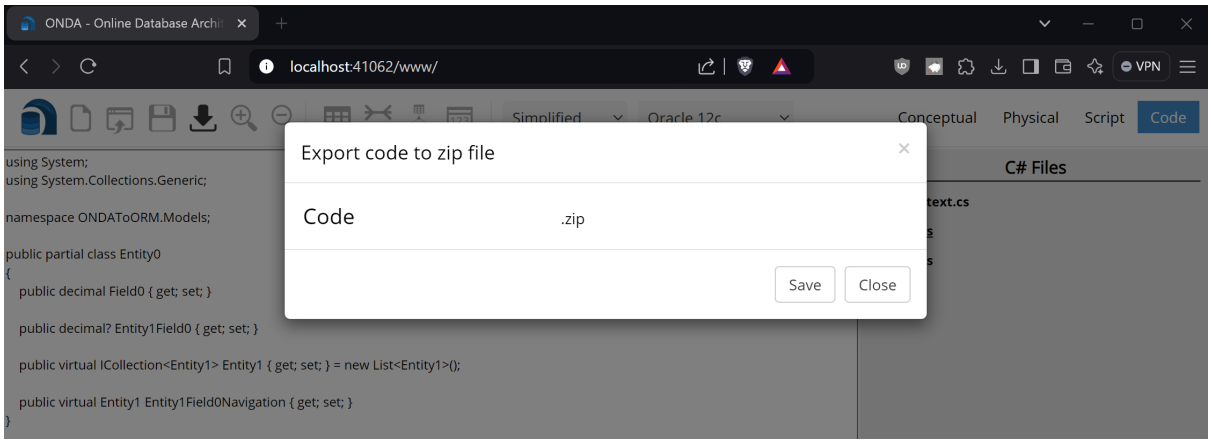

Figura 5.8: Pedido de download das classes C# num arquivo zip.

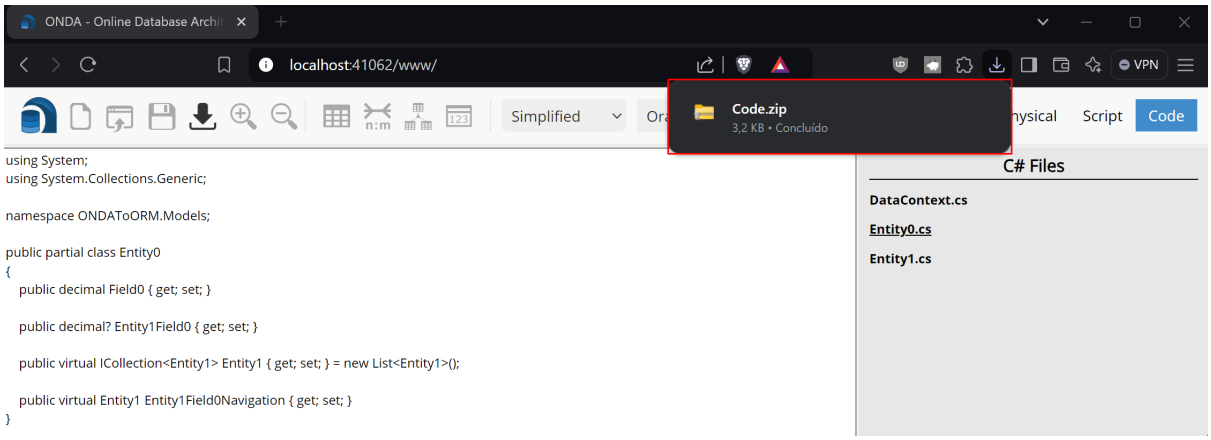

Figura 5.9: Download do arquivo zip processado pelo browser.

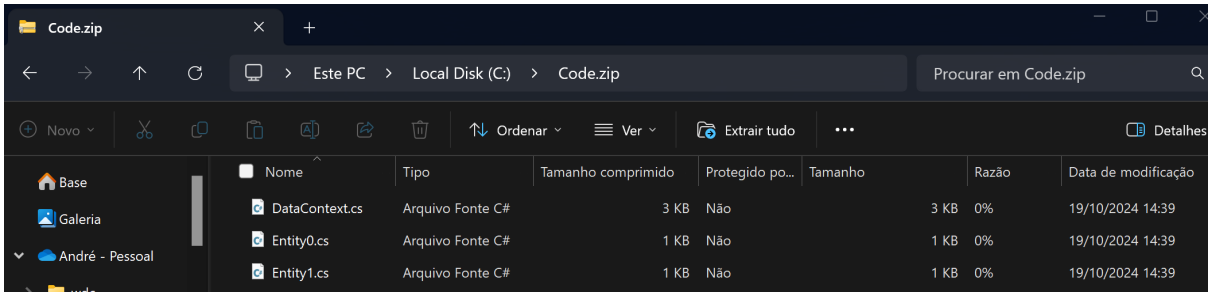

Figura 5.10: Conteúdo do arquivo zip.

5.4 Análise e Discussão dos Resultados

Nesta secção será feita a análise e discussão sobre os resultados obtidos através da implementação das soluções apresentadas ao longo deste capítulo.

A arquitetura da solução proposta através deste projeto torna-a bastante versátil possibilitando o uso em outros contextos que não a plataforma Web ONDA. Será assim possível usar o micro-serviço de conversão para outros casos de uso possíveis.

A solução desenvolvida destaca-se pela sua versatilidade, sendo compatível com os motores de bases de dados, PostgreSQL, Oracle, MySQL, MariaDB e SQLite. Esta versatilidade permite que esta ferramenta seja usada em diferentes ambientes, tanto para projetos de grande escala que tiram partido da robustez do Oracle e PostgreSQL, quanto para aplicações mais leves que usam SQLite ou MariaDB. Ao suportar esta ampla gama de motores de bases de dados a solução torna-se de maior acessibilidade e fácil integração atendendo a diferentes necessidades do mercado. Para além destes motores de bases de dados a solução foi arquitetada para ser facilmente integrável com outros que haja necessidade futuramente. Neste contexto, a escolha da plataforma Web ONDA para a integrar a solução desenvolvida neste projeto oferece uma interface que simplifica a visualização dos resultado das classes C# devolvidas pelo microsserviço ONDAtoORM, facilitando o uso em integrações futuras num software a ser arquitetado.

A solução desenvolvida permite contrastar com as soluções de ORM tradicionais, mencionadas no Capítulo 2, porque permite a um desenhador de um modelo ER antecipar aos programadores quais serão as classes programáticas geradas para integração com o modelo de dados em desenvolvimento, através de uma interface gráfica. Existem outras vantagens deste processo que são:

1. Ser executado em tempo de execução, ou seja, sem que o utilizador se aperceba, e sem que este tenha que possuir qualquer conhecimento sobre o ORM que está a ser executado;
2. Ser totalmente automatizado, não transparecendo assim ao utilizador todo o trabalho de integração e comunicação com o ORM EF Core;
3. Não ser necessário a implementação das bases de dados num determinado ambiente.

O uso de microsserviços nesta solução de software representa uma vantagem significativa pois proporciona modularidade, escalabilidade, e fácil manutenção e publicação da mesma. A separação deste serviço da aplicação Web ONDA reduz os os impactos negativos que esta nova funcionalidade poderia ter na performance do ONDA. Além disso o microsserviço permitiu a integração com múltiplas bases de dados, as quais podem ser executadas em containers Docker, numa arquitetura *CloudReady*. A integração da plataforma ONDA com microsserviços também potencializa a própria solução pois promove uma arquitetura mais ágil e preparada para expansões e evoluções futuras.

Esta solução tem algumas limitações, algumas delas são:

1. A sua modularização poder tornar o sistema mais difícil de ser instalado num ambiente de produção, no caso de não estarem bem definidos os pipelines de instalação;
2. A total dependência de um ORM de terceiros neste caso propriedade da Micro-

soft, apesar de ser de código aberto;

3. A limitação na linguagem de programação de geração das classes ser o C#;
4. A limitação quanto aos motores de bases de dados integrados nesta solução (sendo que a solução está preparada com padrões de arquitetura de software que facilitam o processo).

Como podemos constatar, esta é uma solução bastante útil para a comunidade de programação, e que poderá ser complementada de várias formas, tais como o uso de inteligência artificial para converter as classes geradas em C# para outras linguagens de programação, e a integração de novos motores de base de dados compatíveis com a solução ONDAtoORM.

6 Conclusões e Trabalho Futuro

Neste trabalho o objetivo principal de conversão de modelos ER em classes programáticas na plataforma ONDA foi atingido. Para atingir este objetivo foi desenvolvido um microsserviço que apresenta uma Web API REST pública, responsável por todo o processo de conversão. Também foi adaptada a interface Web da plataforma ONDA para integrar esta nova funcionalidade de apresentação das classes C# ao utilizador.

Este trabalho irá permitir que no processo de desenvolvimento de uma solução sejam antecipadas questões de arquitetura, e desenho de uma solução de bases de dados. Normalmente um ORM é usado numa fase mais avançada de um projeto, e através da linha de comandos. Isto ocorre quando já foram decididos muitos pormenores ao nível da base de dados que poderão impactar negativamente o processo de integração do código com o modelo de dados. Permitindo a atuação numa fase de desenho de um diagrama ER, poderão ser antecipados os problemas mencionados. Além disso, será possível rever o impacto que estes problemas terão na solução, uma vez que agora é possível visualizar em tempo real as classes geradas na plataforma Web ONDA.

No entanto, a solução desenvolvida tem limitações ao nível da linguagem de programação retornada nas classes geradas, o C#. Isso significa que o programador precisará de compreender a linguagem de programação C# para conseguir realizar uma análise teórico-prática no seu contexto de desenvolvimento. Outra limitação diz respeito à modularização da solução em várias aplicações, o que poderá complicar a instalação do sistema. Por fim, existe a dependência de um ORM de terceiros (EF Core), que poderá, no futuro, restringir a integração via código.

Ao longo do desenvolvimento deste projeto, um dos principais desafios foi arquitetar a solução por forma a torná-la *CloudReady*, exigindo a escolha de tecnologia e linguagem de programação que permitisse o desenvolvimento em ambiente de containers e multiplataforma. A escolha do .NET Core permitiu assegurar esses requisitos oferecendo a flexibilidade necessária para o projeto em questão. Outro obstáculo surgiu na necessidade de integração de uma plataforma Web desenvolvida em JavaScript (ONDA), com o microsserviço de conversão desenvolvido em .NET Core que, devido à incompatibilidade entre tecnologias, não permitia uma integração direta. Este obstáculo foi ultrapassado com a criação de um microsserviço que expõem uma Web API REST, permitindo que o código JavaScript comunique via HTTP com o mesmo. Outro desafio relevante foi projetar o core do microsserviço com uma arquitetura flexível para futuras integrações de diferentes motores de bases de dados. Este problema foi ultrapassado

com o uso de padrões de desenvolvimento como o *Strategy*, que facilita a implementação do serviço de conversão de SQL em classes C# através da implementação de uma determinada interface, e o *Factory Method*, que permite a criação de múltiplas estratégias de conversão, consoante o motor de base de dados passado ao microsserviço, exigindo apenas a criação de novas classes conforme necessário.

O microsserviço ONDAtoORM desenvolvido no âmbito deste projeto, para além do seu uso na plataforma ONDA, poderá ser usado via Web API para muitos outros fins em que seja necessário antever as classes geradas pelo ORM EF Core, antes mesmo de a base de dados estar instanciada, mesmo que numa versão primária, num servidor. Esta solução possibilita assim uma redução de requisitos de infraestrutura, segurança, e implementação numa fase preliminar de uma base de dados que não está no seu estado final.

Como trabalho futuro para esta solução, sugere-se implementar e integrar outros motores de base de dados, desde que sejam suportados pela EF Core, a qual já oferece suporte para a maioria dos motores de base de dados mais utilizados no mercado. Além disso, poderá ser implementada uma funcionalidade que permita, ao chamar a Web API, a conversão das classes geradas para outras linguagens de programação, utilizando, por exemplo, inteligência artificial. Isso tornaria a solução mais versátil nos outputs fornecidos.

Referências bibliográficas

[1] J. P. Smith, *Entity Framework core in action*. Manning, 2021.

[2] S. Ambler, "Mapping Objects to Relational Databases: O/R Mapping In Detail - The Agile Data (AD) Method — agiledata.org," https://agiledata.org/essays/mappingObjects.html, [Acedido em 16-10-2023].

[3] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM transactions on database systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[4] ajcvickers, "Overview of Entity Framework Core - EF Core — learn.microsoft.com," https://learn.microsoft.com/en-us/ef/core/, 2021, [Acedido em 11-11-2023].

[5] S. W. Ambler, "Mapping objects to relational databases," *White Paper, Ronin International*, 2000.

[6] M. Lorenz, J.-P. Rudolph, G. Hesse, M. Uflacker, and H. Plattner, "Object-relational mapping revisited-a quantitative study on the impact of database technology on O/R mapping strategies," *Hawaii International Conference on System Sciences (HICSS)*, 2017.

[7] W. Keller, "Mapping objects to tables," in *Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany*, vol. 206. Citeseer, 1997, p. 207.

[8] D. Colley, C. Stanier, and M. Asaduzzaman, "Investigating the effects of object-relational impedance mismatch on the efficiency of object-relational mapping frameworks," *Journal of Database Management (JDM)*, vol. 31, no. 4, pp. 1–23, 2020.

[9] T.-H. Chen, W. Shang, J. Yang, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An empirical study on the practice of maintaining object-relational mapping code in java systems," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176. [Online]. Available: https://doi.org/10.1145/2901739.2901758

[10] "GitHub - ActiveJpa/activejpa: A simple active record pattern library in java that makes programming DAL easier — github.com," https://github.com/ActiveJpa/activejpa, [Acedido em 11-11-2023].

[11] "Core Data," https://developer.apple.com/documentation/coredata, [Acedido em 11-11-2023].

[12] Z. Projects, "Welcome To Learn Dapper ORM - A Dapper Tutorial for C# and .NET Core — learndapper.com," https://www.learndapper.com/, [Acedido em 11-11-2023].

[13] "Making queries | Django documentation — docs.djangoproject.com," https://docs.djangoproject.com/en/5.1/topics/db/queries/, [Acedido em 11-11-2023].

[14] "Doctrine: PHP Open Source Project — doctrine-project.org," https://www.doctrine-project.org/, [Acedido em 11-11-2023].

[15] EclipseLink, "EclipseLink — eclipse.dev," https://eclipse.dev/eclipselink/#jpa, [Acedido em 11-11-2023].

[16] ajcvickers, "Overview of Entity Framework 6 - EF6 — learn.microsoft.com," https://learn.microsoft.com/en-us/ef/ef6/, [Acedido em 11-11-2023].

[17] "GitHub - linq2db/linq2db: Linq to database provider. — github.com," https://github.com/linq2db/linq2db, [Acedido em 11-11-2023].

[18] C. Begin, "MyBatis 3 | Introduction — mybatis.org," https://mybatis.org/mybatis-3/, [Acedido em 11-11-2023].

[19] "Home - NHibernate — nhibernate.info," https://nhibernate.info/, [Acedido em 11-11-2023].

[20] "Sequelize — sequelize.org," https://sequelize.org/, [Acedido em 11-11-2023].

[21] "SQLAlchemy — sqlalchemy.org," https://www.sqlalchemy.org/, [Acedido em 11-11-2023].

[22] M. Myllyaho Forsberg, "An evaluation of .NET Object-Relational Mappers in relational databases : Entity Framework core and Dapper," p. 21, 2022.

[23] C. Xia, G. Yu, and M. Tang, "Efficient implement of orm (object/relational mapping) use in j2ee framework: Hibernate," in *2009 International Conference on Computational Intelligence and Software Engineering*, 2009, pp. 1–3.

[24] mcleblanc, "Entity Data Model: Primitive Data Types - ADO.NET," https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/entity-data-model-primitive-data-types, 2021, [Acedido em 16-10-2023].

[25] W. Premerlani and M. Blaha, "An approach for reverse engineering of relational databases," in [*1993*] *Proceedings Working Conference on Reverse Engineering*, 1993, pp. 151–160.

[26] bricelam, "Reverse Engineering - EF Core," https://learn.microsoft.com/en-us/ef/core/managing-schemas/scaffolding/?tabs=dotnet-core-cli, 2023, [Acedido em 13-11-2023].

[27] N. Laranjeiro and A. M. Pinto, "ONDA: ONline Database Architect," *arXiv preprint arXiv:2401.16552*, 2024. [Online]. Available: https://arxiv.org/abs/2401.16552

[28] BillWagner, "Consulta Integrada linguística (LINQ) em C#," https://learn.microsoft.com/pt-pt/dotnet/csharp/linq/, 2022, [Acedido em 12-11-2023].

[29] ajcvickers, "Provedores de Banco de Dados – EF Core," https://learn.microsoft.com/pt-br/ef/core/providers/?tabs=dotnet-core-cli, 2024, [Acedido em 15-01-2024].

[30] "NuGet Gallery | Home — nuget.org," https://www.nuget.org/.

[31] "PostgreSQL - 8.1. Numeric Types," https://www.postgresql.org/docs/current/datatype-numeric.html, [Acedido em 06-11-2023].

[32] AndriySvyryd, "Provedor de Banco de Dados do Microsoft SQL Server – EF Core," https://learn.microsoft.com/pt-pt/ef/core/providers/sql-server/?tabs=dotnet-core-cli, 2023, [Acedido em 09-11-2023].

[33] "Npgsql Entity Framework Core Provider | Npgsql Documentation," https://www.npgsql.org/efcore/index.html, 2023, [Acedido em 09-11-2023].

[34] ajcvickers, "Breaking changes in EF Core 8.0 (EF8) - EF Core," https://learn.microsoft.com/en-us/ef/core/what-is-new/ef-core-8.0/breaking-changes#sqlserver-date-time-only, 2023, [Acedido em 13-11-2023].

[35] C. Severance, "Roy T. Fielding: Understanding the REST Style," *Computer*, vol. 48, no. 6, pp. 7–9, 2015.

[36] A. G. Rodrigues, "ONDAToORM Docker Container," https://hub.docker.com/r/a21190325/ondatoormwebapi, 2024, [Acedido em 05-07-2024].

[37] O. Foundation, "jQuery," https://jquery.com/, 2014, [Acedido em 26-02-2024].

[38] S. Knightley, "JSZip," https://stuk.github.io/jszip/, 2022, [Acedido em 26-02-2024].

[39] "Postman API Platform — postman.com," https://www.postman.com/, [Acedido em 10-06-2024].

Instituto Superior de Engenharia
Politécnico de Coimbra