

## .NET Core

O .NET Core é uma plataforma para desenvolvimento de aplicações desenvolvida e mantida pela Microsoft como um projeto open source. É uma solução mais leve e modular que o .NET Framework e pode ser usada em diferentes sistemas operacionais.

<https://github.com/aspnet>

<https://dotnet.microsoft.com/download>

<https://dotnet.microsoft.com/download/dotnet-core>

## API

“Application Programming Interface”, Interface de Programação de Aplicações.

As APIs são um tipo de “ponte” que conectam aplicações, podendo ser utilizadas para os mais variados tipos de negócio, por empresas de diversos nichos de mercado ou tamanho,

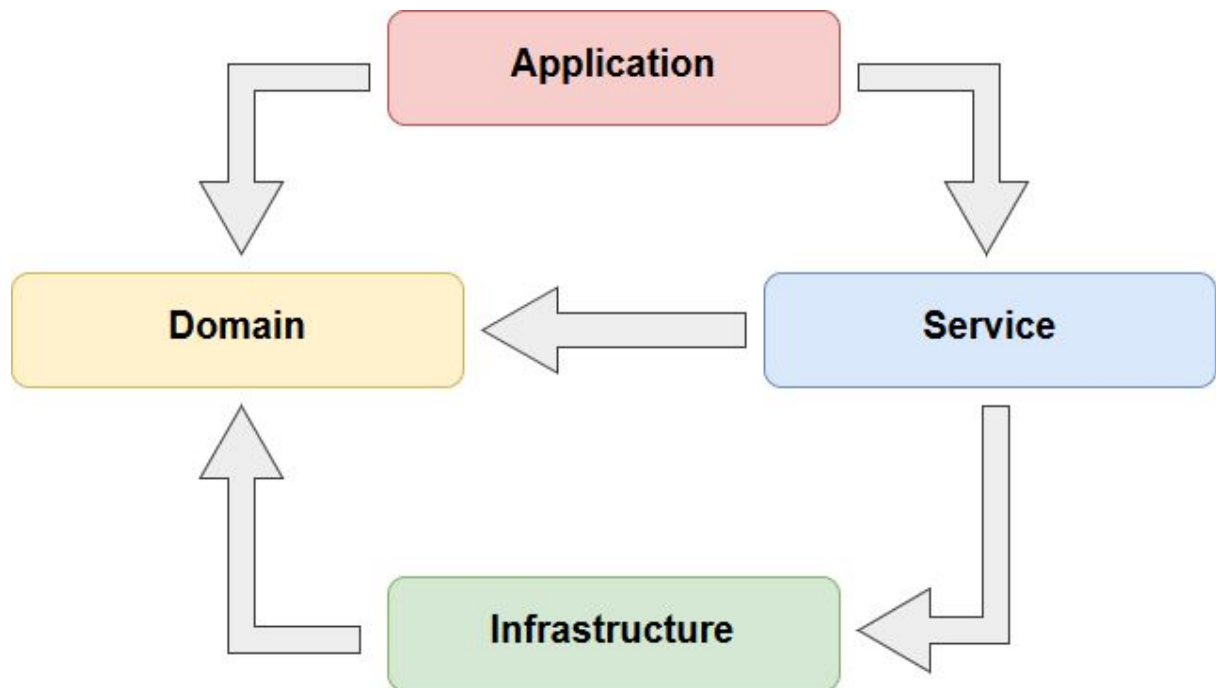
Elas são uma forma de integrar sistemas, possibilitando benefícios como a segurança dos dados, facilidade no intercâmbio entre informações com diferentes linguagens de programação.

## Uma nova arquitetura em .Net Core baseada em DDD

O *DDD (Domain Driven Design)* é uma modelagem de software cujo objetivo é facilitar a implementação de regras e processos complexos, onde visa a divisão de responsabilidades por camadas e é independente da tecnologia utilizada.

Levando em consideração este conceito, é proposto desenvolver uma nova arquitetura baseada na mesma para construção de uma *API (Interface de Programação de Aplicativos)*.

## Entendendo a arquitetura utilizada



1. Camada de **aplicação**: responsável pelo projeto principal, pois é onde será desenvolvido os controladores e serviços da *API*. Tem a função de receber todas as requisições e direcioná-las a algum serviço para executar uma determinada ação.

*Possui referências das camadas **Service e Domain**.*

2. Camada de **domínio**: responsável pela implementação de classes/modelos, as quais serão mapeadas para o banco de dados, além de obter as declarações de interfaces, constantes, **DTOs (Data Transfer Object)** e **enums**.

3. Camada de **serviço**: seria o “coração” do projeto, pois é nela que é feita todas as regras de negócio e todas as validações, antes de persistir os dados no banco de dados.

*Possui referências das camadas **Domain**, **Infra.Data** e **Infra.CrossCutting**.*

4. Camada de **infraestrutura**: é dividida em duas sub-camadas
  - Data: realiza a persistência com o banco de dados, utilizando, ou não, algum *ORM*.
  - Cross-Cutting: uma camada a parte que não obedece a hierarquia de camada. Como o próprio nome diz, essa camada cruza toda a hierarquia. Contém as funcionalidades que pode ser utilizada em qualquer parte do código, como, por exemplo, validação de CPF/CNPJ, consumo de API externa e utilização de alguma segurança.

*Possui referências da camada **Domain**.*

## Criando o projeto

O projeto em questão será um CRUD (**Criar, Ler, Alterar e Deletar**) simples.

Utilizando o Asp. Net Core 2.2 com Linguagem C# e o mesmo pode ser utilizado no Linux, Windows e OS X, ORM Entity Framework utilizando container de Bancos de Dados MySQL e MS-SQL Server no Docker.

Iremos utilizar o Visual Code ou VSCode como Editor de Código e Depuração do mesmo.

<https://code.visualstudio.com/>

Este editor pode ser utilizado em Windows, OS X e Linux.

## Camada Application - Passo a Passo

Vamos utilizar o .NET Core Command-Line Interface (CLI)

Comandos para Help do .NET Core (CLI)

**dotnet --help**

**dotnet new --help**

**dotnet --info**

1 - Criar as seguintes pastas Exemplo Abaixo:

**\api\src**

Digite: **dotnet new sln --name Api**

Será criado uma Solution com o nome de **Api.sln**, dentro da pasta Src

2 - Criar uma Aplicação Asp. Net Core Web API

Digite: **dotnet new webapi -n --help**

Digite: **dotnet new webapi -n Application -o Api.Application --no-https**

**-n** = Nome do Projeto

**-o** = Saída

**--no-https** = Para projeto que não requer HTTPS.

Neste ponto será criado uma aplicação web configurada para ser uma API

Adicionar o projeto Application na Solution (Api.sln)

Digite: **dotnet sln add Api.Application**

Api

└── Src

    └── Api.Application

```
|—— Controllers  
|—— obj  
|—— Properties
```

Acesse a pasta Api\src

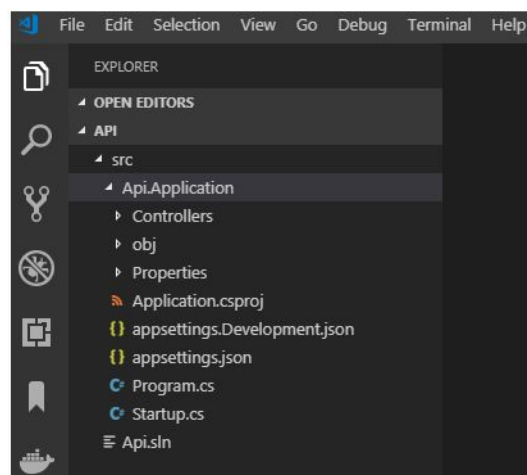
Digite: **dotnet build**

este comando deve executar a Restauração e a Compilação da Aplicação.  
Se a compilação for compilada com êxito.

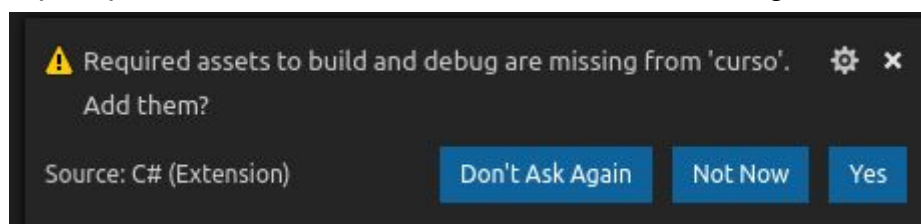
volte para a pasta Api

Digite: **Code .**

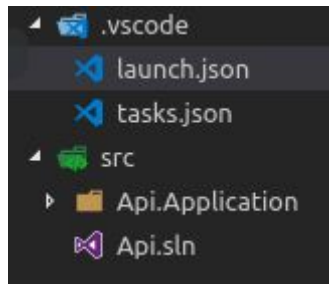
O Editor VsCode será executado abrindo o projeto Api, sempre faz isto um nível abaixo da pasta Src.  
conforme ilustra a imagem abaixo



Ao acessar pela primeira vez o Visual Code exibirá esta mensagem



Neste ponto deve escolher YES, pois o visual code irá criar automaticamente a pasta .vscode com 2 arquivos Launch.json e tasks.json



**OBS:** Para que este arquivo seja criado automaticamente, o visual code tem que ter uma extensão instalada chamada **C#**

Caso o projeto tenha sido criado com o padrão **https** teremos que remover o https do projeto.

Acessar a classe **Startup.cs** na pasta **../api/src/Api.Application**

Comentar a linha

```
app.UseHttpsRedirection();
```

Resultado Esperado

```
//app.UseHttpsRedirection();  
app.UseMvc();
```

Modificar o arquivo **launchSettings.json** da pasta **../api/src/Api.Application/Properties**

Modificar esta linha

```
"applicationUrl": "https://localhost:5001;http://localhost:5000",
```

Para

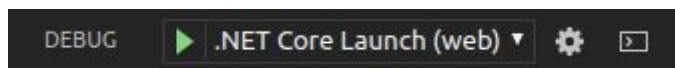
```
"applicationUrl": "http://localhost:5000",
```

Resultado esperado

```
"Application": {  
  "commandName": "Project",  
  "launchBrowser": true,  
  "launchUrl": "api/values",
```

```
"applicationUrl": "http://localhost:5000",  
"environmentVariables": {  
  "ASPNETCORE_ENVIRONMENT": "Development"  
}
```

Agora podemos executar aplicação.



Acessar os endereço abaixo:

<http://localhost:5000/api/values>

<http://localhost:5000/api/values/1>

## O que são as Rotas?

Rota é um caminho, uma direção ou um rumo.

Exemplo

Quando acessamos [www.algumacoisa.com.br](http://www.algumacoisa.com.br) estamos utilizando a rota "/"

Quando acessamos [www.algumacoisa.com.br/contato](http://www.algumacoisa.com.br/contato) estamos utilizando a rota "/contato"

Este é o conceito de rotas, ao receber uma requisição em uma determinada URL o sistema de rotas define o que fazer, como por exemplo redirecionar ou enviar para o controlador (Controller) decidir o que fazer.

## Quais são as rotas possíveis?

As rotas possíveis mediante pasta e nome de arquivos são:

[api/noticias](#)

Verbos	Rotas
GET	/api/noticias
GET	/api/noticias/{id}
POST	/api/noticias/
PUT	/api/noticias/{id}
DELETE	/api/noticias/{id}

Rotas do tipo **GET** é para retornar algo, como por exemplo uma listagem de conteúdo e etc.

Rotas do tipo **POST** normalmente são utilizados para cadastrar algo no sistema.

Rotas do tipo **PUT** ou **PATH** são para editar algum registro.

Rotas do tipo de **DELETE** são para deletar algo.

Neste ponto a nossa aplicação api está funcionando com a controller de exemplo.

Aplicação recebe uma Rota a mesma é identificada automaticamente e se ela existir irá executar o método da controller.

## Criar as Demais Camadas - Passo a Passo

As demais camadas são todas do tipo Class Library  
acessar a pasta ./Api/src

### Domain:

**dotnet new classlib -n Domain -f netcoreapp2.2 -o Api.Domain**

**-n** = Nome do Projeto

**-f** = Framework netcoreapp2.2 se não mencionar ele vai criar netstandard2.0

**-o** = Saída

Adicionar o projeto de domain a Solution

**dotnet sln add** Api.Domain

Executar **dotnet build** na pasta \Api\src

Criar a Camada domain com o comando abaixo:

**dotnet new classlib -n CrossCutting -f netcoreapp2.2 -o**  
Api.CrossCutting



### Infra Cross Cutting:

**dotnet new classlib -n CrossCutting -f netcoreapp2.2 -o**  
Api.CrossCutting

Adicionar o projeto de domain a Solution

**dotnet sln add** Api.CrossCutting

Executar **dotnet build** na pasta \Api\src

### Infra Data:

**dotnet new classlib -n Data -f netcoreapp2.2 -o** Api.Data

Adicionar o projeto de domain a Solution

**dotnet sln add** Api.Data

Executar **dotnet build** na pasta \Api\src

### Service:

**dotnet new classlib -n Service -f netcoreapp2.2 -o** Api.Service

Adicionar o projeto de domain a Solution

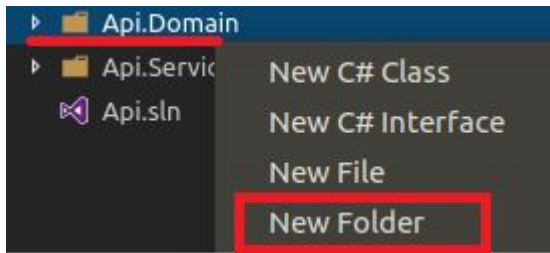
**dotnet sln add** Api.Service

Executar **dotnet build** na pasta \Api\src

**OBS:** Apagar todos os Class1 de todos os projetos Class Library

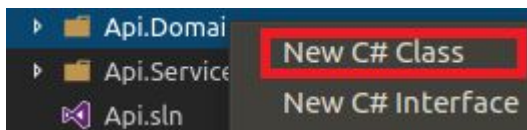
### Api.Domain - Implementação de Entidades

Acessar a Camada de Domain e Criar duas pastas (**Entities e Interfaces**)



```
Api
├── Src
│   ├── Api.Domain
│   │   ├── Entities
│   │   └── Interfaces
```

Criar uma Classe chamada **BaseEntity.cs**



```
using System;
using System.ComponentModel.DataAnnotations;

namespace Api.Domain.Entities
{
    public abstract class BaseEntity
    {
        [Key]
        public Guid Id { get; set; }
        private DateTime? _createAt;
        public DateTime? CreateAt
        {
            get { return _createAt; }
            set { _createAt = (value == null ? DateTime.UtcNow : value) ; }
        }
        public DateTime? UpdateAt { get; set; }
    }
}
```

Criar uma outra classe chamada UserEntity.cs

```
namespace Api.Domain.Entities
{
    public class UserEntity : BaseEntity
    {
        public string Name { get; set; }
        public string Email { get; set; }
    }
}
```

```
}
}
```

Neste ponto criamos uma classe de base de Entidades na qual todas as tabelas no Banco de dados irá ter como Default (Id, CreateAt, UpdateAt)

Na classe UserEntity.cs herdamos da Classe BaseEntity e adicionamos os novos campos que irá compor esta nova tabela (Entidade).

Conforme o padrão DDD, todos as interfaces e Entidades deve pertencer a camada de Api.Domain.

## Api.Data - Instalação Pacotes Entity Framework

Essa camada será responsável por conectar ao banco de dados, no caso será utilizado o MySQL, e realizar as persistências.

Inicialmente instala-se os seguintes pacotes:

- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.Tools
- Pomelo.EntityFrameworkCore.MySql

Para Realizar a instalação acesse o Terminal do Visual code, acesse a pasta ./src/Api.Data

Executar estes comandos do NuGet (Gerenciador de Pacotes)

```
dotnet add package Microsoft.EntityFrameworkCore.Tools --version 2.2.4
```

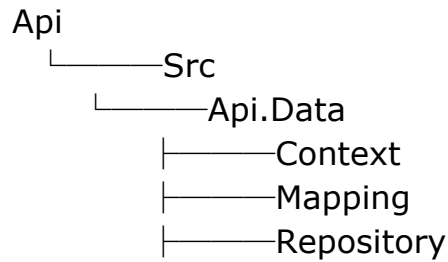
```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 2.2.4
```

```
dotnet add package Pomelo.EntityFrameworkCore.MySql --version 2.2.0
```

Toda instalação de Pacote dentro da camada será referenciada em um arquivo com extensão .csproj, neste caso é Data.csproj.

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="2.2.4" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.2.4">
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>
  <PackageReference Include="Pomelo.EntityFrameworkCore.MySql" Version="2.2.0" />
</ItemGroup>
```

Criar três pastas chamadas *Context*, *Mapping* e *Repository*.



Precisamos criar uma referência para Camada de Domain pois nesta camada contém as Interfaces e Entities.

Acesse a pasta **src**

e execute o comando abaixo para adicionar uma referência entre a camada de Data com a Camada de Domínio.

**dotnet add Api.Data reference Api.Domain**

No arquivo Data.csproj, será adicionada a linha abaixo

```

<ItemGroup>
  <ProjectReference Include="..\Api.Domain\Domain.csproj" />
</ItemGroup>
  
```

## Api.Data - Contexto e Mapeamento

Na pasta Context, criar uma classe **MyContext.cs**

Esta classe tem a função de fazer a conexão com o Banco de Dados e as propriedades do DbSet

```

using Api.Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace Api.Data.Context
{
    public class MyContext : DbContext
    {
        public DbSet<UserEntity> Users { get; set; }
        public MyContext(DbContextOptions<MyContext> options) : base(options) { }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
        }
    }
}
  
```

```
}
```

Continuando na pasta Context, criar uma classe **ContextFactory.cs**

Esta classe será responsável para prover uma conexão em tempo de desenvolvimento para os comandos do Entity Framework (EF)

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
```

```
namespace Api.Data.Context
{
    public class ContextFactory : IDesignTimeDbContextFactory<MyContext>
    {
        public MyContext CreateDbContext(string[] args)
        {
            //Usado para Criar as Migrações
            var connectionString = "Server=localhost;Port=3306;Database=Course;Uid=root;Pwd=admin";
            var optionsBuilder = new DbContextOptionsBuilder<MyContext>();
            optionsBuilder.UseMySQL(connectionString);
            return new MyContext(optionsBuilder.Options);
        }
    }
}
```

Na pasta Mapping, criar uma classe **UserMap.cs**

```
using Api.Domain.Entities;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Api.Data.Mapping
{
    public class UserMap : IEntityTypeConfiguration<UserEntity>
    {
        public void Configure(EntityTypeBuilder<UserEntity> builder)
        {
            builder.ToTable("User");

            builder.HasKey(p => p.Id);

            builder.HasIndex(p => p.Email)
                .IsUnique();

            builder.Property(c => c.Name)
                .IsRequired()
                .HasMaxLength(60);

            builder.Property(c => c.Email)
```

```

        .HasMaxLength(100);
    }
}
}

```

Esta classe você irá configurar o que você espera da entidade que será criada no banco de dados.

Você pode definir nome da tabela, chave primária, índices e definir algumas propriedade para a coluna da tabela

Para que o Mapeamento da entidade funcione precisamos colocar no MyContext.cs no método OnModelCreating

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<UserEntity> (new UserMap().Configure);
}

```

Isto será interpretado pelo próprio entity framework no momento de criar as migrações.

## DOCKER - Container

Vou utilizar docker para subir o MySQL 8 e Microsoft SQL-Server 2017.

O Docker instala normalmente em Windows 10 64 bits: Pro, Enterprise ou Education.

<https://docs.docker.com/docker-for-windows/install/>

Mas o Ideal é rodar em Linux.

<https://docs.docker.com/install/linux/docker-ce/ubuntu/>

Criar um arquivo na pasta Raiz (Api) chamado **docker-compose.yml**

```

version: '3.1'
services:
  Mysql:
    image: mysql:8.0.16
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: admin
    ports:
      - 3306:3306

```

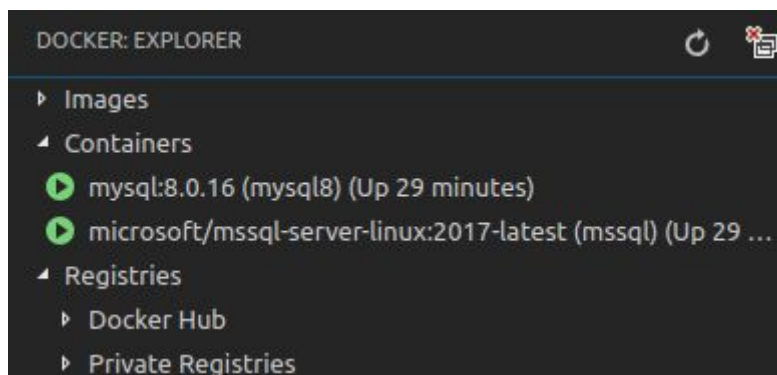
```

container_name: mysql8

MsSql:
  image: microsoft/mssql-server-linux:2017-latest
  environment:
    MSSQL_SA_PASSWORD: mudar@123
    ACCEPT_EULA: "Y"
  ports:
    - 11433:1433
  container_name: mssql

```

Digite: **sudo docker-compose up**  
os container irá subir



Digite: **sudo docker-compose down**  
este comando irá parar os container que está em execução

## Api.Data - Migrações do Entity Framework

Acesse a pasta do projeto `../src/Api.Data`, pois é neste projeto que temos o Entity Framework instalado.

Digite: **dotnet ef --version**

Este comando vai mostrar a versão instalada anteriormente no Nuget

**Entity Framework Core .NET Command-line Tools 2.2.4-servicing-10062**

Digite: **dotnet ef --help**

Este comando vai exibir o help do entity framework

Mas o que faz a migração?

A migração ele cria o banco de dados e suas tabelas configurada anteriormente nas Entidades e Mapeamento.

O Entity Framework ele instância o contexto e pega as Entidades e Mapeamento com estas informações ele cria uns arquivos de migrações para o projeto.

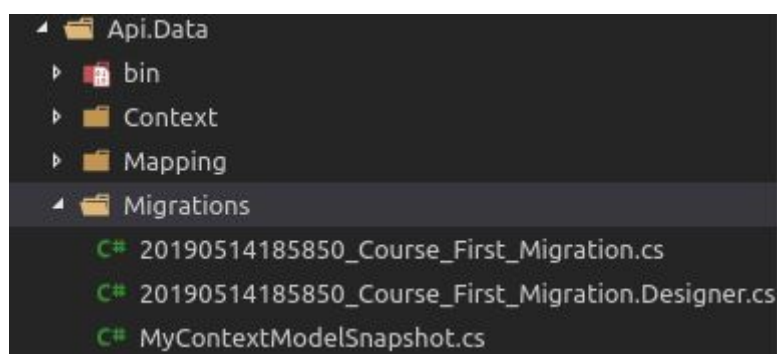
**dotnet ef migrations add** Course\_First\_Migration

se tudo funcionar vai exibir uma mensagem assim:

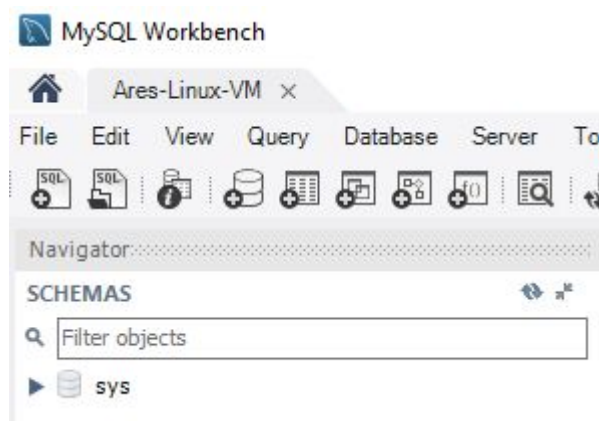
*Done. To undo this action, use 'ef migrations remove'*

Dentro do projeto Api.Data irá criar automaticamente uma pasta chamada Migrations onde ele coloca todos os arquivos gerados como mostra a imagem abaixo, observe que os arquivos começa com data no padrão timestamp.

Todas as classes de migração tem metodos Up e Down, caso precise reverter o comando.



No banco de dados não existe banco de dados referente a este projeto como a imagem abaixo demonstra:



Digite o comando: **dotnet ef database update**

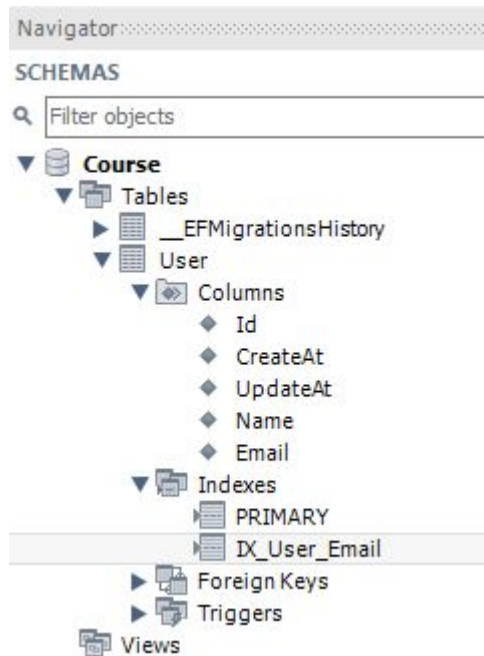
se funcionar irá exibir a seguinte mensagem

*Applying migration '20190514185850\_Course\_First\_Migration'.*

*Done.*

No banco de dados será criado um banco de dados com o nome Course e uma tabela User com as Colunas Id, CreateAt, UpdateAt que está na BaseEntity e Name, email que está na Entidade User.

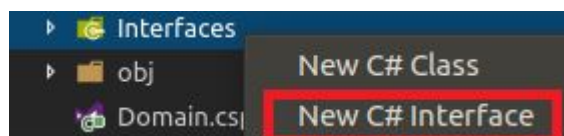




Concluímos, uma etapa muito importante de nosso projeto a camada de Data está comunicando com o banco de dados MySQL criando as migrações baseada no Mapeamento e nas Entidades da Camada de Domínio.

## Api.Domain - Implementação da Interface de Repositório

Criar uma Interface, na pasta Interfaces da Api.Domain.



```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Api.Domain.Entities;

namespace Api.Domain.Interfaces
{
    public interface IRepository<T> where T : BaseEntity
    {
        Task<T> InsertAsync(T item);
        Task<T> UpdateAsync(T item);
        Task<bool> DeleteAsync(Guid id);
        Task<T> SelectAsync(Guid id);
        Task<IEnumerable<T>> SelectAsync();
    }
}
```

```
}
```

Esta interface será o contrato do CRUD do Repositório que será implementado na Camada API.Data.

A Classe espera uma Entidade que tenha a herança da classe BaseEntity, que alguns métodos Recebem como parâmetro a Entidade e alguns faz o retorno do mesmo.

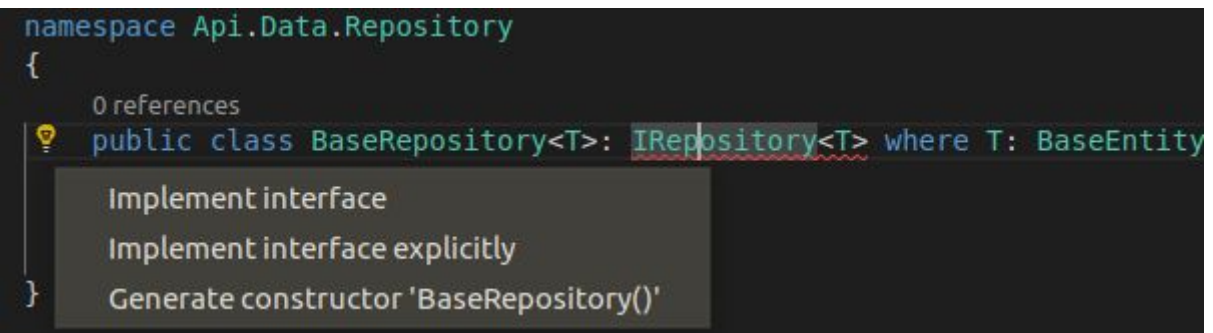
## Api.Data - Repositório

Na camada de Data acesse a Pasta de Repository e criar uma nova classe com o nome BaseRepository.cs

```
using Api.Domain.Entities;
using Api.Domain.Interfaces;

namespace Api.Data.Repository
{
    public class BaseRepository<T>: IRepository<T> where T: BaseEntity
    {
    }
}
```

Ao Criar esta classe você está passando em <T> uma classe de entidade que tenha implementação da BaseEntity, pois estamos construindo um CRUD genérico.



```
namespace Api.Data.Repository
{
    0 references
    public class BaseRepository<T>: IRepository<T> where T: BaseEntity
    {
    }
}
```

The context menu options shown are:

- Implement interface
- Implement interface explicitly
- Generate constructor 'BaseRepository()'

para implementar esta classe baseada nos métodos da interface de um clique na Lâmpada e clique na opção Implement Interface.

Agora temos o esqueleto da classe repository implementada.

No Construtor desta classe vamos receber por injeção de dependência o contexto do banco de dados (MyContext)

O Construtor recebe o contexto e repassa o contexto e dataset para as variáveis locais.

```
protected readonly MyContext _context;
private DbSet<T> _dataset;

public BaseRepository (MyContext context) {
    _context = context;
    _dataset = _context.Set<T> ();
}
```

Método InsertAsync, recebe uma Entidade esta entidade como está herdada da BaseEntity então já temos algumas informações como Id, CreateAt, UpdateAt.

Primeiro passa é certificar se o Id está vazio, caso esteja o método já preenche um novo Guid, preenche a data CreateAt faz a inserção da entidade no DbSet e faz o commit no contexto do Entity Framework como método SaveChangesAsync() o Bloco está dentro de um try...catch que sobe a exceção para cima.

```
public async Task<T> InsertAsync(T item)
{
    try
    {
        if (item.Id == Guid.Empty)
        {
            item.Id = Guid.NewGuid();
        }

        item.CreateAt = DateTime.UtcNow;
        _dataset.Add(item);
        await _context.SaveChangesAsync();
    }
    catch (Exception ex)
    {
        throw ex;
    }
    return item;
}
```

Método UpdateAsync, recebe uma Entidade primeiramente o método certifica-se se o registro existe na base caso exista ele o \_context.Entry irá marcar para o EF fazer a instrução de Atualização no banco quando o contexto fizer o commit como método SaveChangesAsync() o Bloco está dentro de um try...catch que sobe a exceção para cima.

```
public async Task<T> UpdateAsync(T item)
{
    item.UpdateAt = DateTime.Now;
    var result = await _dataset.SingleOrDefaultAsync(p => p.Id.Equals(item.Id));

    if (result == null)
        return null;
}
```

```
item.CreateAt = result.CreateAt;
try
{
    _context.Entry(result).CurrentValues.SetValues(item);
    await _context.SaveChangesAsync();
}
catch (Exception ex)
{
    throw ex;
}
return item;
}
```

Método DeleteAsync recebe um Id, certifica se o mesmo existe na base se existir remove do dbSet e faz um commit.

```
public async Task<bool> DeleteAsync(Guid id)
{
    var result = await _dataset.SingleOrDefaultAsync(p => p.Id.Equals(id));
    try
    {
        if (result != null)
        {
            _dataset.Remove(result);
            await _context.SaveChangesAsync();
            return true;
        }
        else
        {
            return false;
        }
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Método ExistAsync recebe um Id e verificar se o mesmo exista no banco de dados com o método AnyAsync do EF, retornando True/False

```
public async Task<bool> ExistAsync(Guid id)
{
    return await _dataset.AnyAsync (p => p.Id.Equals (id));
}
```

Método `SelectAsync` recebe um `Id` e faz um select utilizando `SingleOrDefaultAsync`, caso o registro não exista na base este método irá retornar `NULL`.

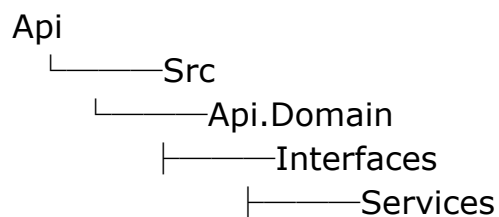
```
public async Task<T> SelectAsync(Guid id)
{
    try
    {
        return await _dataset.SingleOrDefaultAsync(p => p.Id.Equals(id));
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

Método `SelectAsync` retorna a lista da Entidade passada para este contexto.

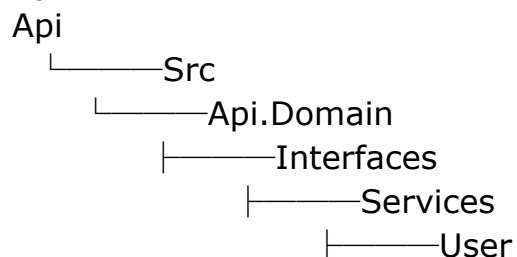
```
public async Task<IEnumerable<T>> SelectAsync()
{
    try
    {
        return await _dataset.ToListAsync();
    }
    catch (Exception ex)
    {
        throw ex;
    }
}
```

## Api.Domain - Interface para Services

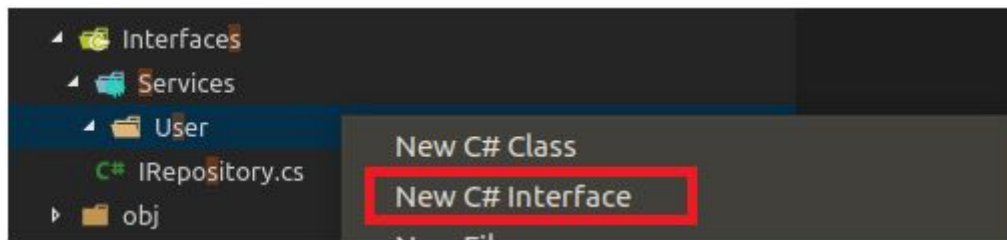
Criar uma pasta com o nome de `Services` dentro da pasta `Interfaces`.



Agora vamos criar uma outra pasta dentro de `Services` com o nome de `User`



Criar uma Interface, com o nome de IUserService.cs



```
namespace Api.Domain.Interfaces.Services.User
{
    public interface IUserService
    {
        Task<UserEntity> Get(Guid id);
        Task<IEnumerable<UserEntity>> GetAll();
        Task<UserEntity> Post(UserEntity user);
        Task<UserEntity> Put(UserEntity user);
        Task<bool> Delete(Guid id);
    }
}
```

## Api.Service - Regras de Negócio

A Camada de Service precisa possuir referências a Api.Domain, Api.Data, Api.CrossCutting acesse o terminal do visual code e acesse a pasta api/src\$

Digite:

**dotnet add Api.Service reference Api.Domain**

**dotnet add Api.Service reference Api.Data**

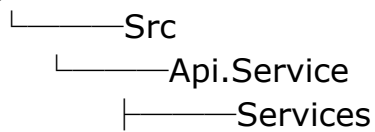
**dotnet add Api.Service reference Api.CrossCutting**

Caso queira verificar as novas referência, acesso o arquivo Service.csproj

```
1 <Project Sdk="Microsoft.NET.Sdk">
2   <ItemGroup>
3     <ProjectReference Include="..\Api.Domain\Domain.csproj" />
4     <ProjectReference Include="..\Api.Data\Data.csproj" />
5     <ProjectReference Include="..\Api.CrossCutting\CrossCutting.csproj" />
6   </ItemGroup>
7   <PropertyGroup>
8     <TargetFramework>netcoreapp2.2</TargetFramework>
9   </PropertyGroup>
10 </Project>
11
```

Criar uma pasta com o nome de Services

Api



Criar uma Classe com o nome UserService.cs, e implementa a interface IUserService, lembrando que todas as Interfaces pertence a camada de Domain.

```

using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Api.Domain.Entities;
using Api.Domain.Interfaces;
using Api.Domain.Interfaces.Services.User;

namespace Api.Service.Services
{
    public class UserService : IUserService
    {
        private IRepository<UserEntity> _repository;
        public UserService(IRepository<UserEntity> repository)
        {
            _repository = repository;
        }
        public async Task<UserEntity> Get(Guid id)
        {
            return await _repository.SelectAsync(id);
        }
        public async Task<IEnumerable<UserEntity>> GetAll()
        {
            return await _repository.SelectAsync();
        }
        public async Task<UserEntity> Post(UserEntity user)
        {
            return await _repository.InsertAsync(user);
        }
        public async Task<UserEntity> Put(UserEntity user)
        {
            return await _repository.UpdateAsync(user);
        }
        public async Task<bool> Delete(Guid id)
        {
            return await _repository.DeleteAsync(id);
        }
    }
}

```

## Api.Application - UserController

Na Camada de Aplicação precisa existir referências a Api.Domain, Api.Service e Api.CrossCutting.

acesse o terminal do visual code e acesse a pasta api/src\$

Digite:

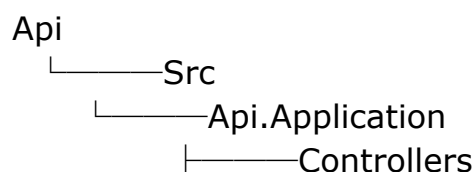
**dotnet add** Api.Application **reference** Api.Domain

**dotnet add** Api.Application **reference** Api.Service

**dotnet add** Api.Application **reference** Api.CrossCutting

Para que uma API exponha seus endpoints ou suas Rotas com os Verbos temos que construir as controllers.

Na pasta Controllers vamos criar uma classe com o nome UserController.cs



**Obs:** Todos os controller precisa ter a convenção de <<Nome>>Controller

```

namespace Api.Application.Controllers
{
    public class UsersController
    {
    }
}
  
```

A controller tem que fazer herança da : ControllerBase

```

using Microsoft.AspNetCore.Mvc;

namespace Api.Application.Controllers
{
    public class UsersController: ControllerBase
    {
    }
}
  
```

Precisamos Adicionar a Rota para esta controller

```
[Route("api/[controller]")]
```



Esta Rota será acessada por <http://localhost:5000/api/users>  
o users vem do nome da classe **usersController**

O atributo [ApiController] pode ser aplicado a uma classe de controlador para habilitar comportamentos específicos à API:

[ApiController]

- Requisito de roteamento de atributo
- Respostas HTTP 400 automáticas
- Inferência de parâmetro de origem da associação
- Inferência de solicitação de várias partes/dados de formulário
- Detalhes do problema dos códigos de status de erro

Esses recursos exigem compatibilidade com a versão 2.1 ou posterior.

Até o momento sua classe de usersController deve estar com as seguinte linhas de código.

```
using Microsoft.AspNetCore.Mvc;

namespace Api.Application.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController: ControllerBase
    {
    }
}
```

Vamos Adicionar um Http do tipo GET para retornar todos os Registros de Usuários.

[HttpGet]

O atributo [HttpGet] Identifica uma ação que dá suporte ao método GET.

ModelState.IsValid

A propriedade IsValid será verdadeira quando os valores forem anexados corretamente ao modelo e nenhuma regra de validação for quebrada no processo de associação.

ArgumentException

O ArgumentException é uma exceção que é gerada quando um dos argumentos fornecidos para um método não é válido.

```
[FromServices] IUserService service
```

O [FromServices] habilita a injeção de um serviço diretamente em um método sem usar a injeção de construtor:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Threading.Tasks;
using Api.Domain.Interfaces.Services.User;
using Microsoft.AspNetCore.Mvc;

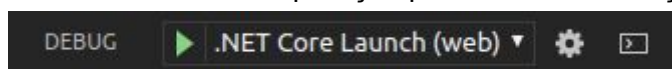
namespace Api.Application.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController : ControllerBase
    {
        [HttpGet]
        public async Task<ActionResult> GetAll([FromServices] IUserService service)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            try
            {
                return Ok(await service.GetAll());
            }
            catch (ArgumentException e)
            {
                return StatusCode((int)HttpStatusCode.InternalServerError, e.Message);
            }
        }
    }
}
```

O Método Get recebe a injeção de dependência do serviço de UserService diretamente ao método, logo depois este método irá validar se o Model State está inválido, se tiver o Método Retorna um BadRequest (400).

Seguindo o código o método irá tentar selecionar todos os registros e retornar um OK (200), em caso de erro é retornado um Internal Error (500)

Vamos Iniciar nossa aplicação para iniciar a execução deste método.



<http://localhost:5000/api/users>

An unhandled exception occurred while processing the request.

InvalidOperationException: No service for type 'Api.Domain.Interfaces.Services.User.IUserService' has been registered.

Ao acessar este endereço api irá retornar um erro que **IUserService** não foi Registrada. Toda e qualquer uso de Injeção de dependência precisa ser registra na classe Startup.cs.

Para deixar o projeto mais robusto iremos colocar estas configurações na camada CrossCutting

## Api.CrossCutting - Configuração de Injeção de Dependência

Na Camada de CrossCutting precisa existir referências a Api.Domain, Api.Service, Api.Data acesse o terminal do visual code e acesse a pasta api/src\$

Digite:

**dotnet add Api.CrossCutting reference Api.Domain**

**dotnet add Api.CrossCutting reference Api.Service**

**dotnet add Api.CrossCutting reference Api.Data**

*Obs: Ao adicionar essas referências pode acontecer alguma referência circular na camada de Services se isto acontecer abra o arquivo Service.csproj e remova a linha que faz referência a CrossCutting*

Adicionar o pacote Nuget para gerenciar as dependencyInjection

Acessar a pasta pasta api/src/Api.CrossCutting\$

```
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection --version 6.1.0
```

Criar uma Pasta com o nome de DependencyInjection

Api

```

├── Src
│   ├── Api.CrossCutting
│       └── DependencyInjection
    
```

Criar uma classe com o nome ConfigureServices.cs

```

using Api.Domain.Interfaces.Services.User;
using Api.Service.Services;
using Microsoft.Extensions.DependencyInjection;

namespace Api.CrossCutting.DependencyInjection
{
    public class ConfigureServices
    
```

```

{
    public static void ConfigureDependenciesService(IServiceCollection serviceCollection)
    {
        serviceCollection.AddTransient<IUserService, UserService>();
    }
}

```

Se tentar rodar novamente aplicação depois destas configurações infelizmente irá dar um outro erro reclamando da Injeção de dependência do Repositório

An unhandled exception occurred while processing the request.

InvalidOperationException: Unable to resolve service for type  
'Api.Domain.Interfaces.IRepository'1[Api.Domain.Entities.UserEntity]' while attempting to activate  
'Api.Service.Services.UserService'.

Continuando na camada de cross cutting vamos criar a configuração das dependências de repositório.

Criar o arquivo ConfigureRepository.cs, na mesma pasta de DependencyInjection

```

using Api.Data.Repository;
using Api.Domain.Entities;
using Api.Domain.Interfaces;
using Microsoft.Extensions.DependencyInjection;

namespace Api.CrossCutting.DependencyInjection
{
    public class ConfigureRepository
    {
        public static void ConfigureDependenciesRepository(IServiceCollection serviceCollection)
        {
            serviceCollection.AddScoped(typeof(IRepository<>), typeof(BaseRepository<>));
        }
    }
}

```

## Api.Application - Startup

Para finalizar as configurações acessamos o arquivo Startup.cs da camada Api.Application, localizar o Método ConfigureServices e Configurar as Dependências de Serviço e Repositório.

```

using Api.CrossCutting.DependencyInjection;
....

```

```

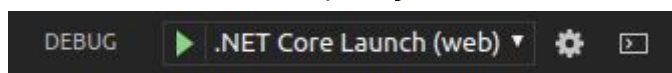
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<MyContext>(
        options=> options.UseMySQL("Server=localhost;Port=3306;Database=Course;Uid=root;Pwd=admin")
    );

    ConfigureServices.ConfigureDependenciesService(services);
    ConfigureRepository.ConfigureDependenciesRepository(services);

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}

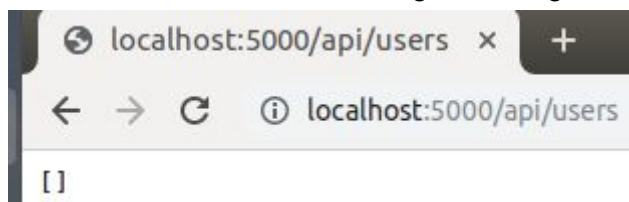
```

Executar novamente aplicação.



<http://localhost:5000/api/users>

o Retorno no browse deve ser igual a imagem abaixo



## Api.Application - UserController

Continuando na construção do User Controller, vamos criar o método Get por id, para isto precisamos adicionar um atributo Route para o método passando um parâmetro.

A Rota principal é esta **/api/users**

```
[Route("{id}", Name = "GetWithId")]
```

se adicionar o atributo route para o método colocando o parâmetro a rota irá ficar assim  
**/api/users/7e5784cd-e6ec-4698-bbba-58d2051c2faf**

```

[HttpGet]
[Route("{id}", Name = "GetWithId")]
public async Task<ActionResult> Get(Guid id, [FromServices] IUserService service)
{
    if (!ModelState.IsValid)
    {

```

```

        return BadRequest(ModelState);
    }

    try
    {
        return Ok(await service.Get(id));
    }
    catch (ArgumentException e)
    {
        return StatusCode((int)HttpStatusCode.InternalServerError, e.Message);
    }
}

```

O Método POST (*INSERT*) Recebe por parâmetro no FromService como injeção de dependência a interface de IUserService e FromBody da Entidade de Usuário em formato de JSON.

Antes de Realizar o Post na Service é Validado do ModelState caso seja invalidado é retornado um BadRequest(400)

Ao Executar o Post na Camada de Service se as Camada abaixo retornar algum tipo de erro é retornado um InternalServerError (500), ao Gravar o Retorno seja o objeto gravado é Retornado um Created (201) com um Link no Header para fazer a consulta do Registro cadastrado.

Se Retornar um objeto null na gravação é Retornado um BadRequest (400)

```

[HttpPost]
public async Task<ActionResult> Post([FromBody] UserEntity user,
                                     [FromServices] IUserService service)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        var result = await service.Post(user);
        if (result != null)
        {
            return Created(
                new Uri(Url.Link("GetWithId", new { id = result.Id })), result);
        }
        else
        {
            return BadRequest();
        }
    }
    catch (ArgumentException e)
    {
        return StatusCode((int)HttpStatusCode.InternalServerError, e.Message);
    }
}

```

```
}
```

O Método PUT (*UPDATE*) Recebe por parâmetro no FromService como injeção de dependência a interface de IUserService e FromBody da Entidade de Usuário em formato de JSON.

Antes de Realizar o Put na Service é Validado do ModelState caso seja invalidado é retornado um BadRequest(400)

Ao Executar o Put na Camada de Service se as Camada abaixo retornar algum tipo de erro é retornado um InternalServerError (500), ao Gravar o Retorno seja o objeto gravado é Retornado um OK(200).

Se Retornar um objeto null na gravação é Retornado um BadRequest (400)

```
[HttpPut]
public async Task<ActionResult> Put([FromBody] UserEntity user,
                                     [FromServices] IUserService service)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        var result = await service.Put(user);
        if (result != null)
        {
            return Ok(result);
        }
        else
        {
            return BadRequest();
        }
    }
    catch (ArgumentException e)
    {
        return StatusCode((int)HttpStatusCode.InternalServerError, e.Message);
    }
}
```

O Método DELETE Recebe por parâmetro no FromService como injeção de dependência a interface de IUserService e Id no formado do Guid.

Antes de Realizar o Delete na Service é Validado do ModelState caso seja invalidado é retornado um BadRequest(400)

Ao Executar o Delete na Camada de Service se as Camada abaixo retornar algum tipo de erro é retornado um InternalServerError (500), ao deletar o registro é Retornado um OK(200).

```
[HttpDelete ("{id}")]
public async Task<ActionResult> Delete(Guid id,
                                         [FromServices] IUserService service)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    try
    {
        return Ok(await service.Delete(id));
    }
    catch (ArgumentException e)
    {
        return StatusCode((int)HttpStatusCode.InternalServerError, e.Message);
    }
}
```

**GET**

<http://localhost:5000/api/users>

**POST**

<http://localhost:5000/api/users>

Json para enviar no Corpo (Body) da Requisição

```
{
  "name": "Marcos",
  "email": "teste@marcos.com"
}
```

Retorno do Post (Insert)

```
{
  "name": "Marcos",
  "email": "teste@marcos.com",
  "id": "c2a12fb0-8394-4138-8fef-be9efe101322",
  "createAt": "2019-05-16T12:50:39.624006",
  "updateAt": "2019-05-16T09:58:35.4381294-03:00"
}
```

**OBS:** Todos os Id são Gerados automaticamente, então efetuar a troca dos IDs para funcionar as requisições abaixo.

**GET**

<http://localhost:5000/api/users/c2a12fb0-8394-4138-8fef-be9efe101322>

**PUT**

<http://localhost:5000/api/users>

Json para enviar no Corpo (Body) da Requisição



```
{
  "id": "c2a12fb0-8394-4138-8fef-be9efe101322", //Aqui tem que modificar com ID Atual
  "name": "Marcos_ALTERADO",
  "email": "teste@marcos.com"
}
```

Retorno do Put (Update)

```
{
  "name": "Marcos_ALTERADO",
  "email": "teste@marcos.com",
  "id": "c2a12fb0-8394-4138-8fef-be9efe101322",
  "createAt": "2019-05-16T12:50:39.624006",
  "updateAt": "2019-05-16T09:58:35.4381294-03:00"
}
```

## DELETE

<http://localhost:5000/api/users/c2a12fb0-8394-4138-8fef-be9efe101322>

Retorno do Delete

true

## Configurando o uso do Swagger

Swagger é para geração de documentação em APIs REST criadas com o ASP.NET Core. O exemplo apresentado nesta ocasião fazia uso de uma versão beta do pacote Swashbuckle, além de envolver ajustes no build do projeto, na classe Startup na Action responsável pelo tratamento de requisições.

Acessar o Terminal do VS-Code e a Pasta de Api.Application  
/src/Api.Application\$

```
dotnet add package Swashbuckle.AspNetCore --version 4.0.1
```

Modificar a classe **Startup.cs**

Método **ConfigureServices**

```
using Swashbuckle.AspNetCore.Swagger;
```

```
public void ConfigureServices(IServiceCollection services)
{
    . . .

    //Configurando o serviço de documentação do Swagger
    services.AddSwaggerGen(c =>
```

```

    {
        c.SwaggerDoc("v1",
            new Info
            {
                Title = "AspNetCore 2.2",
                Version = "v1",
                Description = "Exemplo de API REST criada com o ASP.NET Core",
                Contact = new Contact
                {
                    Name = "Marcos Fabricio Rosa",
                    Url = "https://github.com/mfrinfo"
                }
            });
    });

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}

```

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    . . .

    // Ativando middlewares para uso do Swagger
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.RoutePrefix = string.Empty;
        c.SwaggerEndpoint("/swagger/v1/swagger.json",
            "Projeto em AspNetCore 2.2");
    });

    // Redireciona o Link para o Swagger, quando acessar a rota principal
    var option = new RewriteOptions();
    option.AddRedirect("^$", "swagger");
    app.UseRewriter(option);

    . . .
    app.UseMvc();
}

```

## Configurando Entity Framework para SQL-Server 2017

No Terminal do Visual Code acessar a Pasta

/src/Api.Data\$

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 2.2.4
```

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer.Design --version 1.1.6
```

Acessar a Classe Startup.cs na camada de Api.Application

Comentar a Linha do .UseMySQL e Adicionar a Linha com string de conexão com .UseSqlServer

```
//services.AddDbContext<MyContext>(options =>
options.UseMySQL("Server=localhost;Port=3306;Database=Course;Uid=root;Pwd=admin"));
services.AddDbContext<MyContext>(options =>
options.UseSqlServer("Server=127.0.0.1,11433;Database=Course;User
Id=sa;Password=mudar@123"));
```

Configurar a Classe ContextFactory.cs da Camada Api.Data, comentando as linhas do .UseMySQL e Acrescentar a .UseSqlServer

```
public MyContext CreateDbContext(string[] args)
{
    //Usado para Criar as Migrações
    //var connectionString="Server=localhost;Port=3306;Database=Course;
    //    Uid=root;Pwd=admin";
    var connectionString = "Server=127.0.0.1,11433;Database=Course;
    //    User Id=sa;Password=mudar@123";
    var optionsBuilder = new DbContextOptionsBuilder<MyContext>();
    //optionsBuilder.UseMySQL(connectionString);
    optionsBuilder.UseSqlServer(connectionString);
    return new MyContext(optionsBuilder.Options);
}
```

No Terminal do Visual Code acessar a Pasta

/src/Api.Data\$

**dotnet ef database update**

## Configurando appSettings.json - GetConnectionString

Abrir Arquivo **AppSettings.json** da Camada Api.Application, e coloca o código do ConnectionsStrings.

Esta ConnectionsStrings irá conter o DbAtivo que irá indicar qual banco de dados ativo e duas conexão uma apontando para MySQL e a outra apontando para MySqlConnection.

```
{
  "ConnectionStrings": {
    "DbAtivo": "MySQL",
```

```

    "MySQLConnection": "Server=localhost;Port=3306;Database=Course;Uid=root;Pwd=admin",
    "MsSqlConnection": "Server=127.0.0.1,11433;Database=Course;User Id=sa;Password=mudar@123"
  },
  ....

```

Abrir a classe **Startup.cs** na camada Api.Application

No método `ConfigureServices` vamos descobrir qual `DbAtivo` para o mesmo configurar automaticamente qual das conexões irá configurar o `MyContext`.

```

public void ConfigureServices(IServiceCollection services)
{
    if (Configuration.GetConnectionString("DbAtivo")=="MsSQL")
    {
        services.AddDbContext<MyContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("MsSqlConnection")));
    }
    else
    {
        services.AddDbContext<MyContext>(options =>
            options.UseMySQL(Configuration.GetConnectionString("MySQLConnection")));
    }
}

```

Abrir a classe `MyContext.cs`

No construtor Adicionar a migração quando a API fizer a conexão com o banco de dados

Sem a Migração

```

public MyContext(DbContextOptions<MyContext> options) : base(options) { }

```

Com a Migração

```

public MyContext(DbContextOptions<MyContext> options) : base(options)
{
    Database.Migrate();
}

```

## ANEXOS

### Pré-Requisitos

No Exemplo os Bancos de Dados MySQL e SQL-Server ou utilizar em Docker.  
Caso usar o Docker tem que fazer a instalação.

<https://www.docker.com/get-started>

My-SQL Community 8

<https://dev.mysql.com/downloads/mysql/>

<https://dev.mysql.com/downloads/workbench/>

MS-SQL 2017

<https://www.microsoft.com/pt-br/sql-server/sql-server-editions-express>

<https://docs.microsoft.com/pt-br/sql/azure-data-studio/download?view=sql-server-2017>

AspNetCore 2.2

<https://dotnet.microsoft.com/download/dotnet-core/2.2>

Visual Code

<https://code.visualstudio.com/download>

Instalar três plugins:

C#



C# Extension

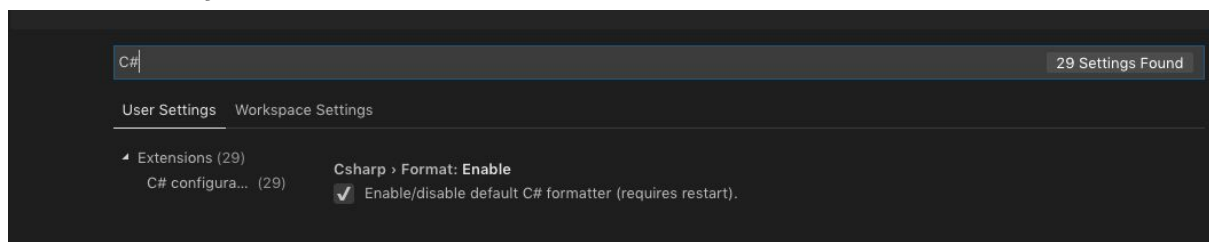


vscode-icons

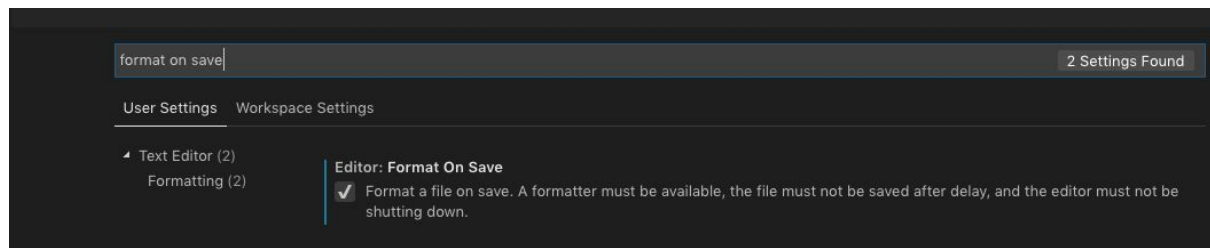


File / preferences / Settings

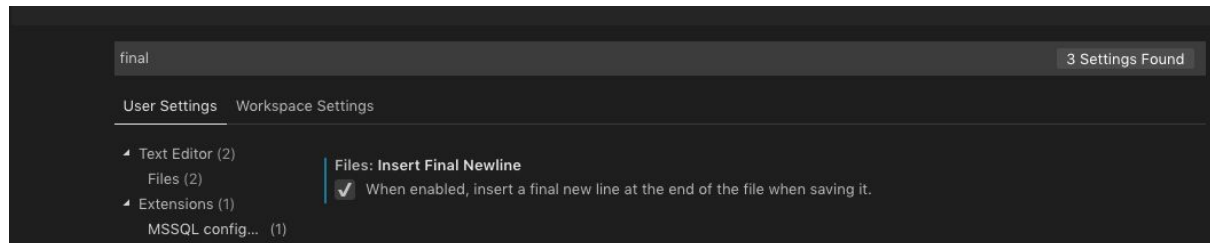
Ativa a formatação padrão do C#



Ativa ao Salvar o Arquivo será formatado automaticamente



Ao Salvar o arquivo deixa uma linha no final do arquivo.



## Códigos de Retorno

Código do Status HTTP (Status-code)	Significado do código HTTP (Reason-Phrase)	Significado do código HTTP
100	Continue	Continuar
101	Switching Protocols	Mudando Protocolos
102	Processing	Processando
200	Ok	Ok
201	Created	Criado
202	Accepted	Aceito
203	Non-Authoritative Information	Não autorizado
204	No Content	Nenhum Conteúdo
205	Reset Content	Resetar Conteúdo
206	Partial Content	Conteúdo Parcial
300	Multiple Choices	Múltipla Escolha
301	Moved Permanently	Movido Permanentemente
302	Found	Encontrado
303	See Other	Veja outro
304	Not Modified	Não modificado
305	Use Proxy	Use Proxy
306	Proxy Switch	Proxy Trocado
400	Bad Request	Solicitação Inválida

401	Unauthorized	Não autorizado
402	Payment Required	Pagamento necessário
403	Forbidden	Proibido
404	Not Found	Não encontrado
405	Method Not Allowed	Método não permitido
406	Not Acceptable	Não aceito
407	Proxy Authentication Required	Autenticação de Proxy Necessária
408	Request Time-out	Tempo de solicitação esgotado
409	Conflict	Conflito
410	Gone	Perdido
411	Length Required	Duração necessária
412	Precondition Failed	Falha de pré-condição
413	Request Entity Too Large	Solicitação da entidade muito extensa
414	Request-URL Too Large	Solicitação de URL muito Longa
415	Unsupported Media Type	Tipo de mídia não suportado
416	Request Range Not Satisfiable	Solicitação de faixa não satisfatória
417	Expectation Failed	Falha na expectativa
500	Internal Server Error	Erro do Servidor Interno
501	Not Implemented	Não implementado
502	Bad Gateway	Porta de entrada ruim
503	Service Unavailable	Serviço Indisponível
504	Gateway Time-out	Tempo limite da Porta de Entrada
505	HTTP Version Not Supported	Versão HTTP não suportada

