

Entity Framework (EFC)

Possui duas classes principais

1) DbContext: serve para estabelecer um conexão com o banco, armazenar os dados e fazer as operações com os dados

2) DbSet^{<T>}: serve para definir os dados que serão mapeados no banco, no caso, as tabelas do banco

Criando string de conexão com o banco

as tabelas serão criadas de forma automática por causa
de migrations, mas para isso precisamos configurar a string
de conexão

String de conexão → expressão com os parâmetros para conexão ao BD

↓ temos 2 tipos, um com autenticação do SQL server e a
com autenticação do Windows
(trusted connection)

↓ Na string consta a instância, nome do DB, detalhes
de autenticação e outros configurações de acordo
com o ambiente

Primeiro obtemos o nome do servidor e o nome da
instância, podemos fazer isso abrindo o Manager do SQL server →
Serviços do SQL → SQL server que estiver em execução → M2 → propriedades
Serviços, com isso conseguimos o nome da instância e do
servidor local.

Que podemos adicionar VS22 em Exibir → Server explorer.

Para criar uma string a partir do VS22, clicamos em conexões de dados →
adicionando informamos o nome do servidor e/o instância,
da lista clicamos (M2) no banco → propriedades e iremos adicionar string
de conexão.

↳ Consequence of the changes in climate & changes in species diversity are same

↳ Following are some of the changes in climate

↳ Global warming & more rainfall are important factors

↳ More rainfall & more changes in vegetation are random

↳ Consequence of the changes in climate & more changes in species diversity

↳ More changes in climate & more changes in species diversity

↳ Consequence of the changes in climate & more changes in species diversity

↳ Climate change & changes in species diversity due to global warming

(Randomness can not afford due to the dependence/pattern)

--- (i). Migration

--- (ii). Death

Mic... Entomoc... SCL surv

↳ Boxcar pattern & Wedge pattern due to pattern

EF Cen Consequence of

LDI definir o service no startep.cs no ConfigureServices
ls services.AddDbContext<arquivo de contexto>(x => x.name =
do provedor do banco(Configuration.GetConnectionString("name -
do string de conexão que definimos no appsettings"));

O serviço é criado com tempo de vida scoped, ou
seja, o cada request será criado uma nova instância.

Data Annotations

São atributos que podem ser aplicados nos membros do seu classe, eles servem para válidos, o modelo de funcionamento como os dados devem ser exibidos, especificar o localizamento das entidades e sobreescriver as convenções do EFC.

Pode-se definir se algo é requerido no largo, max e min de caracteres, se é um da principais e não outros códigos, como válidos números e e-mails

(~~Exemplos~~) Esses atributos/regex devem ser aplicados nos model

Quando preenchemos nossa classe através de um request HTTP, ele verifica os atributos da classe para saber se os dados do request são válidos, assim a validação ocorre no lado servidor.

(aplicado no projeto)

Migrations

Será para atualizar o banco e mantê-lo sincronizado com o modelo de dados.

Ele pode: atualizar o banco, personalizar migração, remover uma migração, reverte uma migração, gerar scripts SQL e aplicar migrações em tempo de execução.

Para usá-lo precisamos do pacote EFCore e para aplicá-lo devemos definir as classes/DBset/Dbcontext...

Poderemos usar no painel de package manager ou o NET CLI

Para aplicar precisamos primeiro criar uma migração e aplicar a migração

COMANDOS

add-migration nome da migração - cria a migração

update-database - aplica a migração

remove-migration - remove a migração

(Quando digitarmos o comando para criá-lo, só geraremos pasto Migration com a configuração das tabelas, para que ele só crie os dados de tabelas sem as configurações (~~as~~) adequado, precisaremos usar o **data anno-**
tations.

Populando as tabelas

Temos vários formas de popular uma tabela, duas delas seriam: fazendo uma nova consulta com as instruções `Insert Into`. E outra seria fazendo um migração vazia e usar os métodos `UP` e `DOWN` e nela utilizar as instruções `Insert Into` no `UP()` e no

(~~SELECT INTO~~ para desfazer a migração)

`DOWN()` `Delete From` para desfazer uma migração

Usamos suas instâncias

(Ver no código)

Banco de dados

(a chave primária
fizou definido como
os ID.)

Chaves

No projeto de lanches, temos as categorias, e os lanches, os lanches possuem uma **chave estrangeira** que é referente a **chave primária** das categorias. Assim, a classe pai é o tabela das categorias e a classe filho é a tabela dos lanches. Daí se, não podemos ter um lanche sem antes definir uma categoria para ele.

Caso deletarmos uma categoria, os lanches ^{já que a categoria} terão uma chave estrangeira (ID da categoria) com um valor inválido, ou seja, não ocorrerá uma violação de chave referencial.

Delete Cascade

Para **evitarmos** esse problema, definimos no nosso migration o **Delete Cascade** (Cascata), isso irá fazer com que, caso (Delete) deletarmos uma categoria, os items que referenciavam a elas serão deletados.

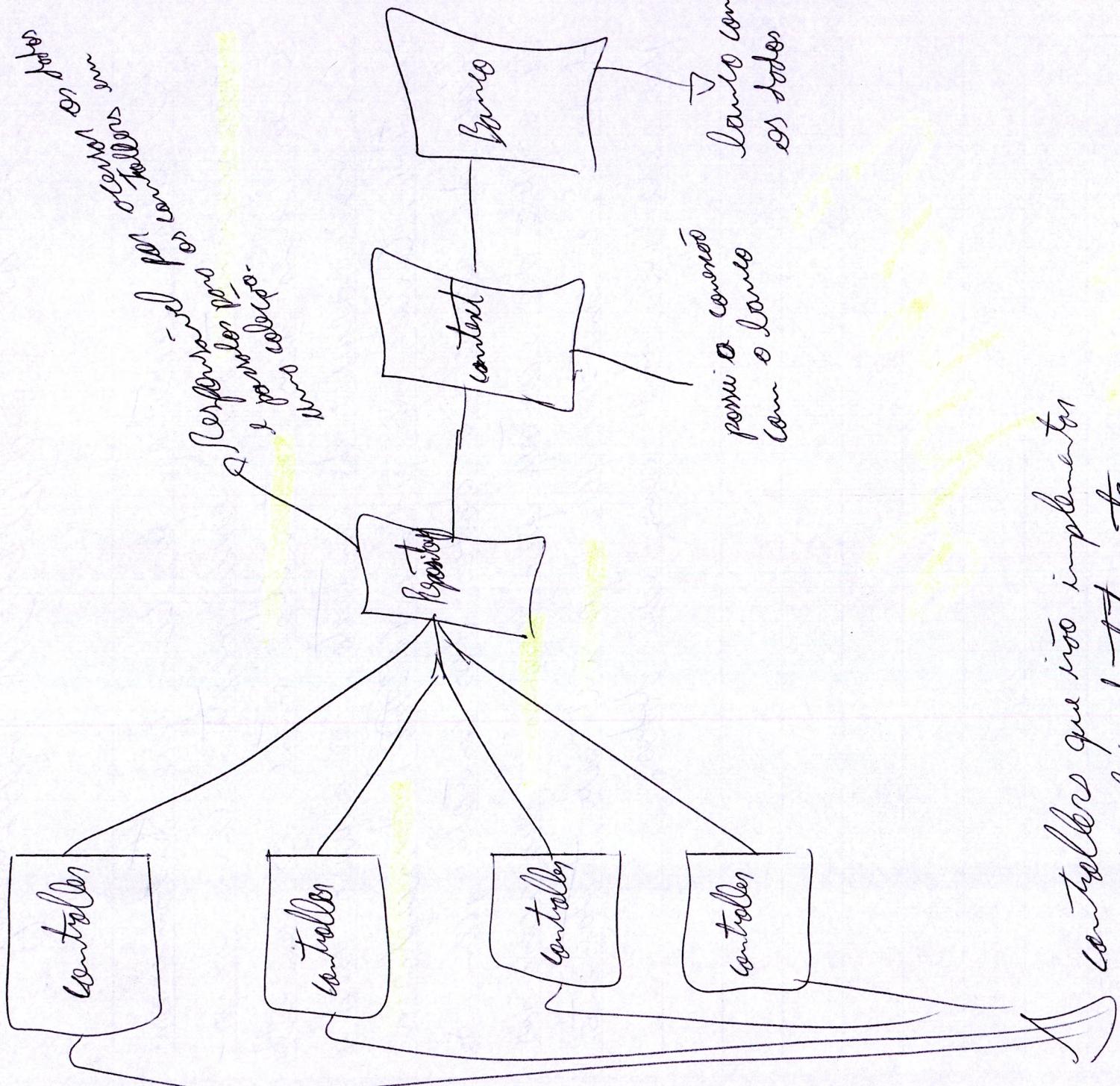
Padrão (Repository) Repository

O padrão repository faz com que tenhamos um baixo acoplamento no nosso projeto, com ele podemos colocar um intermediário entre o controller e o nosso arquivo de contexto, esse arquivo sera o repository, com ele, podemos (~~o controller~~) desacoplar a lógica de acesso a dados dos controllers, com isso o arquivo do repository ficas responsável pelo acesso aos dados e o controller ficas responsável pelo tratamento que ele irá fazer com esses dados.

Exemplo

me

Verso



controller que não implementa seu próprio logico de tratamento

Implação de dependências (DI)

É o ato de ~~delegar~~ solicitarmos os dependências de outro classe.

Por exemplo, podemos criar um construtor que recebe uma classe como parâmetro, podemos atribuir essa classe a uma ~~variável~~ variável e utilizarmos os valores dela.

Se a classe A tem um construtor que recebe a classe B como argumento, podemos armazenar a classe B em uma variável e utilizá-la dentro da classe A. Isso seria a DI, com isso, poderíamos ter vários classes B padronizadas mas com valores diferentes, assim iríamos chamar as vários classes B na classe A sem problemas e com um baixo acoplamento.

Poderíamos instanciar os vários classes B no classe A, mas isso iria gerar um forte acoplamento, porque todo vez que fosse necessário trocar uma das classes B, precisaríamos alterar a classe para alterarmos ela, com a DI basta mudarmos o parâmetro (~~parametro~~) que passamos.

Injeção de dependências Nativas do ASP.NET Core

Usamos a injeção de dependências para diminuir o acoplamento dos nossos projetos.

No ASP.NET fazemos isso um pouco diferente, quem nos fornece a instância no construtor é o ~~container de DI~~ container de DI nativo do ASP.NET.

Para que isso funcione precisamos setar um serviço informando quais as classes que irão usar a DI. Podemos fazer isso no ConfigureService do Program/Startup.

Containers DI

Para que o injecção de dependências pelo container DI ocorra (precisamos) configurar o service primeiro, para isso temos 3 opções de escopo para utilizarmos:
(escopo de vida.)

Transient — gera um novo instâncio todo vez que (houver uma requisição) um serviço é solicitado ao provedor de serviços.

Scoped — Um novo instâncio é gerado todo vez que houver um request.

Singleton — Apenas um instâncio do serviço é criado caso ainda não exista um.

- various products will have own numbers
so company has their own set of numbers

W.M.

- symbol for

probability of numbers of snow = probability of snow = probability of snow / probability of no snow
symbol for no probability of snow = probability of no snow

worked as copy
numbers, or symbols of
probability

probability of numbers of snow = $\frac{1}{2}$ (probability) = 0.50

worked as copy as opposed to
numbers as counting good more numbers and
own numbers as

W.M.

Novo tabela no lance

Propriedades

- ↳ Para retornar o id, podemos definir o nome da propriedade como Id ou colocarmos o nome da tabela e Id no final. Com isso será gerado o id que não sera a chave primária.
- ↳ Caso seja definido uma propriedade com um tipo já existente no lance, a propriedade será gerada como uma chave estrangeira.

Partial View

São pedaços de código que podem ser reutilizados em várias views diferentes, algo similar a um (~~componente~~) componentes do React.

E utilizamos as PartialViews quando houver muito código repetido/parcido em várias views



↳ Eles não executam o arquivo `viewstart`, ou seja, não possuem o arquivo de layout.

↳ O nome de uma PartialView geralmente começa com """ (underline)



Poderemos ter partial views no posto `shared`, eles poderão ser usados em todo o projeto. E podemos (~~ter~~) ter essas em uma pasta de uma view específica assim elas só irão poder ser chamadas por aquela view.

tag helper

Partial View utilizando (PV)

Usamos ele com o tag partial e o atributo name que não referencia a partial view.

```
<partial name="caminho" />
```

Como possuímos o "caminho" do nome com somente o nome PV, ele não faz referência a pasta Shared. Agora se colocarmos a extensão .cshtml ele fará referência a pasta do view.

html helper

Poderemos usar os html helpers também

```
@Html.PartialAsync("name") for="nº"  
(~ ~ ~) RenderPartialAsync("name")
```

A questão da extensão do arquivo também se aplica.

No PV podemos apenas o @Layout para referenciar o tipo.

Se no for do tag paramos o model com os dados usados (delivered)

Código (@) Render (@)