INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

João Pedro Almeida Costa (1231541)

Roberto Oliveira Valente (1231555)

Samuel Oliveira Lemos (1231557)

Dinis Pinto Faryna (1231530)

Pedro Sousa Barbosa (1231553)

Desenvolvimento e Automação de uma API REST para Gestão de Reservas de Restaurantes

(Implementação de Práticas DevOps com CI/CD, Testes Automatizados e Containerização)

Desenvolvimento / Operação de Software

Santa Maria da Feira

2025

Índice

Conteúdo

Texto contém excertos revistos por inteligência artificial para maior precisão.

Introdução

Este projeto final, desenvolvido no âmbito da unidade curricular de DevOps no ISEP, representa 30% da nota final e tem como principal propósito a aplicação prática dos conceitos abordados ao longo do semestre. O trabalho proposto consiste no desenvolvimento de uma Web API REST para a gestão de reservas de mesas em restaurantes, garantindo a implementação de boas práticas de automação, qualidade de software e containerização.

Para alcançar esse objetivo, a API foi desenvolvida utilizando .NET Core, com base de dados em SQL Server, suportado por virtualização com Vagrant e containerização com Docker. O código-fonte segue um rigoroso controle de versões no GitHub, utilizando os branches master e develop, e foi validado através de testes unitários com cobertura mínima de 80%.

Além disso, a solução foi integrada em um pipeline automatizado no Jenkins, garantindo build, testes, análise de qualidade de código com SonarQube e deployment automatizado em containers. A documentação foi gerada automaticamente com Swagger, permitindo a visualização detalhada dos endpoints e suas funcionalidades.

Objetivo

O principal objetivo deste projeto é desenvolver uma solução de software robusta e pronta para produção, aplicando práticas modernas de DevOps e automação de deploy. Especificamente, busca-se:

- Criar uma API REST funcional e eficiente para a gestão de reservas de mesas em restaurantes.
- Garantir qualidade do código por meio de testes unitários e análise estática.
- Implementar integração e entrega contínuas (CI/CD) com Jenkins e Docker.
- Aplicar conceitos de virtualização e containerização para tornar a solução flexível e escalável.
- Documentar de forma clara todo o processo de desenvolvimento, incluindo desafios enfrentados e melhorias futuras.

Este relatório detalha cada uma das etapas do desenvolvimento, justificando as escolhas técnicas e apresentando os resultados obtidos.

Desenvolvimento do Projeto

Criação do projeto e GitHub

O primeiro passo para o desenvolvimento do projeto foi a sua criação e configuração em um repositório no GitHub. Essa abordagem permitiu um controlo de versões eficiente, facilitando a colaboração entre os membros da equipa e garantindo um histórico claro de todas as alterações realizadas no código.

Para iniciar, foi criado um repositório seguindo as instruções de nomenclatura definidas no projeto:

Ctesp2425-final-gX[E/F], onde X representa o número do grupo e E/F indica a localização da instituição (Ermesinde ou Santa Maria da Feira), sendo assim ficou designado como "ctesp24-25-final-gAf".

O repositório foi configurado com as seguintes boas práticas:

- Branches principais:

- master: versão estável do projeto, utilizada para entrega final.

- develop: branch de desenvolvimento onde novas funcionalidades foram implementadas antes do merge na master.

- Organização do código:

- Estrutura inicial do projeto foi gerada utilizando .NET Core.

- Arquitetura baseada em camadas, separando responsabilidades para maior manutenibilidade.

- Gestão de commits e versionamento:

- Foram seguidas as boas práticas de commits, utilizando mensagens descritivas e padronizadas.

- Cada funcionalidade foi desenvolvida em branches individuais, sendo revisadas antes da integração na develop.

Além disso, foi garantido que todos os membros do grupo tivessem permissões adequadas no repositório, e o professor responsável foi adicionado como administrador para acompanhamento e avaliação do trabalho.

Com esta estrutura, o projeto ficou preparado para a implementação dos requisitos definidos, garantindo um desenvolvimento colaborativo, seguro e bem documentado.

Models

Na fase seguinte, foi criada a pasta Models, onde se encontra a classe Reservation. Este modelo foi desenvolvido com os atributos necessários para atender aos requisitos do sistema de reservas.

A classe Reservation inclui os seguintes atributos:

- Id (int) – Identificador único da reserva.
- CustomerName (nvarchar) – Nome do cliente que efetuou a reserva.
- ReservationDate (date) – Data da reserva.
- ReservationTime (time) – Hora da reserva.
- TableNumber (int) – Número da mesa reservada.
- NumberOfPeople (int) – Número de pessoas na reserva.
- CreatedAt (datetime) – Data e hora em que a reserva foi criada.

Este modelo garante que todas as informações essenciais para a gestão das reservas sejam armazenadas de forma estruturada e eficiente.

DbContext

Após a criação do model, avançámos para quesito do desenvolvimento dos endpoints, contudo para implementar os endpoints previstos, foi necessário, em primeiro lugar, estabelecer uma ligação a uma base de dados, com o intuito de garantir o armazenamento e a gestão dos dados manipulados por essas funcionalidades.

Começamos por configurar o DbContext

```
11 referências | Roberto Valente, há 3 dias | 1 autor, 1 alteração
public class AppDbContext : DbContext
{
    1 referência | Roberto Valente, há 3 dias | 1 autor, 1 alteração
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
    14 referências | Roberto Valente, há 3 dias | 1 autor, 1 alteração
    public DbSet<Reservation> Reservations { get; set; }

    0 referências | Roberto Valente, há 3 dias | 1 autor, 1 alteração
    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        modelBuilder.Entity<Reservation>()
            .ToTable("dsosReservation");

        modelBuilder.Entity<Reservation>()
            .Property(r => r.CreatedAt)
            .HasColumnType("datetime2");
    }
}
```

O código apresentado define e configura um contexto de base de dados com Entity Framework Core. A classe AppDbContext gerência a persistência dos dados da entidade Reservation e mapeia a tabela correspondente como "dsosReservation". Além disso, o campo CreatedAt é explicitamente definido como datetime2, garantindo maior precisão para armazenar registos temporais.

Essa abordagem permite que a aplicação interaja de maneira eficiente com a base de dados, seguindo boas práticas de ORM (Object-Relational Mapping) e garantindo flexibilidade na configuração das entidades.

Para complementar todo este processo precisamos de fazer o builder para gestão da execução.

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<AppDbContext>(options => options.UseSqlServer(
    connectionString
    , sqlOptions => sqlOptions.EnableRetryOnFailure(
              maxRetryCount: 5,
              maxRetryDelay: TimeSpan.FromSeconds(10),
              errorNumbersToAdd: null
          )
    )
);
```

A configuração apresentada complementa o contexto AppDbContext, permitindo que a aplicação se conecte à base de dados SQL Server de maneira eficiente e segura. Além disso, a resiliência da conexão garante que falhas momentâneas não impactem a disponibilidade da aplicação, tornando-a mais robusta.

Para que toda a configuração funcione corretamente, foi necessário definir a connection string no arquivo appsettings.json, garantindo que a aplicação possa localizar e acessar a base de dados de forma dinâmica e configurável.

```
"ConnectionStrings": {
  "DefaultConnection": "Server=LOCALHOST,1433;Database=DosBDDocker;User Id=sa;Password=Password@123;TrustServerCertificate=True;"
}
```

Nesta fase do projeto, a criação da base de dados e das respetivas tabelas foi realizada manualmente, a fim de agilizar o processo de desenvolvimento. Embora o Entity Framework Core permita a geração automática da base de dados por meio de migrações, optou-se pela criação manual para maior controle e rapidez na implementação.

Controllers

O controller está definido na classe ReservationsController, que herda de ControllerBase e tem a anotação [ApiController], garante assim que é tratado como um controlador de API. Está configurado para responder a pedidos na rota "api/reservations".

O controller depende do AppDbContext, que representa o contexto da base de dados. A instância deste contexto é injetada através do construtor, permitindo a interação com a base de dados.

1) GetReservations (DateOnly? date) [Get]

   Objetivo: Retorna todas as reservas ativas (StatusReservation == 0). Se um parâmetro opcional date for passado, retorna apenas as reservas para essa data específica.

   Parâmetro da query: date (opcional, DateOnly), filtra as reservas pela data informada.

   Fluxo:

   - Se date for fornecido, busca apenas as reservas dessa data.

- Caso contrário, retorna todas as reservas ativas.
- Retorna 200 OK com a lista de reservas.

2) GetReservation(int id) [Get]

Objetivo: Retorna os detalhes de uma reserva ativa com base no id fornecido.

Parâmetro da query: id (obrigatório, int),identificador da reserva.

Fluxo:

- Pesquisa a reserva através do id, filtrando apenas as reservas ativas (StatusReservation == 0).
- Se a reserva não existir, retorna 404 Not Found.
- Caso contrário, retorna 200 OK com os detalhes da reserva.

3) CreateReservation([Required] String customerName, [Required] DateOnly resDate, [Required] TimeOnly resTime, [Required] int tableNumber, [Required] int numOfPeople) [Post]

   Objectivo: Cria uma nova reserva, validando se a data é futura e se a mesa está disponível.

   Parâmetros do corpo (formato application/json):
   - customerName (obrigatório, string): Nome do cliente.
   - resDate (obrigatório, DateOnly): Data da reserva.
   - resTime (obrigatório, TimeOnly): Hora da reserva.
   - tableNumber (obrigatório, int): Número da mesa.
   - numOfPeople (obrigatório, int): Quantidade de pessoas na reserva.

   Fluxo:

   - Concatena resDate e resTime para criar um DateTime.
   - Se a data/hora for no passado, retorna 400 Bad Request.
   - Verifica se a mesa já está reservada a essa hora. Se sim, retorna 400 Bad Request.
   - Cria uma nova reserva e guarda-a no banco.
   - Retorna 200 OK com os detalhes da nova reserva.

4) UpdateReservation(int id, String? customerName, DateOnly? resDate, TimeOnly? resTime, int? tableNumber, int? numOfPeople) [Put]

   Objectivo: Atualiza os detalhes de uma reserva ativa. Pelo menos um campo deve ser enviado para atualização.

   Parâmetro do corpo:
   - id (obrigatório, int): Identificador da reserva.
   - Parâmetros do corpo (formato application/json, opcionais):

- customerName (string): Nome do cliente.
- resDate (DateOnly): Nova data da reserva.
- resTime (TimeOnly): Novo horário da reserva.
- tableNumber (int): Novo número da mesa.
- numOfPeople (int): Nova quantidade de pessoas.

Fluxo:

- Se não for enviado nenhum campo, retorna 400 Bad Request.
- Procura a reserva pelo id e verifica se está ativa. Se não existir, retorna 404 Not Found.
- Se resDate e/ou resTime forem alterados, verifica se a nova data/hora é futura.
- Se tableNumber for alterado, verifica se a mesa já está reservada para a nova data/hora.
- Atualiza os campos introduzidos.
- Guarda as alterações no banco.
- Retorna 200 OK com os dados atualizados.

5) DeleteReservation(int id) [Delete]

Objectivo: "Apaga" uma reserva marcando-a como inactiva (StatusReservation = 1).

Parâmetro da query: id (obrigatório, int), identificador da reserva.

Fluxo:

- Procura a reserva ativa pelo id.
- Se não o encontrar, retorna 404 Not Found.
- Define StatusReservation = 1 para indicar a eliminação lógica.
- Guarda a alteração no banco.
- Retorna 200 OK com os detalhes da reserva agora inativa.

Testes Unitários (XUnit)

Este tópico tem como objetivo descrever e explicar os testes unitários implementados para a classe ReservationsControllerTests. Os testes utilizam a biblioteca XUnit e visam garantir o correto funcionamento das funcionalidades presentes no controller de reservas (ReservationsController). Foi criado um projeto em XUnit para a implementação dos mesmos.

Antes de executar os testes, um contexto de base de dados em memória é criado utilizando UseInMemoryDatabase(Guid.NewGuid().ToString()), garantindo uma nova base de dados isolada para cada teste. O método CreateContext() inicializa o contexto e insere a base de dados com reservas fictícias para validar os casos de teste.

1) GetReservations_WithoutDate_ReturnsActiveReservations

Objetivo: Este teste verifica se a listagem de reservas sem uma data específica retorna apenas as reservas ativas (StatusReservation == 0).

Passos:

- Chama GetReservations(null).
- Verifica se o resultado é um OkObjectResult.
- Verifica se a resposta é uma lista de reservas.
- Garante que todas as reservas retornadas possuem StatusReservation == 0.
- Verifica se o número de reservas retornadas é exatamente 3.

2) GetReservations_WithSpecificDate_ReturnsCorrectReservations
Objetivo: Este teste verifica se ao passar uma data específica, apenas as reservas dessa data são retornadas corretamente.

Passos:
- Chama GetReservations(new DateOnly(2024, 1, 17)).
- Verifica se o resultado é OkObjectResult.
- Confirma que a lista retornada não é nula.
- Garante que a reserva retornada possui o nome "Dinis Faryna".

3) GetReservation_ExistingActiveReservation_ReturnsReservation
Objetivo: Valida se uma reserva existente e ativa pode ser recuperada corretamente pelo seu ID.

Passos:
- Chama GetReservation(1).
- Verifica se o resultado é OkObjectResult.
- Confirma que o estado da reserva devolvida é 0 (ativa).
- Verifica se o nome da reserva é "João Costa".

4) GetReservation_NonExistentReservation_ReturnsNotFound
Objetivo: Garante que um pedido por um ID inexistente retorna NotFound.

Passos:
- Chama GetReservation(999).
- Verifica se o resultado é um NotFoundResult.

5) CreateReservation_ValidData_ReturnsOkAndStoresCorrectly
Objetivo: Verifica se uma nova reserva é criada corretamente e guardada na base de dados.

Passos:
- Chama-se CreateReservation("João Silva", 10/08/2025, 18:30, 5, 4).
- Verifica se o retorno é OkObjectResult.
- Obtém a reserva criada pelo ID gerado.
- Compara os dados armazenados com os dados enviados para garantir a correta persistência.

6) CreateReservation_PastDateTime_ReturnsBadRequest
Objetivo: Garante que não é possível criar uma reserva com data no passado.

Passos:

- Chama CreateReservation("Past Reservation", 01/01/2023, 12:00, 6, 2).
- Verifica se o resultado é BadRequestObjectResult.

7) UpdateReservation_PartialUpdate_UpdatesSuccessfully
Objetivo: Valida se é possível atualizar corretamente uma reserva existente.

Passos:

- Chama UpdateReservation(1, "Updated Name", 10/08/2025, 18:30, 7, 3).
- Verifica se o retorno é OkObjectResult.
- Obtém a reserva atualizada e verifica se os valores foram alterados corretamente.

8) DeleteReservation_ActiveReservation_SetsStatusToCancelled
Objetivo: Garante que ao eliminar uma reserva ativa, o seu estado é alterado para "Cancelado" (StatusReservation = 1).

Passos:

- Chama DeleteReservation(1).
- Verifica se o retorno é OkObjectResult.
- Confirma se o StatusReservation da reserva foi alterado para 1.

Conclusão
Os testes unitários descritos validam o comportamento esperado do ReservationsController. Garantem que:

- A listagem de reservas apenas devolve reservas ativas.
- A pesquisa por data retorna reservas corretamente.
- A recuperação de reservas pelo ID funciona corretamente.
- A criação de reservas com dados válidos ocorre corretamente.
- A criação de reservas com datas passadas é impedida.
- As atualizações de reservas são aplicadas corretamente.
- A eliminação de reservas altera o estado em vez de as remover.

Migration (Code First)

No projeto, a base de dados SQL Server será criada utilizando Code First do Entity Framework Core. As migrations (migrações) permitem definir e aplicar alterações na estrutura de base de dados diretamente a partir do código, sem necessidade de modificações manuais.

Para este passo precisamos de instalar o Entity Framework Core e a Ferramenta de Migrations, sendo essas ferramentas:

- Microsoft.EntityFrameworkCore (O núcleo do EF Core).
- Microsoft.EntityFrameworkCore.SqlServer (Suporte ao SQL Server).
- Microsoft.EntityFrameworkCore.Tools (Ferramentas CLI para executar migrations).

Além disso, configuramos o DbContext e definimos a Connection String, que permite à aplicação comunicar-se com o SQL Server.

Com essas configurações prontas, podemos gerar uma migration com o comando "dotnet ef migrations add [NomeDaMigration]". Isso criará uma pasta Migrations/ no projeto, contendo um arquivo C# com as instruções para criar ou modificar a base de dados.

Para aplicar essa migration e criar as tabelas no SQL Server, utilizamos "dotnet ef database update".

Para complementar esse processo e garantir que as migrations sejam aplicadas automaticamente ao iniciar a aplicação, adicionamos o seguinte código no Program.cs:

```
// Criar ou aplicar as migra◆◆es automaticamente ao rodar a aplica◆◆o
using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    dbContext.Database.Migrate();
}
```

Esse código garante que, ao iniciar a aplicação, qualquer migration pendente será aplicada automaticamente. No entanto, em ambientes de produção, pode ser mais seguro executar as migrations manualmente para evitar atualizações inesperadas na estrutura da base de dados.

Vagrant

O primeiro passo para utilizar o Vagrant é garantir que tanto o Vagrant quanto o VirtualBox (escolha da nossa máquina virtual) estejam instalados corretamente.

Após isso, criamos o nosso arquivo de configuração, chamado Vagrantfile, e realizamos a configuração das seguintes opções:

- Definição da Box (Imagem Base): Escolhemos uma imagem base que já contém o SQL Server pré-instalado. Também definimos a versão específica dessa imagem para garantir a compatibilidade com o ambiente de trabalho.
- Configuração de Portas: Realizamos o redirecionamento de portas, permitindo que a aplicação acesse o SQL Server dentro da VM. Neste caso, a porta 1433 foi configurada para permitir a comunicação entre a aplicação e a base de dados.
- Definição dos Recursos da VM: Especificamos que a máquina virtual será criada no VirtualBox. Para garantir um desempenho adequado, alocamos 2 GB de RAM e 2

núcleos de CPU à VM, o que é suficiente para rodar o SQL Server de forma eficiente.

```
Vagrant.configure("2") do |config|
  config.vm.box = "gusztavvargadr/sql-server"
  config.vm.box_version = "2019.2102.2409"

  config.vm.network "forwarded_port", guest: 1433, host: 1433

  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = 2
  end
end
```

Após a configuração, utilizamos o comando vagrant up para inicializar a VM. Esse comando realiza o download da imagem do SQL Server (caso ela não esteja no cache).

Para testar se a implementação foi realizada com sucesso, podemos acessar a máquina virtual e tentar conectar à base de dados usando as credenciais definidas na connectionString:

- Server = localhost,1433: Conecta-se à porta 1433, configurada no redirecionamento da VM.
- Database = DosBDDocker: Nome do banco de dados que está sendo utilizado.
- User Id = sa; Password = Password@123: Credenciais de acesso ao SQL Server dentro da VM.

Docker & Docker-Compose

Após a configuração do Vagrant e a instalação do ambiente de virtualização, a fase seguinte envolve a configuração e utilização do Docker para a criação e orquestração de containers que serão responsáveis pelos serviços da aplicação. O Docker Compose é utilizado para gerir múltiplos contentores em simultâneo, facilitando a execução do ambiente completo. Para tal, é necessário ter dois ficheiros principais: o Dockerfile e o docker-compose.yml.

Dockerfile: Construção da Imagem Docker

O Dockerfile define as etapas necessárias para construir a imagem Docker que será utilizada para rodar a aplicação. Este arquivo é dividido em duas fases: a fase de build, onde a aplicação é compilada e publicada, e a fase de execução, onde a aplicação é executada no container.

- Definição da Imagem de Build: A imagem base mcr.microsoft.com/dotnet/sdk:8.0 é utilizada para compilar a aplicação.
- Fase de Execução: Na fase de execução é utilizada a imagem mcr.microsoft.com/dotnet/aspnet:8.0, uma vez que está optimizada para correr aplicações ASP.NET, sem as ferramentas de compilação.

Durante a construção da imagem, o Docker executa os seguintes passos:

- Restaurar dependências: O comando dotnet restore descarrega as dependências do projeto.

- Publicação da aplicação: O comando dotnet publish -c Release cria a versão otimizada para a produção da aplicação, gerando um diretório /out com todos os ficheiros necessários para a execução.

Após a publicação, a imagem da aplicação está pronta para ser executada. Utilizamos o comando "docker build -t [Nome da Imagem] ." para construir a imagem. Este passo só é possível caso o Docker Desktop esteja aberto e em funcionamento. Após verificar se o Docker está ativo, podemos também confirmar se o Swagger está a funcionar ao acessar a aplicação no navegador.

docker-compose.yml: Orquestração de Contentores

Uma vez criada a imagem Docker a partir do Dockerfile, o passo seguinte é a orquestração e gestão de múltiplos contentores utilizando o Docker Compose. O ficheiro docker-compose.yml especifica todos os serviços necessários para correr a aplicação, incluindo a aplicação principal (web [dockerfile]), a base de dados SQL Server (db), bem como ferramentas como o Portainer, Jenkins e SonarQube para gestão e integração contínua.

- Serviço web: Define o serviço principal da aplicação, que será construída a partir do Dockerfile. Mapeia a porta 8080 do contentor para a porta 8050 do host e depende do serviço de base de dados (db), que deve estar em execução antes do serviço web ser iniciado.
- Serviço db: Utiliza a imagem oficial do SQL Server da Microsoft (mcr.microsoft.com/mssql/server). Mapeia a porta 1433 para permitir a comunicação com a base de dados e utiliza um volume db_data para persistir os dados do SQL Server.
- Serviço portainer: Fornece uma interface gráfica para a gestão dos contentores Docker. Mapeia a porta 9000 e partilha o socket Docker, permitindo ao Portainer controlar os contentores em execução.
- Serviço jenkins: Define o Jenkins para integração contínua, mapeando a porta 8080 para o host e utilizando o volume jenkins_data para persistir os dados de configuração e históricos de builds.
- Serviço sonarqube: Define o SonarQube, uma ferramenta para análise da qualidade de código. Mapeia a porta 9001 para o host e monta volumes para persistir os dados, extensões e registos do SonarQube.

Após esta configuração do ficheiro Docker-Compose.yml podemos executar o comando "docker-compose up", desta forma será contruído e iniciado os respetivos containers no Docker-Desktop.

Após executar o comando, os serviços estarão em funcionamento e podem ser acessadas as respetivas interfaces e aplicações:

- A aplicação web é acessível em http://localhost:8050.
- O Portainer é acessível em http://localhost:9001, oferecendo uma interface para gerenciar os containers.
- O Jenkins estará disponível em http://localhost:8080.
- O SonarQube pode ser acessado em http://localhost:9000.

Vale lembrar que o código presente no Program.cs relacionado à migração automatizada garantirá que, ao executar o docker-compose, a base de dados seja criada automaticamente, caso ainda não exista.

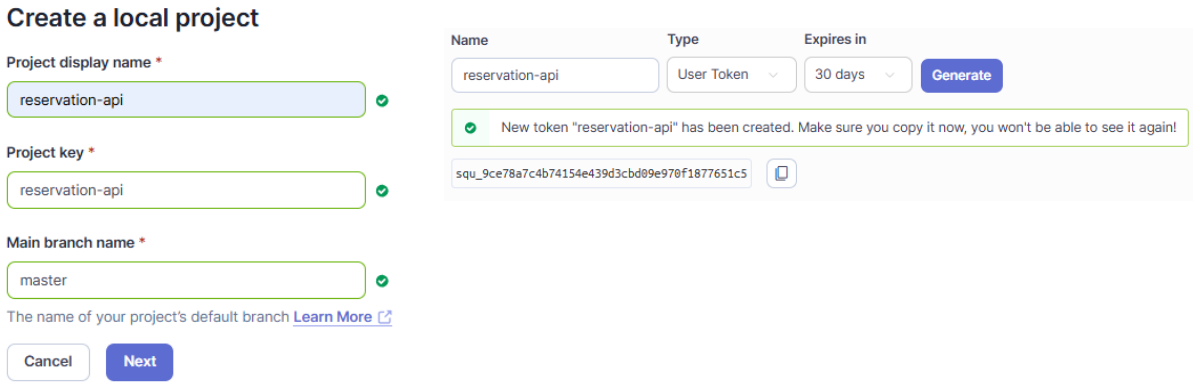
Pipeline (Jenkins & SonarQube)

A etapa final do projeto consiste na utilização do Jenkins para criar um pipeline juntamento com o sonarqube que tem como finalidade analisar a qualidade do código e fazer comentários automáticos.

Começamos então por acessar o nosso Docker-Desktop e daí abrimos os urls correspondentes ao Jenkins e SonarQube para começarmos as configurações de cada um.

Configuração do SonarQube

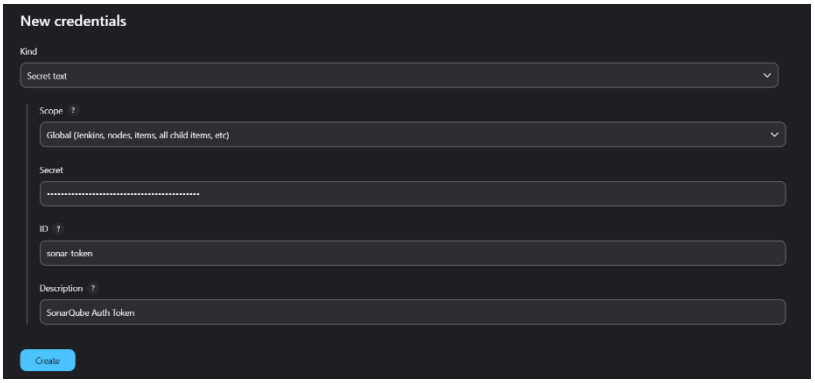
- Acessamos o servidor SonarQube através do navegador (localhost:9000).
- Fazemos Login e criamos um novo projeto "Create Project".
- Colocamos o nome "reservation-api" e definimos a branch "master". Depois de clicar em "Next", escolhemos "Use the global setting" e por fim "Create Project".
- Após isso temos de gerar um token, esse mesmo token é gerado no "My Account – Security" e preenchemos o nome, type e "expires". O token gerado deve ser alterado no ficheiro Jenkinsfile, variável "SONAR_TOKEN".



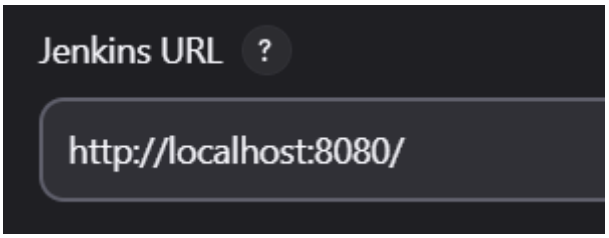
Configuração do jenkins

- Acessamos o servidor Jenkins através do navegador (localhost:8080).
- Fazemos login e instalamos os plugins necessários.
  - Docker Pipeline
  - SonarQube Scanner
  - .NET SDK Support
  - Pipeline Utility Steps

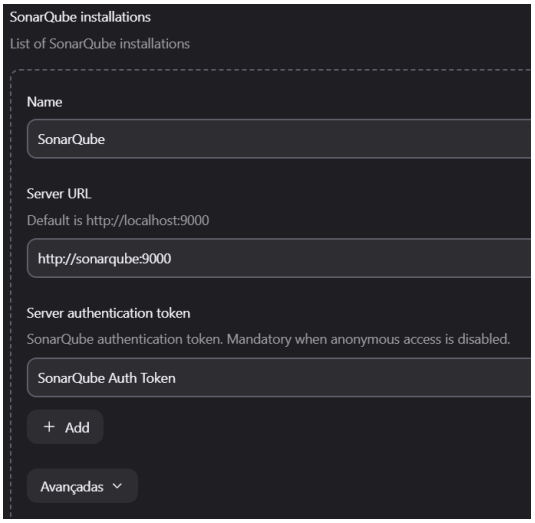
- Acessamos o credentials.
  - Permite configurar as credenciais necessárias para autenticação com o SonarQube.
  - O token secreto é fundamental para que o Jenkins possa se comunicar de forma segura com o servidor SonarQube e enviar os resultados da análise.



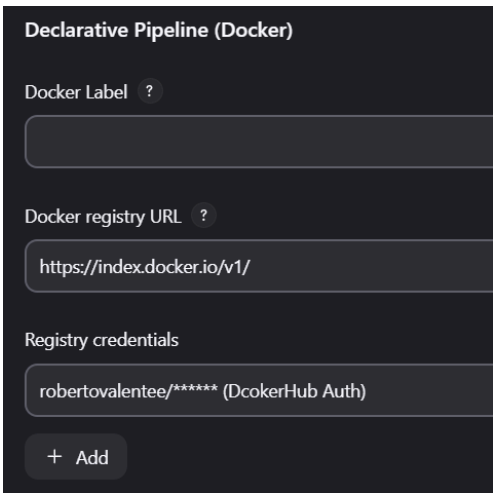
- Acessamos o System
  - Jenkins URL
    - Verificamos se o Jenkins URL está correto.



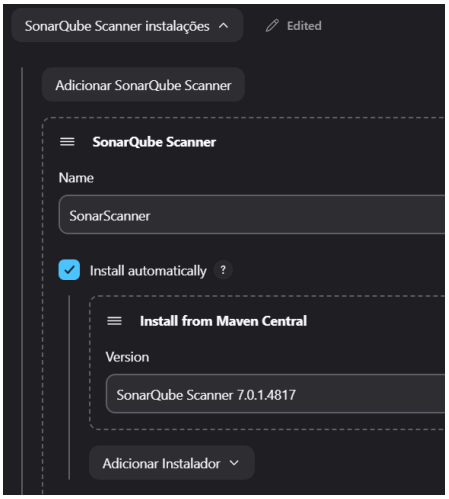
  - SonarQube Installations
    - Define a conexão com o servidor SonarQube (http://sonarqube:9000).
    - O token de autenticação é necessário para que o Jenkins possa se comunicar com o SonarQube (Criado me credentials).
    - Esta configuração permite que o Jenkins saiba onde enviar os resultados da análise de código.



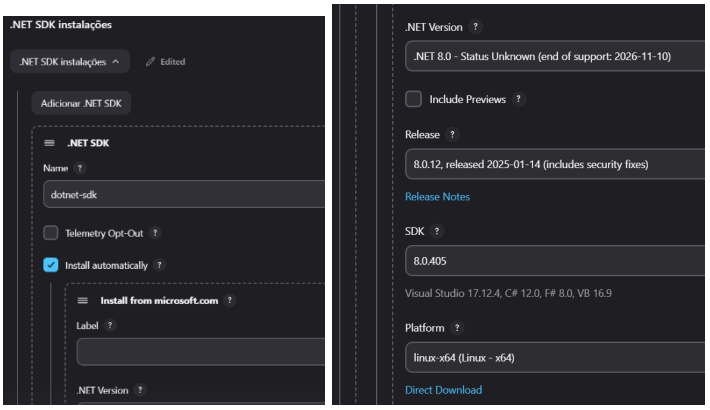
- Declarative Pipeline (Docker)
  - Configuração do registro Docker onde as imagens serão armazenadas.
  - Importante para garantir que o ambiente de build seja consistente e isolado.
  - As credenciais do Docker Hub permitem puxar imagens necessárias para o processo de build.



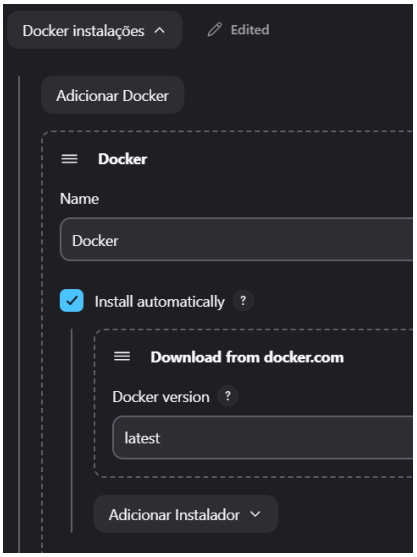
- Acessamos Tools
  - SonarQube Scanner
    - Instalação automática do scanner do SonarQube, que é a ferramenta que efetivamente realiza a análise do código.
    - A versão específica (7.0.1.4817) garante compatibilidade com o servidor SonarQube



- .NET SDK Instalações
  - Configuração do ambiente .NET necessário para compilar o projeto.
  - Define a versão específica do SDK .NET (8.0).
  - Inclui configurações específicas para Linux x64
    - Quanto à plataforma, a decisão dependerá da arquitetura do Node utilizada pelo Jenkins. Para verificar, acesse o dashboard do Jenkins e clique em "Nodes".
  - Necessário para que o Jenkins possa compilar e analisar projetos .NET.



- Docker Instalações
  - Configuração para instalação automática do Docker.
  - Necessário para criar e gerenciar os containers onde a pipeline será executada.
  - Usa a versão mais recente ("latest") do Docker.



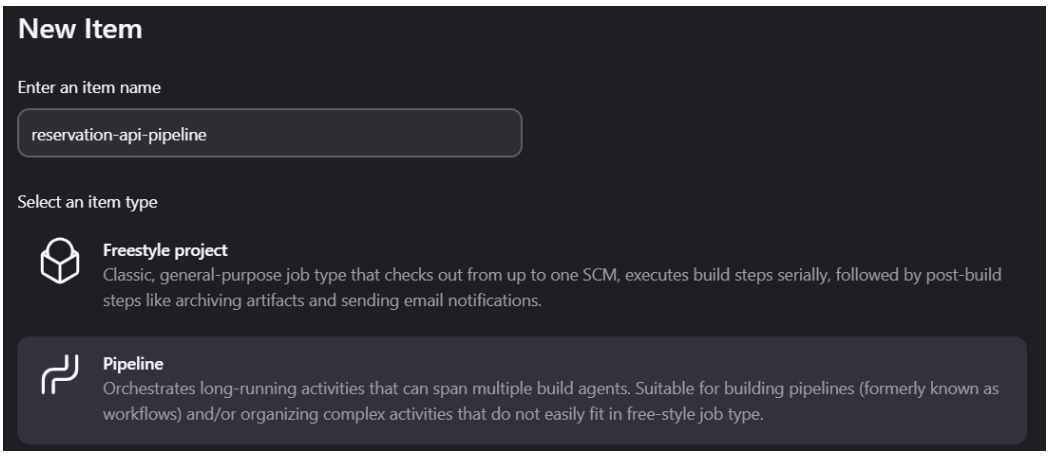
Todas estas configurações trabalham em conjunto para:

- Criar um ambiente isolado usando Docker.
- Compilar o projeto .NET no ambiente correto.
- Executar a análise de código usando o SonarQube Scanner.
- Enviar os resultados para o servidor SonarQube.
- Manter todo o processo seguro através das credenciais apropriadas.

Esta configuração permite uma análise de código automatizada que pode identificar problemas de qualidade, vulnerabilidades de segurança e débitos técnicos no código .NET, tudo integrado ao processo de CI/CD do Jenkins.

Criação da Pipeline

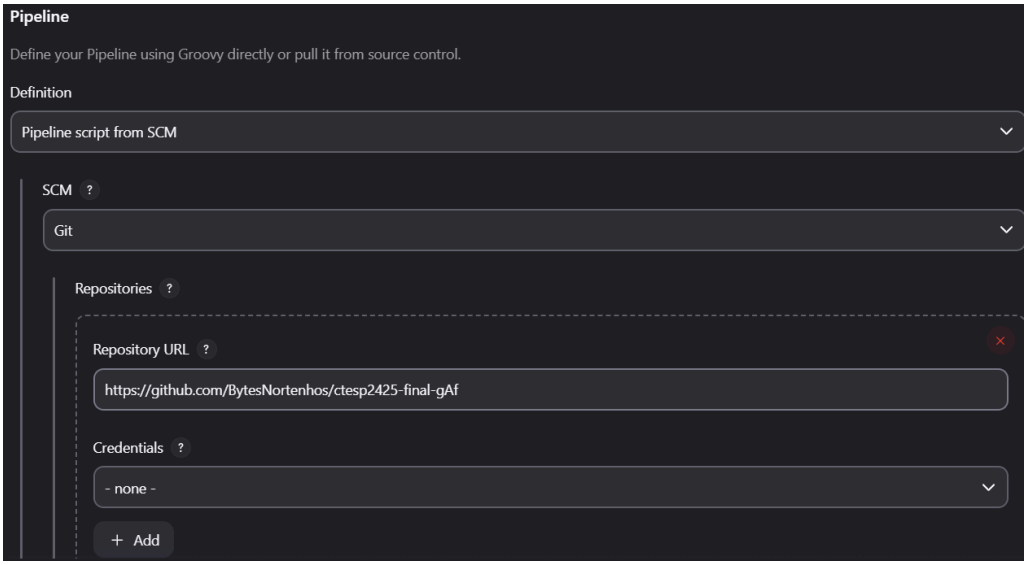
Por fim, temos de criar a pipeline. Começamos por acessar a dashboard e criar um novo item.



Para configurar a pipeline no Jenkins, devem ser preenchidos os seguintes campos:

1. Em "Definition", selecionar "Pipeline script from SCM"
2. Em "SCM", escolher "Git"
3. No campo "Repository URL", inserimos: https://github.com/BytesNortenhos/ctesp2425-final-gAf
4. Em "Branch Specifier", preencher com: */master
5. Em "Script Path", definir: Jenkinsfile
6. Marcar a opção "Lightweight checkout"

Esta configuração indica ao Jenkins onde encontrar o código fonte e o Jenkinsfile que contém as instruções da pipeline, garantindo que o processo de automação seja executado a partir da branch master do repositório especificado.

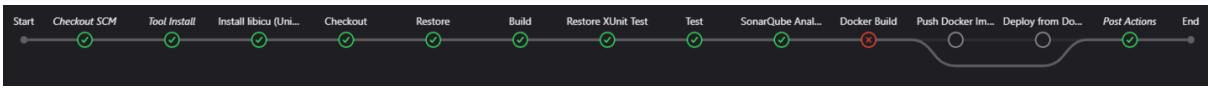


O ficheiro Jenkins presente contêm estes passos:

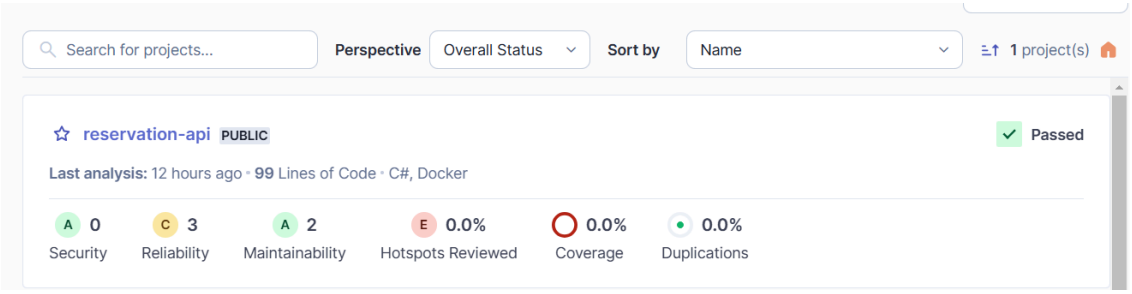
- Install libicu: Instala a biblioteca libicu apenas em sistemas Unix/Linux, necessária para suporte a localização no .NET.
- Checkout: Obtém o código-fonte do repositório.

- Restore: Restaura as dependências do projeto principal (dotnet restore).
- Build: Compila o projeto (dotnet build), garantindo que está pronto para execução.
- Restore XUnit Test: Restaura dependências para os testes unitários.
- Test: Executa testes unitários com cobertura de código.
- SonarQube Analysis: Faz análise de qualidade do código com o SonarQube.
- Docker Build: Constrói uma imagem Docker para a aplicação.
- Push Docker Image to Docker Hub: Faz o push da imagem Docker para o Docker Hub.
- Deploy from Docker Hub: Desliga e remove um container existente, baixa a nova imagem e executa um novo container com a aplicação.

A imagem seguinte representa a execução da pipeline de integração e entrega contínua (CI/CD). O processo inclui etapas como checkout do código-fonte, instalação de dependências, build, execução de testes, análise de qualidade com o SonarQube, push do docker para DockerHub e deploy em produção. Observa-se que a etapa de build do Docker apresentou uma falha, impedindo o progresso para as próximas fases de deploy. Algumas soluções foram testadas, mas não resolveram o problema de imediato. Ainda assim, seria apenas uma questão de tempo para corrigir e prosseguir com a pipeline.



A análise do SonarQube para o projeto reservation-api indica um estado geral "Passed", o que significa que não foram identificados problemas críticos que impeçam a sua execução.



Distribuição de Tarefas

- WebAPI / Base de Dados - Roberto Valente
- Docker / Migration - Dinis Faryna
- Testes Unitários / Vagrant - João Costa
- Jenkins / SonarQube - João Costa, Roberto Valente, Samuel Lemos
- Relatório - Pedro Barbosa

Conclusão

Em conclusão, este projeto proporcionou uma experiência significativa no desenvolvimento de uma API REST para gestão de reservas de restaurantes, incorporando práticas modernas de DevOps. Durante o processo, foram aplicados diversos conceitos fundamentais, desde o desenvolvimento da API com .NET Core até à implementação de testes unitários, containerização com Docker e automatização com Jenkins.

Um dos desafios significativos encontrados foi o erro "docker not found" durante a execução da pipeline no Jenkins. Este problema, embora não tenha impedido a demonstração dos conceitos principais, evidencia a importância de uma configuração adequada do ambiente Docker no servidor Jenkins e a necessidade de garantir que todas as dependências estejam corretamente instaladas e acessíveis. Além disso, é importante mencionar que foi desafiador trabalhar entre diferentes plataformas (Windows, Linux e MacOS) pois existem algumas diferenças na configuração dependendo da plataforma.

A aplicação prática destes conhecimentos é particularmente relevante em cenários reais. Por exemplo, uma empresa de software que precise implementar um pipeline de CI/CD robusto para as suas aplicações, onde as práticas de DevOps aqui demonstradas seriam fundamentais para garantir entregas consistentes e de qualidade.

A experiência adquirida com ferramentas como Docker, Jenkins e SonarQube, bem como a implementação de testes automatizados e práticas de CI/CD, representa um conjunto valioso de competências alinhadas com as necessidades atuais do mercado de desenvolvimento de software.