

# Relatório final DOS

## **Grupo 3**

1231551 \_ Pedro Gonçalves

1231546 \_ Marcelo Ferreira

1231752 \_ João Ferreira

1231547 \_ Marco Machado

---

## **Docente    Orientador**

Amaral, Diogo (JDA)

## **Unidade Curricular**

Desenvolvimento / Operação de software

## Índice

1.	Introdução .....	2
2.	Contextualização do projeto.....	3
3.	Planeamento e organização do trabalho.....	4
4.	Desenvolvimento da solução.....	5
5.	Implementação web API .....	6
6.	<i>Vagrant</i> com base de dados.....	7
7.	Testes unitários.....	9
8.	<i>Docker</i> e containerização.....	10
9.	<i>Pipeline</i> automatizado com <i>Jenkins</i> .....	12
11.	Conclusão .....	19
12.	Referências .....	20

## 1. Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de Desenvolvimento / Operação de software, onde foram adquiridos conhecimentos ao longo do semestre através de exercícios semanais. O objetivo principal foi criar uma aplicação pronta para produção, aplicando práticas de integração contínua e entrega contínua (CI/CD).

A solução proposta consiste numa Web API baseada em REST API, destinada à gestão de reservas de mesas nos restaurantes. O sistema inclui *endpoints* para criar, consultar, atualizar e cancelar reservas, além de garantir que não ocorram conflitos de horários entre diferentes reservas para a mesma mesa.

Foi desenvolvido com o objetivo de aplicar práticas fundamentais de *DevOps* para assegurar a qualidade da aplicação. Foram implementadas ferramentas de automação, containerização e integração contínua para garantir o fluxo de desenvolvimento em conformidade com os requisitos estabelecidos. O projeto incluiu a utilização de *Docker* e *Jenkins* para a gestão de containers e pipelines automatizados e a configuração de uma base de dados numa máquina virtual seguindo a abordagem *Code First*.

Este relatório apresenta todo o processo de desenvolvimento, desde a configuração inicial do ambiente até a entrega do sistema pronto para produção. Também são discutidos os desafios enfrentados, as decisões tomadas e as potenciais melhorias identificadas ao longo do projeto.

## 2. Contextualização do projeto

### Objetivo Geral do Projeto

O principal objetivo deste projeto é desenvolver uma aplicação que permita implementar as principais práticas de *DevOps* que promovem a automação, a integração contínua e a entrega contínua. O foco foi criar uma Web API que suporte a gestão de dados, enquanto se utilizam ferramentas para garantir qualidade e eficiência no ciclo de vida do desenvolvimento de software.

### Requisitos Principais

- Implementação de uma Web API: Criar uma API que suporte operações CRUD e integração com a base de dados.
- Containerização: Utilizar *Docker* para criar containers.
- Configuração de Base de Dados: Configurar uma base de dados numa máquina virtual usando *Vagrant* seguindo a abordagem *Code First* para a criação das tabelas.
- *Pipelines* de CI/CD: Implementar um pipeline automatizado com *Jenkins*, que inclua automação de testes e integração com ferramentas como o *SonarQube* para análise do código.
- Gestão de Conflitos: Utilizar práticas que minimizem conflitos no código, como a adoção de *Git* para controlo de versões.
- Análises Estáticas do Código: Garantir a qualidade do código através da integração com ferramentas como o *SonarQube*.
- Automação de Testes: Implementar testes automatizados.

### 3. Planeamento e organização do trabalho

Inicialmente dividiu-se as responsabilidades pelos membros, utilizando o *Trello* para organizar e acompanhar as tarefas. Cada membro ficou responsável por uma área específica do projeto:

- Marcelo: desenvolvimento da Web API;
- Pedro: configuração do Vagrant e da base de dados
- Marco: implementação e execução dos testes;
- João: configuração do *Jenkins* e do *SonarQube*.

Contudo, devido à simultaneidade com outros projetos, só foi possível dedicar exclusivamente a este projeto durante os últimos 4 dias antes da entrega. Diante deste cenário, ficou decidido adaptar e adotar uma abordagem mais direta e eficiente: todos os membros da equipa trabalharam em conjunto em calls via *Teams*.

Durante as sessões, enquanto um dos membros realizava a implementação, os restantes pesquisavam soluções para os problemas que surgiam e documentavam em relatório. Essa metodologia garantiu um trabalho mais dinâmico e orientado à resolução rápida do projeto permitindo a entrega dentro do prazo estipulado.

Utilizámos o *GitHub* como ferramenta para controlo de versões, onde aplicámos *tags* para organizar o código e identificar as diferentes funcionalidades implementadas

## 4. Desenvolvimento da solução

A aplicação foi desenvolvida seguindo uma arquitetura de camadas, separando as responsabilidades em Modelos, Controladores e Serviços. As configurações da aplicação, como a ligação à base de dados, estão no ficheiro `appsettings.json`.

A solução inclui ainda um projeto separado para testes automatizados, o `RestauranteTestes`, que contém testes desenvolvidos para validar o comportamento da aplicação.

As tecnologias escolhidas para o desenvolvimento foram `.NET Core`, `SQL Server`, `Docker`, `Jenkins` e `SonarQube`. O `.NET Core` foi utilizado para a construção da Web API devido à sua flexibilidade e desempenho, além de ser uma tecnologia utilizada pela equipa ao longo do semestre. O `Docker` foi utilizado para containerizar a aplicação para garantir a portabilidade e facilitando o desenvolvimento em ambientes homogêneos, além de permitir a integração com ferramentas como `Jenkins` e `SonarQube`. O `Jenkins` foi implementado para criar pipelines automatizados e reduzir o esforço manual e garantindo consistência no ciclo de desenvolvimento e integração contínua (CI/CD). Por fim, o `SonarQube` foi integrado para realizar análises estáticas de código, identificando potenciais vulnerabilidades.

As escolhas tecnológicas foram feitas com base no conhecimento da equipa, considerando que todas as tecnologias foram trabalhadas ao longo do semestre. Devido ao tempo limitado, foram priorizadas ferramentas conhecidas para evitar a curva de aprendizagem associada a novas tecnologias. Entre os principais benefícios destas ferramentas, destaca-se o `Docker` para garantir a uniformidade dos ambientes de desenvolvimento e produção, o `Jenkins` para automatizar tarefas repetitivas e o `SonarQube` para aumentar a confiabilidade da solução.

## 5. Implementação web API

### Endpoints Implementados

A Web API do projeto RestauranteFinal tem vários *endpoints* implementados no controlador de reservas:

- GET /api/reservations: Este endpoint retorna todas as reservas existentes no sistema.
- GET /api/reservations/{id}: Este endpoint retorna uma reserva específica com base no ID fornecido.
- POST /api/reservations: Este endpoint permite a criação de uma nova reserva.
- PUT /api/reservations/{id}: Este endpoint permite a atualização de uma reserva existente com base no ID fornecido.
- DELETE /api/reservations/{id}: Este endpoint permite a exclusão de uma reserva específica com base no ID fornecido.

Uma das principais regras de negócio implementadas na API é a validação de conflitos nas reservas. Antes de criar ou atualizar uma reserva, a API verifica se já existe uma reserva para o mesmo horário, isso garante que não haja conflitos e que cada mesa fique disponível apenas para uma reserva.

O Swagger é utilizado para gerar a documentação automática da API, está configurado para permitir que se visualizem e testem os *endpoints* da API diretamente no navegador. A documentação gerada pelo Swagger inclui todos os *endpoints* disponíveis.

## 6. Vagrant com base de dados

Para configurar o SQL Server na máquina virtual *Vagrant*, seguimos os seguintes passos:

- I. Inserir o Vagrantfile: Colocamos o ficheiro Vagrantfile dentro da pasta do projeto.
- II. Configuração do Switch Virtual: No gestor *Hyper-V*, configuramos um switch virtual para permitir a comunicação entre a máquina virtual e o host.
- III. Conteúdo do Vagrantfile: Inserimos o conteúdo necessário dentro do Vagrantfile para configurar a máquina virtual com o SQL Server.

```
Vagrant.configure("2") do |config|
  # Box
  config.vm.box = "gusztavvargadr/sql-server"

  # Configurações do provedor Hyper-V
  config.vm.provider "hyperv" do |hv|
    # Configuração de memória
    hv.memory = 2048
    # Configuração de CPU (opcional, ajuste conforme necessário)
    hv.cpus = 2
  end

  # Redirecionamento de portas
  config.vm.network "forwarded_port", guest: 1433, host: 1234

  # Configuração de rede privada (opcional, descomente se necessário)
  # config.vm.network "private_network", type: "dhcp"

  #config.vm.network "private_network", type: "dhcp", bridge: "ExternalSwitch"
  config.vm.network "private_network", type: "dhcp", bridge: "VagrantSwitch"

  config.vm.synced_folder ".", "/vagrant", disabled: true
end
```

Figura 1 - Vagrantfile do RestauranteFinal

- IV. Inicialização da máquina virtual: Abrimos o *PowerShell* como administrador, navegamos até a localização do nosso Vagrantfile e executamos o comando **vagrant up --provider=hyperv**. A máquina virtual foi então configurada e iniciada.
- V. Configuração da firewall: Para permitir a comunicação entre as máquinas, abrimos o *PowerShell* como administrador e executamos o comando **New-NetFirewallRule -DisplayName "Allow ICMPv4-In" -Protocol ICMPv4 -Direction Inbound -Action Allow** para alterar a configuração da firewall.
- VI. Configuração do SQL Server: Abrimos o Microsoft SQL Management Studio e efetuamos login com Windows Authentication. Alteramos a password do utilizador SA para



"Teste123!" seguindo o caminho Security > Login > SA (clique com o botão direito) > Properties.

VII. Instalação de bibliotecas: No projeto, instalamos as seguintes bibliotecas:

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

VIII. Configuração do appsettings.json: Configuramos o arquivo [appsettings.json](#) com a ligação à base de dados da nossa máquina virtual.

IX. Execução de comandos: Executamos os seguintes comandos dentro da pasta do projeto:

- **dotnet tool update --global dotnet-ef**
- **dotnet ef migrations add InitialCreate**

X. Ativação do protocolo TCP/IP: No SQL Server Configuration Manager, ativamos o protocolo TCP/IP e reiniciamos o serviço de base de dados para permitir a comunicação entre a máquina do projeto e a base de dados.

XI. Atualização da base de dados: Executamos o comando **dotnet ef database update** para aplicar as migrações e criar a estrutura da base de dados.

XII. Verificação das operações da API: Após a configuração, verificamos se as chamadas efetuadas pela API são refletidas na base de dados, confirmando que a integração está a funcionar corretamente.

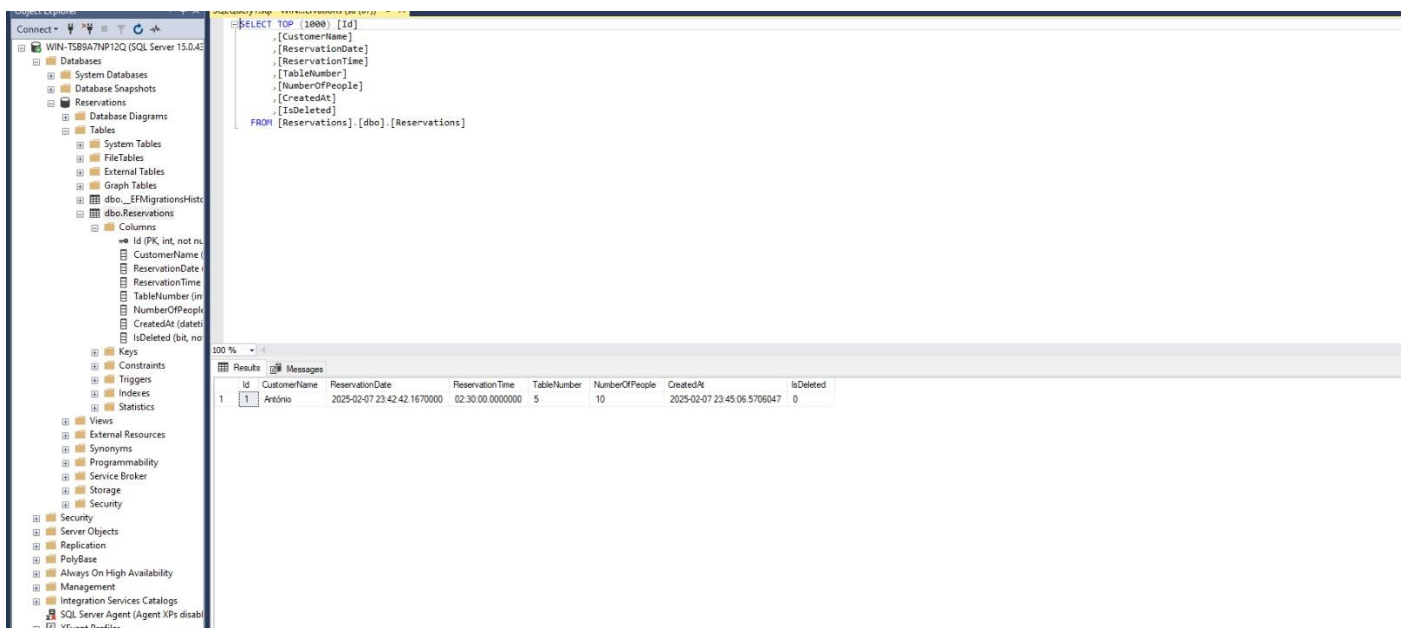


Figura 2 - Tabela criada no SQL Server do vagrant

## 7. Testes unitários

Para a implementação dos testes unitários utilizamos o *framework* xUnit.

Os testes unitários foram desenvolvidos para cobrir os principais cenários de uso da API, que inclui a verificação de todas as operações CRUD e a validação das regras de negócio implementadas.

### Validação de Conflitos

Outro cenário importante testado é a validação de conflitos nas reservas. Antes de criar ou atualizar uma reserva, a API verifica se já existe uma reserva para o mesmo horário e mesa. Os testes garantem que essa validação funcione corretamente, evitando conflitos de reservas.

### Exclusão Lógica de Reservas – *Soft delete*

A exclusão lógica de reservas, onde uma reserva é marcada como apagada sem ser removida fisicamente da base de dados, também é testada. Os testes garantem que a reserva é marcada como apagada e que não volta a ser retornada nas consultas de reservas ativas.

### Exemplos de Testes Implementados

- **GetAllReservations\_ReturnsAllActiveReservations:** Verifica se todas as reservas ativas são retornadas corretamente.
- **GetReservationById\_ReturnsReservation\_WhenExists:** Verifica se uma reserva específica é retornada corretamente quando existe.
- **GetReservationById\_ReturnsNotFound\_WhenDoesNotExist:** Verifica se o retorno é NotFound quando a reserva não existe.
- **CreateReservation\_AddsReservationSuccessfully:** Verifica se uma nova reserva é adicionada com sucesso.
- **UpdateReservation\_UpdatesReservationSuccessfully:** Verifica se uma reserva existente é atualizada corretamente.
- **UpdateReservation\_ReturnsNotFound\_WhenReservationDoesNotExist:** Verifica se o retorno é NotFound quando a reserva a ser atualizada não existe.
- **SoftDeleteReservation\_MarksReservationAsDeleted:** Verifica se uma reserva é marcada como apagada corretamente.
- **SoftDeleteReservation\_ReturnsNotFound\_WhenReservationDoesNotExist:** Verifica se o retorno é NotFound quando a reserva a ser apagada não existe.
- **GetReservationsByDate\_ReturnsReservationsOnSpecificDate:** Verifica se as reservas em uma data específica são retornadas corretamente.

## 8. Docker e containerização

Para containerizar a aplicação RestauranteFinal:

- I. Foi criado um Dockerfile que define o ambiente necessário para executar a aplicação. O Dockerfile utiliza a imagem base do .NET SDK para compilar e publicar a aplicação, e a imagem base do .NET ASP.NET para executar a aplicação.

```
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /app
COPY . .
RUN dotnet restore
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY --from=build /app/out .
EXPOSE 8080
CMD ["dotnet", "RestauranteFinal.dll"]
```

Figura 3 - Dockerfile do RestauranteFinal

- II. Instalamos e iniciamos o software *Docker Desktop* e de seguida, na consola, corremos o comando **docker build -t restaurante-api** e depois o comando **docker run -d -p 8080:8080 --name restaurante-container restaurante-api**
- III. Para adicionar o *Jenkins* e o *SonarQube* ao nosso ambiente, criamos o `docker-compose.yml`, que permite gerir múltiplos containers.

```

version: '3.8'
services:
  api:
    container_name: restaurante-api
    build: .
    ports:
      - "8080:8080"
    depends_on:
      - jenkins
      - sonarqube

  jenkins:
    container_name: jenkinsRestaurante
    image: jenkins/jenkins:lts
    ports:
      - "8081:8080"
      - "50000:50000"
    volumes:
      - jenkins_data:/var/jenkins_home
    restart: unless-stopped

  sonarqube:
    container_name: sonarqubeRestaurante
    image: sonarqube:lts
    ports:
      - "9000:9000"
      - "9092:9092"
    environment:
      SONARQUBE_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONARQUBE_JDBC_USERNAME: sonar
      SONARQUBE_JDBC_PASSWORD: sonar
    depends_on:
      - db
    restart: unless-stopped

  db:
    container_name: sonarqube-db
    image: postgres:15
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
      POSTGRES_DB: sonar
    volumes:
      - sonarqube_db:/var/lib/postgresql/data
    restart: unless-stopped

volumes:
  jenkins_data:
  sonarqube_db:

```

Figura 4 - Docker-compose.yml do RestauranteFinal

- IV. Paramos o container original que estava a correr com o comando **docker stop restaurante-container**.
- V. Executamos o comando **docker-compose up -d** para iniciar os containers.
- VI. Feito, os containers agora estão a correr em simultâneo.

```

[+] Running 7/7
✔ Network restaurantefinal_default Created 0.1s
✔ Volume "restaurantefinal_jenkins_data" Created 0.1s
✔ Volume "restaurantefinal_sonarqube_db" Created 0.0s
✔ Container sonarqube-db Started 3.0s
✔ Container jenkinsRestaurante Started 3.0s
✔ Container sonarqubeRestaurante Started 2.7s
✔ Container restaurante-api Started 3.4s
> C:\Users\maki\Desktop\DevOps\RestauranteFinal>

```

Figura 5 - Containers do RestauranteFinal em execução

## 9. Pipeline automatizado com Jenkins

- I. Utilizamos o comando **docker-compose start** para iniciar o nosso container.
- II. Acedemos ao url <http://localhost:8081> para aceder ao Jenkins.
- III. Inserimos a password que foi apresentada nos logs do Jenkins .
- IV. Seleccionamos a opção “Install suggested plugins”.
- V. Seleccionamos a opção “Skip and continue as admin”.
- VI. Seleccionamos a opção “Save And Finish”.
- VII. Fomos ao menu da Gestão de Plugins que se encontra da seguinte forma: “Manage Jenkins -> Available Plugins”;
- VIII. Instalamos os seguintes plugins:
  - a. Pipeline;
  - b. Docker Pipeline;
  - c. SonarQube Scanner;
  - d. JUnit;
- IX. Acedemos aos containers através dos seguintes hosts:
  - <http://localhost:8081>

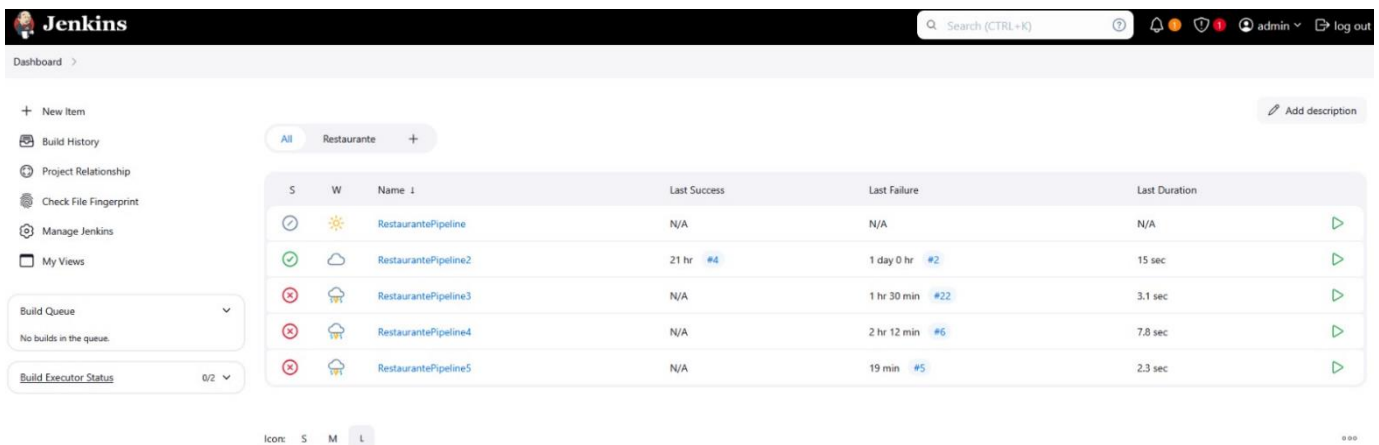


Figura 6 - Jenkins do RestauranteFinal

- <http://localhost:9000>

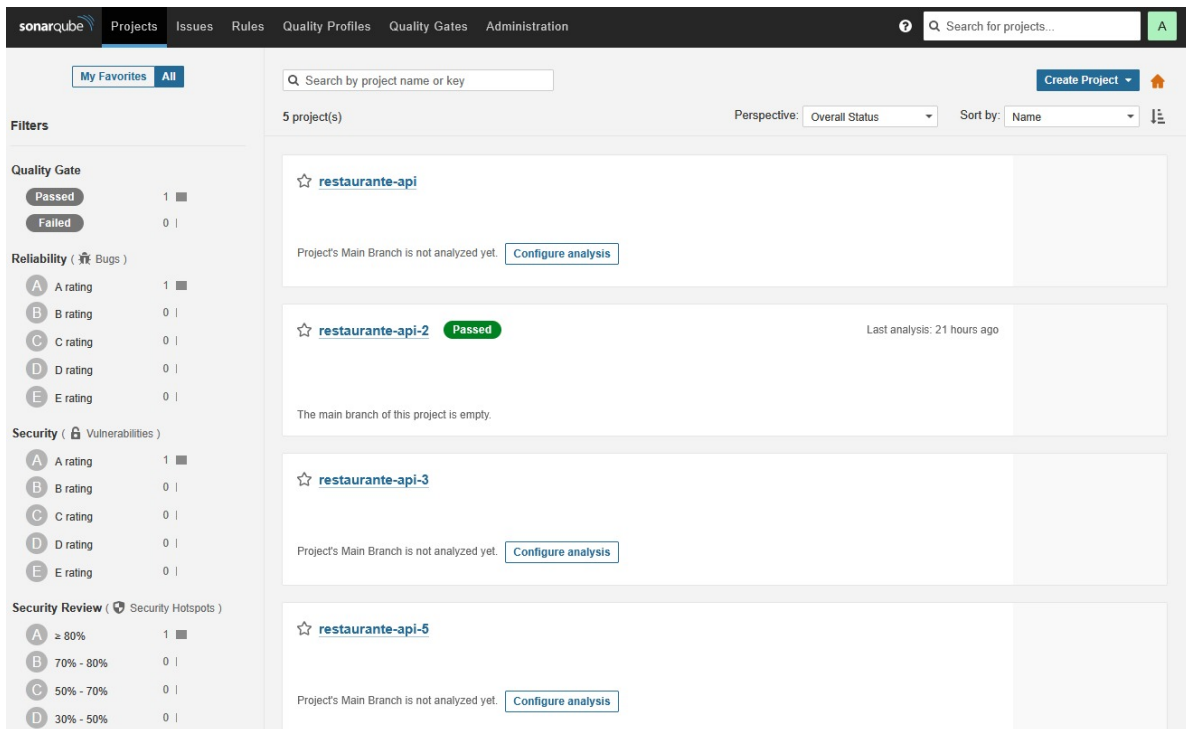


Figura 7 - Sonarqube do RestauranteFinal

- <http://localhost:8080/>

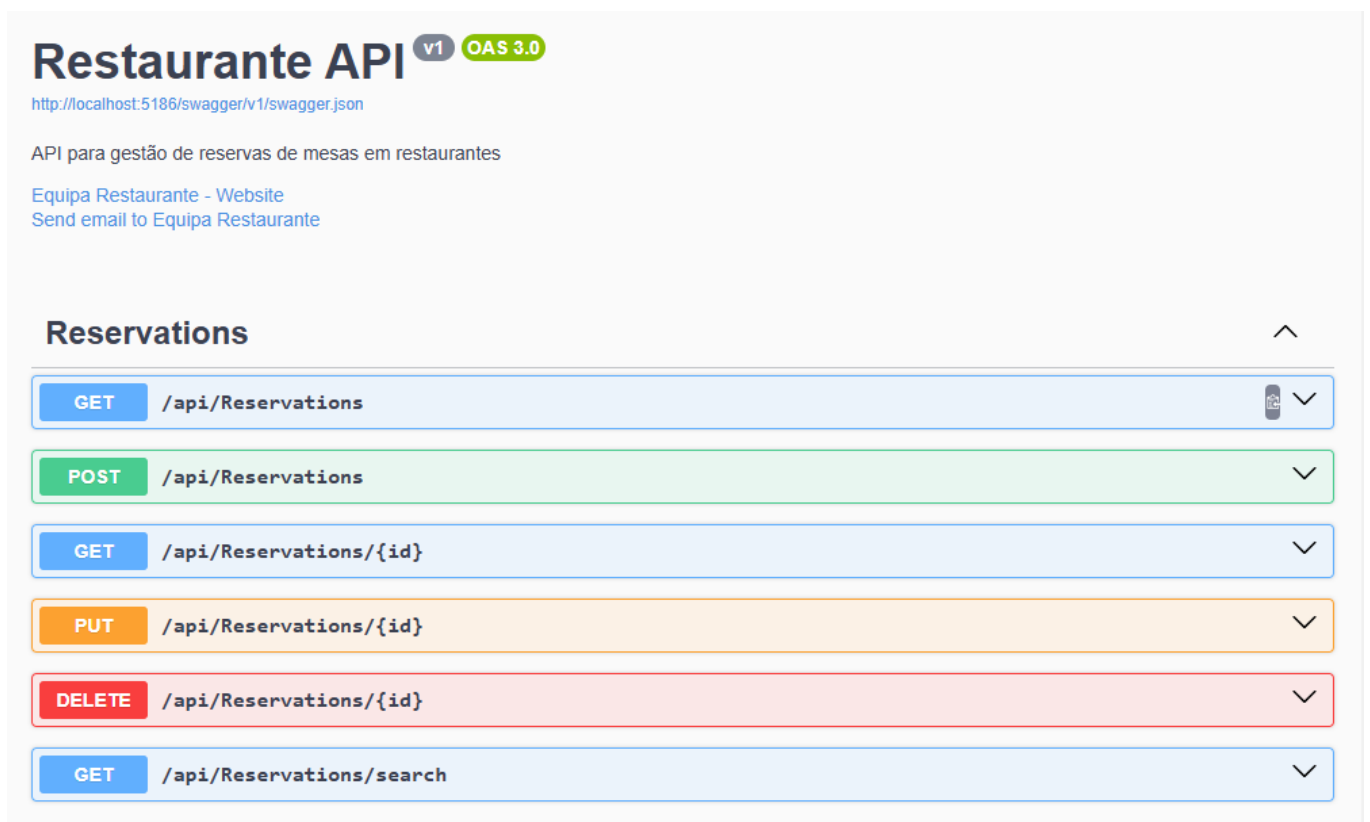


Figura 8 - Web API do RestauranteFinal

## Problemas relacionados com a automação da pipeline

Durante a implementação e execução da pipeline automatizada para o projeto, encontramos dificuldades que impediram a conclusão bem-sucedida do pipeline.

Inicialmente, enfrentamos problemas relacionados à configuração do *.NET SDK* no *Jenkins*. Embora o *.NET SDK 8* estivesse instalado no servidor, o comando *dotnet* não era encontrado durante a execução da pipeline. Tentamos configurar manualmente o *PATH* e remover a seção *tools* do *Jenkinsfile*, mas o problema persistiu.

Optamos por usar *Docker* para garantir um ambiente de build. Criamos um *Dockerfile* para construir a imagem *Docker*, executar testes e fazer o *deploy*. No entanto, ao tentar construir e executar a imagem *Docker* na pipeline, encontramos o problema de que o comando *docker* não estava disponível no ambiente *Jenkins*.

Tentamos tanto com scripts personalizados quanto configurando diretamente no *Jenkinsfile* para resolver os problemas encontrados. No entanto, apesar das várias tentativas, a pipeline automatizada continuou a falhar devido à indisponibilidade dos comandos *dotnet* e *docker* no *Jenkins*.

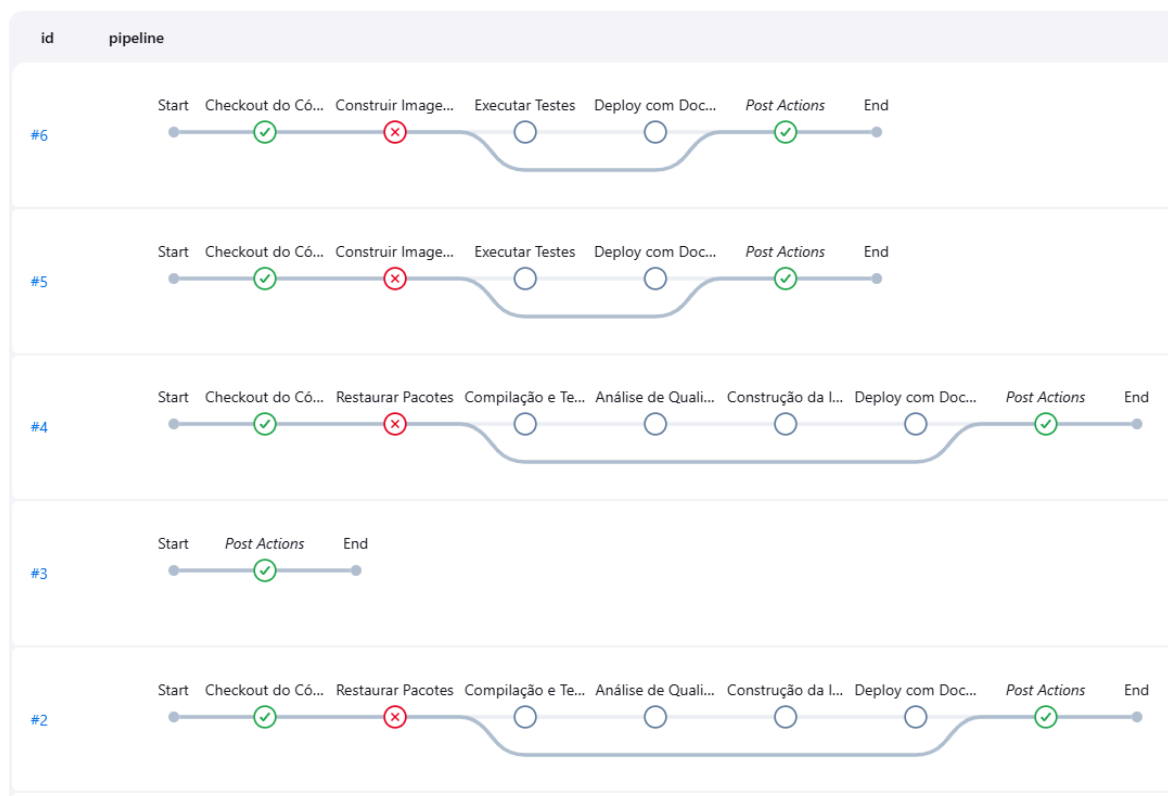


Figura 9 - Stages das pipelines falhadas

## Solução com Freestyle project

Diante das dificuldades encontradas, optamos por criar um projeto *FreeStyle* no *Jenkins*. Neste formato, conseguimos configurar manualmente cada etapa do processo de build e deploy, utilizando scripts personalizados para cada fase. Esta abordagem permitiu maior controle sobre o ambiente e garantiu que todos os comandos necessários fossem executados corretamente.

### I. Criamos um projeto no SonarQube

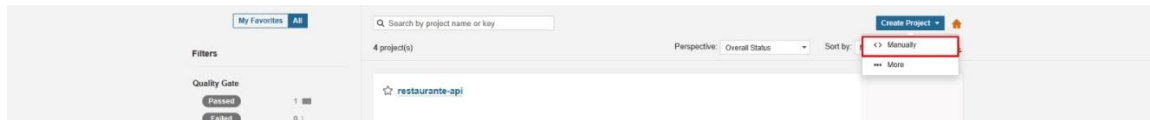


Figura 10 - Configuração SonarQube para FreeStyle Project

### II. Definimos o nome do projeto e a branch que o SonarQube iria analisar.

All fields marked with \* are required

**Project display name \***

   
Up to 255 characters. Some scanners might override the value you provide.

**Project key \***

 **This project key is already taken.**  
The project key is a unique identifier for your project. It may contain up to 400 characters. Allowed characters are alphanumeric, '-' (dash), '\_' (underscore), '.' (period) and ':' (colon), with at least one non-digit.

**Main branch name \***

  
The name of your project's default branch [Learn More](#)

**Set Up**

Figura 11 - Configuração SonarQube para FreeStyle Project

### III. Geramos um token no SonarQube que será utilizado na opção “Credentials” do Jenkins.

Analyze your project

We initialized your project on SonarQube, now it's up to you to launch analyses!

1 Provide a token

☒ Generate a project token

Token name Expires in

Analyze "restaurante-api-apaga2" No expira...

Please note that this token will only allow you to analyze the current project. If you want to use the same token to analyze multiple projects, you need to generate a global token in your user account. See the [documentation](#) for more information.

☐ Use existing token

The token is used to identify you when an analysis is performed. If it has been compromised, you can revoke it at any point in time in your [user account](#).

2 Run analysis on your project

Figura 12 - Configuração SonarQube para FreeStyle Project



IV. No Jenkins clicamos em “Add Credentials”.

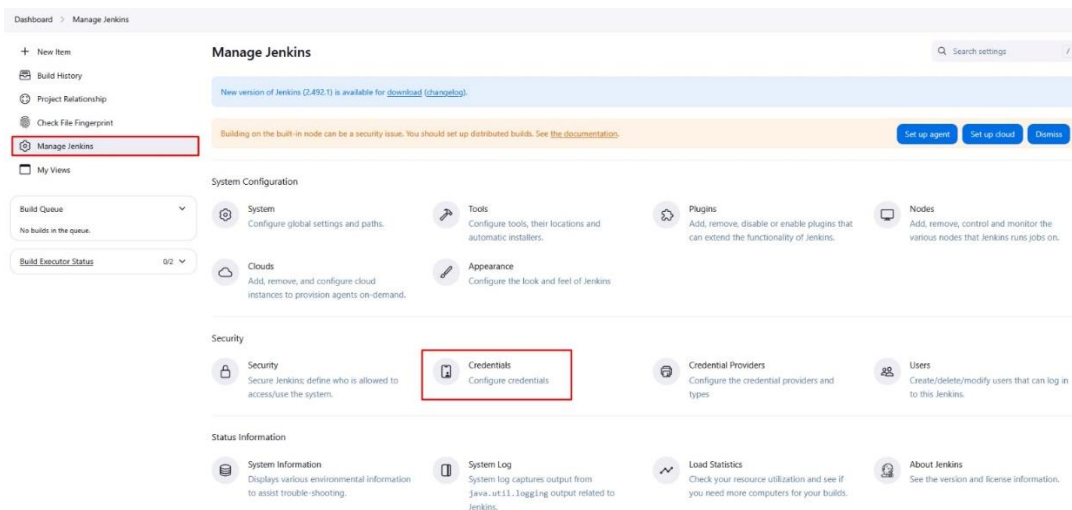


Figura 13 - Configuração Jenkins para FreeStyle Project

V. Configuramos da seguinte forma e inserimos o token gerado no SonarQube.

Figura 14 - Configuração Jenkins para FreeStyle Project

VI. Fomos até o caminho "Manage Jenkins" -> "System" -> Secção "SonarQube servers" e configuramos, salientando que no Server URL devemos colocar o IP onde o Docker está a ser executado:

Figura 15 - Configuração Jenkins para FreeStyle Project

- VII. No Jenkins clicamos em "New Item", demos um nome ao projeto e selecionamos o tipo "Freestyle Project".

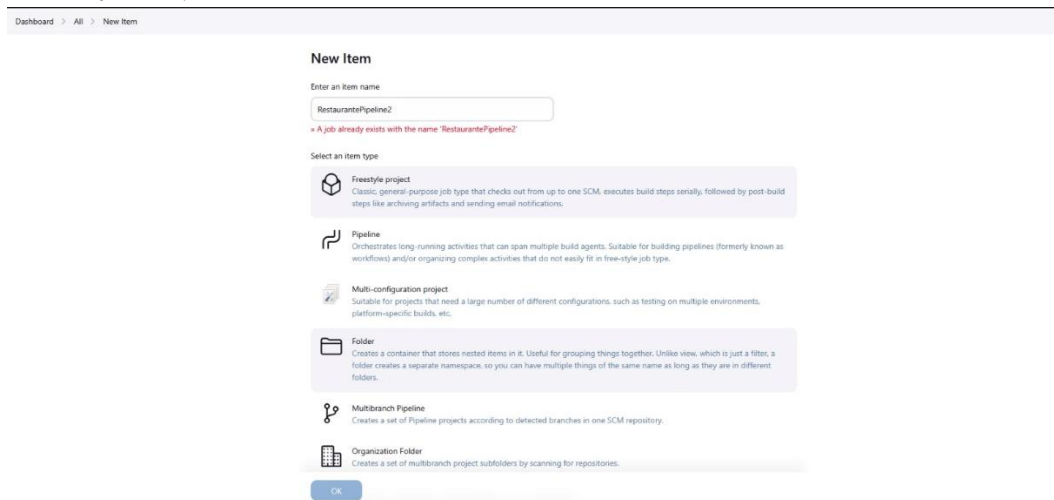


Figura 16 - Configuração Jenkins para FreeStyle Project

- VIII. Clicamos no botão "General" e configuramos da seguinte forma:

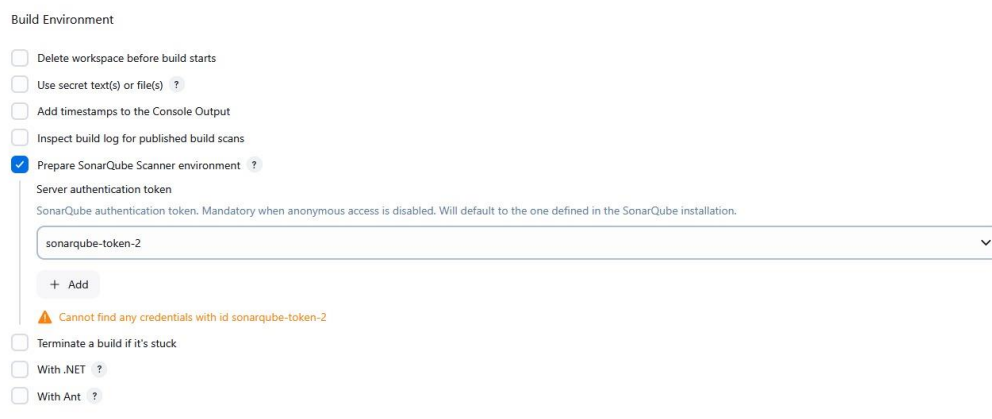


Figura 17 - Configuração Jenkins para FreeStyle Project

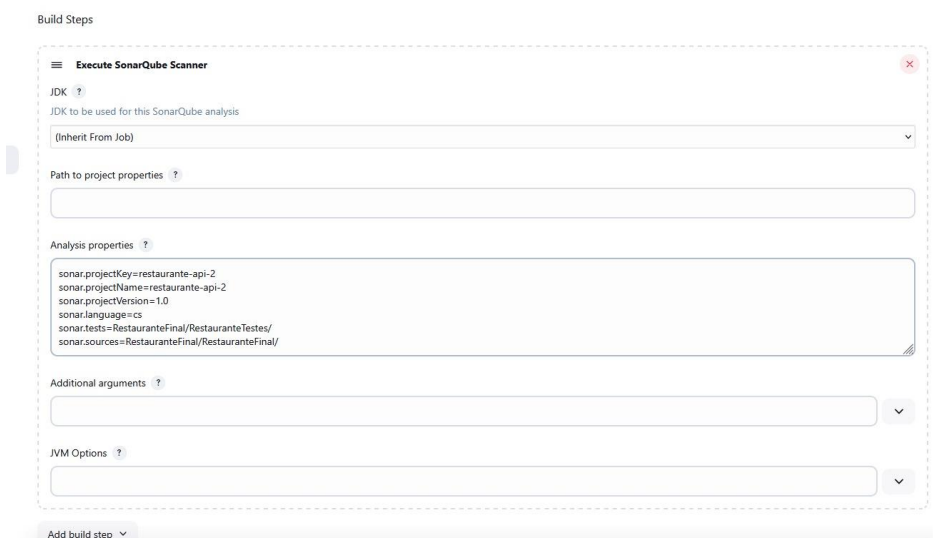


Figura 18 - Configuração Jenkins para FreeStyle Project

## IX. Clicamos em “Build Now”.

```
Dashboard > RestaurantePipeline2 > #4 > Console Output

22:24:06.014 INFO Sensor VB.NET Analysis Log [vbnet]
22:24:06.066 INFO Sensor VB.NET Analysis Log [vbnet] (done) | time=52ms
22:24:06.066 INFO Sensor VB.NET Properties [vbnet]
22:24:06.066 INFO Sensor VB.NET Properties [vbnet] (done) | time=0ms
22:24:06.066 INFO Sensor IAC Docker Sensor [iac]
22:24:06.075 INFO 0 source files to be analyzed
22:24:06.421 INFO 0/0 source files have been analyzed
22:24:06.421 INFO Sensor IAC Docker Sensor [iac] (done) | time=355ms
22:24:06.431 INFO ----- Run sensors on project
22:24:06.517 INFO Sensor C# [csharp]
22:24:06.517 WARN Your project contains C# files which cannot be analyzed with the scanner you are using. To analyze C# or VB.NET, you must use the SonarScanner for .NET 5.x or higher, see
https://redirect.sonarsource.com/doc/install-configure-scanner-msbuild.html
22:24:06.517 INFO Sensor C# [csharp] (done) | time=0ms
22:24:06.518 INFO Sensor Analysis Warnings import [csharp]
22:24:06.519 INFO Sensor Analysis Warnings import [csharp] (done) | time=0ms
22:24:06.519 INFO Sensor C# File Caching Sensor [csharp]
22:24:06.519 WARN Incremental PR analysis: Could not determine common base path, cache will not be computed. Consider setting 'sonar.projectBaseDir' property.
22:24:06.519 INFO Sensor C# File Caching Sensor [csharp] (done) | time=0ms
22:24:06.519 INFO Sensor Zero Coverage Sensor
22:24:06.525 INFO Sensor Zero Coverage Sensor (done) | time=0ms
22:24:06.528 INFO SCM Publisher SCM provider for this project is: git
22:24:06.538 INFO SCM Publisher 18 source files to be analyzed
22:24:06.741 INFO SCM Publisher 18/18 source files have been analyzed (done) | time=209ms
22:24:06.746 INFO CPD Executor Calculating CPD for 0 files
22:24:06.746 INFO CPD Executor CPD calculation finished (done) | time=0ms
22:24:06.836 INFO Analysis report generated in 81ms, dir size=144.1 kB
22:24:06.868 INFO Analysis report compressed in 24ms, zip size=29.2 kB
22:24:07.079 INFO Analysis report uploaded in 219ms
22:24:07.081 INFO ANALYSIS SUCCESSFUL, you can find the results at: http://192.168.1.163:9000/dashboard?id=restaurante-api-2
22:24:07.081 INFO Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
22:24:07.094 INFO More about the report processing at http://192.168.1.163:9000/api/ci/task?id=427npco07h4K5eRc311
22:24:07.095 INFO EXECUTION SUCCESS
22:24:07.095 INFO Total time: 12.969s
Finished: SUCCESS
```

Figura 19 - Resultado do build Freestyle Project

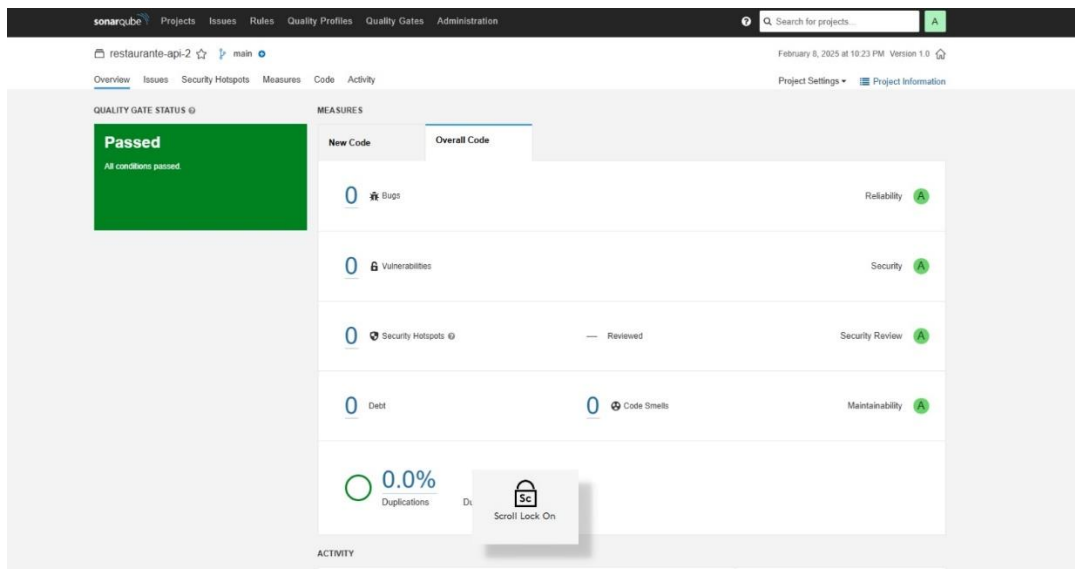


Figura 20 - Resultado do build Freestyle Project

## 11. Conclusão

O desenvolvimento da aplicação para gestão de reservas em restaurantes, no âmbito deste projeto, teve como objetivo implementar práticas de *DevOps*, com foco em automação, integração contínua e entrega contínua. A solução, baseada numa REST API, conseguiu atender à maioria dos requisitos estabelecidos, incluindo operações CRUD, validação de conflitos nas reservas e integração com uma base de dados SQL Server configurada na máquina virtual Vagrant.

Embora não tenhamos alcançado a automação total desejada, a experiência proporcionou uma valiosa lição sobre a configuração de ambientes e ferramentas de CI/CD.

A metodologia de trabalho adotada, em que todos os membros da equipa participaram nas diversas tarefas foi essencial para superar os desafios dentro do prazo disponível, demonstrando a importância da colaboração e da adaptação em projetos de software. A utilização do *GitHub* para controlo de versões, bem como a execução de testes unitários, permitiu um desenvolvimento onde garante que a aplicação atende aos critérios definidos.

A utilização de tecnologia como *Vagrant* e *Docker*, trouxe desafios de configuração, que foram superados. No entanto, devido ao tempo limitado, não conseguimos explorar outras ferramentas além dessas, mas gostaríamos de ter tido a oportunidade, o que poderia trazer melhorias ao projeto.

Este projeto contribuiu significativamente para o desenvolvimento de competências técnicas e de gestão de projetos, além de permitir a aplicação prática de conceitos aprendidos durante o semestre.

## **12. Referências**

- <https://chatgpt.com/> , utilizado como ferramenta de melhoramento de texto;