Códigos Incríveis

```
using System;
namespace Cooperchip.ITDeveloper.Domain.Entities
{
    public abstract class EntityBase
    {
        protected EntityBase()
        {
            this.Id = Guid.NewGuid();
        }
        public Guid Id { get; set; }

        public override bool Equals(object obj)
        {
            var compareTo = obj as EntityBase;
            if (ReferenceEquals(this, compareTo)) return true;
            if (ReferenceEquals(null, compareTo)) return false;
            return Id.Equals(compareTo.Id);
        }

        public static bool operator ==(EntityBase a, EntityBase b)
        {
            if (ReferenceEquals(a, null) && ReferenceEquals(b, null)) return true;
            if (ReferenceEquals(a, null) || ReferenceEquals(b, null)) return false;
            return a.Equals(b);
        }

        public static bool operator !=(EntityBase a, EntityBase b) => !(a == b);

        public override int GetHashCode() => (GetType().GetHashCode() * 13) + Id.GetHashCode();

        public override string ToString() => GetType().Name + " [Id=" + Id + "]";

    }

}
```

Essa classe abstrata em C# chamada `EntityBase` serve como uma base para a criação de outras classes que representam entidades em um sistema. Ela contém propriedades e métodos comuns que podem ser herdados e utilizados por essas classes derivadas.

Aqui está uma explicação detalhada dos componentes da classe `EntityBase`:

- A linha `public abstract class EntityBase` declara a classe abstrata chamada `EntityBase`. A palavra-chave `abstract` indica que essa classe não pode ser instanciada diretamente, mas serve como uma base para outras classes que a estendem.
- A propriedade `public int Id { get; protected set; }` declara uma propriedade chamada `Id`, que representa o identificador único da entidade. O `get` permite acessar o valor do `Id`, enquanto o `protected set` permite que as classes derivadas definam o valor do `Id`.
- O método `public override bool Equals(object? obj)` substitui o método `Equals` da classe base `System.Object`. Ele compara se o objeto passado como parâmetro é igual ao objeto atual com base no valor do `Id`. Primeiro, ele verifica se os dois objetos são a mesma referência de memória usando `ReferenceEquals`. Se forem a mesma referência, retorna `true`. Caso contrário, compara se o objeto passado é nulo ou do tipo `EntityBase` e, em seguida, compara se o `Id` do objeto atual é igual ao `Id` do objeto passado.
- Os operadores `==` e `!=` são sobrecarregados para comparar dois objetos do tipo `EntityBase`. Eles utilizam o método `Equals` para realizar a comparação, verificando se os dois objetos são iguais.
- O método `public override int GetHashCode()` substitui o método `GetHashCode` da classe base `System.Object`. Ele retorna um valor numérico que representa o hash do objeto. Esse valor é calculado multiplicando o hash do tipo da classe pelo número primo 907 e somando o hash do `Id`.
- O método `public override string ToString()` substitui o método `ToString` da classe base `System.Object`. Ele retorna uma representação textual do objeto, que consiste no nome do tipo da classe (`GetType().Name`) seguido pelo valor do `Id`.

Essa classe abstrata `EntityBase` fornece uma base comum para todas as entidades que herdam dela. Ela define comportamentos como comparação de igualdade, cálculo de hash e uma representação textual padrão. As classes derivadas podem estender essa base e adicionar propriedades e métodos específicos para suas necessidades.

Mais detalhes sobre a sobrecarga do operador `==`

```
public static bool operator ==(EntityBase a, EntityBase b)
        {
            if (ReferenceEquals(a, null) && ReferenceEquals(b, null)) return true;
            if (ReferenceEquals(a, null) || ReferenceEquals(b, null)) return false;
            return a.Equals(b);
        }
```

Vamos analisar o método `operator ==` em mais detalhes:

- Esse método é uma sobrecarga do operador de igualdade `==` para a classe `EntityBase`.
- Ele recebe dois parâmetros do tipo `EntityBase`, `a` e `b`, que representam os objetos que serão comparados.
- O tipo de retorno é `bool`, ou seja, o método retorna um valor booleano indicando se os objetos são iguais ou não.
- O objetivo desse método é comparar se dois objetos `EntityBase` são iguais com base no valor do `Id`.

Aqui está o fluxo de execução do método:

1. Primeiro, ele verifica se tanto `a` quanto `b` são nulos, utilizando a função `ReferenceEquals(a, null) && ReferenceEquals(b, null)`. Se ambos forem nulos, isso significa que os objetos são considerados iguais, e o método retorna `true`.
2. Em seguida, ele verifica se apenas um dos objetos é nulo, utilizando a função `ReferenceEquals(a, null) || ReferenceEquals(b, null)`. Se apenas um dos objetos for nulo, isso significa que os objetos são diferentes, e o método retorna `false`.
3. Caso contrário, ou seja, se nenhum dos objetos for nulo, o método chama o método `Equals(b)` no objeto `a` para realizar a comparação. O `Equals` é um método que foi sobrescrito na classe `EntityBase` para comparar os valores dos `Id` dos objetos.
4. Se o método `Equals` retornar `true`, isso significa que os objetos são considerados iguais, e o método `operator ==` retorna `true`. Caso contrário, retorna `false`.

Resumindo, o método `operator ==` compara se dois objetos `EntityBase` são iguais com base no valor do `Id`. Ele trata casos em que os objetos são nulos e utiliza o método `Equals` para realizar a comparação quando ambos os objetos não são nulos.

▼ ViewModel

O arquivo físico de uma View Model pode estar em diferentes lugares, sendo:

Em uma pasta chamada ViewModels na estrutura raiz do projeto MVC (aplicações pequenas);
Uma *.dll referenciada no projeto MVC (aplicações de qualquer tamanho);
Em projetos separados (como uma camada de serviços) para gerar dados específicos (aplicações grandes).

Utilizando AutoMapper para realizar o mapeamento de uma Model e suas variantes:

Uma outra forma de utilizar ViewModels seria criando um mapeamento entre a Model, entidade de domínio, e a ViewModel que será exibida na View.
Alguns autores defendem fortemente a utilização de mapeamento entre as Models e suas possíveis variantes.

Benefícios de usar uma View Model:

Não alterar uma Model para atender as necessidades de uma View.
Agrupar informações de uma ou mais Models em uma única classe, inibindo a realização de N consultas.
Transporte de dados NÃO contidos em uma Model par a View Model.
Facilidade de mudanças sem afetar a Model do domínio.
Não “poluir” as Models de domínio com DataAnnotations de validação de formulário por estarem contidas diretamente na View Model.

Resumo:

O uso de View Models é recomendado para organizar e gerenciar os dados a serem transportados e exibidos, proporcionando flexibilidade para montar conjuntos de dados compatíveis com as necessidades da View.

A separação de responsabilidades no MVC é importante, onde “M” (Model) é o menos importante. A Model nem sempre estará presente e sim numa camada de domínio que proverá estas classes. A Model do MVC está voltada ao uso na View, portanto

algumas técnicas são importantes para separar e mapear Models de entidades de domínio para ViewModels neste tipo de projeto.