Executing a SQL Server Docker Container and connecting to a .Net Project with DDD and Entity Framework Core

Joao Oliveira · Follow

6 min read · Aug 18, 2022

Listen Share More

In this article, I will show you how to create a SQL Server docker container and how to use it in the .Net project with DDD created in this article. In addition to the connection, we are going to use the Migrations feature of Entity Framework Core to create our database and tables without having to write any SQL lines.

Introduction

The SQL Server is one of the more powerful SQL DataBase, possessing versions for Windows, Linux, and Docker Container that, make possible for example your use in Macs and Linux not support official of Microsoft.

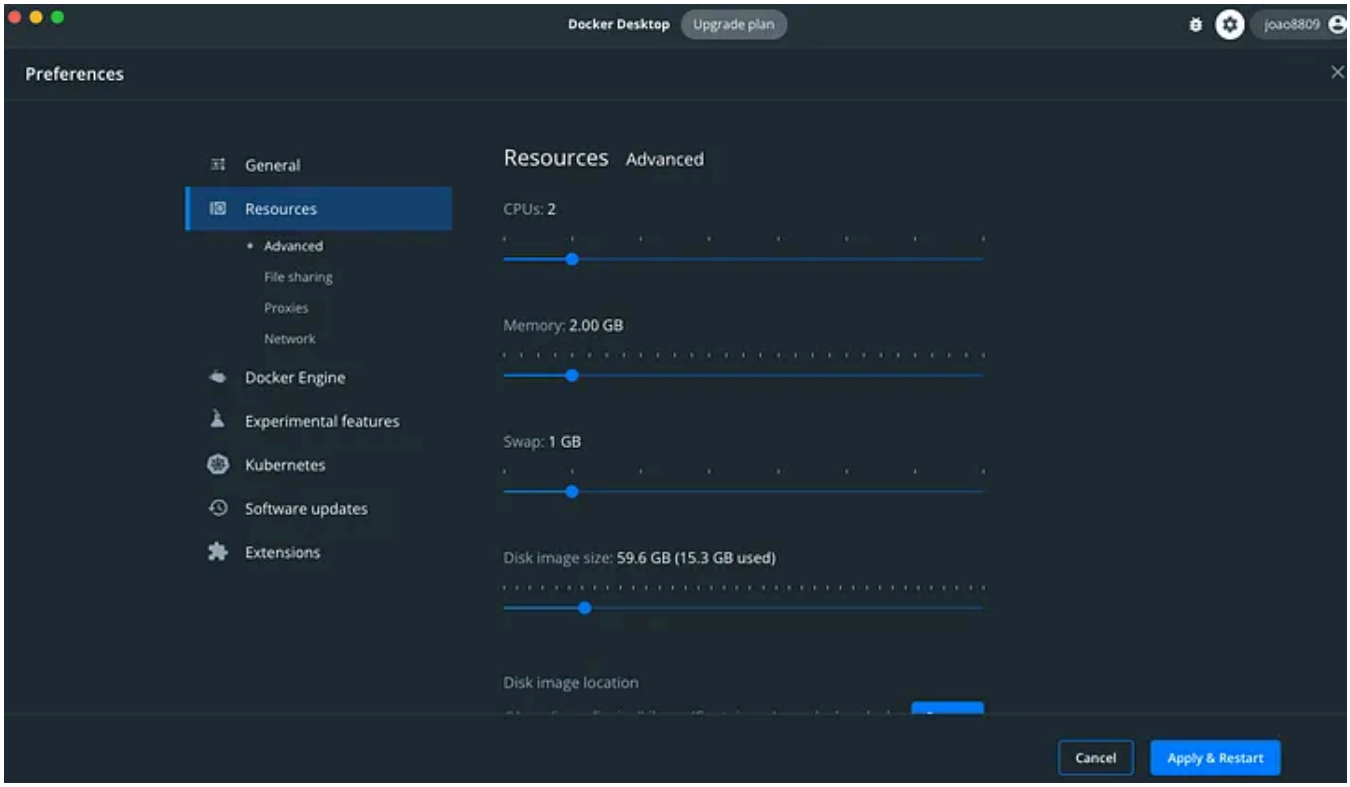
In the next sections will go created our SQL Server image on Docker, create our container a[redacted]ment, in the case will use the DBeaver, more feel free willing to use the tool you think is best. We will also configure our project to connect to the container and use the Migrations feature to create the database and tables, using the Code First approach.

Requirements

- Docker Desktop has versions for Windows, Mac, and Linux.
- DBeaver is a free database management tool with a graphical interface.

Installed Docker

Basically in this step, what you need to do is download the latest version of Docker from the official website and install it on your machine, after installing it, go to preferences and set the CPU usage to 2 and the memory to 2GB (minimum limit requested to run the SQL Server).



Creating a SQL Server Image in Docker

In the terminal of your choice, run the command to download the latest SQL Server image

On Windows:

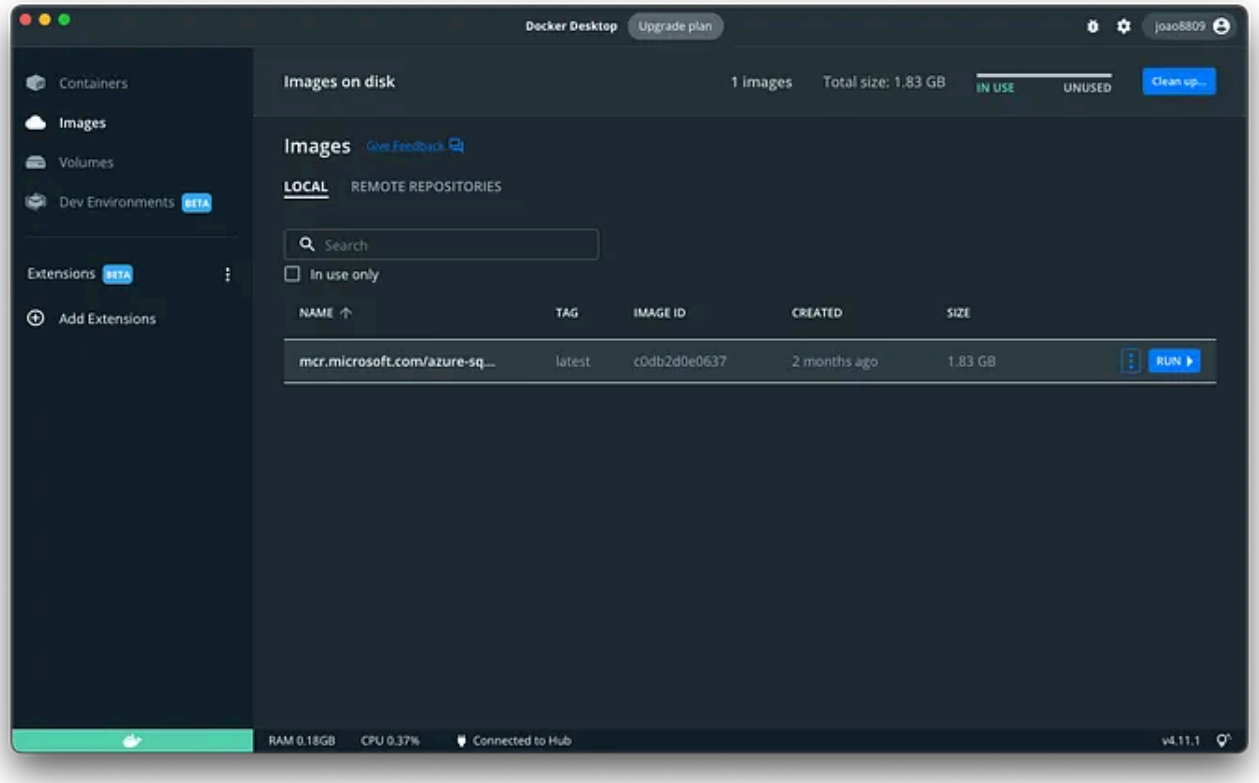
```
docker [redacted]
```

On Mac with M1 or M2 chip we will use the Azure SQL Edge version:

```
docker pull mcr.microsoft.com/azure-sql-edge
```

If you want to download a specific version, check the tags available here.

If everything went well you will see the image downloaded on the docker desktop:



Running the SQL Server image in docker

Now that we have the image downloaded, let's create a container for it, run the command below, but don't forget to change the MSSQL_SA_PASSWORD to the desired password. See password requirements here.

On Windows

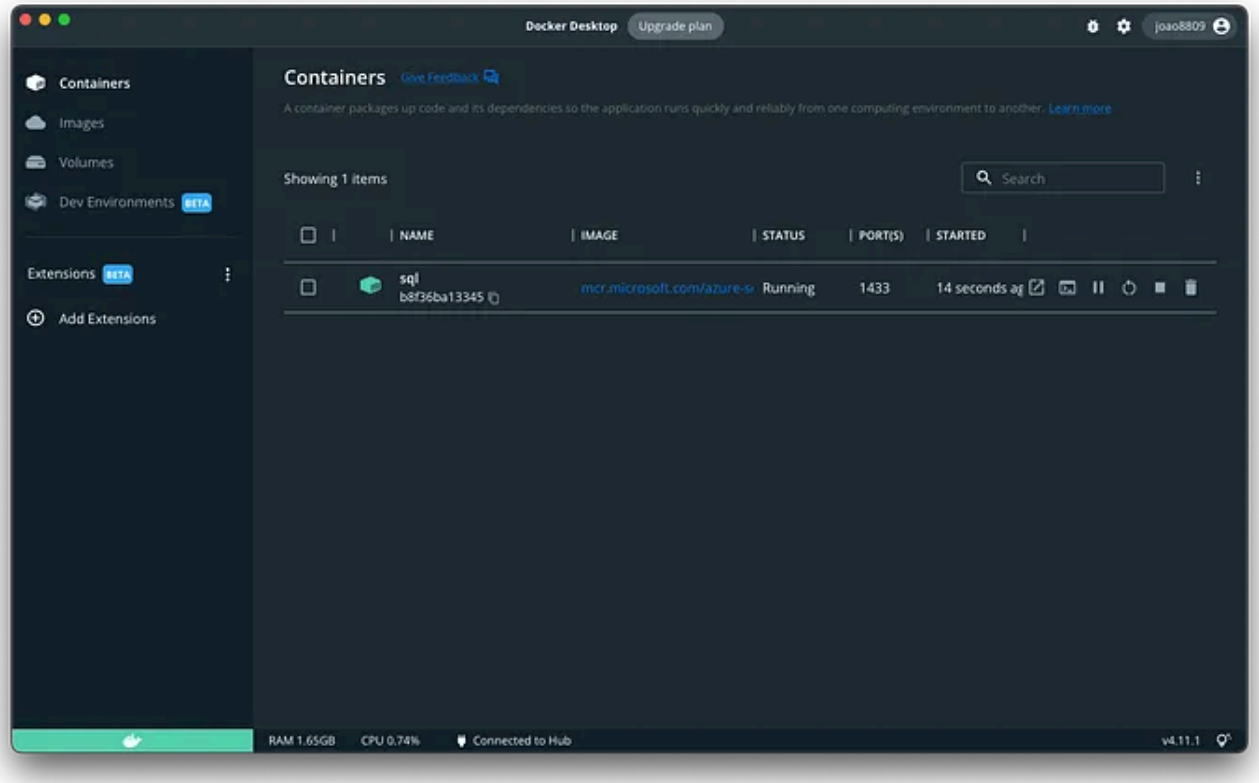
```
docker run -e "ACCEPT_EULA=1" -e "MSSQL_SA_PASSWORD=MyPass@word" -e
"MSSQL_PID=Developer" -e "MSSQL_USER=SA" -p 1433:1433 -d --name=sql
```

```
mcr.microsoft.com/mssql/server
```

On Mac with M1 and M2

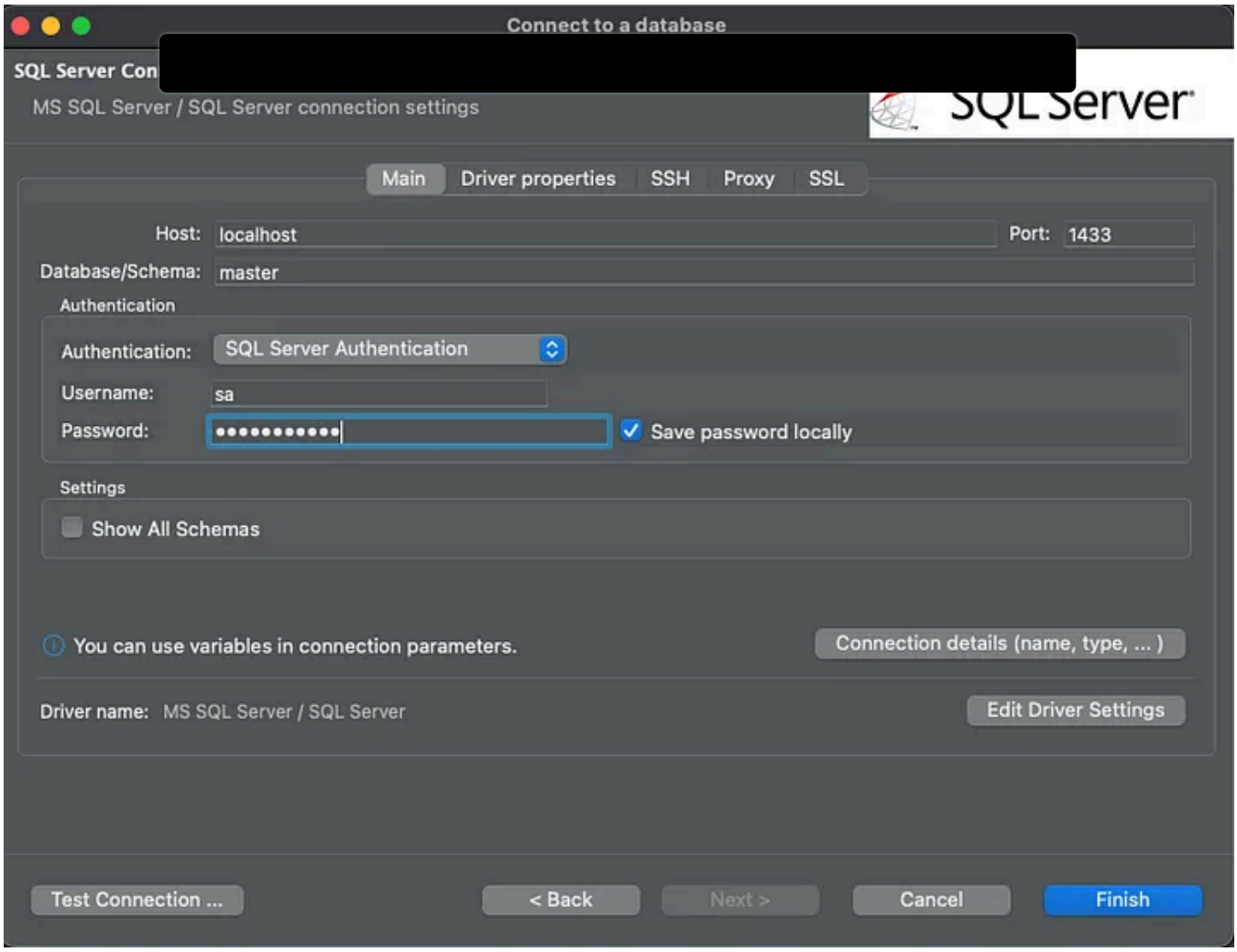
```
docker run -e "ACCEPT_EULA=1" -e "MSSQL_SA_PASSWORD=MyPass@word" -e
"MSSQL_PID=Developer" -e "MSSQL_USER=SA" -p 1433:1433 -d --name=sql
mcr.microsoft.com/azure-sql-edge
```

If everything went well you will see your container created and running as in the image below:



Connecting our SQL Server container with DBeaver

In DBeaver, start a new connection and choose SQL Server, by clicking on next configure as in the image below using the password you defined when you created your SQL Server container.



If everything went well when clicking on Test Connection it will return that the connection was successful.

Connecting our .Net project with SQL Server

Now that we have our SQL Server database running, let's connect our project and use the Entity Framework Core migrations feature to create our database and tables.

In appsettings.Development.json we will use the following connectionstring, but remember to change the password to the one you entered when you created your SQL Server container.

```
"ConnectionStrings": {
  "DefaultConnection": "Data Source=localhost; Initial
Catalog=Developer; User Id=sa; Password=MyPass@word"
},
```

Before we configure our application to get the value of our ConnectionString, let's install the following NuGets packages in our OrderingShop.Api project to use SQL

Server and use the migrations feature

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

In the DependencyInjection class add the code below to get the value of our ConnectionString.

```
1
2  public static class DependencyInjection
3  {
4      public static void AddInfrastructureApi(this IServiceCollection services, IConfiguration co
5      {
6          var connectionString = configuration.GetConnectionString("DefaultConnection");
7
8          services.AddDbContext<ApplicationDbContext>(options => options.UseSqlServer(connectionS
9      }
10 }
```

DependencyInjection.cs hosted with ❤ by GitHub view raw

In our ApplicationDbContext class we map our Product entity.

```
1  public class ApplicationDbContext : DbContext
2      {
3          public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(opti
4          { }
5
6          public DbSet<Product> Products { get; set; } = null!;
7
8          protected override void OnModelCreating(ModelBuilder modelBuilder)
9          {
10         }
11     }
```

ApplicationDbContext.cs hosted with ❤ by GitHub view raw

For our Product Entity, we can use DataAnnotations to define the name of the table and columns, there are many other configurations that we can do here for this mapping, but this is a subject for another article.

```
[Table("PRODUCT")]
public [redacted]
{
    [Column("NAME")]
    public string Name { get; set; } = string.Empty;
    [Column("DESCRIPTION")]
    public string Description { get; set; } = string.Empty;
    [Column("PRICE")]
    public decimal Price { get; set; }
    [Column("STOCK")]
    public int Stock { get; set; }
}
```

Product.cs hosted with ❤ by GitHub view raw

Remember that the Product class inherits from BaseEntity and in this class, we have our Id property.

```
public class BaseEntity
{
    [Key]
    [Column("ID")]
    public int Id { get; set; }
}
```

BaseEntity.cs hosted with ❤ by GitHub view raw

This is a basic mapping, in the future I intend to write an article covering only the main configurations that we can make to map and configure our entities to use with migrations.

Now that we have everything set up, let's run the EF Core command responsible for creating our migration. Before that, install the 3 NuGets packages below in the OrderingShop.Infrastructure project.

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

Now inside the root path of our solution:

"Template-DDD-with-.Net-Microservice/src/main"

Run the following command to create our first migration:

```
dotnet ef migrations add Initial -p OrderingShop.Infrastructure -s
OrderingShop.Api
```

Where **Initial** is the name that we will give our migration, you can change this name with details of what your migration refers to, such as changing a field in a table, adding or removing fields, by convention we try to detail the update that the migration will do.

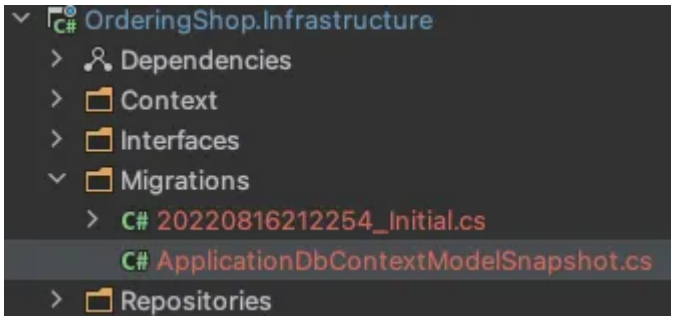
After -p (project) we pass the name of the solution that contains our DbContext in the case of Infrastructure and -s (solution) we pass our main project in the case of the API.

If everything went well after running the command you will get a message like this:

Done. To undo this action, use 'ef migrations remove'

The ***migrations remove*** command is used to remove the created migration if it is not as you wanted.

Within the infrastructure project, a folder called Migrations will be created with the following files:



If you open these files you will see all the configurations that EF Core created and that will be applied when we send it to our bank.

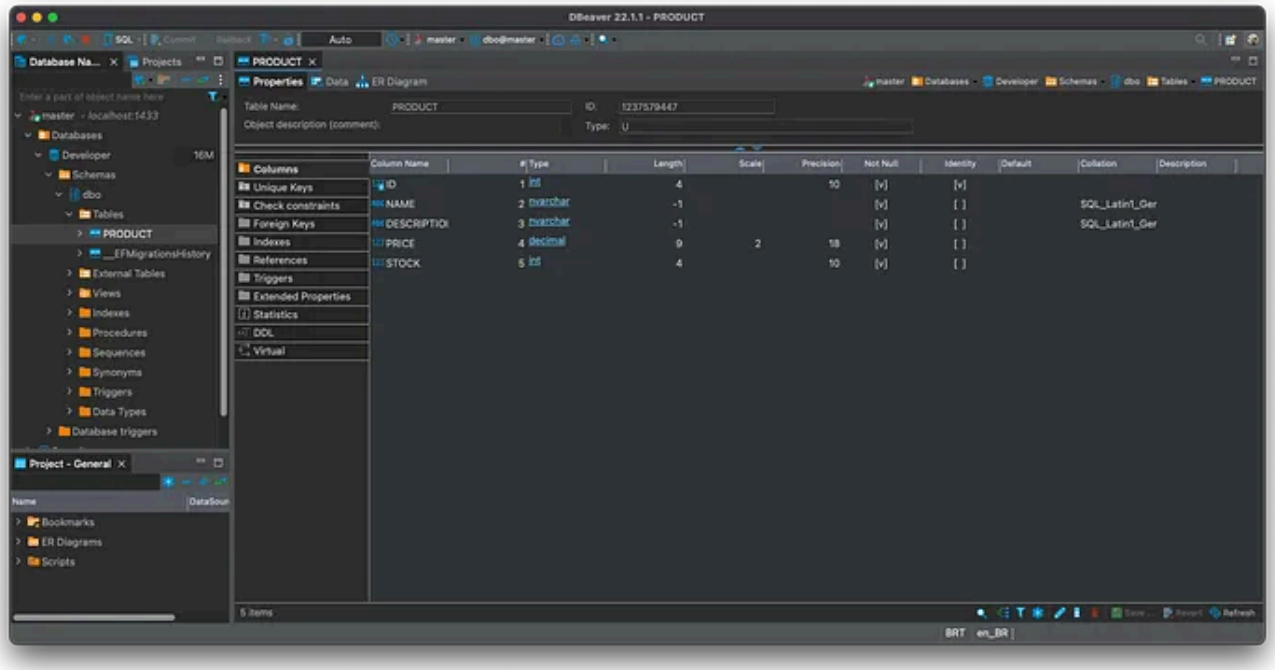
Now that we have our migration created and that everything is ok, it's time to send it to our created SQL Server database, let's use the following command to publish our migration also in the root path:

“Template-DDD-with-.Net-Microservice/src/main”

```
dotnet ef database update -s OrderingShop.Api
```

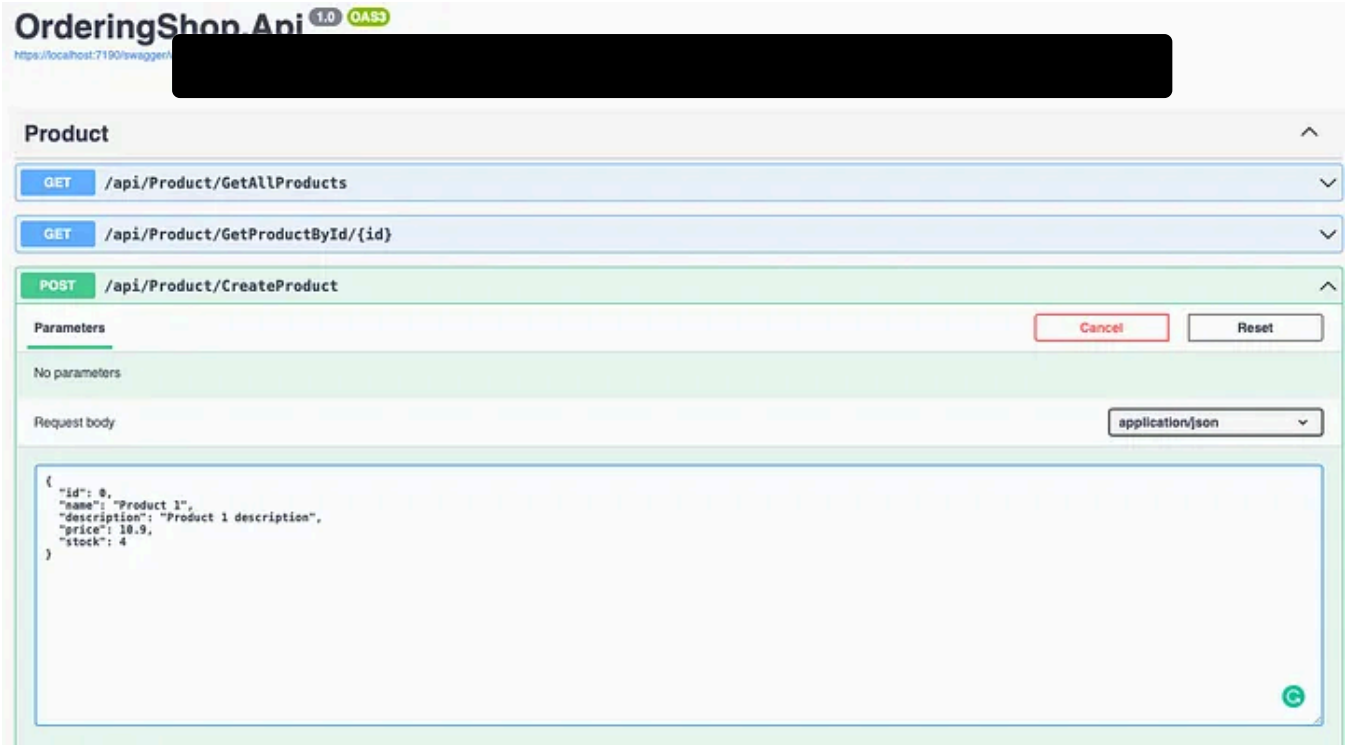
If everything went well you will see an SQL script and a Done message in the log.

Now let’s go back to DBeaver and check if our database was created, expanding the connection we can see that our database and table were created as we configured in our project, without the need to create any SQL line.

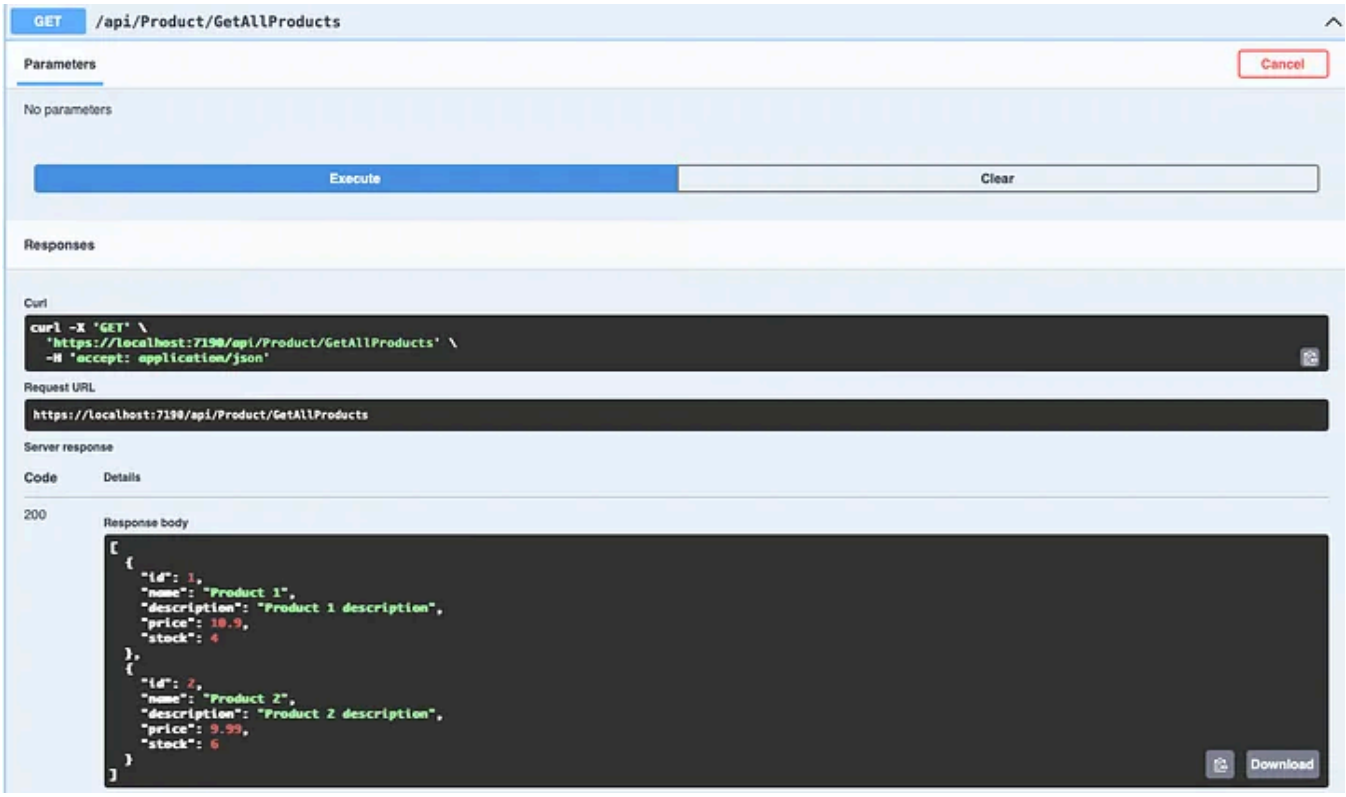


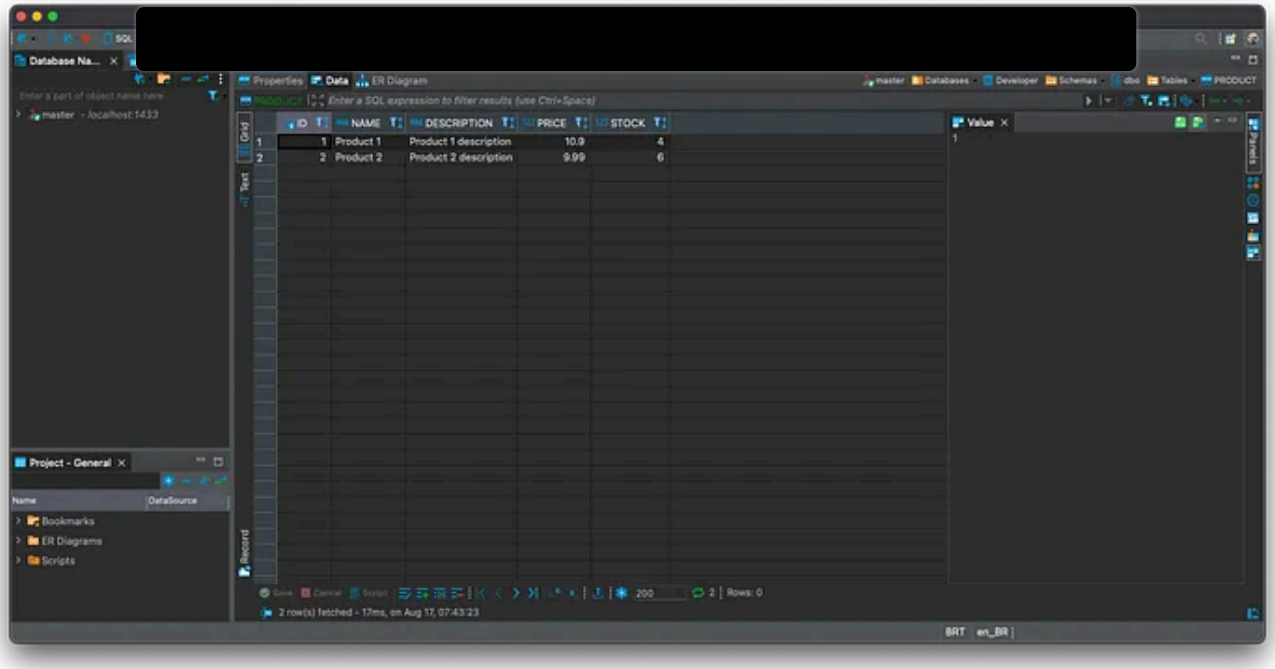
Testing our Endpoints in the API

Now that we have our database already created, let’s test our endpoints in the API, first let’s register 2 products.



After completing the registration, we will use the **GetAllProducts** method to return the 2 registered products.





As we can see in the response body and also in the database, we have the 2 products recorded in our SQL Server database, and with the EF Core migrations feature, we don't need to write any SQL lines.

Good luck and see you around next time!

Link full project in the github

Written by Joao Oliveira

[redacted]

38 Followers · 1 Following

Senior Software Engineer | C# | .Net | Azure | AWS | Docker

Responses (1)

What are your thoughts?

Respond

Fabio Galante Mans
Aug 19, 2022

João você é brasileiro?

2 replies Reply

More from Joao Oliveira

Joao Oliveira

Implementing a microservice in .Net with DDD

The DDD (Drive Domain Desing) is a whole good practices that was disclosed by Eric Evans in 2003 in your book “”. Since then the use of...

Aug 7, 2022 •••

See all from Joao Oliveira

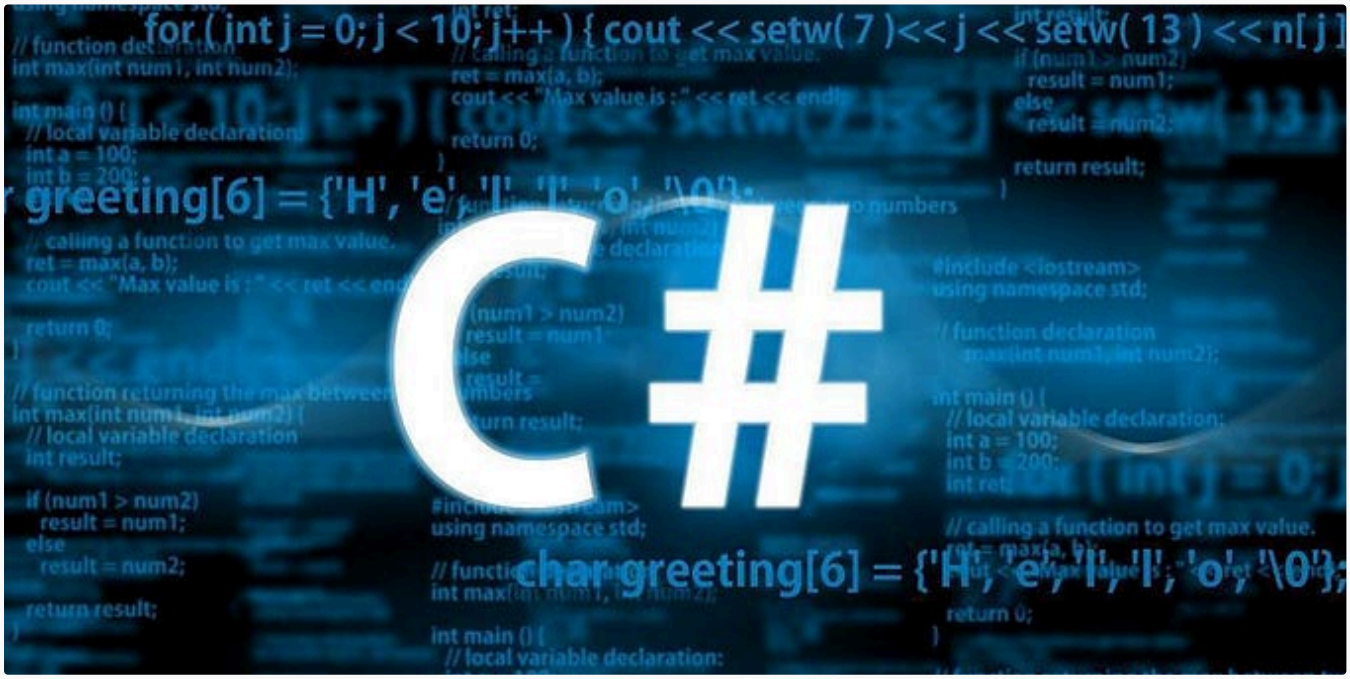
Recommended from Medium

Engr. Md. Hasan Monsur

LINQ and Dapper in .NET: Best Practices for Efficient Data Queries

In this article, “LINQ and Dapper in .NET: Best Practices for Efficient Data Queries,” we explore how to effectively leverage LINQ with...

Oct 18, 2024 21



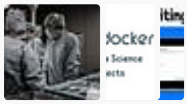
In Dev Genius by .Net Labs

Best practice Generic List Filter using Func<T,V> in C#

In this Chapter we will learn about Func<x,y> delegate and its real time implementations and will see how we can create centralize...

Feb 4 [redacted]

Lists



Coding & Development

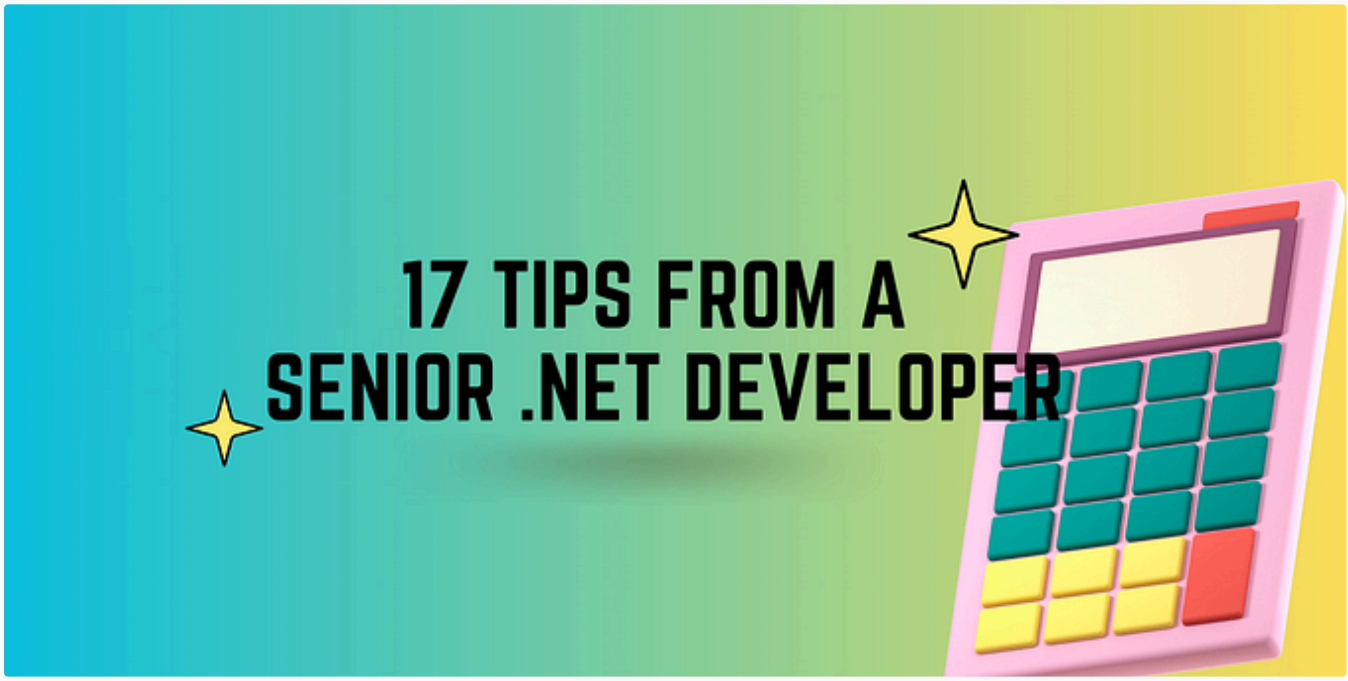
11 stories · 1010 saves

ChatGPT

21 stories · 972 saves

Natural Language Processing

1946 stories · 1598 saves



In Write A Catalyst by Sukhpinder Singh | C# .Net

17 Tips from a Senior .NET Developer

10 years in .NET development have taught me more than just writing clean code.

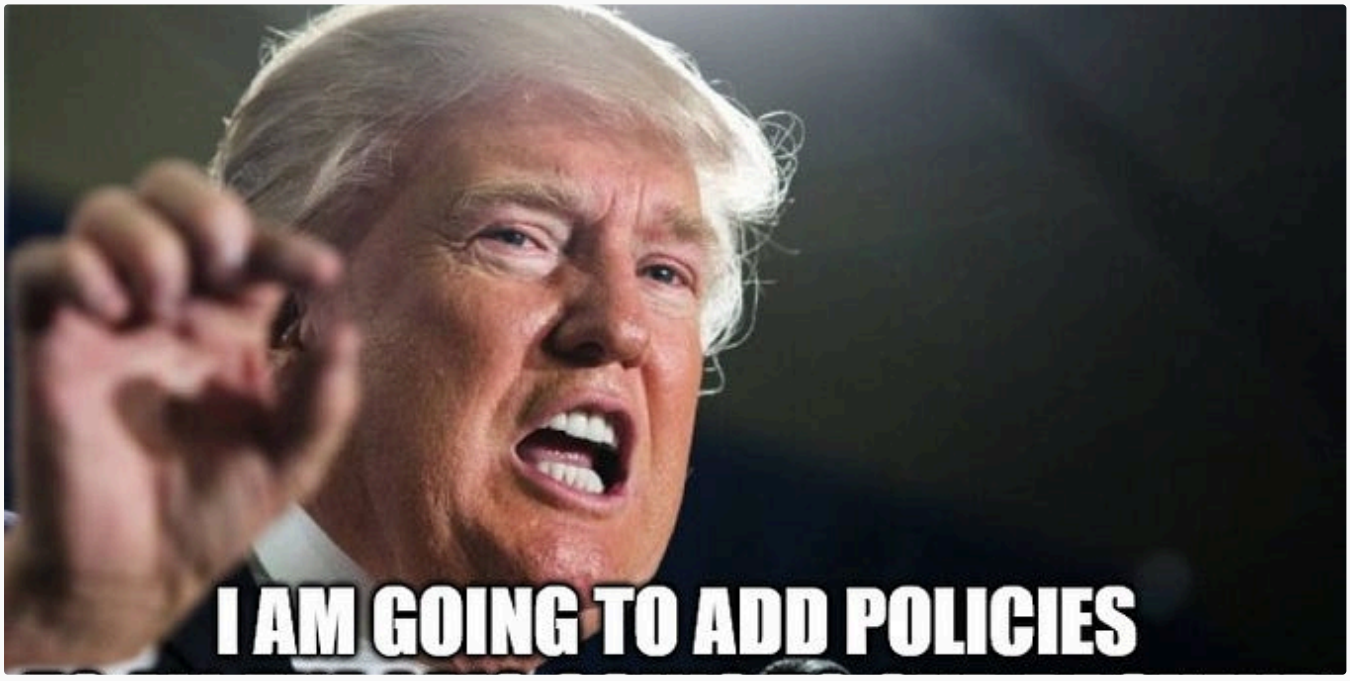
Feb 7 457 12

Harendra

How I Am Using a Lifetime 100% Free Server

Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26, 2024 9K 145

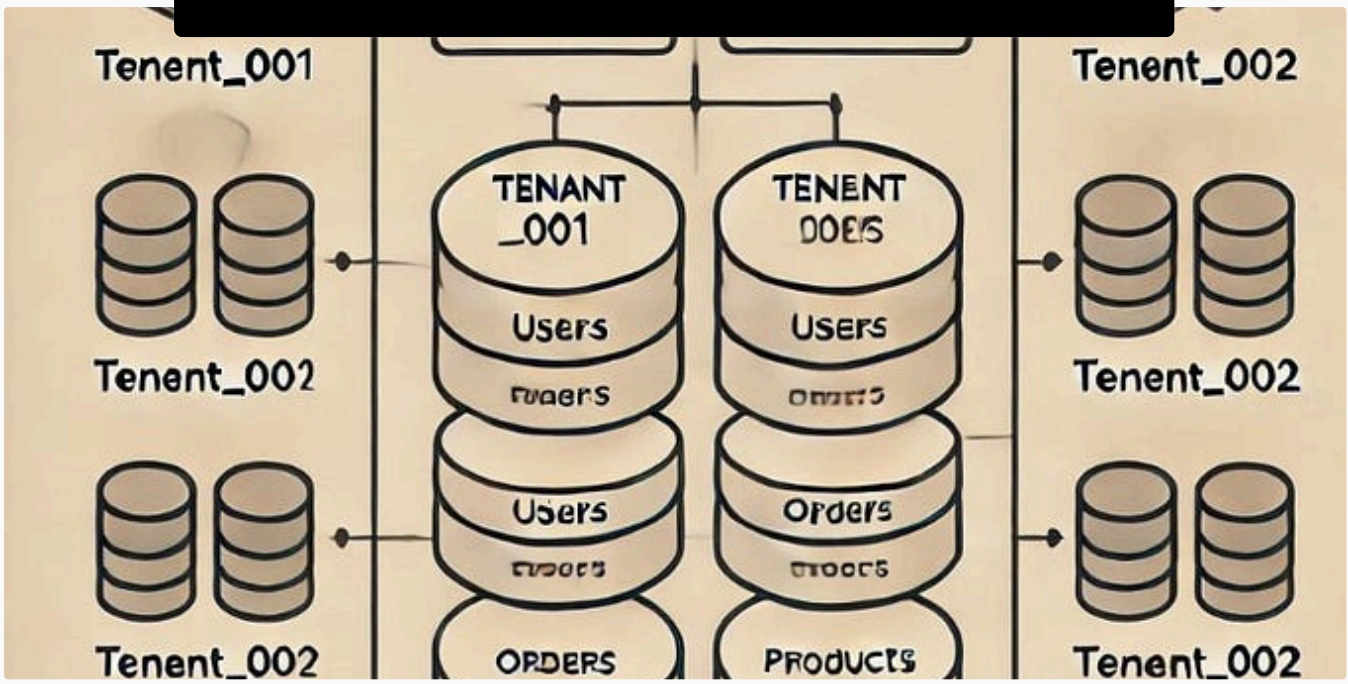


Bob Code

Azure APIM Using Terraform

Using Terraform

Feb 11 110



Maroine Cherif

Building a Multi-Tenant App in ASP.NET 8 MVC with DB Isolation

Scalability and data isolation are crucial for modern SaaS and enterprise applications. In this article, we'll walk you through how to...

Sep 8, 2024 26

See more recommendations