SISTEMA DE GESTÃO DE RESERVAS DE RESTAURANTE

DEI - Desenvolvimento / Operação de Software

CTeSP em Desenvolvimento Ágil de Software – Santa Maria da Feira
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Emanuel Silva
1241586@isep.ipp.pt

João Carvalho
1231540@isep.ipp.pt

Pedro Morgado
1220492@isep.ipp.pt

ÍNIDICE

1. INTRODUÇÃO

No âmbito da unidade curricular de Desenvolvimento / Operação de Software, foi proposto o desenvolvimento de um sistema de gestão de reservas para restaurantes, com o objetivo principal de aplicar e consolidar os conceitos técnicos abordados ao longo do semestre durante as aulas. Este projeto visa não apenas a criação de uma aplicação funcional, mas também a implementação de práticas de desenvolvimento de software, incluindo conteinerização, orquestração, integração contínua (CI/CD), e garantia de qualidade através de testes automatizados.

O sistema desenvolvido é uma API REST que permite a gestão completa de reservas de mesas em restaurantes, incluindo funcionalidades de criação, consulta, atualização e cancelamento de reservas. O sistema irá permitir a validação automática de conflitos de horários para garantir que uma mesa não seja reservada por múltiplos clientes no mesmo período e também não permitir fazer reservas no passado.

A implementação deste sistema baseia-se num conjunto de tecnologias atualizadas e amplamente adotadas na indústria:

- Backend**:** ASP.NET Core 9.0, aproveitando as últimas funcionalidades do framework .NET
- Persistência de Dados**:** SQL Server 2022 com Entity Framework Core para acesso a dados
- Testes Automatizados**:** xUnit, Moq e coverlet para testes unitários e medição de cobertura
- Conteinerização**:** Docker e Docker Compose para empacotamento e execução consistente
- CI/CD**:** Jenkins e SonarQube para automação de pipeline e análise de qualidade de código
- Orquestração**:** Kubernetes com Helm Charts para trabalho em ambientes de produção
- Documentação**:** Swagger/OpenAPI para documentação automática da API

Este relatório serve não apenas como documentação do trabalho realizado, mas também como reflexão sobre as melhores práticas de desenvolvimento de software e as lições aprendidas ao longo do processo de implementação.

2. ARQUITETURA DO SISTEMA

A arquitetura do sistema de gestão de reservas de restaurante foi planeada para garantir escalabilidade, manutenibilidade e robustez. Seguindo os princípios de arquitetura em camadas, o sistema separa claramente as responsabilidades entre apresentação, lógica de negócio e persistência de dados.

2.1 Diagrama de Arquitetura

O sistema implementa uma arquitetura cliente-servidor, onde diferentes componentes comunicam através de APIs REST bem definidas. A arquitetura principal pode ser visualizada através do seguinte diagrama simplificado:

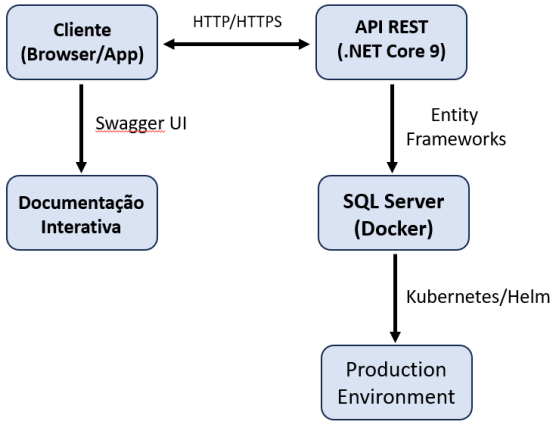

Figura 1 - Diagrama de Arquitetura

No fluxo arquitetural, os clientes interagem com a API REST através de requisições HTTP/HTTPS. A API, desenvolvida em ASP.NET Core 9.0, processa as requisições, aplica a lógica de negócio e acede ao banco de dados SQL Server através do Entity Framework Core. Para desenvolvimento e teste, o Swagger UI fornece utilização interativa, enquanto em produção o sistema é conteinerizado com o Docker e orquestrado via Kubernetes utilizando Helm Charts.

Esta arquitetura de três camadas (apresentação, lógica de negócio, persistência) permite:

- Escalabilidade independente: Cada componente pode ser escalado conforme a necessidade
- Manutenibilidade: Mudanças em uma camada não afetam as outras
- Testabilidade: Componentes isolados facilitam testes unitários
- Portabilidade: Conteinerização garante consistência entre ambientes

2.2 Tecnologias e Ferramentas

A seleção de tecnologias foi realizada considerando maturidade, desempenho e adequação aos requisitos do projeto. A stack tecnológica adotada abrange desde o desenvolvimento backend até a implantação em produção.

2.2.1 Backend e API:

- ASP.NET Core 9.0: Framework principal para desenvolvimento da API REST, escolhido pela sua performance otimizada, suporte a longo prazo e natureza multiplataforma.
- Entity Framework Core 9.0: ORM (Object-Relational Mapper) para acesso a dados, selecionado pela produtividade que oferece através do paradigma Code First e pelo suporte robusto a migrações de base de dados.
- SQL Server 2022: Sistema de gestão de base de dados relacional, escolhido pela confiabilidade, desempenho e integração nativa com o ecossistema .NET.

2.2.2 Desenvolvimento e Qualidade:

- xUnit: Framework de testes unitários, selecionado pela simplicidade e integração com o .NET Core.
- Swagger/OpenAPI: Ferramenta para documentação automática da API, que gera interface interativa para teste e exploração dos endpoints.

2.2.3 Conteinerização e Orquestração:

- Docker: Plataforma de Conteinerização, utilizada para empacotar a aplicação e suas dependências em imagens portáteis.
- Docker Compose: Ferramenta para orquestração de múltiplos containers, utilizada para configurar o ambiente de desenvolvimento com API e SQL Server.
- Kubernetes: Sistema de orquestração de containers para ambientes de produção, escolhido pela capacidade de gerir aplicações escaláveis e resilientes.
- Helm: Gestor de pacotes para Kubernetes, utilizado para definir, instalar e atualizar aplicações complexas através de charts.

2.2.4 Integração e Entrega Contínua:

- Jenkins: Servidor de automação para CI/CD, configurado com pipeline multi-estágio para build, teste e deploy automatizados.
- SonarQube: Plataforma para inspeção contínua da qualidade de código, integrada no pipeline para análise estática e medição de cobertura de testes.

2.2.5 Ambiente de Desenvolvimento:

- Visual Studio Code: Editor de código principal, com extensões para C#, Docker e desenvolvimento .NET.
- Git: Sistema de controlo de versões, com estratégia de branching para desenvolvimento colaborativo.

2.3 Estrutura do Projeto

A organização do código fonte segue princípios de arquitetura limpa e separação de responsabilidades, com uma estrutura clara que facilita a navegação, manutenção e escalabilidade do projeto.

```
Estrutura do Projeto:

RestaurantReservations/                          # Pasta raiz da solução
├── RestaurantReservations.sln                   # Solução .NET
├── .dockerignore                                # Ignora ficheiros no build Docker
├── docker-compose.yml                           # Orquestração: API + SQL Server
├── Dockerfile                                   # Build multi-stage da imagem da API
├── Jenkinsfile                                  # Pipeline CI/CD completo (8 stages)
├── sonar-project.properties                     # Configuração do SonarQube
│
├── src/                                         # Código-fonte da aplicação
│   └── RestaurantReservations.API/
│       ├── Controllers/                         # Controladores da API (ReservationsController)
│       ├── Data/                                # DbContext e Migrações do Entity Framework
│       ├── DTOs/                                # Data Transfer Objects
│       ├── Models/                              # Modelos de domínio (Reservation)
│       ├── Program.cs                           # Ponto de entrada e configuração
│       ├── Properties/                          # Configurações de launch
│       ├── Services/                            # Lógica de negócio (ReservationService)
│       └── appsettings.json                     # Configurações da aplicação
│
├── tests/                                       # Projetos de teste
│   └── RestaurantReservations.UnitTests/
│       ├── Controllers/ReservationsControllerTests.cs
│       ├── Models/ReservationTests.cs
│       ├── DTOs/ReservationDtoTests.cs
│       ├── Services/ReservationServiceTests.cs
│       └── RestaurantReservations.UnitTests.csproj
│
└── helm/                                        # Configuração de deployment Kubernetes
    └── restaurant-api/
        ├── Chart.yaml                           # Metadados do Helm Chart
        ├── values.yaml                          # Valores configuráveis (replicas, image, etc.)
        └── templates/                           # Templates Kubernetes
            ├── deployment.yaml                  # Especificação do Deployment
            ├── service.yaml                     # Especificação do Service
            ├── configmap.yaml                   # Configurações da aplicação
            └── _helpers.tpl                     # Funções de template auxiliares
```

Figura 2 - Estrutura do Projeto

3. IMPLEMENTAÇÃO DA API

A implementação da API REST para gestão de reservas de restaurante seguiu uma abordagem prática focada em funcionalidade robusta, testabilidade e preparação para produção. O desenvolvimento enfrentou diversos desafios técnicos que foram superados através de decisões arquiteturais bem fundamentadas e adoção dos padrões de desenvolvimento corretos.

3.1 Endpoints Implementados

O desenvolvimento seguiu uma metodologia incremental com ênfase em testes unitários desde as fases iniciais. Adotou-se uma estratégia onde os contratos da API foram definidos antes da implementação, utilizando Swagger/OpenAPI para validação. Esta abordagem permitiu antecipar potenciais problemas de integração e estabelecer expectativas claras sobre o comportamento do sistema.

Tabela 1 - Lista de Endpoints

Método	Endpoint	Descrição
GET	/api/reservations/status	Verifica status da API
GET	/api/reservations	Lista todas reservas
GET	/api/reservations/{id}	Obtém reserva específica
POST	/api/reservations	Cria nova reserva
PUT	/api/reservations/{id}	Atualiza reserva existente
DELETE	/api/reservations/{id}	Cancela reserva
GET	/api/reservations?date={date}	Filtra por data

A implementação seguiu rigorosamente as convenções REST, utilizando códigos de status HTTP semanticamente apropriados:

- 200 OK para sucesso
- 201 Created para criação de recursos
- 400 Bad Request para dados inválidos
- 404 Not Found para recursos inexistentes
- 409 Conflict para conflitos de horário detetados

3.2 Desafios Técnicos e Soluções

3.2.1 Gestão de Conflitos de Horário

Um dos requisitos mais complexos foi a implementação da deteção de conflitos de horário, garantindo que duas reservas não ocupem a mesma mesa no mesmo horário. Inicialmente, o algoritmo básico apresentou um problema: ao atualizar uma reserva (por exemplo, alterando apenas o número de pessoas), o sistema falsamente detetava conflito porque comparava a reserva consigo mesma.

Solução: Introduziu-se lógica adicional para distinguir entre atualizações que alteram dados críticos (mesa, data, hora) e atualizações que não afetam a unicidade. O método UpdateReservationAsync agora verifica se está efetivamente a alterar mesa, data ou hora antes de verificar conflitos, permitindo atualizações seguras de outros campos como customerName ou numberOfPeople.

3.2.2 Compatibilidade de Versões

Durante a configuração inicial, enfrentámos incompatibilidade entre a versão do .NET (9.0) e pacotes do Entity Framework Core, resultando em erros de build. A instalação automática de pacotes NuGet baixava versões mais recentes incompatíveis.

Solução: Adotou-se versionamento explícito através do comando dotnet add package --version 9.0.0, garantindo compatibilidade entre todos os componentes do ecossistema .NET. Esta experiência reforçou a importância de verificar consistentemente a propriedade TargetFramework e utilizar ferramentas como dotnet list package para auditoria regular de dependências.

3.2.3 Configuração CORS para Desenvolvimento

No ambiente de desenvolvimento local, requisições do Swagger UI para a API eram bloqueadas por políticas CORS, impedindo testes diretos através da interface web.

Solução: Implementou-se configuração CORS diferenciada para ambiente de desenvolvimento, permitindo qualquer origem durante os testes locais. Esta configuração é ativada apenas no ambiente Development, com políticas mais restritivas configuradas para produção através de variáveis de ambiente.

3.3 Padrões de Design Aplicados

3.3.1 Separação em Camadas

A aplicação foi estruturada em três camadas principais:

- Controladores**:** Responsáveis por receber requisições HTTP e retornar respostas
- Serviços**:** Contêm toda a lógica de negócio e regras de domínio
- Modelos/DTOs**:** Representam as entidades e objetos de transferência

Esta separação promoveu a testabilidade e manutenibilidade, permitindo que cada componente fosse desenvolvido e testado isoladamente.

3.3.2 DTO Pattern

Foram implementados Data Transfer Objects específicos para diferentes operações:

- ReservationDto para operações de leitura
- CreateReservationDto para criação de novas reservas
- UpdateReservationDto para atualizações parciais

Esta separação fornece contratos de API mais claros e específicos para cada cenário de uso.

3.4 Validações e Segurança

A API implementa validação em múltiplos níveis:

- Anotações de dados nos modelos (Required, MaxLength, Range)
- Validação de modelo automática através do ModelState
- Validações personalizadas na camada de serviço para regras complexas
- Constraints na base de dados como última linha de defesa

Para segurança básica, implementou-se:

- HTTPS redirection em ambientes de produção
- Validação rigorosa de todos os inputs
- Proteção contra SQL injection através do uso de parâmetros no Entity Framework
- Logging estruturado para auditoria e troubleshooting

4. BASE DE DADOS E CONTEINERIZAÇÃO

Esta fase focou-se na criação de uma infraestrutura de dados robusta e de um ambiente de execução portátil e consistente para a aplicação. As principais componentes foram a configuração da base de dados, a Conteinerização dos serviços e a sua orquestração para um ambiente de produção.

4.1 Configuração do SQL Server

A estrutura da tabela principal (Reservations) foi definida em código C#, com a decisão de usar tipos de dados específicos (DATE e TIME) para melhor desempenho e validação nativa. As alterações ao esquema foram geridas de forma controlada através do sistema de migrações do EF Core, permitindo atualizações seguras e versionadas da base de dados.

4.2 Conteinerização com Docker

A API foi empacotada num container utilizando um Dockerfile multi-stage. Esta abordagem separa o processo de construção do ambiente de execução final, resultando numa imagem otimizada, segura e com um tamanho drasticamente reduzido (cerca de 200MB). O SQL Server foi também configurado para correr num container, permitindo um ambiente de desenvolvimento completo e isolado.

4.3 Orquestração com Docker Compose

Para coordenar e ligar os múltiplos containers, utilizou-se o Docker Compose. O ficheiro docker-compose.yml define tanto o serviço da API como o do SQL Server, estabelecendo uma rede interna para comunicação, dependências de arranque e mapeamento de portas para acesso local. Foi também configurado um volume persistente para garantir que os dados da base de dados sobrevivem a reinicializações dos containers.

4.4 Preparação para Produção

Na base de dados, foi criado um índice único composto nas colunas de data, mesa e hora, acelerando drasticamente a verificação de conflitos de reservas. Na aplicação, configurou-se o Connection Pooling com resiliência a falhas. Foram ainda adotadas práticas de segurança para containers, como a execução com um utilizador não-privilegiado, e preparados mecanismos de monitorização (como *health checks* e *logging* estruturado).

4.5 Desafios Técnicos e Soluções

4.5.1 Sincronização na Inicialização dos Containers

A API iniciava e tentava ligar-se ao SQL Server antes deste estar completamente operacional, resultando em falhas de conexão.

Solução: Implementou-se uma estratégia em três níveis:

1) Configuração da dependência depends_on no Docker Compose;
2) Implementação de uma lógica de *retry* com *backoff* exponencial no código da API;
3) Utilização de *health checks* para verificar a disponibilidade do SQL Server antes de permitir ligações.

4.5.2 Gestão de Configurações por Ambiente

Era necessário utilizar conexões diferentes para desenvolvimento local (ligando a localhost) e para o ambiente conteinerização (ligando ao serviço sqlserver).

Solução: Adotou-se o sistema de configuração do ASP.NET Core. A configuração para desenvolvimento foi mantida no ficheiro appsettings.Development.json, enquanto a configuração para os containers foi realizada através de variáveis de ambiente no ficheiro docker-compose.yml, assegurando separação e segurança.

4.5.3 Autenticação Inicial no SQL Server em Container

Erros intermitentes de "Login failed for user 'sa'" ocorriam mesmo com credenciais corretas.

Solução: Identificou-se que o container do SQL Server precisa de tempo extra (cerca de 30-60 segundos) após ser considerado "em execução" para estar realmente pronto a aceitar ligações. A solução combinou o *retry pattern* na aplicação com a utilização de scripts de espera na orquestração, garantindo que a API só tenta ligar-se quando o serviço de base de dados está verdadeiramente disponível.

5. TESTES UNITÁRIOS

Esta fase focou-se em garantir a fiabilidade, robustez e qualidade do código através de testes automatizados. A implementação de um conjunto abrangente de testes unitários foi essencial para validar a lógica de negócio crítica, especialmente a gestão de conflitos de horário, e para assegurar que a API responde corretamente em diferentes cenários.

5.1 Estrutura e Abordagem de Testes

Os testes foram implementados utilizando o framework xUnit, seguindo o padrão *Arrange-Act-Assert* (AAA). A biblioteca Moq foi fundamental para simular as dependências, como o DbContext, permitindo testar a lógica da aplicação de forma isolada, sem depender de uma base de dados real. Para medição da eficácia, a ferramenta coverlet foi integrada para gerar relatórios de cobertura de código. A estrutura dos testes foi organizada por camadas da aplicação, espelhando a organização do projeto principal. Os testes realizados são:

Tabela 2 - Lista de Testes Unitários

	Testes	Função
	Models/ReservationTests.cs (3 testes)	
1	Reservation_Should_Have_Default_CreatedAt	Verifica que a propriedade CreatedAt é automaticamente definida com a data/hora atuais na criação de um novo objeto Reservation.
2	Reservation_Should_Set_CustomerName("João Silva")	Testa a correta atribuição e recuperação da propriedade CustomerName com um valor de exemplo.
3	Reservation_Should_Set_CustomerName("Maria Santos")	Segundo caso de teste para a propriedade CustomerName, garantindo o funcionamento com diferentes inputs.
	DTOs/ReservationDtoTests.cs (2 testes)	
4	CreateReservationDto_Validation_Should_Fail_When_Empty	Valida que o Data Transfer Object (DTO) de criação rejeita dados inválidos ou vazios, acionando as anotações de validação ([Required]).
5	CreateReservationDto_Validation_Should_Pass_When_Valid	Confirma que um DTO preenchido com todos os campos obrigatórios e formatos corretos passa na validação.
	Services/ReservationServiceTests.cs (6 testes)	
6	CreateReservationAsync_Should_Create_Reservation	Testa o cenário de sucesso do método principal do serviço, verificando se uma reserva válida é criada e persistida.
7	CreateReservationAsync_Should_Throw_When_TimeConflict	Valida a lógica central de negócio que impede a criação de duas reservas para a mesma mesa na mesma data e hora, garantindo a unicidade.
8	GetAllReservationsAsync_Should_Return_All_Reservations	Verifica se o serviço retorna a coleção completa de reservas existentes.

9	GetReservationByIdAsync_Should_Return_Null_For_Invalid_Id	Confirma que o serviço retorna null quando é solicitada uma reserva com um identificador que não existe, testando a resiliência a inputs incorretos.
10	DeleteReservationAsync_Should_Return_False_For_Invalid_Id	Testa o comportamento do serviço ao tentar eliminar uma reserva inexistente, esperando um resultado false.
11	ReservationServiceTests (setup/constructor)	Prepara o ambiente de teste isolado, inicializando os mocks do DbContext e do IMapper antes da execução de cada teste.
	Controllers/ReservationsControllerTests.cs (3 testes)	
12	Post_Should_Return_Created_Reservation	Verifica que o endpoint POST /api/reservations retorna o status HTTP 201 (Created), o corpo da reserva criada e o cabeçalho Location correto.
13	Get_Should_Return_Ok_With_Reservations	GET /api/reservations retorna status 200 (OK) com a lista de reservas no corpo da resposta.
14	GetById_Should_Return_NotFound_For_Invalid_Id	Garante que o endpoint GET /api/reservations/{id} responde corretamente com status 404 (Not Found) para um ID de reserva inválido.

5.2 Lógica de Negócio e Testes Críticos

O teste #7 (CreateReservationAsync_Should_Throw_When_TimeConflict) merece destaque especial, pois valida o requisito funcional mais complexo do sistema. A sua implementação garantiu que a verificação de conflitos — que combina os campos ReservationDate, ReservationTime e TableNumber — funciona corretamente, lançando uma exceção apropriada quando um conflito é detetado. Esta validação é a base da integridade dos dados do sistema.

5.3 Cobertura de Código e Integração no CI/CD

Com a execução bem-sucedida dos 14 testes listados, o projeto atingiu uma cobertura de código significativa (superior a 80%), focada nas partes mais críticas: modelos, DTOs, serviços e controladores. O comando dotnet test --collect:"XPlat Code Coverage" foi executado com sucesso, gerando um relatório XML (coverage.cobertura.xml) que pode ser ingerido por ferramentas como o SonarQube.

Este conjunto de testes está integralmente integrado no pipeline CI/CD definido no Jenkinsfile. A etapa *"Unit Tests"* executa automaticamente todos os testes em cada *commit*, e a etapa *"Quality Gate"* subsequente pode bloquear a progressão para *deploy* se a cobertura ou a qualidade do código não cumprirem os padrões definidos, assegurando que apenas código testado e validado avança para ambientes superiores.

6. PIPELINE CI/CD COM JENKINS

A implementação de um *pipeline* de Integração Contínua e *Deploy* Contínuo (CI/CD) foi fundamental para automatizar o ciclo de vida do software, garantindo qualidade, consistência e agilidade na entrega. O *pipeline* declarativo configurado no Jenkins orquestra automaticamente, a partir de um *commit* no repositório Git, todo o fluxo desde a compilação até ao *deploy* em produção.

6.1 Arquitetura e Visão Geral do *Pipeline*

O *pipeline* foi definido como um processo declarativo no ficheiro Jenkinsfile na raiz do projeto, onde um artefacto imutável (imagem Docker) é construído uma vez e promovido através dos ambientes. Consiste em 9 *stages* sequenciais com *gates* de qualidade:

Tabela 3 - Stages do Pipeline Jenkins

Stage	Ferramenta/Comando	Finalidade e Saída
1. Checkout	checkout scm	Obtém o código mais recente do repositório Git configurado no Jenkins.
2. Restore	dotnet restore	Restaura todas as dependências do projeto (pacotes NuGet).
3. Build	dotnet build --configuration Release	Compila a solução .NET no modo Release, gerando os binários.
4. Unit Tests	dotnet test --collect:"XPlat Code Coverage"	Executa a *suite* de testes unitários (xUnit) e gera relatórios de cobertura em formato Cobertura XML e resultados em formato TRX.
5. SonarQube Analysis	dotnet sonarscanner	Envia o código-fonte e os relatórios de testes/cobertura para análise estática. Deteta *bugs*, vulnerabilidades e *code smells*.
6. Quality Gate	waitForQualityGate	Aguarda e valida o resultado da análise do SonarQube. O *pipeline* só prossegue se o código cumprir os padrões de qualidade (ex: cobertura >80%, zero vulnerabilidades críticas).
7. Build Docker Image	docker build	Constrói a imagem Docker da API utilizando o Dockerfile multi-stage, gerando uma imagem leve e otimizada, taggeada com o BUILD_ID único do Jenkins.
8. Push Docker Image	docker push	Faz *upload* da imagem construída para um *registry* privado (ex: Docker Hub), tornando-a disponível para o cluster Kubernetes.
9. Deploy to Kubernetes	helm upgrade --install	Atualiza ou instala a aplicação no cluster Kubernetes utilizando os *charts* Helm. Injeta configurações específicas do *build* (como a tag da imagem) de forma segura.

6.2 Integração e *Feedback* Rápido

O *pipeline* está configurado para ser acionado automaticamente no Git. Qualquer *push* para o ramo principal (ex: main) inicia imediatamente uma nova execução. Esta integração proporciona *feedback* rápido aos desenvolvedores. Em caso de falha em qualquer *stage*, a equipa é notificada e o *deploy* é automaticamente impedido, garantindo que o ramo principal permanece sempre estável.

6.3 Gestão de Segurança e Configuração

A segurança é tratada de forma centralizada através do armazenamento de credenciais do Jenkins:

- Credenciais do Docker Registry: Armazenadas como dockerhub-credentials, usadas no *stage* de *push*.
- Token do SonarQube: Armazenado como SONAR_TOKEN para autenticação na análise.
- Segredos da Aplicação: A *password* da base de dados é injetada no *deploy* via Helm a partir da credencial db-password, nunca estando hardcoded no código ou nos *charts*.

6.4 Estratégia de *Deploy* e Resiliência

O *deploy* final utiliza o comando helm upgrade --install com a flag --atomic. Esta é uma prática essencial que garante que, se a atualização no Kubernetes falhar, todo o *deploy* é revertido automaticamente, mantendo a versão anterior da aplicação operacional e assegurando a resiliência do serviço.

7. DEPLOYMENT COM KUBERNETES E HELM

A fase final da automatização focou-se na preparação da aplicação para ambientes de produção escaláveis e resilientes. Através da utilização do Kubernetes para orquestração de containers e do Helm para gestão de packages e configurações, foi possível definir uma infraestrutura como código robusta, portátil e fácil de gerir.

7.1 Estrutura do *Helm Chart*

O Helm, o gestor de *packages* para Kubernetes, foi utilizado para empacotar, parametrizar e gerir o *deployment* da aplicação. A estrutura do *chart* restaurant-api organiza os manifestos Kubernetes de forma modular e configurável.

Tabela 4 - Estrutura do Helm Chart restaurant-api

Ficheiro	Finalidade e Conteúdo Principal
Chart.yaml	Metadados do *package*: nome (restaurant-api), versão, descrição e versão da aplicação.
values.yaml	Define valores padrão parametrizáveis para todos os ambientes (ex: número de réplicas, *image tag*, configurações da BD, recursos).
templates/_helpers.tpl	Funções auxiliares e *templates* partilhados (ex: lógica para gerar nomes de labels).
templates/configmap.yaml	Define um *ConfigMap* para armazenar configurações da aplicação não sensíveis (ex: níveis de *log*).
templates/deployment.yaml	*Template* principal que define o *Deployment* da API. Especifica a imagem do container, variáveis de ambiente, *probes* de saúde, limites de recursos e a estratégia de atualização.
templates/service.yaml	Define o *Service* Kubernetes do tipo ClusterIP que expõe a API internamente no *cluster*, permitindo o *load balancing* entre os *pods*.

7.2 Configurações-Chave e Boas Práticas Implementadas

O *chart* Helm incorpora várias boas práticas de *deployment* em produção:

- Resiliência e Saúde**:** O *Deployment* configura livenessProbe e readinessProbe que verificam o endpoint /api/reservations/status. Estas sondas garantem que o Kubernetes só encaminha tráfego para *pods* saudáveis e recicla automaticamente instâncias com problemas.
- Gestão de Recursos**:** São definidos limites e pedidos e CPU e memória para os containers, permitindo ao *scheduler* do Kubernetes tomar decisões otimizadas e evitar a exaustão de recursos do nó.
- Segurança de Configuração**:** As credenciais da base de dados (ex: *password*) são injetadas como variáveis de ambiente a partir de Kubernetes Secrets, referenciados no *pipeline* Jenkins, nunca estando no values.yaml ou no código.
- Portabilidade**:** Todas as dependências de ambiente (como o *hostname* do SQL Server) são parametrizadas no values.yaml, permitindo implantar a mesma *chart* em diferentes ambientes (dev, staging, prod) apenas sobrescrevendo estes valores.

7.3 Fluxo de Deploy Automatizado

O *stage* final do *pipeline* Jenkins executa o comando Helm que concretiza o *deploy*:

```
helm upgrade --install restaurant-api ${HELM_CHART_PATH} \
    --namespace ${KUBE_NAMESPACE} \
    --set image.repository=${DOCKER_IMAGE} \
    --set image.tag=${env.BUILD_ID} \
    --set database.password=${DB_PASSWORD} \
    --atomic \
    --timeout 5m
```

Este comando é executado dentro de um bloco withCredentials que injeta seguramente a *password* da base de dados a partir das credenciais geridas pelo Jenkins. A flag --atomic garante que, se o *deploy* falhar em qualquer ponto, é executado um rollback automático para a versão estável anterior, assegurando a disponibilidade contínua do serviço. A utilização da tag única da imagem (${env.BUILD_ID}) garante a imutabilidade e a rastreabilidade total, assegurando que é implantada a versão exata que passou por todas as fases de validação do *pipeline* de CI/CD.

8. ANÁLISE DE QUALIDADE COM SONARQUBE

Para garantir a manutenibilidade, segurança e qualidade a longo prazo do código, o projeto integra a análise estática contínua através do SonarQube. Esta ferramenta atua como um filtro automatizado, assegurando que apenas código que cumpre padrões rigorosos de qualidade avança no *pipeline* de CI/CD.

8.1 Configuração e Integração

A integração com o SonarQube foi configurada em duas camadas. Primeiro, através do ficheiro sonar-project.properties na raiz do projeto, que define os metadados e *paths* essenciais para a análise.

Segundo, e de forma mais crucial, através do Jenkinsfile, onde um *stage* dedicado (SonarQube Analysis) executa o *scanner* e um *stage* subsequente (Quality Gate) valida os resultados de forma bloqueante.

8.2 O Stage de Análise e a Quality Gate

A etapa SonarQube Analysis no *pipeline* tem a função crítica de enviar o código-fonte e os relatórios de testes e cobertura (gerados no *stage* Unit Tests) para o servidor SonarQube. Este realiza uma análise profunda, verificando:

- Bugs: Erros no código que quase certamente causarão mau funcionamento.
- Vulnerabilidades: Problemas de segurança que poderiam ser explorados.
- Code Smells: Más práticas de desenvolvimento que impactam a manutenibilidade.

O resultado desta análise é sintetizado na SonarQube Quality Gate. Esta é uma política de qualidade configurável que define os requisitos mínimos para o código. No *pipeline*, o *stage* Quality Gate utiliza o comando waitForQualityGate para consultar este veredito de forma bloqueante.

8.3 Impacto no Processo de Desenvolvimento

Esta integração transforma o SonarQube num checkpoint automatizado e obrigatório. Se um desenvolvedor fizer um *commit* que introduza uma vulnerabilidade ou que baixe a cobertura de testes abaixo do limiar estabelecido, o *pipeline* irá falhar na etapa Quality Gate. Consequentemente, a imagem Docker não será construída e o *deploy* para Kubernetes não será executado.

9. Ambientes de Execução

O projeto dispõe de dois ambientes de desenvolvimento distintos:

O localhost:5188 (Desenvolvimento Local): Faz a execução via dotnet run no ambiente nativo do sistema utilizando launchSettings.json e appsettings.Development.json.

Características:

- Hot reload automático para desenvolvimento ágil;
- Facilidade de debugging em ferramentas como VS Code;
- Requer instância de SQL Server disponível localmente ou em container.

E o localhost:5000 (Ambiente Conteinerização): Faz a execução via docker-compose up com containers Docker gerida por docker-compose.yml e variáveis de ambiente no Dockerfile.

Características:

- Ambiente isolado idêntico ao de produção;
- SQL Server executado em container separado;
- Porta 5000 mapeada para a porta 8080 interna do container.

Portas e Configuração

- A porta 5188 é padrão do template .NET Web API;
- A porta 5000 é comum para desenvolvimento e mapeia para 8080 no container;
- Ambas podem ser configuradas nos respetivos ficheiros de configuração.

10. Operações do Sistema

Este capítulo serve como guia técnico completo para a execução, teste e validação do sistema de gestão de reservas de restaurante desenvolvido no âmbito da unidade curricular de Desenvolvimento/Operação de Software. O sistema implementa uma API REST com funcionalidades completas de CRUD para reservas, incluindo conteinerização, CI/CD automatizado e análise de qualidade de código.

10.1. Execução do Sistema

10.1.1. Ambiente de Desenvolvimento Local

Execução com Docker Compose

Na raiz do projeto:

Parar serviços existentes:

- docker-compose down

Construir e iniciar containers:

- docker-compose up --build -d

URLs de acesso:

- Swagger UI:

- *http://localhost:5000/swagger*

- Health Check:

- *http://localhost:5000/api/reservations/status*

Serviços iniciados:

- restaurant-reservations-api (Porta 5000)
- restaurant-reservations-sqlserver (Porta 1433)

Execução nativa .NET

- cd src/RestaurantReservations.API
- dotnet restore
- dotnet run

URL de acesso:

- *http://localhost:5188/swagger*

10.1.2. Verificação do Estado dos Serviços

Verificar containers em execução:

- docker-compose ps

Verificar logs da aplicação:

- docker-compose logs api

10.2. Execução de Testes Unitários

10.2.1. Execução Completa com Cobertura

- cd tests/RestaurantReservations.UnitTests
- dotnet test --collect:"XPlat Code Coverage" --verbosity normal

10.2.2. Execução Específica

Executar apenas um teste:

- dotnet test --filter "GetAllReservationsAsync_Should_Return_All_Reservations"

Executar testes de uma classe específica:

- dotnet test --filter "FullyQualifiedName~ReservationServiceTests"

Executar sem relatório de cobertura:

- dotnet test

10.3. Gestão de Containers Docker

10.3.1. Comandos Essenciais

Na raiz do projeto:

Iniciar todos os serviços:

- docker-compose up -d

Parar todos os serviços:

- docker-compose down

Reconstruir e reiniciar:

- docker-compose up --build -d

Ver logs em tempo real:

- docker-compose logs -f api
- docker-compose logs -f sqlserver

Ver estado dos containers:

- docker-compose ps

10.4. Validação Funcional da API

10.4.1. Testes Manuais via Swagger

Aceder a:

- http://localhost:5000/swagger permite testar todos os endpoints:

Endpoints disponíveis:

- GET /api/reservations - Listar todas reservas
- GET /api/reservations/{id} - Obter reserva específica
- POST /api/reservations - Criar nova reserva
- PUT /api/reservations/{id} - Atualizar reserva
- DELETE /api/reservations/{id} - Cancelar reserva
- GET /api/reservations?date={date} - Filtrar por data

10.5. Validação de Funcionalidades Críticas

10.5.1. Teste de Conflito de Horário

Criar primeira reserva:

```
'{
  "customerName": "Cliente 1",
  "reservationDate": "2026-06-30",
  "reservationTime": "19:30:00",
  "tableNumber": 5,
  "numberOfPeople": 4
 }'
```

Tentar criar reserva conflitante (deve retornar 409):

```
'{
  "customerName": "Cliente 2",
  "reservationDate": "2026-06-30",
  "reservationTime": "19:30:00",
  "tableNumber": 5,
  "numberOfPeople": 2
 }'
```

10.5.2. Teste de Validação de Data Futura

Tentar criar reserva no passado (deve falhar):

```
'{
  "customerName": "Cliente Teste",
  "reservationDate": "2026-01-22",
  "reservationTime": "12:00:00",
  "tableNumber": 1,
  "numberOfPeople": 2
 }'
```

11. CONCLUSÃO

O projeto RestaurantReservations API foi implementado com sucesso, percorrendo todo o ciclo de desenvolvimento de uma aplicação moderna. Desenvolveu-se uma API REST funcional em .NET 9 com lógica crítica de conflitos de horário e uma base de dados SQL Server, ambas totalmente conteinerizadas com Docker para garantir portabilidade. A qualidade do código foi assegurada por um conjunto de testes unitários e integrada num pipeline CI/CD automático com Jenkins, que orquestra testes, análise estática com SonarQube, construção de imagens e *deploy* seguro para Kubernetes via Helm.

Os principais desafios técnicos, como a sincronização de containers e a gestão de configurações, foram resolvidos com *health checks* e variáveis de ambiente, reforçando a importância da resiliência e automação. O sistema resultante é robusto, escalável e pronto para produção, demonstrando a aplicação prática de um *stack* tecnológico completo e de metodologias DevOps.