Dapper Micro ORM

Pequeno guia do Usuário

inTera Tecnologia

Sumário

[O que é o Dapper? 3](#_Toc465689792)

[Instalação 4](#_Toc465689793)

[Criando o Acesso a Dados 4](#_Toc465689794)

[Conexão com o Banco de Dados 5](#_Toc465689795)

[Consultas no Banco 5](#_Toc465689796)

[Criando o Model Customers 5](#_Toc465689797)

[Criando uma consulta com Like 7](#_Toc465689798)

[Comandos de transação 7](#_Toc465689799)

[Incluindo dados 8](#_Toc465689800)

[Alterando dados 8](#_Toc465689801)

[Referências 9](#_Toc465689802)

# O que é o Dapper?

O Dapper é um produto ORM (Object Relational Mapping) para a plataforma .NET. Ele provê um framework para mapear um modelo de domínio orientado a objeto para um banco de dados relacional. Sua finalidade é aliviar o desenvolvimento de uma parcela significativa de tarefas de relacionadas a persistência de dados em banco de dados relacionais.

O Dapper é grátis e é de código aberto sobre a licença ***Apache License 2.0*** [[1]](#footnote-1) ou ***MIT License***[[2]](#footnote-2)

Como é considerado um Micro ORM, ele não conta com todas as funcionalidades de um ORM mas facilita bastante o desenvolvimento que seja voltado a performance do banco de dados.

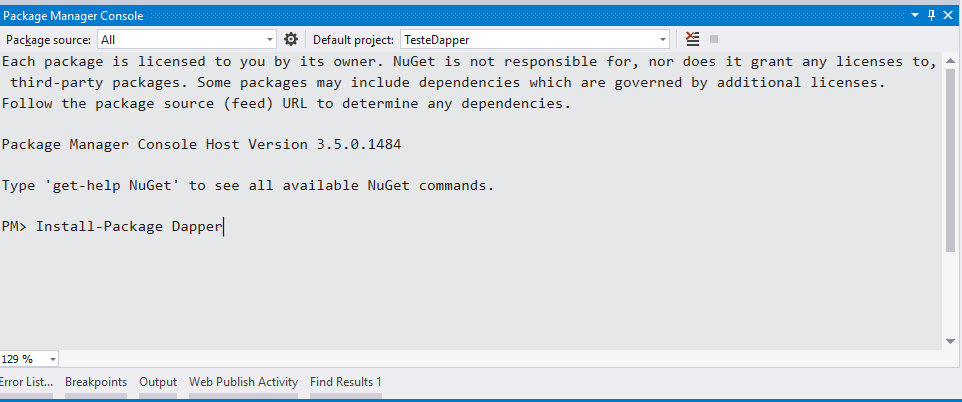
O Dapper ORM nada mais é do que uma extensão dos métodos que fazem acessos a dados no ADO.NET. Dessa forma, quando se faz uma consulta no banco de dados, o resultado pode ser mapeado uma lista de objetos de uma classe.

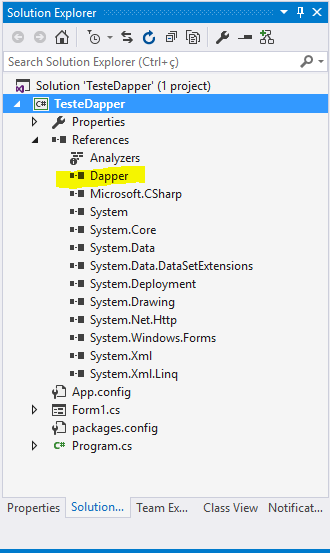
# Instalação

Após ter criado uma solução contendo um projeto que será responsável pela persistência de dados em banco de dados relacional, será necessário usar a ferramenta NuGet para fazer a instalação do Dapper ORM neste projeto.

Abra o menu Tools 🡪 NuGet Package Manager 🡪 Package Manager Console.

Quando a janela abrir, digite a seguinte linha de comando “*Install-Package Dapper”* como mostrado abaixo:





NOTA: Estarei usando o projeto do tipo **Windows Form** para exemplificar o uso do **Dapper**.

Após a instalação, poderemos constatar que dentro do Solution Explorer 🡪 References teremos o Dapper já disponível, como demonstra na ilustração ao lado:

Agora, que o desenvolvedor já tem a funcionalidade do Dapper instalado poderá fazer um simples teste de acesso a dados como é demonstrado no capítulo seguinte.

# Criando o Acesso a Dados

Para demonstrar o acesso a dados usando o Dapper ORM, vou usar o banco de dados Northwind que é um banco de dados de exemplo da Microsoft usando um SQL Server 2014 Express.

Nota: Não entrarei em detalhes os procedimentos da instalação do Northwind no SQL Server.

# Conexão com o Banco de Dados

Vamos criar uma classe no projeto de nome BaseData.cs dentro de uma pasta de nome DAL e terá o seguinte código:

1. **using** System.Data.SqlClient;
2. **namespace** TesteDapper.DAL
3. {
4. **public** **abstract** **class** BaseData
5. {
6. **public** **static** **string** StrConexao { **get**; **set**; } =
7. "Data Source=LOCALHOST;Initial Catalog=NORTHWIND;Integrated Security=True";
8. **public** **static** SqlConnection Conexao { **get**; **set**; }
10. **public** **void** InicializarConexao()
11. {
12. **if** (BaseData.Conexao == **null**)
13. **this**.IniciarConexao();
14. **if** (Conexao.State == System.Data.ConnectionState.Closed)
15. Conexao.Open();
16. }
18. **public** **void** IniciarConexao()
19. {
20. BaseData.Conexao = **new** SqlConnection(BaseData.StrConexao);
21. }
22. }
23. }

Essa classe abstrata servirá como molde das classes que farão consultas nos bancos através de herança.

# Consultas no Banco

A partir de agora vamos realizar consultas no banco de dados e aprender como preencher as listas de dados baseados em Modelos (classes usadas no mapeamento).

## Criando o Model Customers

Lembrando que a classe que iremos criar aqui servirá para mapear os dados que vem do Banco de Dados, logo, não somente os nomes dos campos devem ser iguais aos nomes das propriedades dessa classe, mas também os tipos de dados devem ser compatíveis.

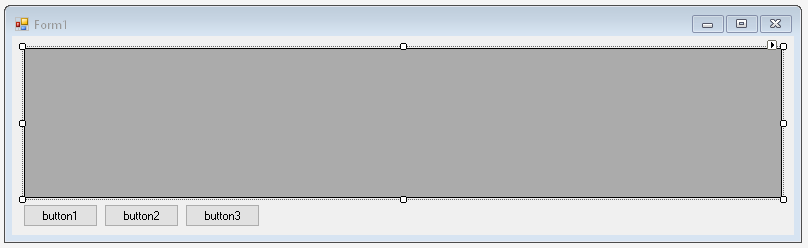
Antes de criarmos essa classe, vamos criar uma pasta de nome Model. Após isso, vamos criar a classe com o seguinte código:

1. **namespace** TesteDapper.Model
2. {
3. **public** **class** Customer
4. {
5. **public** **string** CustomerID { **get**; **set**; }
6. **public** **string** CompanyName { **get**; **set**; }
7. **public** **string** ContactName { **get**; **set**; }
8. **public** **string** ContactTitle { **get**; **set**; }
9. **public** **string** Address { **get**; **set**; }
10. **public** **string** City { **get**; **set**; }
11. **public** **string** Region { **get**; **set**; }
12. **public** **string** PostalCode { **get**; **set**; }
13. **public** **string** Country { **get**; **set**; }
14. **public** **string** Phone { **get**; **set**; }
15. **public** **string** Fax { **get**; **set**; }
16. }
17. }

Agora vamos criar o DAL para as primeiras consultas na tabela Customers(clientes) do banco de dados Northwind.

Dentro da pasta DAL crie uma classe de nome CustomerDAL que herda de BaseData, como demonstrado abaixo:

1. **using** System.Collections.Generic;
2. **using** Dapper;
3. **using** System.Linq;
4. **namespace** TesteDapper.DAL
5. {
6. **public** **class** CustomerDAL:BaseData
7. {
8. **public** List<Model.Customer> GetAll()
9. {
10. **this**.InicializarConexao();
11. List<Model.Customer> lista = Conexao
12. .Query<Model.Customer>("select \* from Customers")
13. .ToList();
14. **return** lista;
15. }
17. **public** List<Model.Customer> GetById(**string** CustomerID)
18. {
19. **this**.InicializarConexao();
20. List<Model.Customer> lista = Conexao
21. .Query<Model.Customer>(
22. "select \* from Customers where CustomerID = @CustomerID",
23. **new** { CustomerID = CustomerID })
24. .ToList();
25. **return** lista;
26. }
28. **public** List<Model.Customer> GetByIds(**string**[] CustomerIDs)
29. {
30. **this**.InicializarConexao();
31. List<Model.Customer> lista = Conexao
32. .Query<Model.Customer>(
33. "select \* from Customers where CustomerID in @CustomerIDs",
34. **new** { CustomerIDs = CustomerIDs })
35. .ToList();
36. **return** lista;
37. }
39. }
40. }

Pronto, temos agora o método que realiza as consultas prontas, agora vamos construir um formulário para realizar a execução desses métodos e preencher uma grade de dados, para isso, adicione no formulário Form1.cs os seguintes componentes:

* DataGridView
* Button (3x)

O formulário deverá ter a aparência da figura ao lado.

Abaixo teremos os códigos para cada botão adicionado ao formulário:

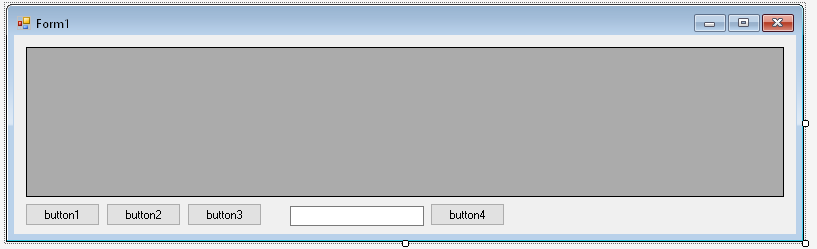
1. **private** **void** button1\_Click(**object** sender, EventArgs e)
2. {
3. DAL.CustomerDAL customerDAL = **new** DAL.CustomerDAL();
4. dataGridView1.DataSource = customerDAL.GetAll();
5. }
7. **private** **void** button2\_Click(**object** sender, EventArgs e)
8. {
9. DAL.CustomerDAL customerDAL = **new** DAL.CustomerDAL();
10. dataGridView1.DataSource = customerDAL.GetById("ALFKI");
11. }
13. **private** **void** button3\_Click(**object** sender, EventArgs e)
14. {
15. DAL.CustomerDAL customerDAL = **new** DAL.CustomerDAL();
16. dataGridView1.DataSource = customerDAL.GetByIds(**new** **string**[] {"ALFKI", "AROUT"});
17. }

Agora é só executar e testar.

## Criando uma consulta com Like

Adicione o seguinte método na classe CustomerDAL:

1. **public** List<Model.Customer> GetByContactName(**string** ContactName)
2. {
3. **this**.InicializarConexao();
4. ContactName = "%" + ContactName + "%";
5. List<Model.Customer> lista = Conexao
6. .Query<Model.Customer>(
7. "select \* from Customers where ContactName like @ContactName",
8. **new** { ContactName = ContactName })
9. .ToList();
10. **return** lista;
11. }

No formulário adicione os seguintes componentes:

* TextBox
* Button

O formulário deve apresentar com na figura ao lado.

Adicione o seguinte método ao button4.

1. **private** **void** button4\_Click(**object** sender, EventArgs e)
2. {
3. DAL.CustomerDAL customerDAL = **new** DAL.CustomerDAL();
4. dataGridView1.DataSource = customerDAL.GetByContactName(textBox1.Text);
5. }

# Comandos de Persistência

Agora vamos trabalhar com comandos de inclusão, alteração e exclusão de dados com o Dapper ORM.

## Incluindo dados

Adicione o seguinte método na classe CustomerDAL.cs:

1. **public** **int** Add(Model.Customer customer)
2. {
3. **this**.InicializarConexao();
5. **string** strComando =
6. @"INSERT Customers
7. (CustomerID, CompanyName, ContactName, ContactTitle, Address,
8. City, Region, PostalCode, Country, Phone, Fax)
9. VALUES
10. (@CustomerID, @CompanyName, @ContactName, @ContactTitle
11. ,@Address, @City, @Region, @PostalCode, @Country, @Phone
12. ,@Fax)";
14. **int** qtdRegistros = Conexao.Execute(strComando, customer);
16. **return** qtdRegistros;
17. }

## Alterando dados

1. **public** **int** Update(Model.Customer customer)
2. {
3. **this**.InicializarConexao();
4. **string** strComando =
5. @"UPDATE Customers
6. SET CompanyName = @CompanyName
7. ,ContactName = @ContactName
8. ,ContactTitle = @ContactTitle
9. ,Address = @Address
10. ,City = @City
11. ,Region = @Region
12. ,PostalCode = @PostalCode
13. ,Country = @Country
14. ,Phone = @Phone
15. ,Fax = @Fax
16. WHERE CustomerID = @CustomerID";
18. **int** qtdRegistros = Conexao.Execute(strComando, customer);
20. **return** qtdRegistros;
21. }

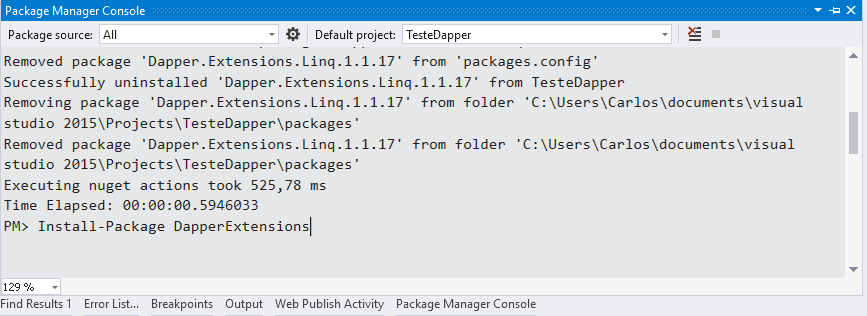
## Excluindo dados

1. **public** **int** Delete(Model.Customer customer)
2. {
3. **this**.InicializarConexao();
4. **string** strComando =
5. @"DELETE FROM Customer
6. WHERE CustomerID = @CustomerID";
8. **int** qtdRegistros = Conexao
9. .Execute(strComando, **new** { CustomerID = customer.CustomerID });
11. **return** qtdRegistros;
12. }

# Dapper Extensions

É uma biblioteca pequena que complementa o Dapper ORM adicionando operações de CRUD básico (Get, Insert, Update e Delete) para o POCO ou DTO.

## Instalando o Dapper Extensions no projeto

Abra novamente a janela do NuGet Package Manager 🡪 Package Manager Console para digitarmos o comando de instalação desse pacote.

**Install-package DapperExtensions**

Após a instalação, poderemos refazer os métodos de consulta e persistência para usar com essa extensão.

## Refazendo as consultas

Abaixo teremos os código dos métodos do GetALL(), GetByID() e getByContactName() reescrito usando o Dappre Extensions.

1. **public** List<Model.Customer> GetAllv2()
2. {
3. **this**.InicializarConexao();
4. List<Model.Customer> lista = Conexao
5. .GetList<Model.Customer>()
6. .ToList();
7. **return** lista;
8. }
10. **public** List<Model.Customer> GetByIdv2(**string** CustomerID)
11. {
12. **this**.InicializarConexao();
13. var predicate = Predicates
14. .Field<Model.Customer>(f => f.CustomerID, Operator.Eq, CustomerID);
16. List<Model.Customer> lista = Conexao
17. .GetList<Model.Customer>(predicate)
18. .ToList();
19. **return** lista;
20. }
22. **public** List<Model.Customer> GetByContactNamev2(**string** ContactName)
23. {
24. **this**.InicializarConexao();
25. var predicate = Predicates
26. .Field<Model.Customer>(f => f.ContactName, Operator.Like, ContactName);
28. List<Model.Customer> lista = Conexao
29. .GetList<Model.Customer>(predicate)
30. .ToList();
31. **return** lista;
32. }

## Refazendo os métodos de persistências

Agora vamos refazer os métodos Add(), Update() e Delete() usando o Dapper Extensions

1. **public** **int** Addv2(Model.Customer customer)
2. {
3. **this**.InicializarConexao();
4. **int** qtdRegistros = Conexao.Insert(customer);
5. **return** qtdRegistros;
6. }
8. **public** **bool** Updatev2(Model.Customer customer)
9. {
10. **this**.InicializarConexao();
11. **bool** executed = Conexao.Update(customer);
12. **return** executed;
13. }
15. **public** **bool** Deletev2(Model.Customer customer)
16. {
17. **this**.InicializarConexao();
18. **bool** executed = Conexao.Delete(customer);
19. **return** executed;
20. }

# Usando transações

Iremos agora trabalhar com transações usando o Dapper ORM controlando o Commit e o Rollback.

Como o Dapper ORM é apenas uma extensão do IDBConnection, iremos usar aqui os conceitos do ADO.NET para realizar essas tarefas.

Seguindo o código abaixo iremos demonstrar o uso de uma transação para realizar a gravação de dados.

1. **public** **int** AddRange(Model.Customer[] customers)
2. {
3. **this**.InicializarConexao();
4. var transacao = Conexao.BeginTransaction();
5. **int** qtdRegistros = 0;
7. **try**
8. {
9. **foreach** (Model.Customer cust **in** customers)
10. {
11. Conexao.Insert(cust, transacao);
12. qtdRegistros++;
13. }
14. transacao.Commit();
15. }
16. **catch** (System.Exception)
17. {
18. qtdRegistros = 0;
19. transacao.Rollback();
20. }
22. **return** qtdRegistros;
23. }

## Salvando pontos de recuperação na transação

A função salvar pontos de recuperação é uma implementação do ADO.NET, logo poderemos usar juntamente com o Dapper ORM.

É possível gravar um ponto executando o método .Save(“nome”) a partir de uma transação criada.

Para desfazer todo que foi feito nesse ponto, execute o método .Rollback(“nome”) na transação e o banco de dados vai realizar o rollback apenas daquele ponto.

Para exemplificar veremos o método AddCustomersAndRegions que adiciona uma lista de Clientes e Regiões, caso a inserção de algumas dessas listas der erro, não deverá impedir a inserção da outra.

1. **public** **void** AddCustomersAndRegions(Model.Customer[] customers, Model.Region[] regions)
2. {
3. **this**.InicializarConexao();
4. var transacao = Conexao.BeginTransaction();
6. **try**
7. {
8. transacao.Save("customers");
10. **try**
11. {
12. **foreach** (Model.Customer cust **in** customers)
13. {
14. Conexao.Insert(cust, transacao);
15. }
16. }
17. **catch** (System.Exception)
18. {
20. transacao.Rollback("customers");
21. }
23. transacao.Save("regions");
24. **try**
25. {
26. **foreach** (Model.Customer cust **in** customers)
27. {
28. Conexao.Insert(cust, transacao);
29. }
30. }
31. **catch** (System.Exception)
32. {
33. transacao.Rollback("regions");
34. }
36. transacao.Commit();
37. }
38. **catch** (System.Exception)
39. {
40. transacao.Rollback();
41. }
42. }

# Referências

Wikipedia, Dapper ORM, Disponível em <https://en.wikipedia.org/wiki/Dapper_ORM>, acessado em: 27/10/2016

1. https://www.apache.org/licenses/LICENSE-2.0 [↑](#footnote-ref-1)
2. https://opensource.org/licenses/MIT [↑](#footnote-ref-2)