Advance Web Technology

Lecture 06
Asrar Ahmad Ehsan

Introduction

- Dapper.NET is one of the popular ORM (Object Relational Mapper) for .NET.
  - Open source, light weight, fast, simple and easy to use ORM.
  - Built directly on top of ADO.NET
- It is not database specific, it works across all .NET ADO providers such as SQL Server, SQLite, Firebird, Oracle, MySQL, PostgreSQL.
- Dapper falls into a family of tools known as micro-ORMs. These tools perform only a subset of the functionality of full-blown Object Relations Mappers, such as Entity Framework Core.
- The primary reason for its existence is ***performance***. The original developers of Dapper were using Entity Framework Core's predecessor – the short-lived Linq to SQL.
- They found that query performance wasn't good enough for the increasing traffic that the site in question (Stackoverflow) was experiencing, so they wrote their own micro ORM.

What is ORM?

- An object-relational mapper provides an object-oriented layer between relational databases and object-oriented programming languages without having to write SQL queries.

```
book_list = new List();
sql = "SELECT book FROM library WHERE author = 'Linus'";
data = query(sql); // I over simplify ...
while (row = data.next())
{
     book = new Book();
     book.setAuthor(row.get('author');
     book_list.add(book);
}
```

- With an ORM library, it would look like this:

```
book_list = BookTable.query({author="Linus"});
```

Dapper Example

```
var sql = "select * from products";
var products = new List<Product>();
using (var connection = new SqlConnection(connString))
{
    connection.Open();
    using (var command = new SqlCommand(sql, connection))
    {
        using (var reader = command.ExecuteReader())
        {
            var product = new Product
            {
                ProductId = reader.GetInt32(reader.GetOrdinal("ProductId")),
                ProductName = reader.GetString(reader.GetOrdinal("ProductName")),
                SupplierId = reader.GetInt32(reader.GetOrdinal("SupplierId")),
                CategoryId = reader.GetInt32(reader.GetOrdinal("CategoryId")),
                QuantityPerUnit = reader.GetString(reader.GetOrdinal("QuantityPerUnit")),
                UnitPrice = reader.GetDecimal(reader.GetOrdinal("UnitPrice")),
                UnitsInStock = reader.GetInt16(reader.GetOrdinal("UnitsInStock")),
                UnitsOnOrder = reader.GetInt16(reader.GetOrdinal("UnitsOnOrder")),
                ReorderLevel = reader.GetInt16(reader.GetOrdinal("ReorderLevel")),
                Discontinued = reader.GetBoolean(reader.GetOrdinal("Discontinued")),
                DiscontinuedDate = reader.GetDateTime(reader.GetOrdinal("DiscontinuedDate"))
            };
            products.Add(product);
        }
    }
}
```

```
products = connection.Query<Product>(sql);
```

Installation

- Dapper is available from Nuget. It is compliant with .NET Standard 2.0 which means that it can be used in .NET applications that target the full framework and .NET Core. You can install the latest version using the following command with the .NET CLI:

```
dotnet add package Dapper
```

- or this command from the Package Manager Console in Visual Studio:

```
install-package Dapper
```

- For MySQL or PostgreSQL you need to install the following packages as well.
- For MySql: `Install-Package MySql.Data`
  For PostgreSql: `Install-Package Npgsql`

Connection to DB

- The first step toward using Dapper is of course after installation is the connection to the database. We need to create our won connection since the dapper doesn’t do that for us.
- SQL Server:

```
var connectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";

using (var connection  = new SqlConnection(connectionString));

  //Do some magic here
```

- MySQL:

```
var connectionString =
"Server=ServerAddress;Database=myDataBase;Uid=myUsername;Pwd=myPassword;";

using (var connection  = new MySqlConnection(connectionString));

  //Do some magic here
```

Retrieving Multiple Data

- Dapper provides a number of methods for selecting data, depending on how you want to work with the data that you retrieve.

Method	Description
`Query`	Returns an enumerable of dynamic types
`Query<T>`	Returns an enumerable of the type specified by the T parameter
`QueryAsync`	Returns an enumerable of dynamic types asynchronously
`QueryAsync<T>`	Returns an enumerable of the type specified by the T parameter asynchronously

Retrieving Multiple Data

```
using (var connection = new SqlConnection(connString))
{
    var sql = "select * from customers";
    var customers = connection.Query(sql);
    foreach(var customer in customers)
    {
        Console.WriteLine($"{customer.CustomerID} {customer.CompanyName}");
    }
    Console.ReadLine();
}
```

Retrieving Multiple Data

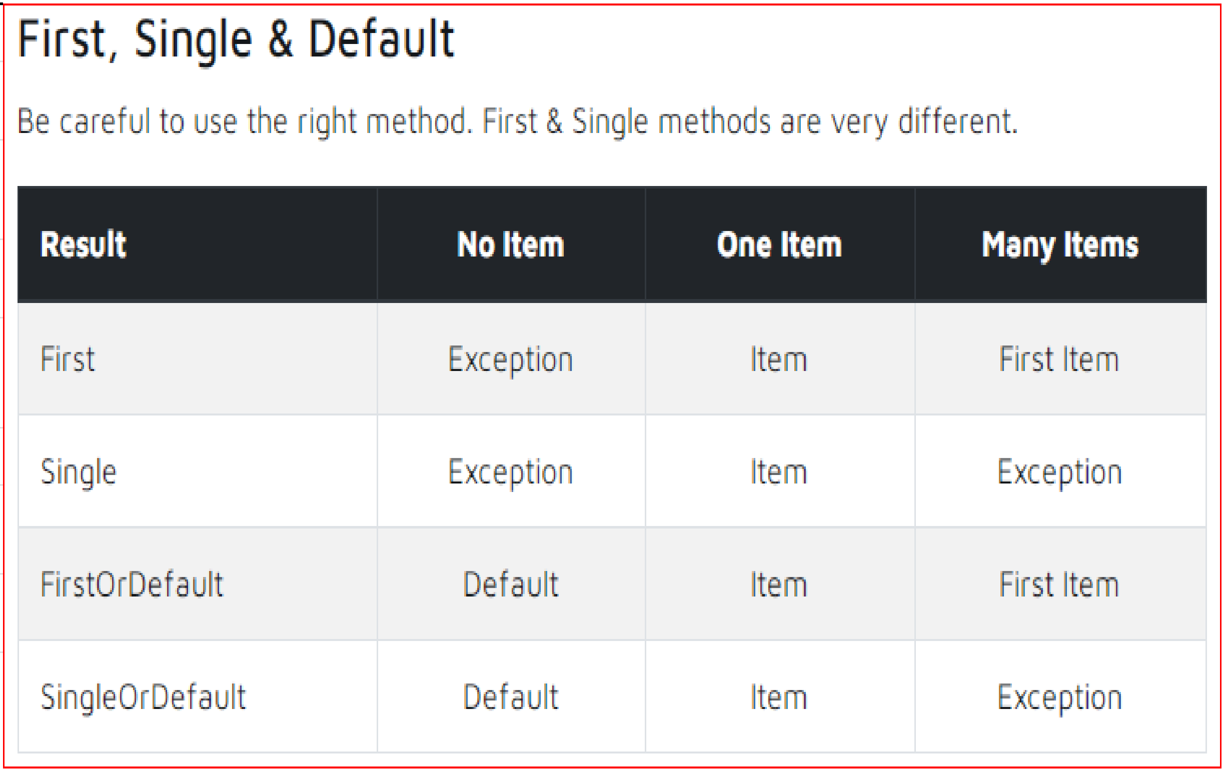
```
Public class Customer
{
    public int CustomerId {get; set;}
    public string FullName {get; set;}
}
```

```
using (var connection = new SqlConnection(connString))
{
    var sql = "select * from customers";
    List<Customer> customers = connection.Query<Customer>(sql);
    foreach(var customer in customers)
    {
        Console.WriteLine($"{customer.CustomerId} {customer.FullName}");
    }
    Console.ReadLine();
}
```

Retrieving Single Row

- Dapper provides a number of methods for selecting single rows of data, depending on how you want to work with the data that you retrieve.

Method
`QuerySingle`
`QuerySingle<T>`
`QuerySingleOrDefault`
`QuerySingleOrDefault<T>`
`QueryFirst`
`QueryFirst<T>`
`QueryFirstOrDefault`
`QueryFirstOrDefault<T>`



First, Single & Default

Be careful to use the right method. First & Single methods are very different.

Result	No Item	One Item	Many Items
First	Exception	Item	First Item
Single	Exception	Item	Exception
FirstOrDefault	Default	Item	First Item
SingleOrDefault	Default	Item	Exception

Retrieving Single Row

```
using (var connection = new SQLiteConnection(connString))
{
    var sql = "select * from products where productid = 1";
    var product = connection.QuerySingle(sql);
    Console.WriteLine($"{product.ProductID} {product.ProductName}");
}

// both queries are equal

using (var connection = new SQLiteConnection(connString))
{
    var sql = "select * from products where productid = 1";
    var product = connection.QuerySingle<Product>(sql);
    Console.WriteLine($"{product.ProductID} {product.ProductName}");
}
```

Scalar Values

- Dapper provides two methods for selecting scalar values along with their asynchronous counterparts.

Method	Description
`ExecuteScalar`	Returns a dynamic type
`ExecuteScalar<T>`	Returns an instance of the type specified by the T type parameter
`ExecuteScalarAsync`	Returns a dynamic type asynchronously
`ExecuteScalarAsync<T>`	Returns an instance of the type specified by the T type parameter asynchronously

Scalar Values

```
using (var connection = new SQLiteConnection(connString))
{
    var sql = "select count(*) from products";
    var count = connection.ExecuteScalar(sql);
    Console.WriteLine($"Total products: {count}");
}



using (var connection = new SQLiteConnection(connString));

var sql = "select count(*) from products";
var count = connection.ExecuteScalar<int>(sql);
Console.WriteLine($"Total products: {count}");
```

Non Commands

- Dapper provides the Execute method (and its async equivalent) for commands that are not intended to return result sets i.e. INSERT, UPDATE and DELETE commands. The Execute method returns an int, representing the number of rows affected by the successful completion of the command.

- **Insert**

```
var sql = "insert into categories (Name) values ('New Category')";
using (var connection = new SqlConnection(connString))
{
    var affectedRows =  connection.Execute(sql);
    Console.WriteLine($"Affected Rows: {affectedRows}");
}
```

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL

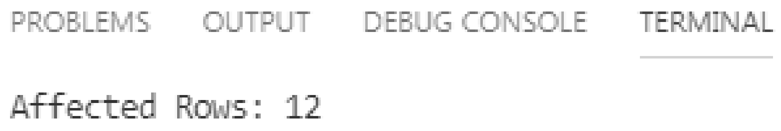
```
Affected Rows: 1
```

Non Commands

- **Update**

```
var sql = "update products set unitprice = unitprice * .1 where
categoryid = 2";

using (var connection = new SqlConnection(connString));
var affectedRows =  connection.Execute(sql);
Console.WriteLine($"Affected Rows: {affectedRows}");
```

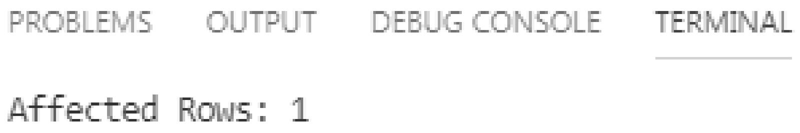


```
Affected Rows: 12
```

- **Delete**

```
var sql = "delete from categories where CategoryName = 'New Category'";

using (var connection = new SqlConnection(connString));
var affectedRows =  connection.Execute(sql);
Console.WriteLine($"Affected Rows: {affectedRows}");
```



```
Affected Rows: 1
```

Query Parameters

- Parameters are represented in the SQL command by placeholders, and the values are passed to the command within the DbCommand object's Parameters collection.
- The format of the placeholder is dependant on what the provider supports. The SqlClient provider supports named parameters, with the parameter name prefixed with an @ character.
- The OleDb provider supports positional parameters. Values are passed to the SQL command based on matching the order in which they have been added to the parameters collection to the order in which placeholders appear in the SQL.
- Parameter placeholders can be named anything, as long as the placeholder names don't match database object names (columns, tables etc).

Parameters As Anonymous Types

- Parameter values can be passed to commands as anonymous types:

```
var parameters = new { UserName = username, Password = password };
var sql = "select * from users where username = @UserName and password
= @Password";
var result = connection.Query(sql, parameters);
```

DynamicParameters Bag

- Dapper also provides a DynamicParameters class, which represents a "bag" of parameter values. You can pass an object to its constructor. Suitable objects include a Dictionary<string, object>:

```
var dictionary = new Dictionary<string, object>
{
    { "@ProductId", 1 }
};
var parameters = new DynamicParameters(dictionary);
var sql = "select * from products where ProductId = @ProductId";
using (var connection = new SqlConnection(connString))
{
    var product = connection.QuerySingle<Product>(sql, parameters);
}
```

DynamicParameters Bag

- Or you can pass a stub representing the object that you are looking to return:

```
var template = new Product { ProductId = 1 };
var parameters = new DynamicParameters(template);
var sql = "select * from products where ProductId = @ProductId";
using (var connection = new SqlConnection(connString))
{
    var product = connection.QuerySingle<Product>(sql, parameters);
}
```

DynamicParameters Bag

- The DynamicParameters type provides an Add method, enabling you to pass explicit parameters, specifying the datatype, direction and size:

```
var parameters = new DynamicParameters();
var customerId = "ALFKI";

parameters.Add("@CustomerId", customerId, DbType.String,
ParameterDirection.Input, customerId.Length);

parameters.Add("@msg",type: DbType.String, direction:
ParameterDirection.Output, 500);

var sql = "select * from customers where CustomerId = @CustomerId";
using (var connection = new SqlConnection(connString))
{
    var customer = connection.QuerySingle<Customer>(sql, parameters);
}

string returnedMessage = parameters.Get<string>("msg");
```

Executing Stored Procedures

- There are two ways to execute stored procedures with Dapper: with the CommandType as Text; or as StoredProcedure.
  - **CommandType.Text**
  - **CommandType.StoredProcedure**

❑CommandType.Text
All DbCommand objects have a CommandType property. By default this is set to Text. If you want to execute a stored procedure in a SQL statement (text) you use the Execute (or Exec) statement:

```
var sql = "exec [Sales by Year] @Beginning_Date, @Ending_Date";
var values = new { Beginning_Date = "2017.1.1", Ending_Date =
"2017.12.31" };
var results = connection.Query(sql, values).ToList();
results.ForEach(r => Console.WriteLine($"{r.OrderID} {r.Subtotal}"));
```

Executing Stored Procedures

❑CommandType.StoredProcedure
Dapper provides access to the CommandType property via the commandType parameter included in all the various query and execute methods. The next example is the equivalent to the above, but with the CommandType set to StoredProcedure:

```
var procedure = "[Sales by Year]";
var values = new { Beginning_Date = "2017.1.1", Ending_Date =
"2017.12.31" };
var results = connection.Query(procedure, values, commandType:
CommandType.StoredProcedure).ToList();
results.ForEach(r => Console.WriteLine($"{r.OrderID} {r.Subtotal}"));
```

Thank You !