**Extensões Úteis:**

- NuGet Gallery: Deixa o Visual Studio Code parecido com o Visual Studio para baixar as extensões úteis para nosso projeto. Ao usar Ctrl + Shift + P, você abre a ferramenta e consegue ver com mais facilidade o que está instalado e instalar novas funcionalidades.

**Ferramentas Úteis:**

- Microsoft.EntityFrameworkCore

- Microsoft.EntityFrameworkCore.Tools

- Microsoft.EntityFrameworkCore.Design

- Microsoft.EntityFrameworkCore.SqlServer

**GLOBAL.JSON**

O arquivo `global.json` é utilizado para definir a versão específica do SDK que desejamos utilizar em um projeto. Isso é especialmente útil quando há múltiplas versões de SDK instaladas na máquina.

**Formato do Arquivo:**

{

"sdk": {

"version": "5.0.102"

}

}

Como Criar o `global.json` via .NET CLI:

Para criar o arquivo `global.json`, utilize o seguinte comando no terminal:

dotnet new global.json --sdk-version 5.0.102

**Sobrescrevendo um `global.json` Existente:**

Se você deseja sobrescrever um arquivo `global.json` já existente, adicione a opção `--force` ao final do comando.

MIGRATIONS

O migrations no Entity Framework Core é uma ferramenta que permite gerenciar alterações no esquema do banco de dados de forma automatizada. Ele registra as alterações feitas nos modelos de dados e as traduz em comandos SQL para atualizar o banco de dados correspondente. Isso facilita o controle das mudanças na estrutura do banco de dados ao longo do desenvolvimento de um aplicativo, garantindo a consistência entre o modelo de dados e o esquema do banco de dados.

**Dotnet ef migrations list** – Vai listar todas as migrations disponíveis no projeto ou retornar nada caso não tenha migrations.

**Dotnet ef migrations add init -o Data/migrations** – Esse comando cria a migrations com base as classes modelos do seu projeto. **Dotnet ->** “invoca o dotnet” **ef -> “**Dentro do dotnet foi invocado o entity framework**” migrations ->** “dentro de ef foi invocado o migrations” **add ->** “Comando do migratiuons para adicionar uma migration” **init ->** ”o nome da migrations obs: pode ser qualquer nome” **-o Data/Migrations ->** “ esse comando cria a migrations dentro da pasta data e dentro da pasta migrations se as pastas não existirem ele cria as mesmas”

**Dotnet ef migrations remove** – Caso queira remover a migration criada.

**Dotnet ef database update** – Esse comando pega a migrations criada no projeto e sobe para um banco.

ADICIONAL:

O código a seguir é feito dentro da minha Conectioncontext e serve para dizer ao migration que estamos adicionando duas foreingkey dentro da tabela palestrante Evento

  protected override void OnModelCreating(ModelBuilder modelBuilder)

        {

            modelBuilder.Entity<PalestranteEvento>().HasKey(PE => new {PE.EventoID, PE.PalestranteID});

        }

**Camadas**

Organizar um projeto em camadas oferece várias vantagens:

1. **Separação de Responsabilidades:** Cada camada tem um propósito específico, o que facilita a compreensão do código e a manutenção do sistema. Por exemplo, uma camada pode lidar com a lógica de negócios, outra com a apresentação da interface do usuário e outra com acesso a dados.

2. **Reusabilidade**: Camadas bem definidas facilitam a reutilização de código. Por exemplo, se a camada de acesso a dados é separada, ela pode ser usada por diferentes partes do sistema sem a necessidade de reimplementação.

3. **Escalabilidade**: Camadas modulares permitem que o sistema seja escalado de forma mais eficiente. Por exemplo, se a camada de interface do usuário precisar ser atualizada, isso pode ser feito sem afetar as outras camadas.

4. **Testabilidade**: É mais fácil testar unidades individuais do sistema quando elas estão em camadas separadas. Isso permite testes de unidade mais eficazes e também facilita a realização de testes de integração.

5. **Manutenção**: Com uma estrutura em camadas, é mais fácil realizar alterações ou atualizações em partes específicas do sistema sem afetar outras partes. Isso ajuda na manutenção a longo prazo do projeto.

Em resumo, organizar um projeto em camadas promove uma arquitetura mais limpa, modular e flexível, o que facilita o desenvolvimento, teste e manutenção do software.

No Visual Studio, temos o arquivo .sln que servirá para referenciar as camadas criadas. Já no VS Code, devemos criar esse arquivo manualmente. Por exemplo:

**dotnet new sln -n ProEventos**

Esse arquivo de SLN deve ser criado dentro de Sln onde mostra todas as camadas do seu projeto. Ele servirá para que possamos referenciar todas as camadas do projeto dentro dele.

Então devemos criar as camadas. Podemos fazer isso usando o comando dotnet new classlib, como mostrado no exemplo abaixo:

dotnet new classlib -n ProEventos.Persistence

dotnet new classlib -n ProEventos.Domain

dotnet new classlib -n ProEventos.Application

Nesse modelo de camadas, a ideia é que a API entre em contato com a Application, que por sua vez utilize o Domain e acesse também a Persistência.

**Referenciando dentro do SLN**

Após todas as camadas do projeto criadas, podemos então referenciá-las dentro do sln usando o seguinte código.

**dotnet sln “nome da sln” add “nome da camada”**

Com isso devemos referenciar todo o projeto dentro da própria SLN.

Após fazer isso devemos referenciar o projeto em si. Por exemplo: associar a API ao Application. Fazemos isso usando o seguinte comando:

**dotnet add “ProEventos.API/ProEventos.API.csproj” reference ProEventos.Application**

Após fazer essa referência inicial pode-se notar que dentro do csproj de api foi adicionado a referência do application. Com o código já gerado podemos copiá-lo para as outras camadas assim ligando nosso projeto inteiro.

**<ItemGroup>**

**<ProjectReference Include="..\ProEventos.Application\ProEventos.Application.csproj" />**

**</ItemGroup>**

Exemplo:

* API referencia o Application
* Application referencia Domain e Persistence
* Persistence recebe Domain
* E por fim o Domain não recebe nenhum

Conclusão:

1. **Adicionar um projeto a uma solução**: Quando você adiciona um projeto a uma solução usando o comando dotnet sln <nome da solução> add <nome do projeto>, você está basicamente informando ao .NET que esses projetos estão relacionados e fazem parte do mesmo aplicativo. Isso permite que você gerencie todos os projetos relacionados em um único lugar.
2. **Referenciando um projeto a partir de outro projeto**: Digamos que você tenha dois projetos, ProjetoA e ProjetoB, e o ProjetoA precisa usar algum código do ProjetoB. Nesse caso, você precisaria adicionar uma referência do ProjetoB ao ProjetoA. Isso é feito usando o comando dotnet add <ProjetoA> reference <ProjetoB>. Isso cria um link entre os dois projetos, permitindo que o ProjetoA use o código do ProjetoB.

**INTERFACE**

O que é e para que serve a interface em Web APIs?

Uma interface em C# é um tipo que define um contrato que outras classes devem seguir. Ela especifica os métodos que uma classe concreta deve implementar. Em uma WebAPI, interfaces são frequentemente usadas para definir contratos para os controladores, fornecendo uma estrutura para expor endpoints HTTP consistentes e bem definidos. Elas ajudam na separação de preocupações e permitem que diferentes partes do código comuniquem-se de forma padronizada, facilitando a manutenção e a escalabilidade do sistema.

Ao declarar uma interface, seguimos a seguinte assinatura:

public interface IItemRepository

{

void Add(ItemModel item);

List<ItemModel> Get();

ItemModel Get(int id);

void Update(ItemModel item);

void Delete(int id);

}

Porém, dessa forma, precisaríamos fazer um método para cada classe do nosso projeto, o que muitas vezes é inviável. Para facilitar, podemos criar métodos genéricos da seguinte forma:

void Add<T>(T entity) where T : class;

Com isso, o método Add recebe qualquer classe do projeto.

-void:Indica que o método não possui retorno.

- Add<T>:\*\* Add é o nome do método, que poderia ser qualquer coisa, e `<T>` indica que queremos adicionar um item genérico.

-(T entity):\*\* `entity` é o nome do parâmetro que você passa para o método quando o chama. É como uma caixa que você usa para enviar informações para dentro do método. O nome `entity` é apenas uma convenção, você pode dar qualquer nome que quiser, mas é uma boa prática escolher um nome que seja descritivo do que está sendo passado para o método.

**Próxima aular revisar a aula 63 que fala de interface e persistência entender bem esse conceito da forma que o professor esta ensinando . Fazer a interface**