MICROSERVICES BASICS

This is a project with the basics about working with micro-services with .NET with docker, creating images, forwarding with their containers storing the data in a Microsoft SqlServer container volume.

> All the credits to InfoToolsSV:

- Tutorial: https://youtu.be/3ftI26leOzA?si=idaZZHJ1nRF6LTTn
- Channel: https://www.youtube.com/@InfoToolsSV

> My contribution is:

- Add SqlServer container to docker compose script to save data.
- .env file to leave the docker compose file cleaner.
- Name 2 different ConnectionStrings in the same .env file for webapi containers.
  - DefaultConnectionOrder
  - DefaultConnectionProduct

Nuget packages to add working with Entity Framework

Got to the project root directory and run the following commands

```
$> dotnet add package Microsoft.EntityFrameworkCore.Design
$> dotnet add package Microsoft.EntityFrameworkCore.Tools
$> dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

Microservice / WebApi configuration

Once you created the project, enter to the project and open the Program.cs file, and write all that's under the WARNING comments:

```
using System.Globalization;
using Microsoft.AspNetCore.Localization;
using Microsoft.EntityFrameworkCore;
using OrderMS.Data;


var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();
// WARNING: Remember to set in .csproj file the next configuration to go
// smoothly with the next builder configuration:
```

~~~~
// <InvariantGlobalization>false</InvariantGlobalization>
builder.Services.Configure<RequestLocalizationOptions>(options =>
{
    options.DefaultRequestCulture = new RequestCulture("en-US");
    options.SupportedCultures = new [] { new CultureInfo("en-US") };
    options.SupportedUICultures = new [] { new CultureInfo("en-US") };
});
// WARNING: If you're going to use docker, this line is getting the
// DB configuration.
builder.Services.AddDbContext<ProductContext>(options => options.UseSqlServer(

builder.Configuration.GetConnectionString("ConnectionStringName_From_Docker_Enviroment_Variables")
));
// WARNING: Because we use MVC in our web api project.
builder.Services.AddControllers();

var app = builder.Build();

// WARNING: In case to use a custom localization configuration
(builder.Services.Configure<RequestLocalizationOptions>).
app.UseRequestLocalization();
// WARNING: To update automatically the Models' schema with Entity Framework.
// With this code we ensure a db exists before running the web api.
using(var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<ProductContext>();
    // It's the same like running in the terminal "dotnet ef database update".
    dbContext.Database.Migrate();
}

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.UseHttpsRedirection();
// WARNING: Additional configuration callbacks for MVC.
/* if your api uses credentials to make requests */
// app.UseAutorization();
app.MapControllers();

app.Run();
```


---------------------------------------------------------


## Testing the webapi using Entity Framework

Don't forget run this command to create schema:

```bash
dotnet ef migrations add [migration_name]
~~~~

~~~
```

----------------------------------------------------------

## Dockerfile general structure

To create an ASP.NET Core Web API image:

```docker
# We need to have a image from the microsoft-dotnet sdk image.
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:[.NET VERSION] AS build

# We create/set the work directory in the container.
WORKDIR /source

# Connect the localization we set in the web api project to docker.
ENV DOTNET_SYSTEM_GLOBALIZATION_INVARIANT=false

# Copy the .csproj file to the project directory.
COPY ["ProjectName.csproj", "ProjectName/"]

# Restore the dependencies (libraries).
RUN dotnet restore "ProjectName/ProjectName.csproj"

# Copy all web api (MicroService) content
COPY . ./ProjectName

# Once we copied the project from local, let's change the work directory
# to the web api (MicroService).
WORKDIR "/source/ProjectName"

# Now we build the project:
# -c: Flag to indicate the executable profile.
# -o: Destination for the executable.
RUN dotnet build "ProjectName.csproj" -c Release -o /app/build

# Multistaging phase: publish.
# Using multiple FROM statements has the end to create temporary
# images that is used to the final image in the last FROM keyword.
FROM build AS publish

# Publishing the result.
RUN dotnet publish "ProjectName.csproj" -c Release -o /app/publish

# Final stage/image from publish stage. The final image should
# an entrypoint with the name "[.csproj filename].dll".
FROM mcr.microsoft.com/dotnet/aspnet:[.NET VERSION] AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "ProjectName.dll"]
```

----------------------------------------------------------
~~~

~~~
# Docker Compose script

```docker
services:
  [microservice_name]:
    build:
      context: ./[project_directory_containing_dockerfile]
    ports:
      - "8001:8080"
    env_file:
      - .env
  sqlserver:
    image: mcr.microsoft.com/mssql/server:2019-latest
    container_name: sqlserver_db
    env_file:
      - .env
    ports:
      - "1433:1433"
    volumes:
      - sql_data:/var/opt/mssql
volumes:
  sql_data:
```

---------------------------------------------------------

# .env file content

```bash
# DOTNET MICROSERVICE CONTAINERS
ASPNETCORE_ENVIRONMENT="Development"
ConnectionStrings__DefaultConnection[Microservice1]="Server=sqlserver_db;Database=
ProductDB;User=sa;Password=sa_pass;Encrypt=false"
ConnectionStrings__DefaultConnection[Microservice2]="Server=sqlserver_db;Database=
OrderDB;User=sa;Password=sa_pass;Encrypt=false"

# SQL SERVER CONTAINER
SA_PASSWORD="sa_pass"
ACCEPT_EULA="Y"`
```
~~~